Tampere University

Alex Ingman

NETWORK API SECURITY TESTING IN MODERN EMBEDDED SYSTEMS

Master's Thesis
Faculty of Information Technology and Communication Sciences
Examiner: University Instructor Marko Helenius
Examiner: University Instructor Matti Monnonen
May 2025

ABSTRACT

Alex Ingman: Network API Security Testing in Modern Embedded Systems
Master's Thesis
Tampere University
Master's Programme in Information Technology
May 2025

The growing interconnectivity of embedded systems has significantly increased their exposure to cyber threats. These systems, which are often deployed in critical environments such as telecommunications, industrial automation, and defence, commonly use modern API technologies, such as REST, WebSocket, and QUIC, for communication, data exchange, and remote management. As these APIs provide external access points into the system, their security becomes a top priority during development and deployment. This thesis focuses on practical methods for security testing of such APIs in embedded systems.

The study begins by examining the unique characteristics of embedded systems that differentiate them from traditional IT environments. These include limited hardware resources, custom protocol implementations, and real-time constraints, all of which can make conventional testing tools less effective. To address these challenges, the research evaluates several testing techniques suitable for embedded use cases, including fuzz testing, vulnerability scanning, and penetration testing. Special attention is given to the use of mutual TLS authentication and its role in securing API communications.

A key objective of this thesis is to explore how security testing can be automated and integrated into the software development process. The implementation combines open-source tools, such as Boofuzz, SSLyze, Scapy, and Robot Framework, to conduct protocol-aware testing that is both repeatable and efficient. The test system used in this research is based on a Yocto Linux distribution running on an SoC-FPGA platform, providing a realistic embedded environment for experimentation. Security testing was applied to APIs using TLS-secured REST and WebSocket interfaces, a QUIC-based data stream, and an SSH service intended for remote access.

The testing methods were evaluated for their effectiveness in identifying potential vulnerabilities and misconfigurations. Results indicate that automated testing can reliably uncover many classes of common issues, while some complex or logic-based vulnerabilities still require manual investigation. Furthermore, the work demonstrates how these testing tools can be integrated into continuous development workflows, helping to detect issues earlier in the product lifecycle.

This thesis provides actionable insights for developers and engineers working with networked embedded systems. It emphasizes the importance of a balanced approach that combines automation with selective manual testing, tailored to the technical and operational constraints typical of embedded platforms. By addressing both the technical and organizational aspects of embedded API security, the work contributes toward building more secure and robust embedded devices.

Keywords: Embedded systems, API security, Fuzz testing, Vulnerability scanning, Automation in security testing, Penetration testing, Security testing tools

TIIVISTELMÄ

Alex Ingman: Verkon ohjelmointirajapintojen turvallisuustestaus moderneissa sulautetuissa järjestelmissä
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Toukokuu 2025

Sulautettujen järjestelmien lisääntyvä verkottuminen on lisännyt niiden haavoittuvuutta tietoturvauhkia vastaan. Näitä järjestelmiä käytetään usein kriittisissä ympäristöissä, kuten tietoliikenteessä, teollisuusautomaation sovelluksissa ja puolustusteknologiassa. Niissä hyödynnetään moderneja ohjelmointirajapintoja, kuten REST, WebSocket ja QUIC, laitteen hallintaan, tiedonsiirtoon ja pilvipalveluihin yhdistämiseen. Koska nämä rajapinnat tarjoavat ulkoisia yhteyspisteitä järjestelmään, niiden tietoturva nousee keskeiseksi kehitysvaiheessa. Tämä opinnäytetyö keskittyy käytännön menetelmiin sulautettujen järjestelmien API-rajapintojen tietoturvatestauksessa.

Työssä tarkastellaan aluksi sulautettujen järjestelmien erityispiirteitä, jotka erottavat ne perinteisestä IT-ympäristöstä. Näitä ovat esimerkiksi rajalliset resurssit, mukautetut protokollatoteutukset ja reaaliaikavaatimukset, jotka tekevät perinteisten tietoturvatyökalujen soveltamisesta haastavaa. Näihin haasteisiin vastaamiseksi työssä arvioidaan erilaisia testausmenetelmiä, kuten fuzz-testausta, haavoittuvuusskannauksia ja tunkeutumistestauksia. Erityishuomio kohdistetaan myös mTLS-todennukseen ja sen merkitykseen rajapintojen suojaamisessa.

Työn keskeinen tavoite on selvittää, kuinka tietoturvatestausta voidaan automatisoida ja ottaa osaksi ohjelmistokehitysprosessia. Käytännön toteutuksessa yhdistetään avoimen lähdekoodin työkaluja, kuten Boofuzz, SSLyze, Scapy ja Robot Framework, testauksen suorittamiseen tavalla, joka ottaa huomioon rajapintojen erityispiirteet ja on helposti toistettavissa. Testausta varten käytetty sulautettu järjestelmä perustuu Yocto Linux jakeluun ja SoC-FPGA-alustaan, tarjoten realistisen testausympäristön. Tietoturvatestausta sovellettiin TLS-suojattuihin REST- ja WebSocket-rajapintoihin, QUIC-protokollaan perustuvaan tiedonsiirtoon ja SSH-etäyhteyksiin.

Menetelmien tehokkuutta arvioitiin haavoittuvuuksien ja virheellisten kokoonpanojen tunnistamisessa. Tulokset osoittavat, että automaattinen testaus paljastaa luotettavasti yleisimmät ongelmatyypit, mutta tietyt monimutkaisemmat tai loogiset virheet edellyttävät edelleen manuaalista testausta. Lisäksi työssä esitetään, miten valitut testausratkaisut voidaan liittää jatkuvaan kehitysprosessiin, jotta tietoturvaongelmat voidaan tunnistaa jo varhaisessa vaiheessa.

Työ tarjoaa käytännönläheisiä näkemyksiä kehittäjille ja tietoturva-asiantuntijoille, jotka työskentelevät verkottuneiden sulautettujen järjestelmien parissa. Se korostaa tasapainoisen lähestymistavan merkitystä, jossa automatisointia täydennetään valikoidulla manuaalisella testauksella. Näin voidaan vastata sulautettujen järjestelmien teknisiin ja toiminnallisiin erityisvaatimuksiin ja parantaa niiden turvallisuutta ja luotettavuutta merkittävästi.

Avainsanat: Sulautetut järjestelmät, API turvallisuus, Fuzz-testaus, Haavoittuvuusskannaus, Turvallisuustestauksen automaatio, Penetraatiotestaus, Turvallisuustestauksen työkalut

USE OF AI IN THESIS

I have utilised AI tools in my thesis:

☐ No
☒ Yes

The AI tools utilised in my thesis and their purposes are described below:

Name and versions of AI tool: Grammarly Free

Purpose of using AI tool: Improving grammar and flow of text while writing.

I acknowledge that I am fully responsible for the entire content of my thesis, including the parts generated by AI, and accept accountability for any violations of ethical standards in publications.

PREFACE

I want to sincerely thank my colleagues at work and my family for their support, encouragement, and understanding during this journey.

I am also deeply thankful to my friends from university for all the unforgettable memories and shared experiences, these years have passed in the blink of an eye.

Now, with a mix of gratitude and nostalgia, it’s time to move forward, feeling proud that the hard work of the past five years has led to this moment of graduation and new beginnings.

Tampere, 20th May 2025

Alex Ingman

CONTENTS

LIST OF FIGURES

LIST OF PROGRAMS

ABBREVIATIONS

ADC	Analog-to-Digital Converter
ALPN	Application-Level Negotiation Field
API	Application Programming Interface
CI/CD	Continuous Integration / Continuous Deployment
DAC	Digital-to-Analog Converter
DAST	Dynamic Application Security Testing
DoS	Denial-of-Service
FPGA	Field Programmable Gate Array
HTTP	Hypertext Transfer Protocol
IT	Information Technology
JSON	JavaScript Object Notation
JWT	JSON Web Token
mTLS	Mutual Transport Layer Security
NMap	Network Mapper (security scanner)
QUIC	Quick UDP Internet Connections
REST	Representational State Transfer
RF	Radio Frequency
SAST	Static Application Security Testing
SCA	Software Composition Analysis
SoC	System on Chip
SSH	Secure Shell
STK	Source-Address Token
TLS	Transport Layer Security
UI	User Interface
XSS	Cross-Site Scripting
ZAP	OWASP Zed Attack Proxy

1. INTRODUCTION

The increasing connectivity of embedded systems has expanded their capabilities and exposure to security threats. As these systems evolve to support modern communication protocols and services, ensuring the security of external interfaces becomes critical. These APIs and services are commonly used for device management, data exchange, and integration with cloud platforms, making them attractive targets for potential attackers. Especially comprehensive security testing of embedded systems is important in critical sectors like telecommunications, defense, and industrial systems.

To address these concerns, this thesis develops network API security testing methodologies tailored for modern embedded systems. The thesis focuses on the security assessment of modern API technologies, specifically REST, WebSocket, and QUIC, as well as remote management services like SSH. Each protocol introduces distinct attack surfaces and implementation challenges that demand dedicated testing strategies. To address this, the testing includes a variety of techniques, including automated fuzzing, protocol-specific test case generation, and manual exploratory testing.

A key goal of this work is to explore the potential of security testing tools for test automation, mostly with Robot Framework, but also by investigating their use in the context of continuous integration and continuous development (CI/CD). The goal is also to identify areas that might require manual security testing, which could be automated. By combining protocol-aware security testing with automation, the designed testing is intended to assist developers and security engineers in identifying vulnerabilities early in the development lifecycle.

Although the technologies discussed in the work are in principle like those used in traditional IT systems, the work also seeks to note the differences in their use as part of an embedded system. In general, testing traditional protocols in embedded systems differs significantly from general-purpose environments due to resource constraints, minimal or custom protocol implementations, limited observability, and hardware-dependent behavior. Embedded devices often lack the CPU, memory, or logging capabilities required for standard security tools, and may use lightweight protocol stacks that diverge from full specifications. Real-time requirements and tight integration with physical hardware also increase the risk of unintended disruptions during testing. Additionally, the difficulty of updating deployed embedded systems makes early, automated, and reliable security

testing essential. These unique challenges demand tailored and adaptable security testing strategies that align with the embedded system's operational context.

1.1 Research questions

The primary objective of this thesis is to investigate the effectiveness, practicality, and limitations of security testing approaches for external APIs in embedded systems. As embedded systems increasingly rely on networked communication, ensuring the security of exposed APIs becomes critical. This study aims to explore both the technical tools available and the testing methodologies best suited for these environments, with a particular emphasis on automation and practical deployment.

To guide the research, the following questions have been formulated:

1. What security testing techniques are most effective for securing external APIs in embedded systems?
2. How does the security testing of common API protocols differ when applied to embedded systems?
3. What tools and frameworks are most suitable for implementing security tests on embedded system external APIs?
4. How can security testing of external APIs in embedded systems be automated?
5. Which areas of embedded system external API security testing require manual effort in addition to automation capabilities?

2. EMBEDDED SYSTEM API SECURITY TESTING OVERVIEW

This chapter presents the key concepts and technologies that form the foundation of this thesis. It begins with an overview of security testing principles, followed by a discussion of how these practices are adapted for embedded systems. Since external APIs are a critical interface in embedded environments, their role and characteristics are also explored in detail. The chapter then introduces common security testing techniques. This includes fuzzing, penetration testing, and vulnerability scanning. Finally, the concept of authentication is examined, as it plays a central role in securing API interactions. Together, these sections provide the theoretical and technical background necessary to understand the scope and approach of the research.

2.1 Security testing overview

Security testing is an important part of the software development lifecycle, which aims to discover vulnerabilities, potential threats, and weaknesses in information systems. While functional testing ensures that the system performs as expected under normal conditions, security testing assesses the resilience of the system against malicious attacks, unauthorized access, and data breaches. These considerations highlight the importance of comprehensive security testing in the overall evaluation of software systems. Functional and security testing combined ensure that the system maintains its functionality and security under different conditions. It is also particularly important to consider the inclusion of security in all phases of the software development lifecycle, such as requirement-gathering, development, testing, deployment, and maintenance [1].

Security testing is not only a technical process but also an important aspect of demonstrating a company's commitment to responsible product development. By investing in security testing, organizations aim to protect sensitive data and maintain the integrity of their systems. This contributes to compliance with legal and industry standards and helps foster customer trust. When products are systematically tested for vulnerabilities, customers can have greater confidence in their security and reliability.

Each information system demands careful design decisions to ensure its information security. Technologies are chosen for data transmission, data encryption, authentication, etc. Security testing aims to ensure the robustness of the implemented information security solutions. This also requires security testers to be creative because attackers will

find innovative ways to compromise the system. For them, any way to gain access to some valuable information or, for example, to disrupt the operation of a system, is a potential path to achieve their goal. It is also worthwhile to check carefully different lists of common vulnerabilities, such as the OWASP top 10, and investigate whether systems under test might have some commonly known vulnerabilities and security problems [1] [2].

In security testing, it is important to consider potential attacks from various threat actors. An attacker could be a completely external party or an authenticated user with legitimate access to the system. While their motives may differ, both scenarios pose significant risks. Therefore, security testing should address a broad range of attack vectors to ensure comprehensive coverage. For example, it is essential to ensure that a client can only access data they are authorized to view and have modification or update rights only where explicitly permitted.

It is also important to recognize that security requirements can differ depending on the stakeholder's perspective. From the manufacturer's point of view, protecting intellectual property, such as proprietary algorithms and system designs, is a key concern. In contrast, end-users are often more focused on the confidentiality of their data, the integrity of operations, and the secure integration of the device into their networks. Security testing must therefore address both perspectives to ensure that the product meets the expectations and requirements of all parties involved.

Integrating security testing throughout the product development process is essential, and automation plays a crucial role in making this possible. By embedding automated security tests from the early stages of design to the final deployment, teams can catch potential issues quickly and efficiently. Key benefits of test automation include resource savings and test repeatability, which reduces the potential for human error in testing. This continuous, automated approach helps identify and fix vulnerabilities before they become major problems. It also develops a culture of ongoing improvement and accountability. In today’s fast-paced development environment, using automation to distribute security testing across every phase ensures that products are not only built with care but also remain protected against the ever-changing landscape of cyber threats. There are also some tasks related to comprehensive security testing of products that require more manual work, for example, manual penetration testing. To achieve comprehensive security testing coverage, a combination of manual and automated testing is required. While automated tools can efficiently detect a wide range of common vulnerabilities, certain issues, such as logic flaws or context-specific weaknesses, often require manual

testing to be identified. Manual testing, therefore, remains an essential complement to automation.

2.2 Security testing in embedded systems

There are some unique challenges in security testing for embedded systems compared to traditional information technology applications, since embedded systems are often a customized combination of hardware and software. That opens the possibility of unexpected vulnerabilities. It is also important to point out that when traditional embedded systems have commonly been isolated systems, they are nowadays interconnected, which makes them vulnerable to new kinds of threats and vulnerabilities [3]. This aspect must be considered not only when implementing security in embedded systems, but also during security testing.

Security is often extremely important in embedded systems as they operate in such highly critical areas. For example, in the defense industry, in critical infrastructure such as communication networks or electricity grids, or in the health sector. In industries like these, the security of equipment must be guaranteed in all situations. [3]

As embedded systems are nowadays interconnected, attacks and potential vulnerabilities at their external interfaces are becoming more common. External interfaces are targeted by multiple kinds of threats, including known-key, meet-in-the-middle, and man-in-the-middle attacks [4]. These are examples of how potential vulnerabilities might be exploited if APIs are not designed securely.

The security risks associated with the hardware must be considered in an embedded system. These include the physical interfaces and debug interfaces of the system, which need to be carefully configured in the finished product, as they can provide pathways to bypass the device's protection. Also, physical tampering and side-channel attacks are possible attack variations that focus on hardware. However, this is not exactly related to the topic of this work, so it will not be discussed further.

Common challenges in embedded system security implementations are that resources available are often very limited, and applications work in real-time conditions. Security features are often heavy, which causes more overhead and lot of resource usage, which might cause challenges.

Attacks related to software are also a challenge. For example, security concerns related to software updates are a real problem, as security issues can allow malicious code to run on an embedded system, or the system to revert to old, insecure versions [5]. Additionally, a secure boot configuration is a critical component of embedded system security,

as it prevents attackers from injecting malicious code into the software of an embedded system. It also establishes a root of trust, providing a foundation upon which other security features are built.

By focusing on security testing based on the previously mentioned aspects, such as limited resources, hardware–software integration, physical and debug interfaces, communication protocols, real-time performance, and secure update mechanisms, testers can help ensure that embedded devices are robust against a wide array of potential attacks.

2.3 Embedded system APIs

Embedded systems are utilizing external interfaces for critical functions such as data exchange and system management. These interfaces allow embedded devices to interact with external systems, enabling communication, control, management and data transmitting. Data exchange through external interfaces is essential for sensor data retrieval, control command transmission, and real-time system monitoring [6]. In addition, interfaces for system management tasks facilitate configuration updates, fault management, and remote diagnostics [6].

Embedded systems commonly utilize both external and internal application programming interfaces (APIs) to support communication and data exchange. External APIs are typically used to expose services or functionalities to other systems or clients. For example, through REST, WebSocket, or other network-based interfaces, which enable integration into broader systems and networks. These APIs facilitate interoperability, remote control, and monitoring in modern embedded applications.

On the other hand, internal APIs, such as hardware abstraction layers (HALs), device drivers, and communication stacks like TCP/IP, Bluetooth, and CAN, are used to abstract hardware details and provide standardized ways to interact with embedded components. Middleware frameworks also fall into this category by offering structured access to lower-level functionality. [7] These internal interfaces simplify software development by enabling developers to write reusable code that runs across different hardware platforms.

APIs for embedded devices are increasingly moving towards standardized communication protocols, leveraging existing web-based architectures to improve connectivity and interoperability. This trend aligns embedded systems more closely with mainstream computing, making them easier to integrate into modern IT infrastructures [8]. As a result, the protocols and technologies used in this work are not limited to specialized embedded solutions but are also well-known in traditional information technology contexts. One con-

tributing factor to this convergence is the increasing use of Linux-based operating systems in embedded systems, which enables the adoption of familiar networking stacks, development tools, and security mechanisms.

In modern embedded systems, advanced protocols such as WebSocket, HTTP, QUIC, and SSH are widely adopted due to their robustness, performance, and scalability. This thesis primarily focuses on these protocols, and the practical security testing efforts target implementations based on them. Since these protocols operate over IP networks, they are often exposed to external access, which increases their attack surface compared to lower-level protocols that typically require physical access to exploit. This exposure makes them a higher-priority target for security testing and justifies their selection as the core focus of this work.

The WebSocket protocol offers full-duplex communication channels over a single TCP connection. That significantly reduces the latency and network overhead compared to traditional request-response models like HTTP [8]. This makes it suitable for embedded systems requiring real-time data streams, such as industrial monitoring or real-time control applications. When WebSocket is used on top of an SSL/TLS connection, it is also encrypted and secure to use.

REST APIs (Representational State Transfer) are mainly popular for their usage in traditional web applications. They are extensively used due to their simplicity, scalability, and stateless interactions. REST APIs leverage standard HTTP methods, simplifying integration with web services and cloud infrastructures, thereby bridging embedded systems to Internet services [9]. REST-based architecture is commonly applied in IoT systems, smart homes, and industrial automation for exchanging data and control signals efficiently.

QUIC (Quick UDP Internet Connections) instead is a modern, UDP (User Datagram Protocol) based protocol developed to minimize latency and improve connection robustness, providing features such as multiplexed connections and built-in encryption by default . These characteristics make QUIC suitable for embedded systems in dynamic network environments, such as automotive telematics or IoT devices with intermittent connectivity. [10]

One protocol also included in this work is SSH (Secure Shell). It is widely implemented in embedded systems for secure management tasks. It enables secure remote logins, command execution, and file transfers, utilizing robust cryptographic algorithms to safeguard communications. [11] SSH ensures secure administrative access to embedded devices, protecting sensitive management and configuration data.

External interfaces are necessary for embedded systems as they serve a critical role in data exchange and system management. Traditional embedded APIs like HALs and middleware frameworks simplify device integration and enhance software portability. Furthermore, modern protocols such as WebSocket, REST APIs, QUIC, and SSH significantly improve embedded system performance, interoperability, and security, aligning embedded devices closer to modern IT infrastructure standards.

2.4 Authentication verification

Authentication is the process of verifying the identity of an entity, be it a user, device, or system, before granting access to sensitive resources. It is fundamental in preventing unauthorized access, safeguarding data integrity, and ensuring that communications occur only between trusted parties. In the context of embedded systems, robust authentication is crucial given the potentially wide exposure of devices in distributed networks and possible critical functions embedded systems perform in larger systems. Common authentication methods include passwords, API keys, and access tokens. However, this work focuses specifically on authentication through mTLS (mutual Transport Layer Security), which provides strong identity verification and encrypted communication between both parties. This is because mTLS is used widely in modern embedded systems for authentication and encryption in connections.

MTLS is a prominent mechanism to enforce strong authentication. Unlike standard TLS, which authenticates only the server, mTLS requires both the client and the server to present valid digital certificates during the handshake process. This bidirectional verification helps to mitigate risks such as man-in-the-middle attacks and impersonation by ensuring that both endpoints are verified using certificates issued by trusted authorities [12].

Embedded systems, such as IoT devices, industrial controllers, or automotive systems, often operate in environments where computational resources are limited. In these cases, mTLS is adapted through lightweight implementations that reduce overhead while still delivering robust security. Each embedded device is provided with a unique digital certificate, which plays a crucial role in establishing mutual trust during secure communications.

In an mTLS setup, the device certificate is used to authenticate the device to the server, proving its identity and ensuring that only legitimate devices can connect. Conversely, the server certificate allows the device to verify the authenticity of the service it is communicating with, protecting against man-in-the-middle attacks and rogue servers. This

dual validation process enforces mutual authentication, strengthening security by preventing unauthorized access from either end. Additionally, certificate-based authentication facilitates scalable and centralized management of credentials, making it well-suited for maintaining secure communications in large, distributed embedded networks. [12]

In summary, authentication is essential for protecting systems and data, and mTLS provides a powerful method to achieve mutual authentication in high-risk and resource-constrained environments like embedded systems. By leveraging digital certificates for both endpoints, mTLS enhances the overall security of communications, ensuring that only authorized entities participate in privileged network transactions.

2.5 Security testing methods

This sub-chapter presents the main methods used in security testing, with a focus on techniques relevant to embedded systems and their external APIs. It considers both automated and manual approaches, highlighting their strengths, limitations, and practical applications. The selected methods, such as fuzzing, penetration testing, and vulnerability scanning, form the basis for evaluating the security posture of embedded systems in the context of this thesis.

2.5.1 Fuzzing

Fuzzing, also known as fuzz testing, is an automated software testing technique that creates and injects many random inputs into a software system to find vulnerabilities or bugs. It has become a widely recognized method in both academia and industry as an effective way to find security-critical bugs in a variety of software systems, network protocols, and complex applications [13]. Fuzzing is widely used in software security analysis mainly because it is a highly efficient way to uncover unexpected vulnerabilities and bugs from large input states by automation. It works by repeatedly testing the target with incorrect or unexpected data and attempting to trigger errors, crashes, or bugs, which are monitored and analyzed to identify potential security vulnerabilities. For example, AFL (American Fuzzy Lop), a popular software component fuzzing tool, has been successful in finding hundreds of critical bugs in widely used programs [14]. In research, it has been significantly more efficient in finding vulnerabilities than more sophisticated tools such as symbolic execution engines [14].

Network protocol fuzzing differs fundamentally from general software fuzzing due to unique challenges posed by network protocols. Network protocol software also requires structured communication involving statefulness, strict adherence to protocol specifications, and interactions over network interfaces. These factors cause specific constraints

on the fuzzing approach. Unlike traditional software fuzzing, which often deals with file inputs or simple command-line parameters, network protocol fuzzing must handle sequences of structured messages over network links. That introduces complexities related to network delays and stateful transitions [13]. Furthermore, the highly structured nature of network inputs means that traditional fuzzing methods, which typically rely on random or semi-structured input mutations, frequently produce invalid packets rejected early in the processing stage. In contrast, software component fuzzing often deals directly with the software's logic without needing to define complex state transitions or network communications [14].

Network protocol fuzzing specifically addresses vulnerabilities in the implementation of network protocols, which are specifications governing device communication across networks. Protocol fuzzing must accurately model protocol syntax and state transitions to effectively discover deep, stateful vulnerabilities. New, more advanced tools have emerged, introducing techniques such as state-sensitive mutation and coverage-guided test generation to navigate stateful protocols more effectively [13]. Additionally, specialized fuzzers have demonstrated the usefulness of providing detailed protocol specifications for effective fuzzing. Since that allows the generation of inputs that respect the expected message formats and stateful interactions. Recent advancements in network protocol fuzzing include innovative techniques such as snapshot-based state restoration and virtual machine-level network simulations to significantly reduce execution overhead and improve fuzzing efficiency [13].

Software component fuzzing, conversely, is typically concerned with individual software units or programs without the added complexity of stateful network interactions. Component fuzzing generally employs simpler input models and uses code coverage feedback to optimize the generation of effective test cases. AFL, for instance, operates effectively as a greybox fuzzer by using execution feedback to identify inputs that exercise new code paths, thus enhancing bug detection efficiency [14]. However, the evaluation of software fuzzing tools is notably challenging due to inherent randomness, the importance of the chosen data seeds, and difficulties in reliably distinguishing unique bugs. Despite these challenges, software fuzzers like AFL and AFLFast have shown their effectiveness across various non-networked applications, highlighting the necessity for experimental evaluations to validate fuzzing strategies [14].

Fuzzing remains one of the most powerful techniques in modern software testing and is especially valuable in identifying and mitigating security vulnerabilities. Its application spans from complex network protocol fuzzing to more straightforward software component testing, with specialized approaches required by each case due to their specific

characteristics. In this work focus is mostly on network protocol fuzzing, since embedded system external APIs could be fuzzed exactly with network protocol fuzzing tools. Figure 1 presents the idea of embedded system external interface fuzzing with network protocol fuzzing tools.

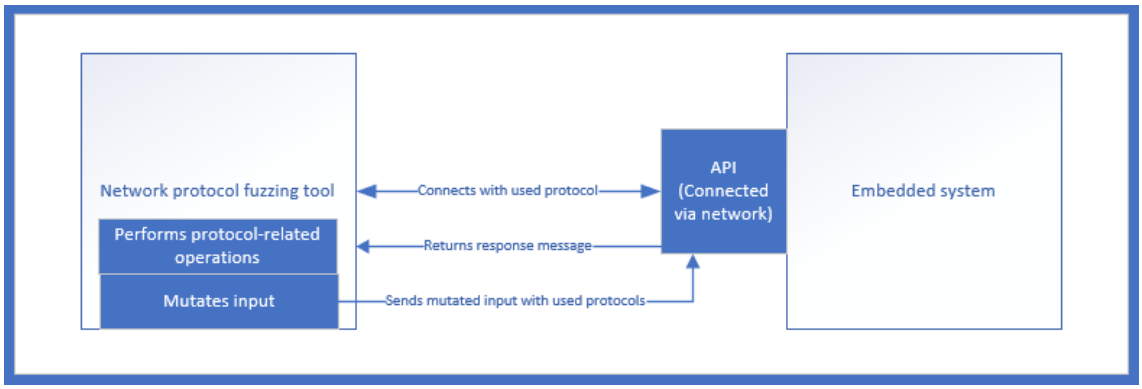

Figure 1. *Embedded system API fuzzing idea with network protocol fuzzer.*

This illustrates well how a network protocol fuzzer must also behave as a client that connects to the API, especially when the API functions as a server implemented using a specific technology. In the case of embedded systems, it is important to note that simply monitoring the connection to the interface is not sufficient for assessing the status of the target system during fuzzing. This is because a crash or malfunction in the embedded system may not result in a dropped connection or visible network-level error. The system might become unresponsive, behave incorrectly, or enter a partial failure without terminating the connection. Therefore, more advanced, system-specific monitoring methods must be developed to enable the fuzzer to accurately detect the internal state and stability of the target system.

2.5.2 Penetration testing

Penetration testing, also known as ethical hacking, involves simulating cyberattacks to identify vulnerabilities within a specific system, such as a software application, embedded platform, or connected device. This practice is crucial, as individual systems can serve as entry points for attackers, especially when integrated into larger environments.

System-level penetration testing functions as a targeted stress test designed to expose weaknesses in the system's design, implementation, or configuration before they can be exploited by malicious actors. The primary goal is to evaluate the robustness of the system's security mechanisms through hands-on, offensive testing techniques tailored to its architecture and intended use. [15]

Developers and integrators perform penetration testing to validate the security of their systems, ensure compliance with industry standards, and mitigate risks before deployment. This includes simulating attacks against APIs, firmware, communication protocols, and user input processing. Common threats assessed in system-level penetration testing include buffer overflows, insecure authentication, improper data handling, injection attacks, and denial-of-service (DoS) attacks [15].

Automation in penetration testing has also evolved significantly. For example, the European Space Agency (ESA) has developed a sophisticated automated penetration testing framework known as PenBox. PenBox automates traditional penetration testing phases such as reconnaissance, vulnerability assessment, and exploitation, minimizing human intervention and improving efficiency. [16] There is potential in AI-driven penetration testing to significantly reduce cost and effort. Machine learning methods enable automated penetration testing tools to learn optimal attack paths without predefined knowledge of the target network, surpassing the limitations of traditional penetration testing methodologies that heavily depend on pre-existing network configurations and detailed system information.

Regarding external interfaces, penetration testing has a critical role in safeguarding these potentially vulnerable points. External interfaces such as web applications, APIs, and networks are common targets of cybercriminals. Penetration testing specifically addresses these areas by making detailed assessments to uncover vulnerabilities associated with external exposure. External penetration tests usually simulate attacks originating from outside the organization's network, providing insights into vulnerabilities that could be exploited remotely. Thorough testing of these interfaces ensures that security controls effectively mitigate risks related to external threats.

Penetration testing of embedded systems presents unique challenges due to their constrained computing environments and different threat models compared to traditional IT systems. Limited computational power and memory can restrict the use of many conventional security testing tools, which are often designed for general-purpose operating systems with ample resources. However, modern embedded systems, for example, Linux-based ones, often provide enough functionality and performance to support a wide range of traditional security tools, making them more accessible to standard testing workflows. Still, many embedded devices lack standard interfaces for monitoring or debugging, making it more difficult to observe system behavior during testing. In contrast to IT systems, embedded devices may also be physically accessible to attackers, increasing the risk of hardware tampering or direct data extraction. While tamper protection is an important part of embedded security, it falls outside the scope of this work. This thesis focuses on

the security of external interfaces that are exposed over the network, rather than physical attack surfaces or the full internals of the embedded system.

2.5.3 Vulnerability scanning

Vulnerability scanning is a fundamental cybersecurity practice that uses automated tools to identify weaknesses in IT systems, networks, and applications. It involves both active scanning and passive scanning. In active scanning, a tool is directly interacting with a scanned system, and in passive scanning, a tool scans for vulnerabilities without directly interacting with the system, for example by scanning source codes or monitoring data traffic. This dual approach helps to create a comprehensive view of the security posture of the system under test.

Vulnerability scanning is part of a broader vulnerability assessment and penetration testing strategy. By regularly scanning and evaluating systems, organizations can prioritize remediation efforts based on risk severity and potential impact. This proactive approach not only minimizes the chances of successful cyberattacks but also assists in meeting regulatory and compliance standards.

Network scanners can also be categorized by purpose as follows. One type is the so-called port scanners, which can scan open network ports in the target system. Another type is application scanners, which can scan an application behind a network. A third type is vulnerability scanners, which scan for vulnerabilities that could be used by a hostile attacker to hack or otherwise compromise the entire system. [17]

Vulnerability scanning also has other uses, such as scanning source code for known vulnerabilities. Since this work focuses primarily on external interfaces, vulnerability scanning is also focused on targeting network-level vulnerabilities, external interfaces, and the applications found on them.

3. SECURITY TESTING DESIGN

This chapter presents the security testing developed in this work. It begins by describing the system under test, including the various external interfaces. A threat model is then defined to outline the key security assumptions and risks, with a particular focus on external interfaces and authentication.

Next, the chapter explores available tools and techniques for API fuzzing, vulnerability scanning, and penetration testing, followed by an overview of test automation and result analysis methods. This includes an investigation into continuous testing practices using CI/CD pipelines and tools like Robot Framework.

Finally, the chapter describes the design and architecture of the overall testing, along with the structure of the implemented test cases. The goal is to provide a systematic and repeatable approach for evaluating the security of external APIs in embedded systems with the help of security testing tools and scripts.

3.1 System under test description

The system under test used in this work is an embedded system built on a System-on-Chip (SoC) FPGA platform that integrates programmable logic, ARM-based processing cores, and embedded radio frequency (RF) analog interfaces, including high-speed Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs). This architecture enables efficient, high-performance processing for RF applications while optimizing size, weight, and power (SWaP) characteristics. [18]

The SoC platform operates on a customized Linux distribution generated with the Yocto Project, an open-source collaboration project that provides tools and methodologies to create customized Linux distributions optimized for embedded environments. Yocto allows extensive configuration of system components, providing a minimal, secure, and highly efficient operating system tailored for resource-constrained embedded systems. [19]

Utilizing Linux as the operating system significantly enhances the platform's capability to implement common communication protocols and API technologies used in information technology systems. Due to its foundation of Linux, the platform seamlessly supports well-established network communication methods, facilitating interoperability and integration into diverse technological ecosystems.

This work focuses on testing the security of the external APIs and communication paths, so it is also worth presenting the external APIs of the system under test at a high level. In the physical level for communication, all the APIs mentioned in this section are using the Ethernet protocol.

3.1.1 QUIC API

The system uses QUIC as the transport protocol for one of its main interfaces. mTLS is used for encryption and mutual authentication. QUIC's TLS handshake requires that the ALPN (Application Layer Protocol Negotiation) field match between the system and the client. [10] Figure 2 presents the protocol stack of QUIC API.

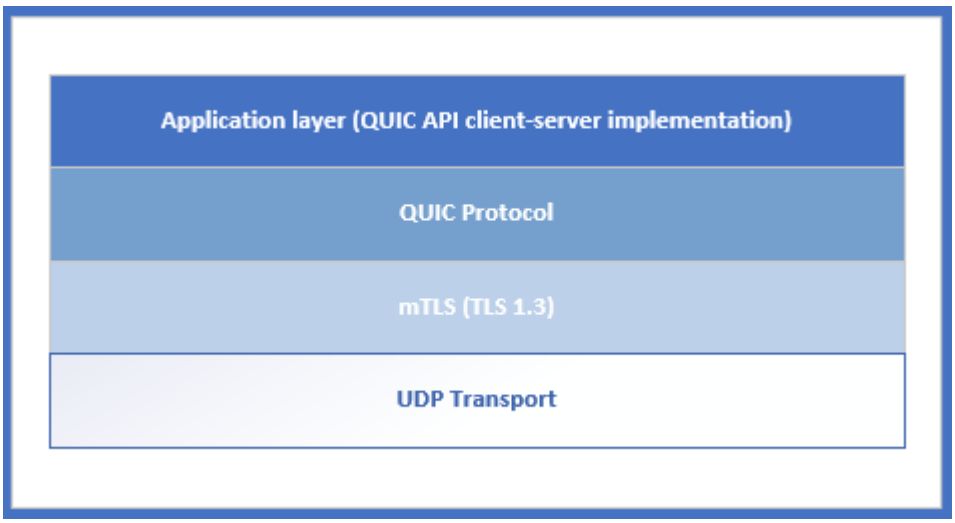

Figure 2. *QUIC API protocol stack presentation*

This protocol stack presentation clarifies the usage of QUIC on top of mTLS. It also clarifies that QUIC works on top of UDP transport.

On top of the transport level implementation, a custom application protocol is implemented. In this system, the client connects to the QUIC API and receives a unidirectional data stream from the system. Any messages sent from the client to the system are ignored entirely. This interface serves as the primary data output mechanism of the system.

3.1.2 WebSocket API

Another main external interface in the system uses WebSocket, which works on top of a TCP connection. Communication between the client and system uses JSON-formatted messages. For authentication and encryption, this API also uses mTLS. [8] Figure 3 presents the protocol stack of the WebSocket API.

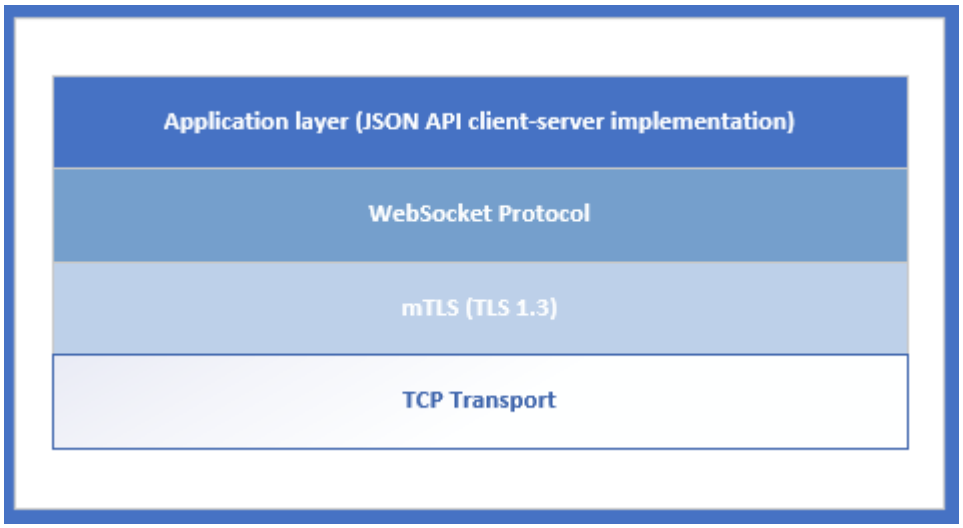

Figure 3. *WebSocket API protocol stack presentation*

This clarifies the protocols used in WebSocket API implementation in the transport layer and application layer. This presentation does not include network layer or physical layer implementations.

3.1.3 REST API

The third main interface in the system under test uses HTTP requests in communication between the client and the system. This interface uses TLS for encryption. This API also requires a JSON Web Token (JWT) to access certain REST API endpoints. Payloads of HTTP requests are structured in JSON format in this API. In addition, there is a user interface web application for this API, which is hosted on the system under test. [9] Figure 4 presents the protocol stack of the REST API.

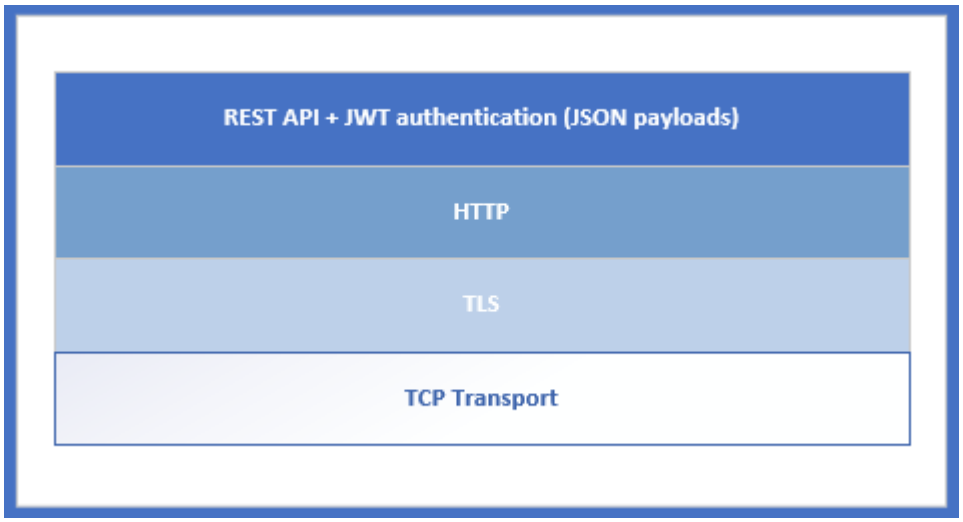

Figure 4. *REST API protocol stack presentation*

This clarifies the protocols used in REST API implementation for the transport layer and application layer. User interface, physical layer implementation, and network layer implementation are not included. Notable here compared to other APIs is that authentication is implemented in the application layer, and not with mTLS.

3.1.4 SSH

The system under test also supports the use of SSH. This is not intended to be used in the daily usage of the system, but its existence and security testing must also be included in this work, as it is one possible interface to connect to the device.

3.2 Threat model

In embedded systems, external APIs pose multiple threats, which target confidentiality, integrity, and availability of the system's information security. Security problems with system external APIs can be caused by, for example, man-in-the-middle attack variations or denial of service attacks.

One major concern is intellectual property (IPR) theft, where attackers may attempt to reverse-engineer the system or gain access to proprietary algorithms and implementations. Another risk involves data spoofing, where malicious actors feed false measurements or manipulated inputs through the API, potentially disrupting the system’s normal operation. Additionally, APIs may expose access points to sensitive information, such as sensor data, user recordings, or diagnostic logs, which can compromise user privacy or system confidentiality if not properly secured.

One of the most important things to remember is to always use the latest and secure versions of the technologies and protocols in use, as they are usually well protected against known vulnerabilities and attacks. The use of secure protocols can protect against most attacks, as an easy way for an attacker to break into a system is to look for outdated service versions and attack their known vulnerabilities.

The important thing at a general level is also to use a firewall or similar tool to restrict network traffic to only the desired ports and protocols in the system. This limits the possible entry routes an attacker can take into the system, which in turn means less chance for the attacker to negatively affect the system.

In the future, it will also be important to consider new threats such as those posed by quantum computing and AI-based attacks. Threats related to these are evolving all the time, which means security implementations require constant review for technological developments. However, some threats have already been identified as a reality in the

coming years. With the rapid progress in quantum computing, traditional public-key algorithms like RSA and ECC are becoming vulnerable, making the early adoption of quantum-safe cryptography crucial for protecting data against future quantum attacks [20]. For example, mTLS is usually based on public key cryptography and is therefore at risk of becoming vulnerable in the future, when quantum computing becomes more common. RSA and ECC algorithms with 112 bits of security strength are defined as deprecated as soon as in year 2030. With larger bits of security strength, RSA and ECC algorithms are disallowed after 2035. [21] In general, the rationality for using RSA is beginning to be questioned as the required key sizes increase, while ECC algorithms can provide the same security much more effectively with smaller key sizes.

It is also good to be aware of privacy-related threats. Although messages are encrypted by default, the metadata of messages can reveal things such as the frequency of message traffic, packet size, and timings. By combining these, it is possible to infer useful information about message traffic.

3.2.1 Authentication and mTLS

The mTLS encryption and authentication used for system-to-device connections also have their security risks. One possible threat is certificate misconfigurations. If either side has expired, incorrectly chained, or improperly issued certificates, an attacker might try to exploit these.

Possible weak certificate validation is also a common threat in mTLS. For example, if servers or clients do not enforce revocation checks or do not confirm that certificates come from trusted authorities, attackers might exploit those. In addition, an attacker may try to force the connection to use an older version of TLS, which is no longer secure today, and this must also be prevented.

In a dynamic environment, special attention must also be paid to the management and renewal of certificates so that their inadequate management does not pose a security risk. [22] However, this work's scope does not include the management of certificates.

In addition, the mTLS handshake consumes system resources such as CPU and memory. If an attacker floods the service with many mTLS handshake requests, it could lead to a denial-of-service (DoS) condition by exhausting those resources. Similarly, a flood of packets at the network level could also cause DoS conditions, although such low-level network attacks are outside the scope of this work and were not included in the testing.

3.2.2 QUIC API

QUIC is seen as a security-oriented protocol, but there are still different ways to attack QUIC implementations. Since the server in a system under test does not handle received messages, attacks based on messages sent from the client to the server, such as various fuzzing or suspicious data transmission, can be ignored at this point. Of course, one of the things included in the functional testing of the system is to test that the API does not handle the messages received as it is defined, but that is not really in the scope of this work.

One key area is the connection resumption functionality of QUIC. This feature allows a client to resume a previously established connection without performing a full handshake, thereby reducing latency and improving performance. However, it can also enable so-called replay attacks, where an attacker hijacks network traffic, captures transmitted packets, and then resends those packets in an attempt to disrupt or manipulate the connection [23].

Another way to attack the API is to try to DOS the server by trying to make many connections in a small amount of time, thus limiting the ability of the actual client to connect to the device. In this context, an attacker can also try to overload the server with a so-called Slowloris attack, which uses incomplete handshakes or connection setup messages to keep the connection open for too long while the server waits for the handshake to complete [23].

There are also two common manipulation attacks used against QUIC connections. The first one targets the Source-Address Token (STK), which is commonly used to validate the client address in QUIC connections. Attackers try to manipulate or replay this token during the handshake, leading to possible unauthorized connections or facilitating further attacks like session hijacking. The second one targets connection identifiers that QUIC connection uses to maintain connection state, especially during IP changes. [23] An attacker altering the connection identifiers can disrupt connection stability or even hijack an existing connection by misdirecting packets. This is why secure binding of these tokens and identifiers to cryptographic keys is essential to prevent such manipulation.

Also, request forgery attacks are possible against QUIC. An attacker can give unrecognized or manipulated version numbers, causing the server to engage in unnecessary version negotiation processes. Attackers can forge the initial connection request to make the server believe it is initiating a connection with a legitimate client. This may cause the server to respond with sensitive configuration data or allocate resources improperly. Also, by spoofing packets with altered source addresses during a connection migration,

attackers can force the server to redirect connections to an unintended destination, effectively hijacking or disrupting ongoing sessions. [23]

3.2.3 WebSocket API

WebSocket implemented on top of mTLS connection is also possibly vulnerable against multiple attack types. This only focuses on WebSocket-related threats, and mTLS security threats are handled in their chapter.

Threats can be divided into two scenarios, one in which a completely unauthenticated party attempts to disrupt or hijack the device or its use. In the second scenario, an attacker has obtained the mTLS certificates required for authentication in one way or another and can attack the device using them.

One significant threat from an unauthenticated attacker is the disruption of the interface by denial-of-service attacks. Here, an attacker may attempt to establish large numbers of connections or send large numbers of messages to disrupt or prevent access by real users of the interface. Another potential problem is replay attacks, which are a situation where an attacker would intercept real network traffic and try to use these real messages to, for example, reopen a connection or resend real messages. [24] In the transport layer level, mTLS has good protection mechanisms against replay attacks, but replay attacks are still a potential threat.

If the attackers could authenticate themself, they also have different ways to disrupt the device usage. This does not address what functionalities an authenticated user has access to, but focuses primarily on what vulnerabilities an authenticated attacker could theoretically exploit to disrupt the normal operation of the device or to compromise security. One possibility is JSON injections, which an attacker could try to include in the JSON payload of a message they send to the server. Another possibility is that an attacker may try to include unexpected data in JSON messages, such as unexpected data types or a large amount of data [24]. This can exploit vulnerabilities such as buffer overflow in unexpected places in the system.

3.2.4 REST API

REST API combined with a web application user interface contains a lot of possible threats, which are similar when hosted from an embedded system than in traditional web applications. In this API, the threat model for encryption-related threats is handled in a

different chapter. The focus in this chapter is on common threats targeted at the REST API backend and its web application user interface.

In this case, a denial-of-service attack is also a threat. They are used by the attacker to disrupt the normal user's activity and connectivity to the API. [25] In the case of the system under test, this is not a particularly critical threat because this API is not used continuously and does not need to be accessible all the time. Even though it is a big problem if the attacker can prevent normal usage of this API, that is why it is handled in this threat model.

Potential threats related to the payload of sent messages include various injections and unexpected message contents. One threat is JSON-injection, where the data structure of the JSON payload includes malicious data. Another threat is unexpected data, such as extremely large messages, which possibly result in vulnerabilities like buffer overflow. Since the system under test also hosts a user interface for the REST API, it opens the possibility of Cross-Site Scripting (XSS) attacks. At a general level, SQL injection is also a common vulnerability in web applications like this. [25] [26] In case of malicious message content attacker also needs to somehow actually get messages through to the server, since use of JWT blocks non-authenticated messages.

Another threat is the situation where an attacker can capture network traffic. The use of TLS ensures that the data is encrypted and cannot be directly read by the attacker, but there are other threats as well. One notable threat is replay attacks, where an attacker tries to resend recorded packets to the API to hijack or disrupt the connection [25]. An attacker may also try to manipulate packet traffic or messages, or, for example, use packet metadata to try to gather information.

Since the system under test uses JWT for authenticating REST endpoints, threats related to JWT should also be considered. Three common threats related to generating and receiving JWT are a weak secret key, which allows for brute-force attacks, insecure communication channels, and possible sensitive information included in JWT [27]. In a system under test, mTLS provides a secure data transfer that can mitigate these threats. Another JWT-related threat is related to the storage of the JWT token. For example, storing JWT in browser local storage risks it being accessed by malicious scripts in case of an XSS attack [27]. An attacker might also try to manipulate JWT token data. For example, an attacker could try to include malicious data or manipulate the alg field in the header [27]. The alg-field specifies the algorithm used to sign the token, and if improperly validated, an attacker could exploit it to bypass verification mechanisms.

3.2.5 SSH

SSH is a secure way for remote access and data transmission. Despite its robust encryption and authentication mechanisms, SSH is susceptible to various security threats, which should be considered.

One of the most common threats to SSH is related to poor authentication implementation. Attackers use brute-force methods to break authentication, for example, by using an insufficiently weak password or even a default password for remote SSH access. Once inside, they may escalate privileges or move laterally within the network, posing substantial risks to system integrity.

Another vulnerability lies in the potential for information leakage through traffic analysis. Research has demonstrated that SSH's handling of interactive sessions can inadvertently expose sensitive information. For instance, the timing of keystrokes can be analyzed to infer password lengths or even specific inputs, undermining the confidentiality that SSH aims to provide. [28]

In conclusion, ensuring robust authentication is fundamental to SSH security because it verifies the identities of clients, thereby preventing impersonation and man-in-the-middle attacks. Keeping the SSH service updated is equally critical, as timely patching of vulnerabilities helps mitigate newly discovered exploits and maintains the integrity of the encryption protocols.

3.3 Security testing tool investigation

This section provides a high-level description of the tools and methods chosen to test the different techniques.

3.3.1 API fuzzing

When selecting API fuzzing tools and methods for an embedded system, several important factors must be considered. Embedded systems often expose multiple interfaces that utilize different communication technologies, which may vary from or be customized versions of those found in conventional software environments. Therefore, chosen tools should either provide broad support for a wide range of protocols and technologies or offer flexibility to easily extend the tool with custom protocol definitions as needed.

Another important factor is the ability to automate fuzz testing to make testing easier and get rid of manually clicking the user interface every time when performing fuzz tests. Important things to consider here are the ability of the tool to work as part of CI/CD pipelines and various test automation solutions, such as the Robot Framework.

One particularly important factor in the case of fuzzing an embedded system is the ability to monitor changes at different levels of the embedded system and services because of fuzz testing. Modern embedded systems include different program-level services, but failures at the lower level might also occur, for example, in hardware. Those also need to be monitored to receive comprehensive results from fuzz testing.

This work has also excluded from further investigation commercial services offering API fuzz features, such as Burp Suite, and security features of Gitlab's Ultimate subscription. This was decided because there were also very comprehensive and functional open-source tools for this very need. It is worth mentioning and keeping in mind that commercial products for this purpose are also certainly functional and could be generally easier and more comprehensive to use, thanks to the wider range of features they offer.

3.3.1.1 ZAP

The first potential API fuzzing tool investigated in this work is OWASP Zed Attack Proxy (ZAP). ZAP has versatile fuzzing capabilities and potentially useful fuzzing tools to fuzz embedded system APIs. ZAP provides user-friendly UI, ready-made fuzzing payload sets, possibility for add-on fuzzing payloads, and option to run own custom fuzz scripts. ZAP also supports different protocols widely, for example, WebSocket connections are supported in addition to the common web application HTTP messaging. ZAP also has great possibilities for custom authentication methods when connecting to authenticated APIs.

A major shortcoming of ZAP is the lack of fuzz support in the ZAP API. This significantly limits the possibilities for test automation and integration of testing with other testing. Overall, ZAP is a great alternative for API fuzzing in embedded systems, if the manual fuzzing with the ZAP UI is sufficient for the use case. It is also worth noting that ZAP's original purpose is security testing of web applications, which is why the range of supported protocols is mostly made to serve the original purpose. However, in this work, the aim is to create solutions with versatile automation possibilities. ZAP was then left out of the practical fuzz implementation because of a lack of fuzzing support in the ZAP API.

3.3.1.2 Boofuzz

Another option for fuzzing APIs is a Python library named Boofuzz. It is an open-source source actively maintained library that supports widely fuzzing embedded device APIs over network. Boofuzz provides multiple ready-made connection options, with support for authentication methods, for example TLS. Boofuzz provides a fuzzing generator that automatically fuzzes defined parts of the message with malicious or unexpected input. It also has versatile logging features and the ability to monitor the status of the object being

fuzzed. As it is an open-source Python library, it is also very customizable for use cases. At the same time, it also requires programming expertise. But since the goal is to automate fuzz testing and possibly integrate it into other testing, using Boofuzz provides very nice and versatile possibilities.

3.3.1.3 Tool comparison

ZAP and Boofuzz both provide great fuzzing capabilities over the network, but their strengths differ slightly. Table 1 presents a high-level comparison between using a pre-built application (ZAP) and a Python-based fuzzing library (Boofuzz), highlighting scenarios where each approach may be most suitable.

Table 1. *API fuzz tool comparison*

Fuzz tool	Strengths	Weaknesses	Use case
Boofuzz	Customizable. Good monitoring and logging options.	Requires programming expertise. A lot of work in the setup.	Customizable fuzz testing. Test automation options.
ZAP	Easy to use UI. Setting up is simple.	Limited possibilities in automation.	Manual fuzzing with UI.

If the use case is more about performing simple fuzz testing manually with simple setup and easy-to-use features, using ZAP or some other application that supports fuzzing naturally would be a good idea. Instead, if the goal is to automate test cases or integrate fuzzing into other testing made for the target system, it could be a good idea to invest time in implementing fuzz testing with Boofuzz.

3.3.2 Vulnerability scanning and penetration testing

This sub-chapter examines and presents various security testing tools related to vulnerability scanning and penetration testing. Open-source tools are preferred in this work, but there are also some free versions of commercial tools tested.

3.3.2.1 Nmap

Network Mapper (Nmap) is an open-source tool designed for network exploration and security auditing. Nmap is widely used to discover hosts and services on a computer network by sending packets and analyzing the responses. Nmap supports features such as host discovery to identify active devices on a network and port scanning to scan for open ports on a target host. Nmap also supports operating system detection and service detection. [29]

Originally developed for Linux, Nmap has been ported to multiple operating systems, including Windows and macOS. That makes it a versatile tool for network administrators and security professionals. Its effectiveness and reliability have also been demonstrated in comparative analyses with other port scanning tools [30].

Nmap supports various scanning techniques, such as TCP connect scans, SYN scans, and UDP scans, enabling flexible and efficient network auditing. It also offers options for stealth scanning to evade detection by intrusion detection systems. Nmap can be used as an application, or it could be used with programming libraries such as Python nmap3. Interest in this work is in the possibility of creating automated port scans with the help of the Nmap Python library.

3.3.2.2 Nessus

Nessus is a valuable tool for helping organizations understand and improve their security posture. For example, a company's IT team can use Nessus to perform regular scans across their network, quickly spotting vulnerabilities like outdated software, default passwords, or configuration mistakes. This means they can proactively address security issues before attackers have a chance to exploit them. [31]

Nessus has three different subscription levels, and to unlock all the features user needs to buy more expensive licenses. This work focuses on the free version, Nessus Essentials. Unlike the professional edition, Nessus Essentials allows users to scan up to 16 IP addresses, making it ideal for individuals, students, or small teams who need reliable security assessments without the cost of enterprise-level software. In this use case, this is enough since the system under test is just a device with one IP address.

With Nessus Essentials, users have access to many core capabilities, including scanning for common vulnerabilities, misconfigurations, and outdated software. It features preconfigured scan templates, so even users without extensive cybersecurity experience can quickly identify security weaknesses. Overall, Nessus Essentials is an accessible entry point into vulnerability scanning, providing valuable insights into improving security without any financial investment. [31]

In this work, the benefits of Nessus could include its port scanning capabilities and the ability to detect outdated components behind services such as SSH. In principle, Nessus is designed to scan larger networks, so its designed purpose is slightly different from this work, but it can still provide some useful features for this work as well.

3.3.2.3 Metasploit

Metasploit is an open-source penetration testing framework widely used by cybersecurity professionals to identify, exploit, and validate vulnerabilities in computer systems. It is a

comprehensive suite of tools developed by Rapid7 to facilitate tasks such as vulnerability testing, network inventory, attack execution, and detection mechanism circumvention. The framework's architecture includes various components such as exploits, payloads, auxiliary modules, and encoders that allow users to tailor their testing strategy to specific environments and objectives. [32]

Beyond penetration testing, Metasploit serves multiple purposes in the cybersecurity domain. It is employed for vulnerability research, allowing analysts to study and validate potential security flaws, thereby contributing to the development of more secure systems. Additionally, Metasploit is used in security awareness training, providing a practical platform for simulating cyberattacks and educating personnel on threat mitigation strategies. The framework also aids in incident response by enabling security teams to replicate attack scenarios, assess the effectiveness of existing defenses, and enhance their preparedness against real-world threats. [33]

In this work Metasploit framework could be most useful in identifying and exploiting known vulnerabilities of common technologies supported by Metasploit. For example, Metasploit has modules to test SSH and HTTP REST interfaces used in the system under test.

3.3.2.4 Wireshark

Wireshark is a free and open-source network protocol analyzer that enables users to capture and examine data packets traversing a network in real-time. It supports live capture and offline analysis, allowing for detailed inspection of hundreds of protocols across various network types, including Ethernet and IEEE 802.11. Wireshark is cross-platform, running on Windows, Linux, macOS, and other operating systems. [34]

The software features a standard three-pane packet browser, providing a comprehensive view of captured data. Users can apply display filters to focus on specific traffic patterns and utilize color coding to quickly identify different packet types. Wireshark also offers rich VoIP analysis capabilities and supports import and export of packet data in various formats. [34]

In this work, Wireshark is most useful in capturing packet data sent to the system under test and packets received from the system under test. These can be used in penetration testing to perform different kinds of attacks, to verify connection encryptions, and to examine the metadata contained in connection traffic.

3.3.2.5 Scapy

Scapy is a powerful and flexible Python-based interactive packet manipulation tool used primarily for network testing, scanning, and penetration testing. It allows users to craft,

send, receive, and analyze network packets, supporting a wide variety of protocols such as TCP, UDP, ICMP, ARP, and even custom-defined protocols. Scapy is used for tasks like network scanning, protocol analysis, packet crafting, and automated network discovery. Scapy's Python interface makes it easily scriptable, enabling cybersecurity professionals and network administrators to create customized network testing scripts tailored specifically to their needs. [35]

In addition to its capability for packet generation and analysis, Scapy provides a versatile platform for various tasks like tracerouting, network discovery, fuzz testing, and firewall rule validation. Users can leverage Scapy's interactive shell or integrate it directly into Python scripts, offering flexibility across different testing scenarios. Its ability to forge or decode a wide range of packets and protocols, combined with powerful visualization and reporting features, makes Scapy a valuable resource in both security research and practical network administration contexts. [35]

In this work, Scapy is mostly used in packet crafting and sending. Packet crafting tools could be used, for example, to resend captured packets with the purpose of testing system durability against replay attacks. In the Program 1 is presented example scripts for Scapy to replay captured packets.

```
from scapy.all import rdpcap, sendp
import time

def replay_pcap(pcap_file, iface="eth0"):
    packets = rdpcap(pcap_file)
    print(f"Replaying {len(packets)} packets...")
    sendp(packets, iface=iface, verbose=True)

def replay_flood(pcap_file, duration, iface="eth0", sleep=0):
    start_time = time.time()
    while time.time() - start_time < duration:
        replay_pcap(pcap_file, iface)
        if sleep != 0:
            time.sleep(sleep)
```

Program 1. *Packet replay scripts with Scapy.*

These scripts in Program 1 are using the Scapy tool with Python. For the replay_pcap-function user gives a path to a pcap-formatted file as a parameter. For example, the Wireshark tool could be used to capture and save packets in PCAP format. Then the function uses Scapy to replay the captured packet. The replay_flood-function shows a test to see how a server reacts if replayed packets are flooding into the server. This is tested to see if it is possible to create a denial-of-service attack by replaying packets in high volume.

3.3.2.6 OpenSSL

OpenSSL is an open-source toolkit that provides robust implementations of the SSL and TLS protocols. It's widely used for a range of cryptographic functions such as encryption, decryption, signing, and certificate management. [36]

A possible use of OpenSSL in this work is to generate certificates for authentication testing. This can be used to easily generate, for example, outdated or forged certificates to assist in mTLS security testing of the system.

3.3.2.7 ZAP

ZAP, also presented as an option in fuzzing, includes the possibility to scan web applications. This is useful in this work as part of the vulnerability scanning of the HTTP/REST API. Also, as mentioned earlier, ZAP has a Python API that can be used to automate testing.

ZAP is also useful as part of penetration testing, as it allows you to run various known attacks on web applications and test their resistance against them. It can also be used to perform so-called man-in-the-middle attacks on web applications, where traffic between the web application and the client is captured and various attacks are performed. For example, by resending the packets and performing so-called replay attacks. ZAP also has WebSocket support, which is useful in the security testing of the WebSocket API in the system under test.

3.3.2.8 SSLScan

SSLScan is a lightweight, command-line tool designed to query SSL/TLS-enabled services to determine their supported protocols, cipher suites, and certificate details. It systematically connects to a target service, such as a web server or mail server, on a specified port and performs a series of SSL/TLS handshakes using different cipher suites and protocol versions. By analyzing the responses, SSLScan generates a detailed report on which cipher suites and SSL/TLS protocols are accepted, helping to identify potential security weaknesses like the use of deprecated protocols or weak ciphers. This tool is designed to be easy to use and fast, making it an ideal choice for quickly assessing the cryptographic strength of a server's configuration. In this work, SSLScan is used for a quick and simple scan against mTLS authentication used to authenticate and encrypt APIs.

3.3.2.9 SSLyze

SSLyze is a fast, powerful, and comprehensive SSL/TLS scanning tool and Python library. It connects to target servers to assess their SSL/TLS configurations, checking for

supported protocols, cipher suites, and potential vulnerabilities such as Heartbleed, ROBOT, and OpenSSL CCS injection. It is designed with performance in mind, and it leverages concurrent scanning techniques, for example multi-threading and multi-processing. It also provides detailed, structured output that can be exported in JSON or XML formats. It contains a fully documented Python API with rich type annotations, which facilitates integration into automated security workflows, continuous monitoring systems, or CI/CD pipelines. That makes SSLyze a great option for in-depth mTLS authentication scanning and testing. [37]

In this work, the SSLyze API is used to implement test cases that can be run as part of automated testing. This allows testing for the absence of known mTLS-related vulnerabilities in the system.

3.4 Test automation and result analysis

One goal in this work is to implement security testing solutions that are easy to use and repeatable. Therefore, for each of the security testing areas discussed in the work, the aim is also to investigate the possibilities for automating the testing area. In security testing, automated test methods also need to be complemented by, for example, manual penetration testing to achieve comprehensive testing coverage. Therefore, for each area, it is necessary to think carefully about which parts could be automated and which require manual effort.

This work focuses on the investigation of the possibilities of automating security testing using two slightly different approaches. One focus is the integration of testing into CI/CD pipelines, and another is the creation of automated test suites using the Robot Framework. One thing to note is that these approaches are not mutually exclusive, since Robot Framework can also be integrated into CI/CD workflows to enable automated and continuous testing. Testing that could be integrated into CI/CD pipelines could be simple scans for different vulnerabilities, for example, by assessing the security of the source code. However, the main objective would be to keep these tests simple and short to run. In contrast, test cases implemented with the Robot Framework can create more comprehensive and resource-intensive testing. This could include, for example, different types of regression testing and more comprehensive vulnerability scanning.

One thing to consider with embedded system testing is also what testing can be run without having dedicated hardware and what requires it. In this work, the breakdown is decided to be done in such a way that it would be possible to run testing in the CI/CD pipeline that does not require running tests on the hardware itself. This could be, for

example, testing the code components behind APIs. Whereas test suites created with the Robot Framework might make more sense to run against the embedded device itself, these could be external vulnerability scanning or external API fuzzing.

3.4.1 Continuous integration and Continuous development

Continuous Integration and Continuous Deployment (CI/CD) have transformed software development by automating the building, testing, and deployment of applications. Automation facilitates the rapid and reliable delivery of software updates and ensures that code changes are consistently integrated and deployed with minimal manual intervention. CI/CD pipelines improve code quality and accelerate the development lifecycle by incorporating automated testing at various stages. [38]

Integrating security testing into CI/CD pipelines is crucial to identify and mitigate vulnerabilities early in the development process. By incorporating security testing, such as static application security testing (SAST), dynamic application security testing (DAST), and software composition analysis (SCA) into the pipeline, organizations can proactively address potential threats before they enter the production environment. [39] In the context of embedded development, where the traditional division between development and operations is often absent, most security responsibilities typically fall on the development team. This makes early and automated security testing especially important and aligns with DevSecOps principles, fostering a development culture where security is embedded throughout the process.

The use of CI/CD pipelines for fuzzing has also been investigated, especially from the point of view of finding security-related bugs. It has been studied that 77% of code bugs are caused by code changes that have just been made, so security testing could also be useful already in the CI/CD pipeline [40]. Things to consider when using fuzzing are when the target has changed enough to warrant fuzz testing, and finding a balance in how much resource usage for fuzz testing is useful and reasonable. Research has found that short fuzzing sessions of a few minutes can detect the most critical bugs. Running fuzzing sessions that are too long continuously, on the other hand, does not provide the same benefit in terms of resources used [40]. In the case of embedded systems, it must also be considered that the CI/CD pipeline may require hardware for testing. Pipelines can also be executed for the software components within the Git repository, as these do not require dedicated hardware.

3.4.2 Robot Framework

The Robot Framework is an open-source automation tool designed for acceptance testing and acceptance testing driven development. It uses a keyword-based testing approach that allows users to create test cases in a tabular format that is both readable and maintainable. This structure facilitates collaboration between team members with different technical expertise, as the syntax is accessible to both developers and non-developers. The extensibility of the framework is one of its special features. It supports the integration of external Python or Java libraries, allowing users to tailor its features to specific testing requirements. [41]

In practical applications, the Robot Framework has been effectively used to automate functional regression test cases, which has significantly improved software quality and reduced testing time. Its ability to define reusable high-level keywords from existing keywords ensures easy extensibility and reusability, making it a cost-effective solution for complex testing scenarios. In addition, the Robot Framework's compatibility with different operating systems, such as Windows, macOS, and Linux, increases its adaptability to different development environments. [42] The combination of these features makes the Robot Framework a valuable tool for test automation, providing a structured yet flexible approach to test management and execution.

In this work, tests are implemented with the Robot Framework in three parts. An illustration of this is shown in Figure 5.

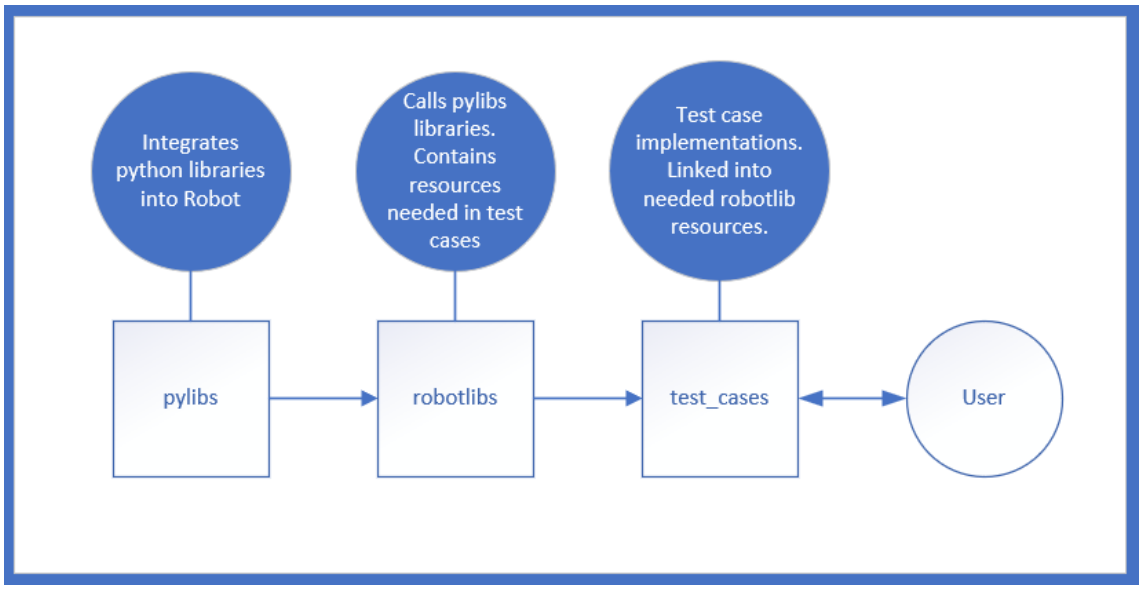

Figure 5. *Robot Framework test automation design of this work.*

If the tests require it or are easier to implement that way, a Python library callable directly from Robot Framework is created. The resources contained in the tests, the initialization of the tests, and other actions required for testing are implemented in the Robot Framework's resource files corresponding to each test case file. The test cases are implemented in robot files, and these are the only files the user must interact with when running the tests.

Another reason for using a Robot Framework in test automation is the result services it provides. After running the tests, the test results and detailed test logs are automatically generated in HTML format under the working directory. This allows the tester to quickly check the test results and to perform a more detailed analysis if the tester wants to. Figure 6 presents an example of the generated report.

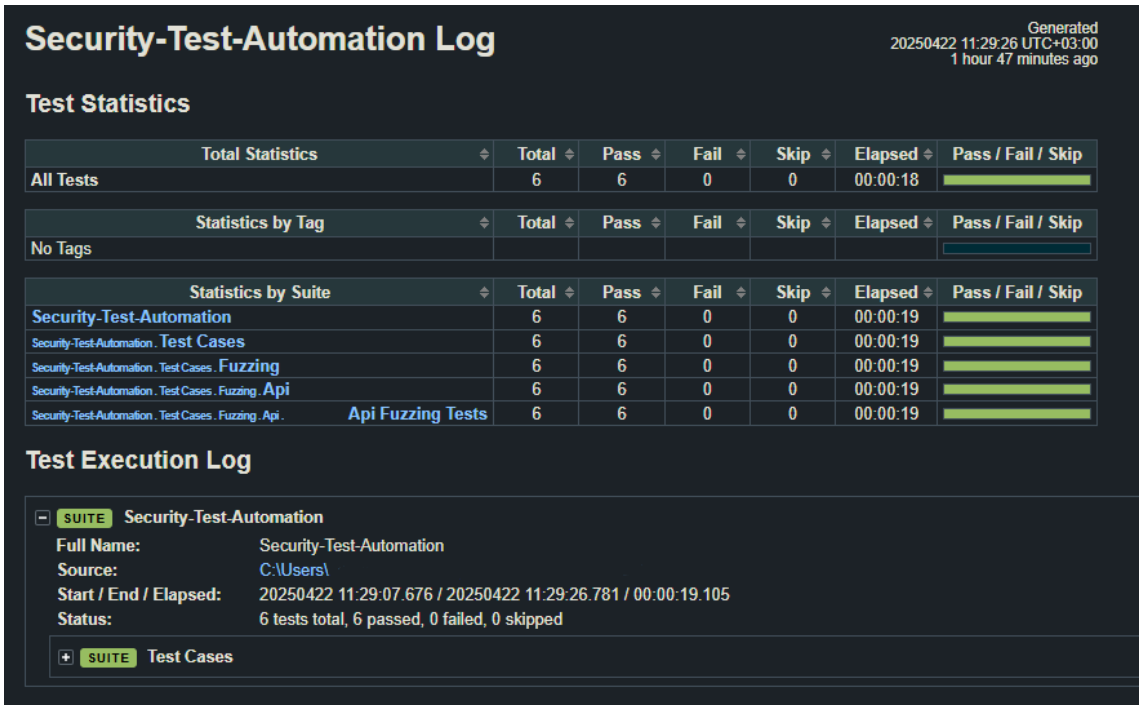

Figure 6. *Robot Framework generated result report example.*

In this example, the result report includes only six test cases executed within a single sub-area. The report organizes test results according to their respective sub-areas and provides detailed logs for each executed test.

3.5 Test design

Under this section is a detailed description of testing broken down into all the tested areas. This contains only the common design of testing for each area. Actual test cases are described in the chapter 4.

3.5.1 Test design and architecture

This sub-chapter gives a general overview of security testing implemented in this work. Figure 7 presents a general structure for the testing approach, including tested areas.

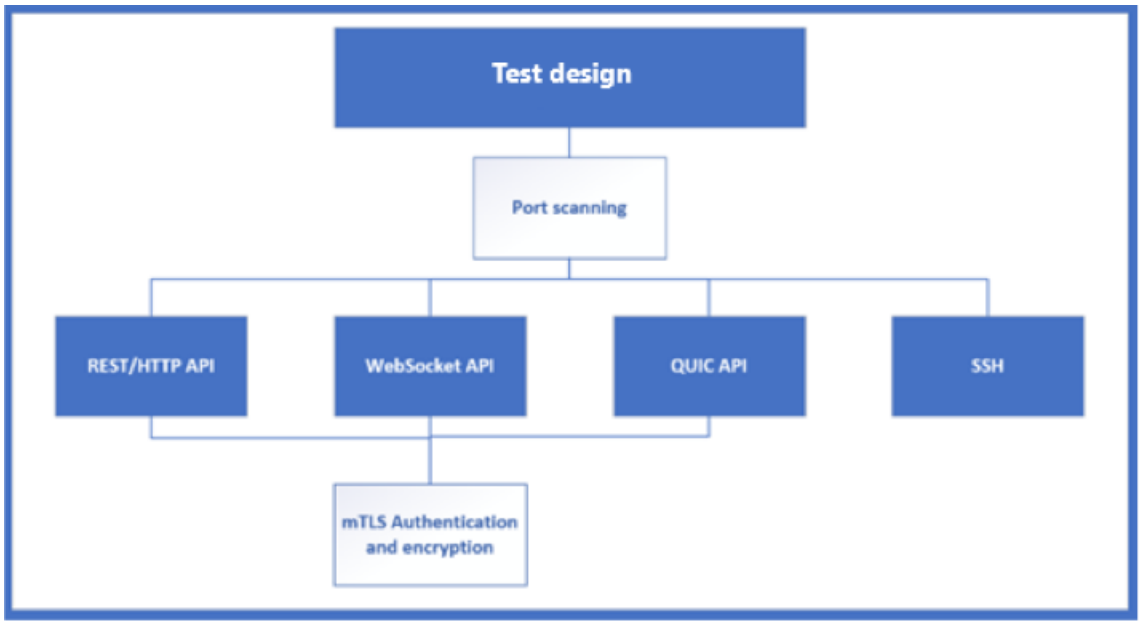

Figure 7. *Testing the general design*

Testing starts by running a port scan to make sure that only necessary ports are open. Each of the APIs uses a port for its service, and that is why in the figure, port scanning is connected to all APIs. Authentication and encryption with mTLS are tested in their section, since many APIs are using mTLS. SSH is an exception since it does not use mTLS authentication and it is not an API, but it is still one of the external interfaces of the system and needs to be included in testing.

3.5.2 API fuzzing

Based on the fuzz tool investigation, fuzz test design and implementation are based on the Boofuzz library. The goal is to create a fuzz-testing procedure suitable for the system under test, and to create a basis for test automation and result analysis.

Fuzz testing of APIs can be resource-consuming and primarily tests the robustness of the whole system under test against unexpected or malicious data input to external interfaces. Integrating API fuzzing into the CI/CD pipeline could be an option, but it would require having hardware permanently reserved for running the pipelines. In a situation where there is enough hardware available to dedicate to the CI/CD pipeline usage, running fuzz testing in the CI/CD pipeline would be a good option. However, in this work decision for API fuzz testing is to implement test automation with Python and Robot Framework.

In the system under test, there is an API using WebSocket, and there is also an HTTP REST API, which are the main APIs that are fuzz-tested at this point. That is why this work focuses on implementing fuzz testing for APIs with these two technologies. The

goal is also to implement a solution where other possible API technologies could easily be added if needed.

High-level design of the fuzzing test solution is shown in the Figure 8.

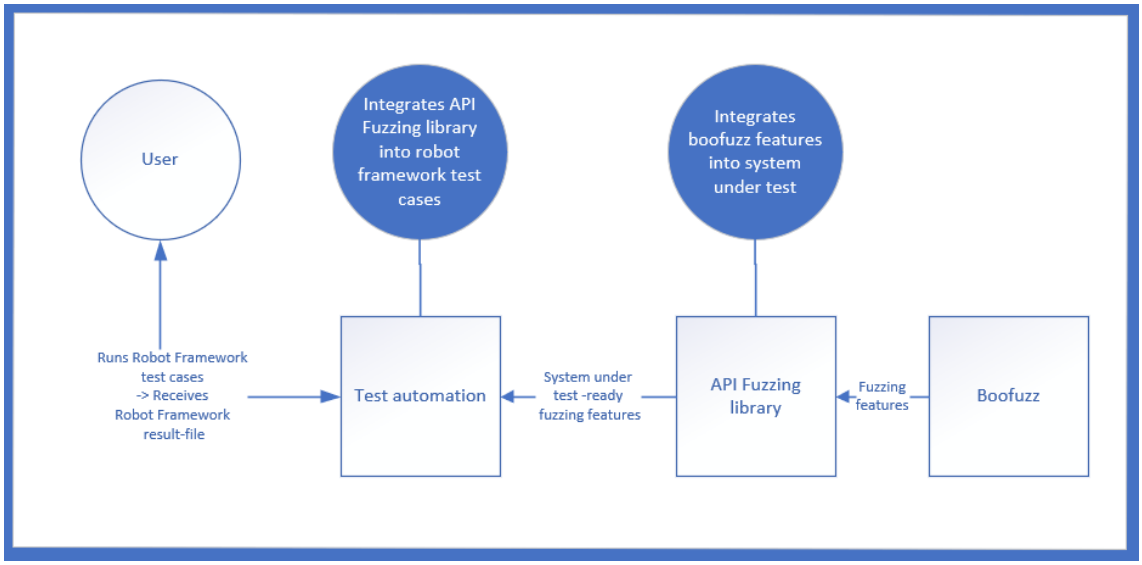

Figure 8. *API fuzzing design*

The goal is that the user can run the desired Robot Framework test cases and receive a test result report. All other technical details of fuzzing are handled in the background, and users do not have to worry about them. The advantages of this architectural solution are test case repeatability and usability without certain fuzzing or information security expertise.

The API fuzzing library is designed to be implemented in Python based on the Boofuzz fuzzing library. The main goal for this library is that it is compatible with running fuzz testing directly into the system under test, and could also monitor the current system status while fuzz tests are running.

The test automation component is designed to implement Robot Framework test automation test cases and support libraries to run fuzzing functionalities of the API fuzzing library. This is also the component user interacts with to run test cases. Test results are generated into test result reports by Robot Framework.

Another effective approach for fuzzing could be to fuzz all the software components that handle data transmitted to external interfaces. In this case, running fuzzing would be simple at the component level, but fuzzing needs to be performed on multiple components. In this approach, running fuzzing in CI/CD pipelines could be a great idea for automated testing. In this work, fuzzing is run only through external interfaces because

of the topic and a more system-level approach, but running fuzzing at the software component level could be a great way to complement security testing.

3.5.3 Vulnerability scanning and penetration testing

This sub-chapter outlines the plan for vulnerability scanning and penetration testing of the various APIs in the system under test. Different interfaces are divided into different chapters. In addition, authentication and port scanning tests are in their own chapters.

3.5.3.1 Port scanning

The plan is to create an easy way to test that a system under test has the desired ports open for the desired protocol. Here, test automation is implemented using the Robot Framework, which is widely used in this work. The NMap tool is used via the Python nmap3 library to make it easy to work together with the test automation. In this case, it is chosen to provide the allowed open ports in JSON format. The design is also intended to make it easy to add a feature to read the allowed ports in other ways. In the Figure 9 is port scanning test automation design is drawn as a simple chart.

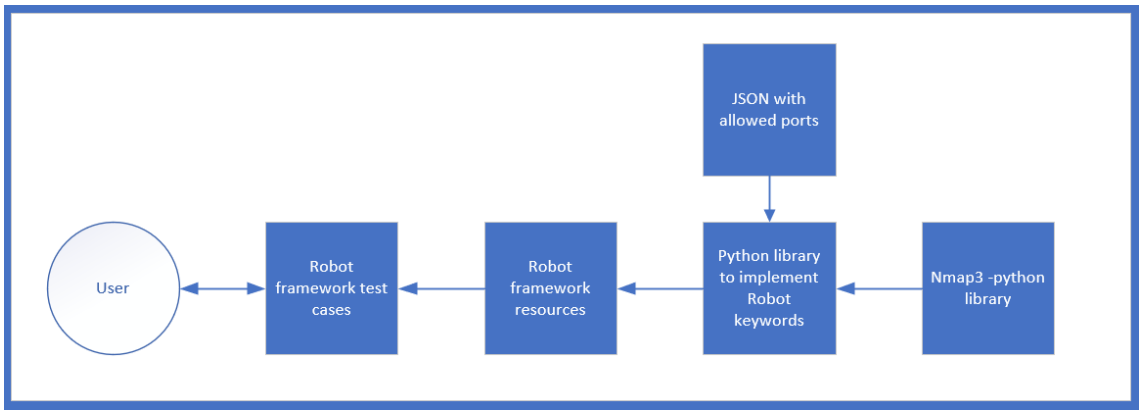

Figure 9. *Port scanning test automation design.*

This is using a design from the chapter 3.4.2 about Robot Framework test automation. The idea is that the Python library implements keywords for Nmap scans that could easily be called from a Robot test case file. This Python library also reads the allowed ports for certain protocols. The user only needs to run Robot Framework test cases, and then all the set Nmap scans are performed, and test cases check that there are no unallowed open ports.

3.5.3.2 Authentication and mTLS

This work focuses mostly on testing the security of the mTLS configuration, handshake, and authentication process. Two tools have been chosen for this purpose, SSLScan and SSLyze, which have already been presented in sections 3.3.2.8 and 3.3.2.9.

SSLScan is included here for ease of use and simple deployment. It is used to perform a quick general-level scan using a command-line tool. SSLyze, on the other hand, is suitable for more comprehensive testing and gives possibilities for easy integration into test automation from its Python API.

Figure 10 illustrates how the SSLyze tool could be integrated into Robot Framework test automation.

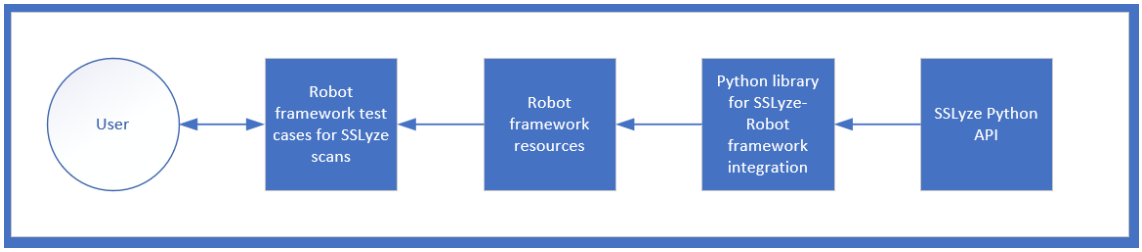

Figure 10. *Port scanning test automation design.*

The same design model from the chapter 3.4.2 has been used here, as in other Robot Framework test automation implementations, in this work to achieve consistency. The Python library in the picture creates Robot Framework suitable keywords, which perform certain SSLyze scans and return the scan SSLyze test automation design. results. This makes it easy to run the scans directly on the Robot test cases.

Authentication based on mTLS also needs to be tested so that only valid certificates can be used to successfully initialize the connection. This includes testing that expired, revoked, or otherwise invalid certificates cannot be used. In this work, this is considered a part of functional testing, and this testing is not implemented in this work. Error handling and the management of certificates are also not addressed in this work. In any case, these are an important part of testing a secure mTLS implementation, and must be done, even though these tests are not implemented in this work.

3.5.3.3 WebSocket API

For the WebSocket API, a lot of security testing is already designed in fuzzing and authentication security testing. This testing is focused mostly on manual penetration testing related to WebSocket.

One tested area is WebSocket APIs' durability against denial-of-service attacks. This is tested with simple Python scripts designed to generate a lot of data traffic into the API. API is tested against many connection attempts at the same time, with real certificates and fake certificates. API is also tested against many messages sent after successful

initialization of the connection. By designing tests both with and without user access certificates, protection can be strengthened against external attackers as well as those with authorized access to the API.

Another tested area replays the attack simulation by replaying real captured packets. This is performed with the help of Wireshark and Scapy tools. Wireshark is used to capture and save real packets, and Scapy is used to resend packets.

In addition, WebSocket APIs' behavior against different JSON-injections is tested. This is performed by manually creating malicious JSON payloads into WebSocket messages sent to the API. This is also testing the system's robustness against users with authorized access to the API.

3.5.3.4 REST API

A lot of testing for the REST API has already been performed in fuzzing and authentication security testing. This testing focuses mostly on manual penetration testing related to the REST API. This testing also includes a lot of similarities in performing security testing to traditional web applications, since a REST backend combined with the user interface is a common web application structure.

One area tested is API security against different kinds of DoS attacks. This includes totally external attacker connection creation, DoS testing with different parameters. It also includes authenticated message DoS testing to see how the API reacts against flooding messages into the server.

Then, testing is done against different common injections. Manual injection tests are run with different malicious inputs into HTTP message payloads. This includes testing for XSS-injection, JSON-injection, and SQL injection. Effective testing for injections also requires some review of the code implementation behind the API to see how data is handled.

Man-in-the-middle testing is performed by capturing and analyzing data. This also includes testing for replay attacks. Likely, the system under test is well protected against these attacks with mTLS, but this testing needs to be investigated for potential issues.

Another tested area is JWT-related testing. Testing includes investigation and testing for issues related to JWT token storage security. Then the JWT token and the data it includes are investigated. One tested area is also manipulating the JWT token with malicious data or editing headers.

In addition, traditional web applications have a lot of good vulnerability scanning tools. Testing here also includes running a vulnerability scan with the ZAP tool against the REST API and user interface.

3.5.3.5 QUIC API

Testing for the external interface with QUIC protocol contains multiple different sub-areas, and while QUIC is generally considered to be a safety-focused protocol, security testing of the interface itself is carried out for possible issues.

In the system under test, the QUIC interface is only used to receive data from the API, and messages sent from the client to the server after the connection is opened are not processed. Furthermore, whether messages are processed is more functional testing than security testing, and not within the scope of this work, so we can exclude, for example, extended fuzzing or sending malicious messages from the testing.

One area to test is to perform a denial-of-service attack by simultaneously generating many connection requests to a device, and to examine how the interface copes with these connection requests without disrupting the real user's experience while using this interface. This is done by using the Python aioquic library to create connections with various parameters.

In addition, a simulated replay attack has been implemented, where an attacker can access the network to record the actual packets used to create the connection. Testing retransmits and manipulates these packets to disrupt the server or hijack the connection.

3.5.3.6 SSH

SSH security testing is performed by two tools. Metasploit SSH scanning methods are used to get information on the SSH implementation. This data can be manually examined for possible vulnerabilities.

In addition, the Nessus Essential tool is used for automated vulnerability scanning, which can detect common vulnerabilities in the network. In this case, Nessus scan is run against the target system's SSH service.

4. IMPLEMENTATION AND TOOL SETUP OF SECURITY TESTS

This chapter presents all the test implementations that require programming or setting up tools. Developing and setting up tools has been done mostly in Kali Linux 2024.4 operating system in a VirtualBox environment, unless something else is specifically mentioned. The decision for this environment is because Kali Linux distribution is specifically made for security testing purposes, and it contains a lot of great tools pre-installed. Running in VirtualBox is decided so that there is no need to dedicate separate computer hardware for this research, and it could still run security testing in an isolated environment.

4.1 API fuzzing

API fuzzing library component's high-level structure is designed in the Figure 11.

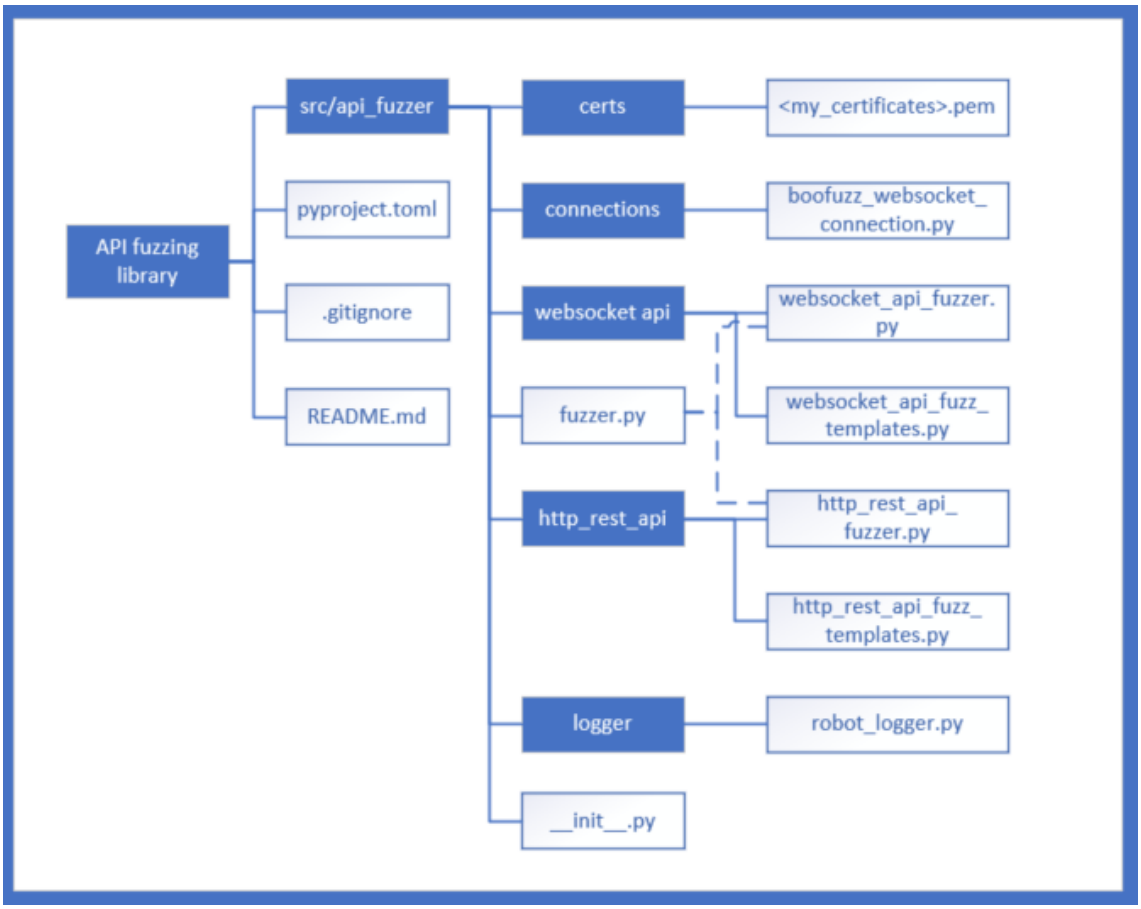

Figure 11. *API fuzzing library structure*

The figure shows the folders as blue boxes and the files in the library as white boxes. The file structure of the folders is connected by solid lines, while the dashed line represents the inheritance of the Python class in this figure. In addition to source code and resource files, the figure presents a common Python library structure and version control files, which are useful to provide to present the programming library. The documentation of the library itself is not implemented and is a major area that could be implemented and improved in the future.

In the system under test, one external interface uses WebSocket technology, but one problem with the Boofuzz library was a lack of direct WebSocket support. Therefore, a key addition in the API fuzzing library was to add WebSocket support to the Boofuzz library, to allow fuzzing of this WebSocket interface as well. Connection to the WebSocket API in the system under test is handled with mTLS. So, the decision is to implement WebSocket connection on top of the Boofuzz library's SSLSocketConnection class, because that class allows for easy implementation of mTLS connection into the API. The code implementation for this is presented in Program 14.

WebSocketConnection -class in Program 14 inherits the SSLSocketConnection class from the Boofuzz library, and adds the features required by WebSocket to open a connection and send a message. This is done by adding a WebSocket handshake to the connection opening context and wrapping the content of the sent messages into WebSocket format. This could be further developed by creating a common WebSocket wrapper for different connection types and possibly even adding this as an open-source contribution to the Boofuzz library itself, but to maximize the ratio of time spent on benefits here, the implementation was created based on the connection type used only.

Program 15 presents a common fuzzer class implementation. On top of Boofuzz fuzzing functionalities, here is an added example of embedded device service status monitoring. Status check works so that baseline status is captured before start of fuzzing, and then after each fuzz test case, status check is called to see if any of the defined service statuses have failed. In embedded systems, this is essential to be able to monitor the status of internal components and services within the system, in addition to just checking the status of the connection. Boofuzz offers a lot of possibilities to implement this based on the target system, such as monitoring received messages, running custom code after every test, and other customizations of test cases. The implementation of checking and reporting on the status of services is system-specific. The implementation of this is beyond the scope of this work, and this work includes only a pseudo-level example of it. Note also that to use the default certificates in an SSL connection, they must be set by the user to match the device to be connected in a .pem file under the certs folder.

Fuzzer for WebSocket API is inherited from the common fuzzer. The implementation of that is presented in Program 2.

```
    from api_fuzzer.connections.boofuzz_websocket_connection import Web-
SocketSSLConnection
    from api_fuzzer.websocket_api.websocket_api_fuzz_templates import Web-
SocketAPIFuzzTemplates
    from api_fuzzer.fuzzer import Fuzzer
    from boofuzz import Target

    class WebSocketAPIFuzzer(Fuzzer):
        """
        Class to fuzz WebSocket-api.

        Init parameters:
        device_ip(str): IP-address of device you want to fuzz
        device_port(int): Port number you want to fuzz
        ssl_context: Put nothing to use default certs,
                     or give python ssl-context for custom certs.
        logger: None for basic console logging,
                "robot" for Robot framework logging,
                or own IFuzzLogger custom logger.
        """
        def __init__(self, device_ip, device_port,
                     ssl_context=None, logger=None):
            # Init parent class
            super().__init__(device_ip= device_ip,
                             device_port=device_port,
                             ssl_context=ssl_context,
                             logger=logger)

            # Initialize template class
            self.templates   =   WebSocketAPIFuzzTemplates(self.device_ip,
self.device_port, self.ssl_context)

            # Init connection/target at start with basic parameters
            self.init_connection()

        def init_connection(self, ws_autoframe = True):
            # Create WebSocket/SSL connection
            connection  =  WebSocketSSLConnection(self.device_ip,  self.de-
vice_port,    ssl_context=self.ssl_context,    ws_autoframe=ws_autoframe,
recv_timeout=0.5)
            self.target = Target(connection=connection)

        def reset(self):
            """
            Resets all the session specific variables
            to original status after fuzz session.
            """
            self.templates   =   WebSocketAPIFuzzTemplates(self.device_ip,
self.device_port, self.ssl_context)
            self.session = None
            self.baseline_bit_status = None
            self.failed_cases = None
```

Program 2. *API fuzzing library's WebSocketFuzzer*

The goal for this class is that the user can just initialize this class and start fuzzing the wanted WebSocket API easily. There are also functionalities to initialize the connection again with different parameters between fuzz sessions, and functionality to reset all the session-specific variables between fuzz sessions. These are useful features added specifically for test automation purposes.

Program 16 presents the Boofuzz fuzzing template class for the WebSocket fuzzer class above. These templates are created with the fuzzing message creation tools provided by Boofuzz. The program includes a few fuzzing template examples for how to fuzz WebSocket message payloads and WebSocket message frames. An important part of fuzz testing is to precisely define these fuzzing message templates to match the system under test, so that the fuzz testing itself is efficient. This is because carefully defined messages can get to the heart of different parts of the services inside the device and can detect unexpected problems in different places inside the device. If we just send completely unthinkable junk, we're just testing the device's ability to block completely unexpected messages. Of course, this must also be included in the testing, but it is also essential to create partially correct message structures by fuzzing only, for example, a single value field in a correct JSON message sent into the WebSocket API.

Another very similar API fuzzer implemented in this work is the HTTP REST API fuzzer, which is presented in Program 17. It is implemented to fuzz HTTP REST APIs by inheriting the common fuzzer class in Program 15. It contains functionality to obtain a JWT token if the REST API under test uses a JWT token for authentication. This uses Boofuzz’s ready-made SSL connection to establish the connection, and uses Boofuzz templates defined in its class to define the content to be fuzzed. Currently, support is mainly for JWT token-based authentication, where a JWT token is requested from a specified sign-in endpoint using a password and username, and this JWT token is used for endpoints requiring authentication. The Boofuzz template class is shown in Program 3.

```
    from boofuzz import s_initialize, s_static, s_string, s_block_start,
s_block_end

class RESTAPIFuzzTemplates:
    """
    Class for all the REST-API boofuzz templates.
    Init parameters:
    device_ip(str): Fuzzing target device IP-address.
    jwt_token(str): JWT-token to access authenticated endpoints.
    """
    def __init__(self, device_ip, jwt_token):
        self.device_ip = device_ip
        self.jwt_token = jwt_token
        # List of all fuzzing template names
        self.names = []

    def define_all_templates(self):
        """
        Calls all template definition functions to register them.
        """
        self.general_fuzz()
        # Add more template functions here as needed.

    def general_fuzz(self):
        """
        General HTTP REST API request boofuzz template
        """
        name = "general_fuzz"
        self.names.append(name)
        s_initialize(name)
        if s_block_start("HTTP_Request"):
            # Fuzz HTTP Method
            s_string("GET", name="HTTP_Method")
            s_static(" ")
            # Fuzz endpoint path
            s_string("/api/endpoint", name="endpoint")
            s_static(" HTTP/1.1\r\n")
            # Fuzz Host header
            s_static("Host: ")
            s_string(f"{self.device_ip}", name="Host_Header")
            # Fuzz Authorization header (e.g., for JWT tokens)
            s_static("Authorization: Bearer ")
            if self.jwt_token:
                s_string(f"{self.jwt_token}", name="JWT_Token")
            else:
                s_string("example-jwt-token", name="JWT_Token")
            s_static("\r\n")
            # Fuzz content-type
            s_static("Content-Type: ")
            s_string("application/json", name="Content_Type")
            s_static("\r\n")
            s_static("\r\n")
            s_static("\r\n")
            s_block_end("HTTP_Request")
```

Program 3. *API fuzzing library's WebSocketFuzzer*

This example shows a general fuzzing template for an HTTP message sent to a REST API. This is very generic and contains mainly the normal message structure, and its fuzzing section by section and does not include possible payloads, and their fuzzing. Here also, system-specific, well-designed, and targeted templates need to be created to achieve the best fuzzing test results and test coverage.

Boofuzz also offers the possibility to create your custom logger by implementing the methods of the abstract IFuzzLogger class. This feature is used to create a class linking the Boofuzz logger and the Robot Framework logger, which allows the Boofuzz log to be streamed directly into the Robot Framework. This makes it a lot easier to check the detailed results after fuzz testing, as it allows you to read the log directly from the Robot Framework’s result files. This logger implementation is presented in Program 18. The implementation is mostly sending only Robot loggers' info messages, but there are exceptions. The robot logger's error logging is used to send an error message, and the WebSocket frame of the received message is decoded if the received message is a WebSocket.

Next, a test automation implementation using Robot Framework is presented, based on the API fuzzing library. Python libraries have been created for this purpose, from which it is easy for the Robot Framework to call keywords and get the desired result as a return value directly in the format processed by the Robot Framework. Below is a Python library in Program 4 with common fuzzing keywords for different APIs that can be called by a robot.

```
import json

class APIFuzzRobot:
    """
    Python support class for common functionalities
    of API fuzzing robot tests
    """
    ROBOT_LIBRARY_SCOPE = 'GLOBAL'

    def __init__(self):
        self.fuzzer = None

    def load_template(self, template_function_name, *args, **kwargs):
        template_function = getattr(self.fuzzer.templates,
                                    template_function_name)
        template_function(*args, **kwargs)
        return f"Successfully loaded: {template_function_name}"

    def reset_session(self):
        self.fuzzer.reset()
        return "Fuzz session reset successful"

    def fuzz(self, template_name, fuzz_depth_level=1):
        fuzz_result = self.fuzzer.fuzz(self.fuzzer.target,
                                       template_name,
                                       fuzz_depth_level)
        if fuzz_result:
            return "Fuzzing test cases successfully completed"
        else:
            return "Fuzzing test cases failed"

    def get_failed_test(self):
        return json.dumps(self.fuzzer.failed_cases, indent=4)
```

Program 4. *API fuzzing common Python library for Robot Framework keywords*

This includes keywords for fuzzing session reset between tests, as well as the generic fuzz command itself. There is also a keyword to receive failed fuzzing test cases if needed. Notable here is that the fuzzer object used by the class is abstract, and it must be implemented in another class, which inherits this class to use the features of this class. However, due to the repetitiveness of the code, the decision is made to make a separate parent class for the common properties.

For the interface technologies discussed in this work, HTTP REST and WebSocket, separate classes have been created to add technology-specific keywords to use for the Robot Framework. In the Program 5 below is a Python class for WebSocket-specific Robot Framework keywords.

```
    from api_fuzzer.websocket_api.websocket_api_fuzzer import WebSocketA-
PIFuzzer
    from APIFuzzRobot import APIFuzzRobot

    class WebSocketAPIFuzzRobot(APIFuzzRobot):
        """Python support class for WebSocket API robot fuzzing tests"""
        def __init__(self):
            super().__init__()

        def initialize_fuzzer(self, device_ip, device_port,
                              ssl_context=None, logger="robot"):
            self.fuzzer = WebSocketAPIFuzzer(
                device_ip=device_ip,
                device_port=int(device_port),
                ssl_context=ssl_context,
                logger=logger
            )
            return "Fuzzer initialized successfully."

        def init_connection(self, ws_autoframe="True"):
            bool_value = ws_autoframe.lower() == "true"
            self.fuzzer.init_connection(bool_value)
            if bool_value:
                return "Connection initialized with auto constructed WS
frame"
            else:
                return "Connection initialized without auto constructed WS
frame"
```

Program 5. *API fuzzing WebSocket Python library for Robot Framework keywords.*

Robot Framework keywords are created for initializing the WebSocketAPIFuzzer-class, and for restarting the connection if the user wants to fuzz the frame structure of the WebSocket connection itself. The initialize_fuzzer keyword of this class initializes the fuzzer object and must be run on the robot test cases before any other keywords can be run.

Program 6 shows the corresponding Python class defining the keywords for the Robot Framework, also for the REST API.

```
from api_fuzzer.rest_api.rest_api_fuzzer import RESTAPIFuzzer
from APIFuzzRobot import APIFuzzRobot

class RESTAPIFuzzRobot(APIFuzzRobot):
    """Python support class for REST API robot fuzzing tests"""

    def __init__(self):
        super().__init__()

    def initialize_fuzzer(self, device_ip, device_port,
                          ssl_context=None, logger="robot",
                          use_jwt_token=True, signin_endpoint=None,
                          user=None, password=None):
        self.fuzzer = RESTAPIFuzzer(
            device_ip=device_ip,
            device_port=int(device_port),
            ssl_context=ssl_context,
            logger=logger,
            use_jwt_token=use_jwt_token,
            signin_endpoint=signin_endpoint,
            user=user,
            password=password
        )
        return "Fuzzer initialized successfully."
```

Program 6. *API fuzzing Python library for REST API's Robot Framework keywords.*

Currently this does not implement any keywords other than the initial keyword of the REST API fuzzer. This is because the necessary parameters are largely provided in the fuzzer initialization, and when the test cases are also only at the example level, there is no need at this point to add keywords for changing a parameter value or keywords that implement new functionality.

The resources needed to fuzz the WebSocket API with a robot are shown in Program 7, in Robot Framework form.

```
*** Settings ***
Documentation    Resource for WebSocket API fuzzing tests
Library    ../../../pylibs/fuzzing/api/WebSocketAPIFuzzRobot.py

*** Variables ***
${DEVICE_IP}                <device-ip>
${DEVICE_PORT}              <device-port>
${SSL_CONTEXT}              <custom-ssl-context or nothing>
${LOGGER}                   robot

${FUZZ_SUCCESSFUL}          Fuzzing test cases successfully completed
${RESET_SUCCESSFUL}         Fuzz session reset successful
${CONNECT_INIT}                 Connection initialized without auto
constructed WS frame
${CONNECT_INIT_WS_FRAME}      Connection initialized with auto con-
structed WS frame

*** Keywords ***
Initialize WebSocket API Fuzzer
    [Documentation]     Initializes WebSocket API fuzzer with given
parameters.
    ${init_result}=     Initialize Fuzzer     ${DEVICE_IP}     ${DE-
VICE_PORT}    ${SSL_CONTEXT}    ${LOGGER}
    Log    ${init_result}

Run Fuzz Test
    [Documentation]    Runs fuzz tests for given template and resets
state after tests.
    [Arguments]    ${template_name}
    ${load_template}=    Load Template    ${template_name}
    Log    ${load_template}
    ${fuzz_result}=    Fuzz    ${template_name}
    ${fuzz_status}=    Run Keyword And Return Status    Should Be Equal
As Strings    ${fuzz_result}    ${FUZZ_SUCCESSFUL}
    ${failed_case_info}=    Run Keyword If    not ${fuzz_status}    Get
Failed Test
    ${reset}=    Reset Session
    ${reset_status}=    Run Keyword And Return Status    Should Be Equal
As Strings    ${reset}    ${RESET_SUCCESSFUL}
    Run Keyword If    not ${fuzz_status}    Fail    ${failed_case_info}
```

Program 7. *API fuzzing robot resources for the WebSocket API.*

The resources contain necessary keywords to initialize fuzzing, as well as the variables needed for fuzzing, such as the IP address of the target device. A similar resource file has also been created for the REST API, shown in Program 8.

```
*** Settings ***
Documentation   Resource for REST API fuzzing tests

Library    ../../../pylibs/fuzzing/api/RESTAPIFuzzRobot.py

*** Variables ***
${DEVICE_IP}                <device-ip>
${DEVICE_PORT}              <device-port>
${SSL_CONTEXT}              <custom-ssl-context or nothing>
${LOGGER}                   robot

${USE_JWT_TOKEN}            True
${SIGNIN_ENDPOINT}          </path/to/signin-endpoint>
${USER}                     <username>
${PASSWORD}                 <password>

${FUZZ_SUCCESSFUL}          Fuzzing test cases successfully completed
${RESET_SUCCESSFUL}         Fuzz session reset successful

*** Keywords ***
Initialize REST API Fuzzer
    [Documentation]    Initializes REST API fuzzer with given parameters.
    ${init_result}=     Initialize Fuzzer     ${DEVICE_IP}     ${DEVICE_PORT}        ${SSL_CONTEXT}        ${LOGGER}        ${USE_JWT_TOKEN}
${SIGNIN_ENDPOINT}    ${USER}    ${PASSWORD}
    Log    ${init_result}

Run Fuzz Test
    [Documentation]    Runs fuzz tests for given template and resets state after tests.
    [Arguments]    ${template_name}
    ${load_template}=    Load Template    ${template_name}
    Log    ${load_template}
    ${fuzz_result}=    Fuzz    ${template_name}
    ${fuzz_status}=    Run Keyword And Return Status    Should Be Equal As Strings    ${fuzz_result}    ${FUZZ_SUCCESSFUL}
    ${failed_case_info}=    Run Keyword If    not ${fuzz_status}    Get Failed Test
    ${reset}=    Reset Session
    ${reset_status}=    Run Keyword And Return Status    Should Be Equal As Strings    ${reset}    ${RESET_SUCCESSFUL}
    Run Keyword If    not ${fuzz_status}    Fail    ${failed_case_info}
```

Program 8. *API fuzzing robot resources for REST API.*

This also includes the Robot keywords for initializing the REST API fuzz and the necessary variables. Then the robot test case files containing the test cases themselves need to only add the corresponding resource, and run the tests with the desired keywords. In Program 9 presented example test case for WebSocket API fuzzing.

```
*** Settings ***
Resource      ../../../robotlibs/fuzzing/api/websocket_api_fuzzing.resource
Suite Setup    Initialize WebSocket API Fuzzer

*** Test Cases ***
Example Message Fuzz
    [Documentation]    Fuzz WebSocket API Get-message fields
    Run Fuzz Test    example_message_fuzz

Example Message Additional Field Fuzz
    [Documentation]     Fuzz all supported WebSocket API Get-messages with additional field
    Run Fuzz Test    example_message_additional_field_fuzz

WebSocket Frame Fuzz
    [Documentation]    Fuzz WebSocket frame structure

    # To fuzz WebSocket Frame we turn automatic WebSocket frame construction off.
    ${connection_init}=    Init Connection    False
    Run Keyword     Should Be Equal As Strings     ${connection_init}    ${CONNECT_INIT}

    Run Fuzz Test    ws_frame_fuzz

    # Reset connection back to normal
    ${normal_connection_init}=    Init Connection
    Run Keyword     Should Be Equal As Strings     ${normal_connection_init}    ${CONNECT_INIT_WS_FRAME}
```

Program 9. *WebSocket API fuzzing test cases example with Robot.*

This shows how the test setup is implemented by just running the fuzzer initialization keyword. After that, writing the test cases is straightforward by just calling the name of the desired fuzz test, which corresponds to the desired Boofuzz template from the WebSocket templates in Program 16. Creating test cases for the REST API is just as simple. Here is an example for REST API test case creation:

```
{
*** Settings ***
Resource    ../../../robotlibs/fuzzing/api/rest_api_fuzzing.resource
Suite Setup    Initialize REST API Fuzzer

*** Test Cases ***
General Fuzz
    [Documentation]    Fuzz REST API HTTP request in general
    Run Fuzz Test    general_fuzz
}
```

Currently, the implementation has focused on the creation of the API fuzzing general structure, and the implementation of the test cases themselves has been de-emphasized, as the test suites are very strongly tied to the system under test.

4.2 Vulnerability scanning and penetration testing

This sub-chapter includes test implementation and tool setup considerations for all the tested areas.

4.2.1 Penetration testing Python library

For penetration testing and vulnerability scanning scripts that are not used with test automation or Robot Framework, a separate Python library has been implemented. This library contains all the necessary scripts and resources, such as certificates. Figure 12 illustrates the structure of this library.

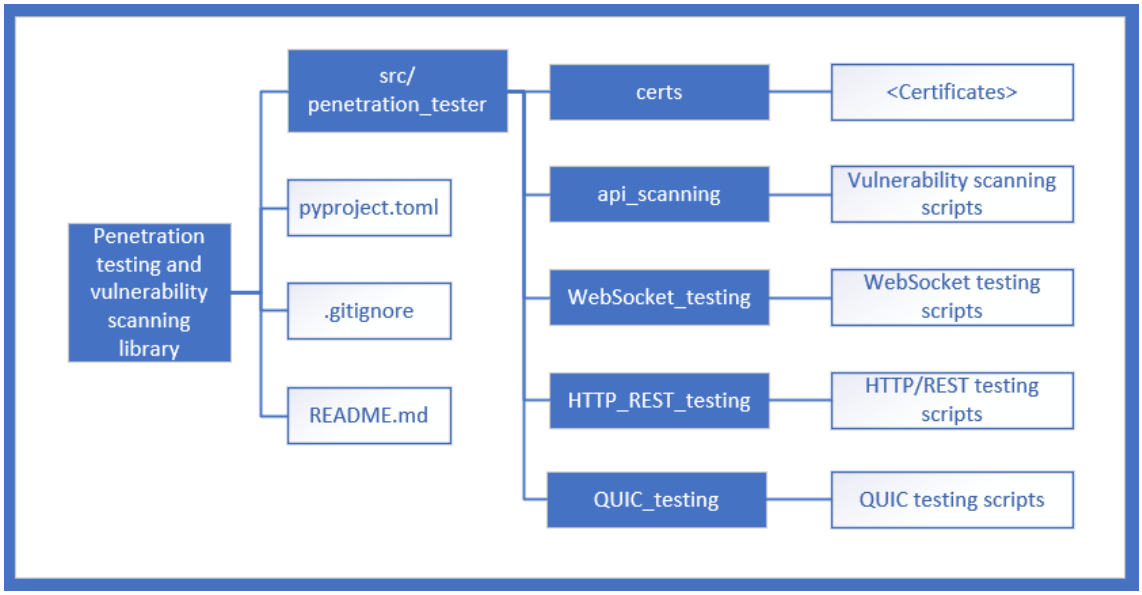

Figure 12. *Penetration testing and vulnerability scanning Python library.*

This library includes testing scripts for QUIC, HTTP/REST, and WebSocket APIs and certificates needed in testing. These test scripts are meant for manual penetration testing and vulnerability scanning. For test automation scripts is a different implementation, which is presented in the chapter 3.4.2.

4.2.2 Port scanning

The implementation is based on the designed test case structure. The JSON file structure shown below is used to represent the allowed ports by protocol.

```
{
   {
       "tcp": [80, 110, 443],
       "udp": [53, 67, 123]
   }
}
```

This JSON structure specifies the name of the protocol in the JSON keyword, followed by the port numbers accepted in the JSON array.

A Python class for creating Robot Framework keywords to run Nmap scans is presented in the Program 19. The program implements its own Robot Framework keyword to perform necessary initializations before scanning. This means, for example, initializing the NMap object and reading the allowed ports from the file. In addition, the program includes custom keywords for running different scans, which run the NMap scan for the desired protocol, compare the open ports found with the allowed open ports, and finally return a list of the forbidden open ports found. By default, this list should be empty if no unallowed open ports are found.

To define all the Robot keywords and variables needed in Robot test cases, there is a resource file in Robot Framework format. This is presented in the Program 10.

```
    *** Settings ***
2   Documentation   Resource for nmap port scanning tests

4
    *** Variables ***
6   ${TARGET}    <device-ip>

8   *** Keywords ***
    Initialize Nmap Port Scanner
10      [Documentation]    Initializes nmap port scan python library to run
nmap scans
12      ${init_result}=    Initialize Port Scan    ${TARGET}
        Log    ${init_result}
14
```

Program 10. *Port scanning Robot Framework resources*

Here is a Robot Framework keyword that calls an initialization method created in the Python library, which does the necessary initialization for the scan tests. In addition, there is the target device's IP address, which is defined by the user based on the target device.

Finally, there are test cases themselves, which the user can run to receive test results. These are presented in Program 11.

```
*** Settings ***
Resource    ../../robotlibs/port_scan/nmap_scan.resource
Suite Setup    Initialize Nmap Port Scanner

*** Test Cases ***
TCP Port Scan
    [Documentation]    Runs TCP port scan and compares open ports to
given list of allowed open ports.
    ${scan_result}=    Tcp Port Scan
    Should Be Empty    ${scan_result}    TCP port scan failed! Open
ports that should not be open noticed: ${scan_result}
    Log    TCP port scan successful: no unexpected open ports found.

UDP Port Scan
    [Documentation]    Runs UDP port scan and compares open ports to
given list of allowed open ports.
    ${scan_result}=    Udp Port Scan
    Should Be Empty    ${scan_result}    UDP port scan failed! Open
ports that should not be open: ${scan_result}
    Log    UDP port scan successful: no unexpected open ports found.
```

Program 11. *Port scanning Robot Framework test cases*

Here are example test cases for TCP and UDP scans. By adding other supported scans from Nmap to the Python library in Program 19 and the corresponding test case to this program is easy to add, as all the supported scans by Nmap are.

Another way to implement this could be the use of RoboNmap library which is using Nmap straight from Robot Framework [43]. Or to directly use Nmap program from Robot Framework. The python library used in this implementation has been more recently updated and better documented, which is why it was decided to be used in this work.

4.2.3 Authentication and mTLS

The first tool used in mTLS testing is SSLScan. Setting it up is simple since it is pre-installed in the Kali Linux environment used in this work. SSLScan is supposed to be run from a command-line tool in this work, and because of that, no other setup is needed.

The SSLyze tool is also pre-installed in the Kali Linux environment used in this work, but in this work, SSLyze scans are run through the provided Python API to automate testing. Test automation is implemented with Robot Framework, and the first step is a Python class, which implements Robot Framework keywords for different SSLyze scans. This is presented in Program 20. Design has been structured by creating a Python class that contains all the SSLyze scans as functions. These functions could also be called from Robot Framework as keywords. In addition, the class contains an initialization function, which initializes all the needed configurations to run SSLyze scans.

Based on the used test automation design, Robot Framework resources are isolated into their file. This is presented in Program 12.

```
*** Settings ***
Documentation   Resource for SSLyze scanning tests
Library    ../../pylibs/mtls_scanning/SSLyzeAPIUser.py
Library    Collections

*** Variables ***
${DEVICE_IP}    <ip_address>
${PORT}         <port_number>


*** Keywords ***
Initialize SSLyze Scanner
    [Documentation]    Initializes SSLyze python library to run SSLyze scans.
    ${init_result}=    Initialize Scanner    ${DEVICE_IP}    ${PORT}
    Log    ${init_result}_result}
```

Program 12. *SSLyze scan Robot Framework resources*

A resource file imports related Python classes as a library, which makes it possible to run Python functions as Robot Framework keywords just by importing this resource file. The initialization keyword is also defined in the resource file because it uses resource variables, like the target IP address and port.

Robot Framework example test cases for SSLyze scans are presented in the Program 13.

```
*** Settings ***
Resource    ../../robotlibs/mtls_scanning/sslyze_tests.resource
Suite Setup    Initialize SSLyze Scanner

*** Test Cases ***
SSL 3.0 Scan
    [Documentation]    Runs SSLyze SSL 3.0 scan and checks that it is disabled and no cipher suites is allowed.
    ${ssl3_result_dict}=    Ssl 3 Scan Results

    ${tls_version_support}=    Get From Dictionary    ${ssl3_result_dict}    tls_version_support
    Should Not Be True    ${tls_version_support}

TLS 1.3 Scan
    [Documentation]    Runs SSLyze TLS 1.3 scan and checks that it is enabled and only needed and secure cipher suites are allowed.
    ${tls_1_3_result_dict}=    Tls 1 3 Scan Results

    ${tls_version_support}=    Get From Dictionary    ${tls_1_3_result_dict}    tls_version_support
    Should Be True    ${tls_version_support}

    ${accepted_ciphers}=    Get From Dictionary    ${tls_1_3_result_dict}    accepted_cipher_suites
    Should Not Be Empty    ${accepted_ciphers}
    # If we know exact number of allowed cipher suites:
    Length Should Be    ${accepted_ciphers}    3

    # We can check that some exact wanted cipher suites are allowed
    ${accepted_cipher_names}=    Get Dictionary Keys    ${accepted_ciphers}
    List Should Contain Value    ${accepted_cipher_names}    TLS_CHACHA20_POLY1305_SHA256
    List Should Contain Value    ${accepted_cipher_names}    TLS_AES_256_GCM_SHA384

    # If we want to make sure that some exact cipher suites are not allowed
    List Should Not Contain Value    ${accepted_cipher_names}    TLS_AES_128_CCM_8_SHA256
    List Should Not Contain Value    ${accepted_cipher_names}    TLS_AES_128_CCM_SHA256

Heartbleed Scan
    [Documentation]    Runs SSLyze Heartbleed scan and checks that target is not vulnerable.
    ${is_vulnerable}=    Vulnerable To Heartbleed
    Should Not Be True    ${is_vulnerable}

Robot Scan
    [Documentation]    Runs SSLyze Robot scan and checks that target is not vulnerable.
    ${is_vulnerable}=    Vulnerable To Robot
    Should Not Be True    ${is_vulnerable}
```

Program 13. *SSLyze scan Robot Framework test case examples*

This includes some example test cases for SSLyze scan results. This does not include all the supported scans, but it works as an example for how scan results could be tested with Robot Framework. In example test cases is tested for example that the TLS implementation under test does not support some outdated SSL/TLS versions like SSL 3.0. Similar test cases could be made for other outdated versions like TLS 1.0 or TLS 1.1. The example test case in row 15 shows how supported cipher suites could be tested for allowed TLS versions. In addition, SSLyze supports scans for common TLS vulnerabilities and other TLS features. To these scans could be added test cases like for example, in row 45 for Heartbleed vulnerability.

4.2.4 WebSocket API

A Python class is implemented for the WebSocket API manual security testing to test the API under different conditions. This Python class is in Program 22. In the WebSocketTester class implemented in that program, there are multiple functions to perform different manual tests for the WebSocket API. This class has multiple functionalities for normal authorized usage of the API and unauthorized usage of the API. It also has functions to perform denial-of-service attacks, for example, by flooding many connection requests or real messages.

It can also use ZAP as a proxy when sending messages. This gives the possibility to use it combined with ZAP's features. ZAP could be used in manual penetration testing for WebSockets by, for example, manipulating message payloads, intercepting and reading messages, and fuzzing messages. To use ZAP as a proxy, it needs to be installed and executed on the user's operating system. ZAP also needs certificates configured to create a connection to the system under test.

In addition, message capturing of messages and replay-attack simulations are tested by capturing traffic with Wireshark and replaying messages with Scapy. Replay attacks are performed with the help of Scapy example scripts in Program 1.

4.2.5 REST API

To support manual security testing of the REST API is implemented simple tester class with Python. It is created to send messages to the REST API with different parameters to support different scenarios. This Python class is presented in Program 23. That Python class includes functions to easily send different HTTP requests to the target API with real or fake authentication certificates. The user can also run the sign-in function with their parameters to receive a JWT token for authentication. Python class also supports using ZAP as a proxy between the program and the REST API. This gives the possibility to

use all ZAP features while sending messages. This class also has functions to run a denial-of-service attack against the REST API with different parameters. This also requires configuring ZAP with real API certificates and other required configurations.

In addition, a ZAP vulnerability scan is performed. This requires configuring ZAP to scan the API, which requires, for example, configuring authentication to ZAP for better scan results. ZAP supports a wide variety of different authentication methods and even supports scripting custom authentication methods. This is required for better scan results in APIs that require authentication. ZAP vulnerability scan executed with ZAP API from Python is also implemented in Program 24. It is just an example and requires configuration from the user, for example user needs to implement a real authentication method into the program.

Man-in-the-middle attack and replay attack testing setup with Wireshark and Scapy is simple in this case also, since both applications are pre-installed in Kali Linux, used as an operating system in testing. Message replaying is performed with the help of Scapy replay scripts presented in Program 1.

4.2.6 QUIC API

A simple Python class made for QUIC connectivity testing is presented in Program 21. That program is implemented as part of the penetration tester Python library structure, designed in Figure 12. The main goal of this program is that users can test different connections to manually implement security testing for the QUIC server. The user can create a connection with real certificates or fake certificates. The user can test the connection with a real ALPN field or try to manipulate it. A user can also create a denial-of-service attack by creating many connection requests in a short time. At this point, this testing class for QUIC is simple, and it should be complemented with more comprehensive testing capabilities.

In addition, tests for how a server reacts to replay-attack attempts are performed with the help of scripts in Program 1. Wireshark is used to capture packets, which directly supports capturing QUIC packets and saving them in PCAP format.

4.2.7 SSH

For SSH scans and tests, two tools are tested. The first one is Metasploit. For automation of Metasploit, there are Python libraries, but for SSH scans, test automation for the

Metasploit tool is not implemented in this work, and tests are run manually from a command-line application. This is because the Metasploit SSH scan provides data about the tested SSH service and does not return information if there are some vulnerabilities, which would make test automation harder in this case. Metasploit is pre-installed in the Kali Linux environment, which means that no setup is needed if the user does not want to run some tests that require plugins to be downloaded. In SSH testing of this work, no additional plugins are downloaded.

Another tool used is Nessus Essentials. Nessus is used from the user interface in this work, and no automation is implemented for it. Setting up of tool is made by downloading the Nessus installer and installing it in a Kali Linux VirtualBox environment. Since security testing is often performed in an isolated network environment, it is good that Nessus supports installing plugins and activating licenses in the offline environment, even though it is a commercial product. In this work, all the plugins are installed into Nessus.

4.3 Test execution and result analysis

This sub-chapter is about running designed tests and analysing the results. Result analysis is mostly done from the perspective of what the advantages of the used tools and test methods are and what could be improved, rather than directly analysing test results in detail.

4.3.1 API fuzzing

Fuzzing APIs over the network is executed by running Robot Framework test cases, like in the example test cases. Executing tests with Robot Framework is very effortless, since test cases are designed so that the user can just run all the Robot Framework test cases with one click and receive a result report. The result report itself is generated as a very clearly readable HTML file, showing the results of the tests. Under each test, there is also provided more detailed information and a log of the test run. Particularly useful in this case is the possibility provided by the used Boofuzz fuzzing tool to program our message logger based on the Boofuzz library's IFuzzLogger abstract class. Robot Framework logger for fuzzing runs is created based on that is useful in this case, since now under each test case, all the logs and details from the fuzzing run are presented and saved into the result report. Log details from one example fuzz case are shown in Figure 13.

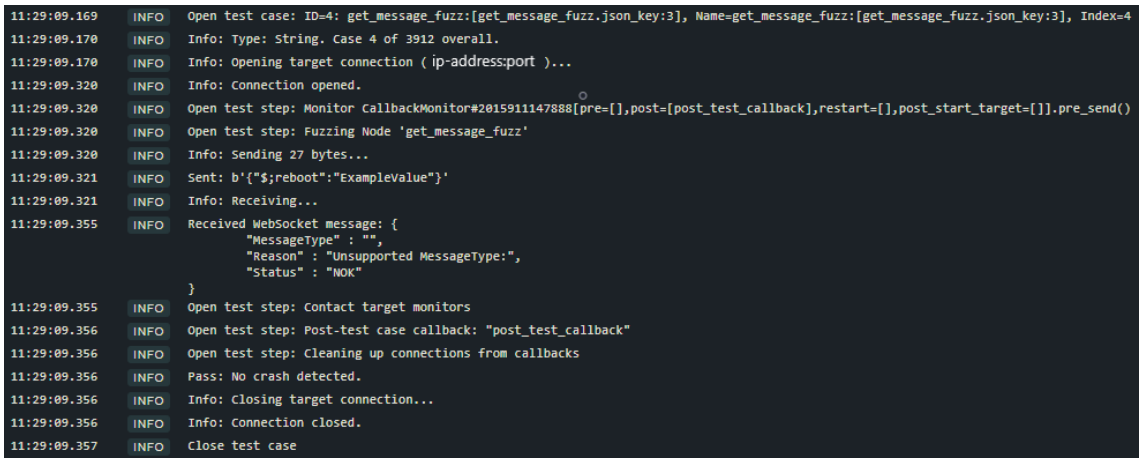

Figure 13. *Example fuzz case log in Robot Framework result report.*

A similar log is found in the result report for each fuzz case of a test case. This allows users to analyze test runs at a deeper level if there are some possible vulnerabilities or other problems in test cases. This detailed log shows, for example, sent messages, received messages, and test steps from start to end of the fuzz test case.

Another handy feature, especially in this case, is the ability to run our code after each fuzz case, as this allows the user to monitor the state of the embedded system while fuzzing. During the test drive, this worked perfectly, and with the help of this is possible to monitor the status of the device continuously during the test drive. The point to note here is that this is just as useful as how well the status reporting of the embedded system works. The requirement for this to be useful is that the status reporting system could report the detailed status of all the necessary services with high accuracy continuously. That is because when fuzzing an embedded system API over the network, there is no way to see if some service behind the API has crashed if there is no status monitoring system.

Another useful feature noticed in the test runs is the ability in Boofuzz to easily create very complex message structures, for example, by fuzzing only one value in a complex JSON. This makes it easy to target the fuzzing to a specific point and service in the system that handles this value. This improves test coverage a lot compared to just sending completely garbage data. The biggest problem here is that to achieve broad test coverage for fuzzing complex message structures, a lot of different fuzzing message templates must be designed. Therefore, for even better coverage, it is also useful to perform fuzzing at the component level for the software components that handle the API data.

One thing to consider when running fuzzing as part of test automation is also how long fuzz run is required to cover most of the bugs. For example, Boofuzz has thousands of

fuzzing mutations for a single fuzzed value, which increases time and resources fuzz testing takes for larger systems with a lot of messages. Also, due to the nature of fuzzing and randomness of sent data, test cases are not repeatable. In some test setups, it might be a requirement that test cases are repeatable and results are same in every run. So, this requires some consideration for whether fuzzing is to be run as part of bigger test sets or as its own test set, or maybe it could be included in a test set with regression testing or other similar testing, where test cases are not repeatable.

In conclusion, using Boofuzz fuzz embedded system APIs over the network seems to be a valid approach. That is because Boofuzz is very customizable, which provides possibilities to create fuzzing test cases tailored exactly to the system under test. Since embedded systems might require tailoring because of a wide range of used technologies and protocols, and also might require system status monitoring possibilities. Problems can be caused by the large number of message fuzzing templates required to achieve test coverage, which in turn means that fuzzing requires a lot of time and resources to run.

4.3.2 Vulnerability scanning and penetration testing

This sub-chapter includes vulnerability scanning executions and result analysis considerations. In addition, penetration testing performed with automated tools and manually is handled in this chapter.

4.3.2.1 Port scanning

Port scanning is performed automatically with Robot Framework test cases designed. This seems to be a suitable approach for this, since it is easy to write down once ports that should be open for a certain service and then just compare the port scan result. This is an easily repeatable test area as part of test automation.

The Nmap3 Python library used for this seems to work well, at least for running basic scanning functions. One point that requires tweaking is Nmap's requirement for admin rights to perform some scans, for example UDP scan. It means that port scanning tests require the test runner to have admin rights for a system where tests are executed.

One noticed issue is that UDP scans are taking a very long time, which is not optimal. The reason for that is likely the nature of UDP, where packets are sent without confirmation from the receiver side. This makes scanning for UDP-based services like QUIC irritating. In addition, looks like Nmap UDP scan does not always find which ports are open. This happens because UDP services might not actively refuse packets if the data is not exactly in the expected format, which means the service just drops scan packets and no

scan result is obtained, even though the port might be open. One notice is also that if service recognition is turned on, scans might find open ports effectively, but scans are taking a lot longer. TCP scan instead is working well, and with all the test runs it found all the supposed open ports from the system under test.

4.3.2.2 Authentication and mTLS

First, both SSLScan and SSLyze scans are executed from a command line tool to make a comparison between them. SSLScan execution is very simple, just by giving the IP address and port of the scanned service. It provides very easily readable results about TLS implementation. For example, there is information about which SSL/TLS protocols are supported, what TLS features are supported, what cipher suites are supported, and some information about the certificate it uses. SSLScan provides all the most important information for testers about TLS implementation in a summarized package. The benefit is that it is simple to read and understand, but also simple to execute scans. This is recommended if tester wants to run a simple SSL/TLS scan easily.

SSLyze is just as easy to execute from the command line tool as SSLScan. The results SSLyze provides are a bit more detailed than SSLScan results. It also tests common vulnerabilities more extensively than SSLScan. The problem here is that the result provided by the command line tool is a lot harder to understand, since there is a lot more information. That means it requires more time to analyze scan results, but also enables the possibility for deeper analysis.

In this work, SSLyze scans are also executed through Robot Framework to research how it is suitable for test automation. This requires a lot of effort to set up tests, but once those are set up, it performs well, since there are a lot of characteristics to easily test as part of test automation. For example, it could run scans against common vulnerabilities and return information if the system under test is vulnerable. Another possibility is scanning for supported TLS protocols and testing that no vulnerable protocols are supported. In addition, if only certain cipher suites are desired to be allowed, a test case could be designed for that purpose easily by comparing the scanned supported cipher suites and the ones wanted to be supported. One thing to notice is that not all the information SSLyze scan provides could be automated, for example, some data about certificates it provides is not easy to test as part of automation, and it also requires manual effort from a tester. But as a conclusion, test automation seems to be a effective way to perform SSLyze scans for TLS implementation, even though security testing of TLS implementation needs to be complemented with manual effort from a tester.

4.3.2.3 WebSocket API

Execution of DoS testing with the help of the implemented Python testing class for WebSocket requires a diverse approach. It requires running DoS scripts from one system and at the same time testing normal usage of the API from another system to see if the API is under DoS conditions at some point of testing. For an individual system, it is also possible to estimate how much data needs to be flooded to the API before its performance starts to degrade, and for example, whether the system has some explicit limits on, for example, the number of connection requests or messages from a single IP address in each period.

Capturing WebSocket messages with Wireshark is simple, but once the WebSocket API uses mTLS in the system under test, all the data is encrypted after connection initialization, which makes capturing and reading messages pointless. Also, performing a replay attack by resending packets is extremely difficult since mTLS has robust packet replay protection. However, at the general level, man-in-the-middle attack simulations and replay-attack simulations are useful tests to perform, even though at this point, these are not that important since the system under test uses mTLS for encryption and authentication.

Testing different injections and other message payload manipulations is simple by sending messages with custom payloads with a Python testing class. One great way is also to send real messages through ZAP proxy and then edit the real messages payload to contain known possible injections like JSON-injection or SQL-injection. This makes it easy to manually edit WebSocket payloads from the UI. Here could also consider automated testing for some common injections, but probably more efficient way is to manually try injections at points where it could be potentially vulnerable. That also requires knowledge about the design of the system under test. Since the tester needs to know where vulnerabilities might arise, for example, in places where API data is handled.

4.3.2.4 REST API

DoS testing with created Python scripts seems to work fine, as it gives the user a lot of opportunities to run DoS with different parameters and then try to target the API with a DoS attack in many ways. Use of TLS and JWT-based authentication helps protect against many types of DoS attacks in the system under test. For example, without authentication user is not able to send heavy HTTP requests to the server without these getting dropped early. Authenticated user instead might be a problem if the API has not implemented any DoS protection, like rate limiting, timeouts, or resource controls against

authenticated users. Also, the TLS handshake itself takes some resources, and someone flooding the server with TLS handshake requests with fake certificates might cause DoS conditions.

Testing REST API against different injection attacks manually is easy and effective with a Python class implemented, combined with ZAP. ZAP has great features in editing and resending real messages into REST API endpoints again, which makes it easy to test, including different injections into real message payloads, for example. This also requires manual research related to the API to see where data is handled and where XSS or JSON-injection vulnerabilities exist. One problem that requires configuration is that ZAP, with default settings, wants to snip the port number out from the REST API HTTP request host, even though the originally sent message includes it. That might cause issues since embedded system REST APIs and user interfaces are usually running behind some IP address and port, and not behind a URL domain name.

ZAP vulnerability scan also seems to work fine when running on an embedded system, as it seems to find REST endpoints and possible vulnerabilities in these. ZAP's support for customized authentication is extremely necessary here, since information received from an unauthenticated scan is a lot more limited. ZAP also supports running vulnerability scans through its API, used with Python, which seems to work well too, even though configuring authentication and other required configurations in the API might take time from the user. Running ZAP from API also opens possibilities in test automation, which could be investigated, but in this work, test automation for it is not implemented since it could be complex and not give that much additional value on top of running that scan manually.

JWT-token related testing is also easy to perform by requesting for JWT token from the API. Then it could be decoded to analyze if there is some potentially dangerous information. JWT token could also be manipulated and used after that to see how the server reacts. For example, testing executed for header manipulation is very easily performed this way.

4.3.2.5 QUIC API

QUIC API tests are performed manually by running QUIC scripts with different parameters and receiving information. Testing is mostly run for connection creation at this point. First tests are related to DoS-attack simulation for the system under test. This is now performed only by flooding connection requests to the server, since the system under test does not handle messages sent into the system. The first test is about spamming connection requests with a fake certificate and a wrong ALPN field. This DoS attempt is

not that effective in general since, with the wrong ALPN field, QUIC can drop the connection attempt early, which does not consume many resources. A problem with QUIC might occur if an attacker gets the ALPN field right, since then the QUIC server starts a full mTLS handshake, which is a resource-consuming operation compared to dropping the connection with the wrong ALPN field early. That is why QUIC server implementations need to handle this somehow, because the ALPN field is usually some normal word and not a secret or cryptographically safe string.

Another tested area is performing replay attacks, because QUIC might be vulnerable to replay attacks in some cases. At least if QUIC implementation supports connection resumption, it might be vulnerable if not designed and implemented carefully. Running tests for replay attack is again performed by capturing QUIC messages with Wireshark and trying to resend them in different scenarios with Scapy scripts presented in Program 1. Wireshark and Scapy work nicely as tools since Wireshark recognizes and supports QUIC packets directly, and Scapy could be used to send these packets easily with customized scenarios.

4.3.2.6 SSH

SSH scanning with Metasploit starts by running an SSH version scan. This provides information about the used SSH service, but it also provides information about supported encryption algorithms and key exchanges. That information might provide valuable information for testers about the SSH service from the perspective of an external attacker. Service information like service software and versions should be ones with no known vulnerabilities, since otherwise an attacker might try to exploit a vulnerability that the service is known to be vulnerable to. Testers should also be interested in supported cryptographic algorithms for in case there are some known weaker algorithms, because otherwise an attacker might try to force usage of these for compromised security.

Metasploit also offers other possibly useful SSH-related modules. If a tester wants to try to force a login, they might try to use brute-force login modules. There are also modules to help brute force logins, like ones that try to exploit little timing differences or response behaviors between the right username and both wrong username and password. These are not maybe the most important tools for testers, but it is always good to recognize that hackers might also be very creative in finding vulnerabilities, and testers should be aware of the tools attackers might try to use.

In addition, a test run for SSH service security is run with Nessus Essentials. This is different since it provides automated scanning for vulnerabilities related to, for example, some SSH versions. This is also a solid way of finding common vulnerabilities from SSH

versions more easily than manually researching SSH service information. Nessus also provides UI for running scans, which makes the user experience a lot better. At this work focus is only on free versions of both Nessus and Metasploit. Both provide commercial versions with a lot of possibly useful features, but with more effort and expertise, free tools are also a solid option.

5. DISCUSSION

This chapter discusses the results of thesis work and a comparison with previous research on the subject. Limitations and future development areas related to thesis topic are also considered.

5.1 Evaluation and comparison with existing research

This thesis successfully assembled methods that had been proven successful in different studies into a coherent set of methods for testing the security of interfaces in an embedded system. The studies from which these methods were derived will be discussed in detail later in this chapter. Existing studies were searched from multiple online libraries, such as IEEE Xplore and Tampere University's online library Andor. Searching tools such as Google Scholar were also used.

One of the main themes of this thesis is protocol-level fuzz testing over the network. Jiang et al. [13] identified unique challenges for this testing technique, which are protocol statefulness, network interface dependencies, and highly structured inputs. Liang et al. [44] also noticed similar challenges in network protocol fuzzing, like strict input validation, which prevents fuzzing reaching deeper into logic. They also studied many different fuzzer tools and compared their characteristics and performance. In addition, they researched different fuzzing tools and techniques related to network protocol fuzzing in theory. Sorsa [45] researched fuzz testing as part of test automation with Robot Framework in embedded systems with software level protocol fuzzers.

This thesis distinguishes itself from previous research by combining test automation using the Robot Framework with fuzz testing of modern web protocols, including REST and WebSocket, in the context of embedded systems. In this thesis, previously mentioned challenges, like network protocol statefulness, network interface dependencies, and strict protocol input validation were noticed and still seen as a challenge. One additional challenge arising from test automation in embedded environments is the complexity of monitoring system behavior during fuzz testing. The work also implements fuzzing support for the WebSocket protocol and Robot framework logging in Boofuzz programming library.

In addition, Ashfaq [46] researched similar topic related to security testing of Yocto Linux–based distributions. This thesis is also using Yocto Linux–based system as test environment but is focusing more on just the security of network interfaces, which makes

it possible to get deeper level security testing in network interfaces, for example by focusing on fuzzing, penetration testing, and certain protocols. For example, Ashfaq [46] researched network security testing tools like Nessus, OpenWAS and Nmap, but tools for fuzzing are not included. In addition, this thesis includes the Robot Framework test automation perspective, which is not included in Ashfaq's [46] work.

There are also numerous studies on penetration testing of web protocols and web applications, which are discussed below. For example, web applications vulnerabilities like SQL-injection and XSS-injection are very common in traditional systems [47]. This thesis noticed that in embedded systems these vulnerabilities are still valid, since SQL-databases and web application user interfaces hosted from embedded system are common nowadays. Broken access control and error handling are also common security issues in web applications [48]. This thesis also identifies error handling as a common concern in the embedded system context. One note about access control in embedded system context is that web applications are often single-user applications where access control is not complicated, even though getting access to single user needs to be strictly handled. The reason for this is that embedded systems are commonly used in environments like industrial automation, smart homes, and medical devices, where a single technician or administrator often manages the device. This reduces the need for complex multi-user access control. Of course, there are also embedded system applications where more complex access control is required, such as hospital devices where, for example, nurses, doctors and mechanics may have different levels of access.

Also, various DoS attacks are quite typical for the type of interfaces discussed in this thesis. For example, slowloris and HTTP flooding attacks against REST API are common [49]. Also, QUIC is vulnerable to different DoS-attacks variation, for example, QUIC flood DDoS, abusing version negotiation, and initial message flooding [23]. This thesis also noticed concern about the DoS attacks against QUIC, REST, and WebSocket APIs handled in this thesis, since embedded systems are often working with very limited resources, which makes DoS attacks even easier for attackers to perform against embedded systems. This requires interface design, that particular attention must be paid to DoS protection. For example, by implementing different rate limits and firewall protections, or considering usage of DoS protection service tools.

Also, Dong et al. [12] researched mTLS and TLS related connections and their use cases, concerning certificate practices, and privacy issues. This thesis focuses on creating test automation with Robot Framework to find the most common mTLS problems and weaknesses, but also this thesis considers the need for manual penetration testing

in mTLS related problems. TLS related security testing is also executed in real-world environment embedded system.

As presented earlier in this chapter, many studies have examined security testing tools and their performance around a single topic. This thesis has combined many different security testing tools, which are serving the exact need for security testing of embedded systems' network interfaces. This thesis researched the usage of many tools for fuzzing, port scanning, vulnerability scanning and penetration testing in the embedded Yocto Linux system network interfaces.

5.2 Challenges and future work

This sub-chapter handles challenges noticed while performing research about embedded system API security testing. This also contains discussion about the limitations of testing implemented now and how testing could be improved and developed in the future.

5.2.1 Challenges encountered

The first challenge noticed is related to the wide range of protocols and technologies used in embedded system APIs. For example, in the system under test of this work there are 3 APIs using different technologies and an additional SSH service. Each protocol and technology have its requirements and characteristics that affect how security testing should be carried out. Therefore, a single standardized test method is rarely sufficient. Different protocols require the use of different tools and test logic, which adds complexity to the testing process and increases the time needed for preparation and customization.

Another challenge is that security testing of embedded systems usually involves interaction with physical hardware, which introduces unique constraints. Unlike in purely software-based systems, an incorrect input or test case may not only cause an error message but may also crash or freeze the entire device. In some cases, physical recovery actions, such as turning on the power or reloading firmware, are required to restore functionality. This can slow down the test cycle and cause difficulties in maintaining consistent test environments.

In many embedded systems, a single API endpoint can trigger functions in several different services or subsystems. This poses a challenge for comprehensive monitoring of the impact of security testing. Without centralized logging or consistent status monitoring, it can be difficult to determine which part of the system was affected or whether a security test had unintended side effects. Furthermore, limited debugging interfaces or a possible lack of real-time logging on an embedded platform can also complicate observability.

Security testing tools often assume that standard interfaces and standard system architectures are in use. However, in embedded systems, APIs are often highly customized in both structure and behavior. As a result, premade tools usually require considerable adaptation to work properly in this context. For example, test scripts might have to deal with binary payloads, customized authentication, or unusual response codes. This increases the learning curve and slows down the test development process. Fortunately, in modern embedded systems such as the system under test in this work, the technologies and protocols involved are often already very close to the standardized ones used in traditional information technology systems, which also facilitates the use of standard testing tools.

Many embedded systems use advanced or proprietary authentication mechanisms such as mTLS, token-based systems, or hardware-specific encryption. Implementing these authentication schemes in automated tests can be challenging. Security testing tools may not support these mechanisms out of the box, requiring manual integration of certificate processing or custom authentication flows. This can become particularly complex when tags or credentials are short-lived or tied to dynamic session modes.

Automation is a key objective of modern security testing to enable repeatability and scalability. However, automating tests of embedded APIs is often not straightforward. Test results can vary depending on system state, network conditions, or timing, making it difficult to define clear pass/fail criteria. Some vulnerabilities may only occur intermittently or under certain conditions, making automation even more difficult. In addition, error handling and recovery logic could be included in the testing to deal with device crashes or inconsistent responses.

5.2.2 Limitations of testing and improvement considerations

While the developed security testing implementation provided a valuable platform for exploring the security of embedded system APIs, several limitations were identified during its implementation and use. These limitations highlight areas for improvement to evolve the testing into a more robust, maintainable, and scalable toolset.

The current testing approach consists of loosely coupled tools, scripts, and components, tailored for specific protocols or testing purposes. This fragmentation leads to a lack of cohesion and makes both maintenance and usability difficult. While each tool can work well in isolation, the overall workflow would benefit from a more coherent architecture. In addition, many protocols have common features, such as authentication methods, data

encoding, or request and response models, which are currently handled separately. Unifying these overlapping elements across protocols could significantly improve consistency and reduce redundancy in the testing.

Currently, the testing approach requires a lot of prior knowledge from its users, especially for manual penetration testing tasks. Many procedures and tool configurations are based on tacit understanding rather than well-defined instructions. Thorough documentation would be essential to improve the usability of testing design in real-life situations, especially by external testers or organizations. This would include user guides, examples, configuration templates, and explanations of test case design principles. Without proper documentation, designed testing runs the risk of remaining usable only by its original developers.

Testing currently supports only a narrow subset of protocols and technologies used in embedded systems. This limits its suitability for systems using rare or proprietary interfaces. In addition, there is no standard process for integrating new protocols or extending existing components. Defining a clear extension mechanism, such as additional interfaces or protocol abstraction layers, would facilitate other developers' participation and the scalability of the testing approach. Standardization of how new protocol handlers are developed and validated would be crucial for long-term maintainability.

Many of the tool implementations in the designed testing are linked at some points to specific configurations or technologies used in the target system. This reduces their reusability across projects or environments. Ideally, tools should be more general-purpose and configurable to support a wider range of use cases. Removing the specific details and focusing on modularity would make the test design more flexible and suitable for wider use.

5.2.3 Future work

This chapter considers development work that should be done to improve security testing implementation based on limitations noticed in this work. This chapter also includes a short analysis of promising areas and technologies in the security testing field, like post-quantum cryptography and the use of artificial intelligence and machine learning as part of security testing.

One of the main areas for improvement is the integration of existing tools into a coherent and modular testing approach. Currently, the toolbox is fragmented, with overlapping functionality implemented separately for different protocols. Future development should

focus on creating common components for common tasks such as authentication processing, input encoding, logging, and result analysis. The introduction of extensible architecture would also allow easier integration of new protocols or test modules, thus supporting extensibility without requiring major changes to the core testing design.

To make the testing approach suitable for wider use, either by industry or by other researchers, it needs to be accompanied by comprehensive documentation. This includes installation instructions, configuration examples, descriptions of supported test types, and instructions for adding new protocol handlers or automation logic. Better documentation would also lower the threshold for users without in-depth knowledge of the domain, making designed testing more accessible and easier to maintain in the long term.

Currently, some parts of the designed testing are tightly coupled to the system under test. Future versions should aim to decouple the tools from device-specific defaults, allowing reuse across a wider range of embedded platforms and applications. This could be achieved through configuration files, schema-based test definitions, or abstract interfaces that separate protocol logic from system-specific behavior.

The results of safety tests are often difficult to interpret, especially when they affect several subsystems. Future work could focus on automatic correlation of logs, alerts, and behavioral anomalies to more effectively identify potential security problems. The integration of monitoring tools capable of monitoring multiple services in parallel would be useful to detect cascading failures or unexpected side effects due to API misuse. One problem is that system monitoring is system-specific, usually, which means that monitoring the system needs to be done by a user tailored to the system. The idea is that some kind of abstract level implementation could be made that the user could use to easily integrate a monitoring system into designed security testing.

In the future, with the advancement in quantum computing, many existing algorithms used in cryptography are becoming vulnerable. In the future, one research area will be how quantum-safe algorithms could be used in embedded systems, but more directly related to this thesis work is the problem of how a transition to post-quantum cryptography is integrated into security testing.

Another interesting possibility in the future is the use of AI in security testing. AI and ML techniques are increasingly being explored in the context of automated vulnerability discovery and behavioral anomaly detection. In the future, these approaches could be applied to, for example, automatically generate test cases based on learned API patterns. Another use case could be to detect unusual system behavior during testing by comparing against baseline models or to predict likely vulnerability locations using training data

from previous findings. Although the integration of AI into embedded security testing is still in its early stages, it holds significant promise, especially for systems with complex, dynamic behaviors that are difficult to fully test manually.

6. CONCLUSION

The main result in this research was the success in designing comprehensive testing for the security of network APIs for modern embedded systems with the help of open-source security testing tools and custom scripts with Python. The work also succeeded in defining several areas of embedded system API security testing where test automation can be used with the help of Robot Framework. However, the designed testing is now only an initial set of tools and needs to be further developed in many areas to create a versatile and flexible test framework. There is still work to be done to seamlessly integrate different tools, technologies, and protocols to get one coherent framework.

The work found that the API protocols and technologies of modern embedded systems are close to the technologies and protocols used in traditional IT systems, and the same tools could be used in security testing of both, even though some modifications and customizations might be needed. However, the rarer or newer the protocols are, the more difficult it is to find premade tools that can be integrated into the testing. In addition, the large number of different protocols and technologies and possible custom implementations also causes a lot of work to implement robust security testing.

Many different techniques were used in this work to see which are most effective and promising. Fuzz testing turned out to be an effective way to find unknown vulnerabilities in the system. It also finds well vulnerabilities related to unexpected input, like buffer overflows or unhandled exceptions. Fuzzing embedded system API turned out to be possible to automate with Robot Framework, even though it requires a system-specified comprehensive monitoring system integrated into fuzz testing to monitor possible crashes or other issues in the system while fuzzing.

Different vulnerability scanning tools also worked well for embedded system APIs, at least for popular standardized technologies and protocols. Once protocol or technology is customized, it requires a lot of configurations in many scanning tools, which requires a lot of work.

Maybe the hardest part to include in security testing design is manual penetration testing with the help of tools and custom scripts. That is because manual penetration testing is hard to standardize, and it requires a lot of expertise in the security testing field. Security testing areas that cannot be automated easily need to be tested, and to perform manual security testing as part of test design, it requires specific documentation and guidelines of what needs to be tested, how it can be tested, and standardized test result logging.

In conclusion, creating a security testing for embedded system APIs with the help of test automation requires careful design and continuous development effort, as both the underlying technologies and the threat landscape are constantly evolving. Standardized security testing, supported by clear methodologies and proven tools and techniques, is essential for ensuring long-term reliability, maintainability, and resilience of embedded products. As systems grow in complexity and connectivity, the role of systematic and well-integrated security testing will only become more critical in the product development lifecycle.

REFERENCES

[1] F. M. A. Hassan, S. R. Das, and M. Hussain, “Importance of Secure Software Development for the Software Development at Different SDLC Phases,” vol. 2023.

[2] “OWASP top 10”, [Online]. Available: https://owasp.org/www-project-top-ten/

[3] W. Burleson *et al.*, “Grand Challenges for Embedded Security Research in a Connected World,” May 13, 2020, *arXiv*: arXiv:2005.06585. doi: 10.48550/arXiv.2005.06585.

[4] M. Bond and R. Anderson, “API-level attacks on embedded systems,” *Computer*, vol. 34, no. 10, pp. 67–75, Oct. 2001, doi: 10.1109/2.955101.

[5] Y. Wu *et al.*, “Your Firmware Has Arrived: A Study of Firmware Update Vulnerabilities,” vol. 2024.

[6] M. Wolf, *Computers as components: principles of embedded computing system design*, Third edition. Waltham, MA: Elsevier/Morgan Kaufmann, 2012.

[7] T. Noergaard, *Embedded systems architecture: a comprehensive guide for engineers and programmers*. in Embedded technology series. Amsterdam ; Boston: Elsevier/Newnes, 2005.

[8] I. Fette and A. Melnikov, “The WebSocket Protocol,” RFC Editor, RFC6455, Dec. 2011. doi: 10.17487/rfc6455.

[9] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” 2000.

[10] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” RFC Editor, RFC9000, May 2021. doi: 10.17487/RFC9000.

[11] T. Ylonen and C. Lonvick, “The Secure Shell (SSH) Protocol Architecture,” RFC Editor, RFC4251, Jan. 2006. doi: 10.17487/rfc4251.

[12] H. Dong, Y. Zhang, H. Lee, K. Du, G. Tu, and Y. Sun, “Mutual TLS in Practice: A Deep Dive into Certificate Configurations and Privacy Issues,” in *Proceedings of the 2024 ACM on Internet Measurement Conference*, Madrid Spain: ACM, Nov. 2024, pp. 214–229. doi: 10.1145/3646547.3688415.

[13] S. Jiang, Y. Zhang, J. Li, H. Yu, L. Luo, and G. Sun, “A Survey of Network Protocol Fuzzing: Model, Techniques and Directions,” Feb. 27, 2024, *arXiv*: arXiv:2402.17394. doi: 10.48550/arXiv.2402.17394.

[14] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating Fuzz Testing,” Oct. 18, 2018, *arXiv*: arXiv:1808.09700. doi: 10.48550/arXiv.1808.09700.

[15] S. R. Mamilla, “A Study of Penetration Testing Processes and Tools,” vol. 2021.

[16] A. Confido, E. V. Ntagiou, and M. Wallum, “Reinforcing Penetration Testing Using AI,” in *2022 IEEE Aerospace Conference (AERO)*, Big Sky, MT, USA: IEEE, Mar. 2022, pp. 1–15. doi: 10.1109/AERO53065.2022.9843459.

[17] D. Railkar, “A Study on Vulnerability Scanning Tools for Network Security,” vol. 2022.

[18] “Zynq UltraScale+ RFSoC Overview.” Xilinx, 2021. [Online]. Available: https://www.xilinx.com

[19] “Yocto Project Documentation.” 2023. [Online]. Available: https://www.yoctoproject.org

[20] K. Cherkaoui Dekkaki, I. Tasic, and M.-D. Cano, “Exploring Post-Quantum Cryptography: Review and Directions for the Transition Process,” *Technologies*, vol. 12, no. 12, p. 241, Nov. 2024, doi: 10.3390/technologies12120241.

[21] A. Regenscheid, “Transition to Post-Quantum Cryptography Standards,” National Institute of Standards and Technology, Gaithersburg, MD, NIST IR 8547 ipd, 2024. doi: 10.6028/NIST.IR.8547.ipd.

[22] G. Nookala, “The Role of SSL/TLS in Securing API Communications: Strategies for Effective Implementation,” 2024.

[23] Y. A. Joarder and C. Fung, “Exploring QUIC Security and Privacy: A Comprehensive Survey on QUIC Security and Privacy Vulnerabilities, Threats, Attacks, and Future Research Directions,” *IEEE Trans. Netw. Serv. Manage.*, vol. 21, no. 6, pp. 6953–6973, Dec. 2024, doi: 10.1109/TNSM.2024.3457858.

[24] J. Karlström, “The WebSocket Protocol and Security: Best Practices and Worst Weaknesses,” 2015.

[25] B. Reddy Bhimireddy, A. Nimmagadda, H. Kurapati, L. Reddy Gogula, R. Rani Chintala, and V. Chandra Jadala, “Web Security and Web Application Security: Attacks and Prevention,” in *2023 9th International Conference on Advanced Computing and Communication Systems (ICACCS)*, Coimbatore, India: IEEE, Mar. 2023, pp. 2095–2096. doi: 10.1109/ICACCS57279.2023.10112741.

[26] Aruna S and VIT Vellore, “Security in Web Services- Issues and Challenges,” *IJERT*, vol. V5, no. 09, p. IJERTV5IS090245, Sep. 2016, doi: 10.17577/IJERTV5IS090245.

[27] I. Zulkarneev and K. A. Basalay, “JSON Web Tokens Lifecycle-Based Threat Classification,” in *2024 IEEE 25th International Conference of Young Professionals in Electron Devices and Materials (EDM)*, Altai, Russian Federation: IEEE, Jun. 2024, pp. 1920–1924. doi: 10.1109/EDM61683.2024.10615042.

[28] D. X. Song, D. Wagner, and X. Tian, “Timing Analysis of Keystrokes and Timing Attacks on SSH£”.

[29] “Chapter 15. Nmap Reference Guide | Nmap Network Scanning.” Accessed: May 19, 2025. [Online]. Available: https://nmap.org/book/man.html

[30] J. M. Pittman, “A Comparative Analysis of Port Scanning Tool Efficacy,” Mar. 20, 2023, *arXiv*: arXiv:2303.11282. doi: 10.48550/arXiv.2303.11282.

[31] “Nessus Documentation.” Tenable. Accessed: Apr. 16, 2025. [Online]. Available: https://docs.tenable.com/Nessus.htm

[32] “Metasploit Framework Documentation.” Rapid7. [Online]. Available: https://docs.rapid7.com/metasploit/msf-overview

[33] “Exploring the Depths of Cybersecurity: A Comprehensive Guide to the Metasploit Framework,” *OSINT Team*. [Online]. Available: https://osintteam.blog/exploring-the-depths-of-cybersecurity-a-comprehensive-guide-to-the-metasploit-framework-d3b74ab0539

[34] “Wireshark User’s Guide.” Accessed: May 19, 2025. [Online]. Available: https://www.wireshark.org/docs/wsug_html/#ChIntroWhatIs

[35] “Scapy 2.6.1 documentation.” Accessed: May 19, 2025. [Online]. Available: https://scapy.readthedocs.io/en/latest/index.html#

[36] “OpenSSL Documentation.” Accessed: May 19, 2025. [Online]. Available: https://docs.openssl.org/master/

[37] “SSLyze 6.1.0 documentation.” Accessed: May 19, 2025. [Online]. Available: https://nabla-c0d3.github.io/sslyze/documentation/

[38] “What is CI/CD?” Accessed: May 19, 2025. [Online]. Available: https://about.gitlab.com/topics/ci-cd/

[39] “Security for Continuous Integration and Continuous Deployment Pipeline,” *IRJMETS*, Mar. 2024, doi: 10.56726/IRJMETS50676.

[40] T. Klooster, F. Turkmen, G. Broenink, R. T. Hove, and M. Böhme, “Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines,” in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, Melbourne, Australia: IEEE, May 2023, pp. 25–32. doi: 10.1109/SBFT59156.2023.00015.

[41] V. Shetty, “An Empirical Study on Robot Test Automation Framework,” vol. 2020.

[42] S. Stresnjak and Z. Hocenski, “Usage of Robot Framework in Automation of Functional Test Regression,” 2011.

[43] A. Bhargav, “RoboNmap Python library.” Accessed: May 19, 2025. [Online]. Available: https://github.com/abhaybhargav/RoboNmap

[44] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, “Fuzzing: State of the Art,” *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018, doi: 10.1109/TR.2018.2834476.

[45] S. Sorsa, “Protocol Fuzz Testing as a part of secure software development life cycle,” vol. 2018.

[46] M. A. B. Ashfaq, “Development of a Security Testing Process for Yocto Linux-Based Distributions:,” vol. 2023.

[47] A. Goutam and V. Tiwari, “Vulnerability Assessment and Penetration Testing to Enhance the Security of Web Application,” in *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, Mathura, India: IEEE, Nov. 2019, pp. 601–605. doi: 10.1109/ISCON47742.2019.9036175.

[48] S. Disawal and S. Ugrasen, “An Analysis and Classification of Vulnerabilities in Web-Based Application Developmen,” *8th International Conference on Computing for Sustainable Global Development (INDIACom)*, vol. 2021, pp. 782–785.

[49] C. Kemp, C. Calvert, T. M. Khoshgoftaar, and J. L. Leevy, “An approach to application-layer DoS detection,” *J Big Data*, vol. 10, no. 1, p. 22, Feb. 2023, doi: 10.1186/s40537-023-00699-3.

APPENDIX A: API FUZZING

```
import base64
from boofuzz.connections import SSLSocketConnection
import os
import struct

class WebSocketSSLConnection(SSLSocketConnection):
    """
    Extension for boofuzz SSLSocketConnection.
    Adds support for WebSocket connections.
    """
    def __init__(self, device_ip, device_port, ssl_context,
                 recv_timeout=1, send_timeout=1, ws_autoframe = True):
        super().__init__(device_ip, device_port,
                         recv_timeout=recv_timeout,
                         send_timeout=send_timeout,
                         sslcontext=ssl_context)
        self.ws_autoframe = ws_autoframe

    def open(self):
        """
        Opens SSLSocketConnection and builds WebSocket handshake
        on top of that
        """
        # Open underlying TLS/SSL connection.
        super().open()

        # Build the handshake request:
        ws_path = "/ws"  # Adjust as needed
        random_key = base64.b64encode(os.urandom(16)).decode('utf-8')
        handshake = (
            f"GET {ws_path} HTTP/1.1\r\n"
            f"Host: {self.host}:{self.port}\r\n"
            "Upgrade: websocket\r\n"
            "Connection: Upgrade\r\n"
            f"Sec-WebSocket-Key: {random_key}\r\n"
            "Sec-WebSocket-Version: 13\r\n\r\n"
        )

        # Send/receive handshake response
        self._sock.send(handshake.encode("utf-8"))
        response = self._sock.recv(1024)
        if b"101 Switching Protocols" not in response:
            raise Exception("WebSocket handshake failed: " +
                            response.decode(errors="ignore"))

    def send(self, data):
        """
        Wraps data into WebSocket-format and sends it
        with SSLSocketConnection's send-function
        """
        # Ensure data is bytes (if text, encode it)
        if isinstance(data, str):
            data = data.encode("utf-8")

        # Wrap data in a WebSocket frame.
        if self.ws_autoframe:
```

```
                data = create_ws_frame(data)

            return super().send(data)


    def create_ws_frame(message_bytes):
        """
        Helper to create a WebSocket text frame with masking (client-to-server)
        """
        frame = bytearray()
        # FIN=1 and opcode=0x1 for text frame.
        frame.append(0x81)
        length = len(message_bytes)
        if length < 126:
            frame.append(0x80 | length)  # 0x80 to set mask bit
        elif length < (2**16):
            frame.append(0x80 | 126)
            frame.extend(struct.pack("!H", length))
        else:
            frame.append(0x80 | 127)
            frame.extend(struct.pack("!Q", length))
        # Generate a random 4-byte mask
        mask = os.urandom(4)
        frame.extend(mask)
        # Mask the payload
        masked_payload = bytearray(b ^ mask[i % 4] for i,
                                   b in enumerate(message_bytes))
        frame.extend(masked_payload)

        return bytes(frame)
```

Program 14. *API fuzzing library's WebSocket SSL-connection*

```
    from api_fuzzer.logger.robot_logger import RobotFuzzLogger
    import asyncio
    from boofuzz import Session, s_get
    from importlib import resources
    import ssl

    class Fuzzer:
        """
        General fuzzer that creates fuzzing sessions with provided target
        Init parameters:
        device_ip(str): IP-address of the target device.
        device_port(int): Port where fuzzed service
                          is running in target device.
        ssl_context: Put nothing for standard certs,
                     or give python ssl-context for custom certs.
        logger: None for basic console logging,
                "robot" for Robot Framework logging,
                or own IFuzzLogger custom logger.
        """

        def __init__(self, device_ip, device_port,
                     ssl_context=None, logger=None):
            self.device_ip = device_ip
            self.device_port = device_port
```

```
26            # Initialize used logger
              if logger == None:
28                self.session_logger = None
              elif logger == "robot":
30                self.session_logger = [RobotFuzzLogger()]
              else:
32                self.session_logger = [logger]

34            # Get default certs if sslcontext is not provided
              if ssl_context == None:
36                cert_file      =      resources.files("api_fuzzer").join-
    path("certs", "default_cert.pem")
38                self.ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
                  self.ssl_context.check_hostname = False
40                self.ssl_context.load_cert_chain(certfile=cert_file)
                  self.ssl_context.load_verify_locations(cert_file)
42            else:
                  self.ssl_context = ssl_context
44
          def fuzz(self, target, fuzz_template_name, fuzz_depth_level = 1):
46            """
              Fuzz single node.
48            Parameters:
              target: boofuzz target device
50            fuzz_template_name: Name of boofuzz-node you want to fuzz.
              fuzz_depth_level: Fuzz complexity level.
52            """

54            self.session = Session(target=target,
                                     keep_web_open=False, # Closes web interface
56  after sessios
                                   post_test_case_callbacks=
58  [self.post_test_callback], # Call this function after every test
                                     fuzz_db_keep_only_n_pass_cases=10,
60                                   fuzz_loggers=self.session_logger,
                                     receive_data_after_fuzz=True
62                                   )
              self.session._fuzz_data_logger.log_info(f"Fuzzing session for
64  message template '{fuzz_template_name}' started.")
              # Initialize device service-status dict
66            asyncio.run(self.capture_device_status_baseline())
              # Connect using the given template node.
68            self.session.connect(s_get(fuzz_template_name))

70            # Run fuzzing
              self.session.fuzz(max_depth=fuzz_depth_level)
72
              self.session._fuzz_data_logger.log_info(f"Fuzzing session for
74  message template '{fuzz_template_name}' done.")

76            self.failed_cases        =        self.session._fuzz_data_log-
    ger.failed_test_cases
78            if self.failed_cases not in ({}, None):
                  # Return false if some test cases failed
80                return False
              return True
82
```

```
84          def post_test_callback(self, target, fuzz_data_logger, session,
    sock, **kwargs):
86              """
                Function that is called after every boofuzz test case.
88              Used to check device services status

90              Parameters:
                target (Target): Target with sock-like interface.
92              fuzz_data_logger (ifuzz_logger.IFuzzLogger):
                    Allows logging of test checks and passes/failures.
94                  Provided with a test case and test step already opened.
                session (Session): Session object calling post_send.
96                  Useful properties include last_send and last_recv.
                test_case_context (ProtocolSession):
98                  Context for test case-scoped data.
                """
100             device_status = asyncio.run(self.device_check())
                if device_status == True:
102                 fuzz_data_logger.log_pass("Device status check made suc-
    cessfully. Service status values are as expected.")
104             else:
                    error_string = (
106                     f"Device status check found degraded service status! "
                        )
108                 fuzz_data_logger.log_fail(error_string)
                    # End fuzzing if some test case fails
110                 self.session._index_end = self.session.total_mutant_index

112         async def get_device_status(self):
                """
114             Pseudo code example for getting device status,
                needs to be replaced with real device status getter function.
116             """
                device_connection      =      await      DeviceConnection.con-
118 nect(self.ssl_context, self.device_ip)
                status = await device_connection.status_query()
120             await device_connection.destroy()
                return status
122
            async def capture_device_status_baseline(self):
124             """
                Pseudo code example to store the device baseline status
126             before start of fuzzing.
                """
128             self.baseline_device_status = {}
                device_status = await self.get_device_status()
130             for service in device_status:
                    self.baseline_device_status[f"{service}"]      =      ser-
132 vice.get("Status")
                self.session._fuzz_data_logger.log_info(f"Baseline device sta-
134 tus captured!")

136         async def device_check(self):
                """
138             Pseudo code example to check device service statuses
                against status baseline
140             """
                device_status = await self.get_device_status()
142             for service in device_status:
```

```
                current_status = service.get("Status")
144             baseline_status = self.baseline_device_status.get(f"{ser-
    vice}")
146             # If the baseline status was NORMAL and now it's not,
                  mark as problematic.
148             if baseline_status == "NORMAL" and current_status != "NOR-
    MAL":
150                 self.session._fuzz_data_logger.log_info(f"Device sta-
    tus changed for {service}: Baseline={baseline_status}, Current={cur-
152 rent_status}")
                    return False
154
            # Return True if nothing has changed
156         return True
```

Program 15. *API fuzzing library's common fuzzer class*

```
    import asyncio
2   from boofuzz import s_byte, s_group, s_initialize, s_static, s_string

4   class WebSocketAPIFuzzTemplates:
        """
6       Class for all the websocket-API boofuzz templates.

8       Init parameters:
        device_ip(str): IP address of target device.
10      device_port(int): WebSocket API port
        ssl_context: Python sslcontext that holds your ssl configuration
12      """
        def __init__(self, device_ip, device_port, ssl_context):
14          self.device_ip = device_ip
            self.device_port = device_port
16          self.ssl_context = ssl_context

18          # Request supported message types from sensor
            msg = asyncio.run(self.get_supported_messages())
20          self.supported_message_types = msg["Parameters"]["Messages"]

22          # List of all fuzzing template names
            self.names = []
24
        async def get_supported_messages(self):
26          """
            Pseudo code example function to receive all supported message
28          types from websocket API of connected device.
            Needs to be implemented based on system under test.
30          """
            connection      =      await      WebSocketAPIConnection.con-
32  nect(self.ssl_context, self.device_ip)
            msg = await connection.get_supported_messages()
34          await connection.destroy()
            return msg
36
        def define_all_json_templates(self):
38          """
            Calls all template definition functions to register them.
40          """
            self.example_message_fuzz(),
```

```
42              self.example_message_additional_field_fuzz()

44          def ws_frame_fuzz(self):
                """
46              WebSocket frame fuzzing template
                When fuzzing WebSocket frame,
48              turn off automatic frame construction
                from websocket-api fuzzing class.
50              """
                s_initialize("ws_frame_fuzz")
52              # First byte: FIN and opcode, normally 0x81
                s_byte(0x81, name="first_byte")
54              # Second byte: mask bit and payload length;
                # default for a small payload (e.g., 10 bytes)
56              s_byte(0x80 | 10, name="length_byte")
                # Four mask bytes as fuzzable fields:
58              for i in range(4):
                    s_byte(0x00, name=f"mask_byte_{i}")
60              # Payload not fuzzed here
                s_static("payload", name="payload")
62
            def example_message_fuzz(self):
64              """
                Function that fuzzes example message value and message type.
66              """
                name = "example_message_fuzz"
68              self.names.append(name)

70              s_initialize(name)
                s_static("{\"")
72              s_string("MessageType", name="json_key")  # Fuzz the key
                s_static("\":\"")
74              s_string("ExampleValue", name="json_value")  # Fuzz the value.
                s_static("\"}")
76
            def example_message_additional_field_fuzz(self):
78              """
                Function that fuzzes additional key for all the supported mes-
80  sage types.
                """
82              name = "example_message_additional_field_fuzz"
                self.names.append(name)
84
                s_initialize(name)
86              s_static("{\"")
                s_static("MessageType")
88              s_static("\":\"")
                s_group(name="SupportedMessages",
90                      values=self.supported_message_types)
                s_static("\", ")
92              s_string("AdditionalKey", name="additional_key")
                s_static("\":\"")
94              s_string("AdditionalKeyValue", name="additional_key_value")
                s_static("\"}")
```

Program 16. *API fuzzing library's WebSocket fuzzing templates*

```
from api_fuzzer.rest_api.rest_api_fuzz_templates import RESTAPIFuzz-
Templates
from api_fuzzer.fuzzer import Fuzzer
from boofuzz import Target, SSLSocketConnection
import httpx

class RESTAPIFuzzer(Fuzzer):
    """
    Class to fuzz REST-API with boofuzz.

    Init parameters:
    device_ip(str): IP-address of the target device.
    device_port(int): Port where fuzzed service is running
                      in target device.
    ssl_context: python ssl-context or nothing for default certs
    logger: None for basic console logging, "robot" for Robot Framework
            logging, or own IFuzzLogger custom logger.
    use_jwt_token(bool): Bool the tell if you want authenticated
                         fuzzing or non-authenticated fuzzing.
    signin_endpoint(str): REST endpoint for sign-in and
                          JWT-token receiving.
    user(str): Sign-in username
    password(str): Sign-in password
    """
    def __init__(self, device_ip, device_port, ssl_context=None,
                 logger=None, use_jwt_token=True,
                 signin_endpoint=None, user=None, password=None):
        super().__init__(device_ip=device_ip,
                         device_port= device_port,
                         ssl_context=ssl_context, logger=logger)

        self.use_jwt_token = use_jwt_token
        self.signin_endpoint = signin_endpoint
        self.user = user
        self.password = password
        self.jwt_token = None

        # Get JWT-token from sign-in endpoint
        if use_jwt_token:
            self.get_jwt_token()

        # Initialize template-class
        self.templates     =     RESTAPIFuzzTemplates(self.device_ip,
self.jwt_token)

        # Init SSL/TLS connection and target
        connection   =   SSLSocketConnection(self.device_ip,   self.de-
vice_port, sslcontext=self.ssl_context, recv_timeout=0.5)
        self.target = Target(connection=connection)

    def get_jwt_token(self):
        """
        Function to receive JWT-token from sign-in endpoint.
        """

        if not (self.device_ip and self.device_port
                and self.signin_endpoint):
            print("No password, username, or signin-endpoint
                   provided to get jwt-token.")
```

```
60                return

62            url = "https://" + self.device_ip + ":" + str(self.device_port)
    + self.signin_endpoint
64            data = {"username": self.user, "password": self.password}
              try:
66                with httpx.Client(verify=self.ssl_context) as client:
                      response = client.post(url, json=data)
68                    if response.status_code == 200:
                          jwt_token = response.json().get("jwt")
70                        if jwt_token:
                              print("JWT-token received successfully!")
72                        self.jwt_token = jwt_token
                      else:
74                        print("Signin failed with status:",
                                response.status_code)
76            except Exception as e:
                  print("Error during signin:", e)
78
          def reset(self):
80            """
              Resets all the session specific variables to original status
82            after fuzz session.
              """
84            self.templates = RESTAPIFuzzTemplates(self.device_ip,
                                                    self.jwt_token)
86            self.session = None
              self.baseline_bit_status = None
88            self.failed_cases = None

90
```

Program 17. *API fuzzing library's REST API fuzzer*

```
      from boofuzz import IFuzzLogger
 2    from robot.api import logger

 4    class RobotFuzzLogger(IFuzzLogger):
          """
 6        Boofuzz custom logger implementation (IFuzzLogger)
          to get boofuzz logs into Robot
 8        """
          def close_test(self):
10            logger.info("Close test")

12        def close_test_case(self):
              logger.info("Close test case")
14
          def log_check(self, description):
16            logger.info(f"Check: {description}")

18        def log_error(self, description):
              logger.error(f"Error: {description}")
20
          def log_fail(self, description=''):
22            logger.info(f"Fail: {description}")

24        def log_info(self, description):
```

```
            logger.info(f"Info: {description}")

        def log_pass(self, description=''):
            logger.info(f"Pass: {description}")

        def log_recv(self, data):
            # Check if data is a WebSocket frame by verifying
            # that it's a byte string.
            if isinstance(data, bytes) and len(data) >= 2:
                # The first byte contains FIN and opcode.
                # For a final text frame, FIN is 1 and opcode is 0x1,
                # so the byte is typically 0x81.
                first_byte = data[0]
                if first_byte == 0x81:
                    second_byte = data[1]
                    # Most significant bit indicates masking
                    mask_flag = second_byte & 0x80
                    # Lower 7 bits is the payload length
                    payload_length = second_byte & 0x7F

                    index = 2
                    if payload_length == 126:
                        # Extended payload length: next 2 bytes.
                        payload_length   =   int.from_bytes(data[index:in-
dex+2], 'big')
                        index += 2
                    elif payload_length == 127:
                        # Extended payload length: next 8 bytes.
                        payload_length   =   int.from_bytes(data[index:in-
dex+8], 'big')
                        index += 8

                    if mask_flag:
                        # The next 4 bytes are the masking key.
                        mask = data[index:index+4]
                        index += 4

                        # The payload is masked; unmask it.
                        masked_payload = data[index:index+payload_length]
                        unmasked = bytearray(payload_length)
                        for i in range(payload_length):
                            unmasked[i] = masked_payload[i] ^ mask[i % 4]
                        try:
                            decoded = unmasked.decode('utf-8')
                        except UnicodeDecodeError:
                            decoded = repr(unmasked)
                    else:
                        # If not masked, simply extract the payload.
                        decoded   =   data[index:index+payload_length].de-
code('utf-8')

                    logger.info(f"Received WebSocket message: {decoded}")
                    return

            # If it doesn't match the expected WebSocket frame structure,
            # log as-is.
            # This also logs other received message types than WebSocket
            logger.info(f"Received: {data}")
```

```
84          def log_send(self, data):
                logger.info(f"Sent: {data}")
86
            def open_test_case(self, test_case_id, name, index,
88                             *args, **kwargs):
                logger.info(f"Open test case: ID={test_case_id},
90                          Name={name}, Index={index}")

92          def open_test_step(self, description):
                logger.info(f"Open test step: {description}")
```

Program 18. *API fuzzing library's boofuzz logger for Robot Framework.*

APPENDIX B: PORT SCANNING

```
import json
import nmap3
import os
from robot.api import logger

class NMapScanner:
    """Python support class for running nmap scans in Robot"""
    ROBOT_LIBRARY_SCOPE = 'GLOBAL'

    def __init__(self):
        pass

    @classmethod
    def initialize_port_scan(self, target):
        """
        Init function. Initializes class to run nmap scans.
        """
        self.nmap = nmap3.NmapScanTechniques()
        self.target = target

        # Load allowed ports from JSON
        script_dir = os.path.dirname(os.path.abspath(__file__))
        json_path = os.path.join(script_dir,
                                 'resources', 'allowed_ports.json')
        with open(json_path) as json_file:
            self.allowed_ports = json.load(json_file)

        return "Port scan initialized"

    # Scans
    def tcp_port_scan(self):
        """
        Runs nmap tcp scan and compares found open ports
        to given list of allowed open ports for tcp-protocol.
        """
        # Run tcp scan to initialized target
        result = self.nmap.nmap_tcp_scan(self.target)

        # Compare open ports to allowed open ports
        failed_ports = self.port_comparison(protocol="tcp",
                                            result=result)
        return failed_ports

    def udp_port_scan(self):
        """
        Runs nmap udp scan and compares found open ports
        to given list of allowed open ports for udp-protocol.
        """
        # Run udp scan to initialized target
        result = self.nmap.nmap_udp_scan(self.target)

        # Compare open ports to allowed open ports
        failed_ports = self.port_comparison(protocol="udp",
                                            result=result)

        return failed_ports
```

```
58
        # Support functions
60      def port_comparison(self, protocol, result):
            """
62          Gets open ports from scan and allowed ports one protocol.
            Returns list of open ports which are not allowed to be open.
64          (or 'error' if scan has failed)
            """
66
            if "error" in result:
68              raise Exception(f"{protocol} scan error: {result["msg"]}")

70          protocol = str(protocol).lower()
            # Get open ports
72          open_ports = []
            target_data = result.get(self.target, {})
74          ports = target_data.get("ports", [])
            for port_info in ports:
76              if  port_info.get("protocol").lower()  ==  protocol  and
    port_info.get("state").lower() == "open":
78                  open_ports.append(port_info.get("portid"))
            logger.info(f"Open  ports  found  for  protocol  {protocol}:
80  {open_ports}")

82          # Get allowed ports for given protocol
            if not self.allowed_ports[protocol]:
84              raise Exception(f"Protocol {protocol} not defined in al-
    lowed ports JSON!")
86          else:
                allowed_ports_for_protocol = self.allowed_ports[protocol]
88              logger.info(f"Allowed ports for protocol {protocol}: {al-
    lowed_ports_for_protocol}")
90
            # Check for if open ports contains non allowed open ports.
92          failed_ports = []
            for port in open_ports:
94              if int(port) not in allowed_ports_for_protocol:
                    failed_ports.append(f"{protocol} - {port}")
96          if failed_ports == []:
                logger.info(f"No unallowed open ports found for protocol
98  {protocol}")
            return failed_ports
```

Program 19. *Python class to implement Nmap port scanning keywords for Robot.*

APPENDIX C: MTLS TESTING

```
from datetime import datetime
import os
from pathlib import Path
from robot.api import logger
from sslyze.json.json_output import SslyzeOutputAsJson, ServerScanResultAsJson
from sslyze.plugins.scan_commands import ScanCommand
from sslyze.plugins.robot import _robot_tester
from sslyze.scanner.models import *
from sslyze.scanner.scanner import Scanner
from sslyze.scanner.scan_command_attempt import ScanCommandAttemptStatusEnum
from sslyze.server_setting import ServerNetworkLocation, ServerNetworkConfiguration, ClientAuthenticationCredentials

class SSLyzeAPIUser:
    """
    Python library to use SSLyze tool from Python API in
    Robot Framework test cases.
    This class contains Python functions,
    which also work as Robot Framework keywords,
    for all the supported SSLyze scans.
    """
    ROBOT_LIBRARY_SCOPE = 'GLOBAL'

    def __init__(self):
        pass

    @classmethod
    def initialize_scanner(self, ip_address, port):
        """
        Function that initializes all the needed configurations
        to run SSLyze scans.
        At the moment used certs are default ones inside cert-folder.

        Parameters:
        ip_address(str): Target device IP address
        port(int): Port to target service
        """
        port = int(port)
        self.server_location = ServerNetworkLocation(hostname="", port=port, ip_address=ip_address)

        current_file_path = os.path.abspath(__file__)
        current_folder = os.path.dirname(current_file_path)
        cert_chain_path = os.path.join(current_folder, "certs", "default_cert_and_ca.pem")
        key_path = os.path.join(current_folder, "certs", "default_cert_pk.pem")

        self.client_credentials = ClientAuthenticationCredentials(
            certificate_chain_path=Path(cert_chain_path),
            key_path=Path(key_path),
        )

        self.network_configuration = ServerNetworkConfiguration(
```

```
                tls_server_name_indication="server",
                tls_client_auth_credentials=self.client_credentials
            )

            return "SSLyze initializations are ready."

        """ SSLyze scan functions"""
        def ssl_2_scan_results(self):
            """
            Description: SSLyze scan for SSL 2.0.
            Checks if SSL 2.0 is supported and what cipher suites it accepts
            Returns: Python dictionary about scan results
            """
            scan_results = self.run_scan_commands([ScanCommand.SSL_2_0_CI-
PHER_SUITES])
            scan_result = scan_results[0]
            scan_attempt = scan_result.scan_result.ssl_2_0_cipher_suites
            scan_result  =  self.cipher_suites_scan_result_parse(scan_at-
tempt)
            return scan_result

        def ssl_3_scan_results(self):
            """
            Description: SSLyze scan for SSL 3.0.
            Checks if SSL 3.0 is supported and what cipher suites it accepts
            Returns: Python dictionary about scan results
            """
            scan_results = self.run_scan_commands([ScanCommand.SSL_3_0_CI-
PHER_SUITES])
            scan_result = scan_results[0]
            scan_attempt = scan_result.scan_result.ssl_3_0_cipher_suites
            scan_result  =  self.cipher_suites_scan_result_parse(scan_at-
tempt)
            return scan_result

        def tls_1_0_scan_results(self):
            """
            Description: SSLyze scan for TLS 1.0.
            Checks if TLS 1.0 is supported and what cipher suites it accepts
            Returns: Python dictionary about scan results
            """
            scan_results = self.run_scan_commands([ScanCommand.TLS_1_0_CI-
PHER_SUITES])
            scan_result = scan_results[0]
            scan_attempt = scan_result.scan_result.tls_1_0_cipher_suites
            scan_result  =  self.cipher_suites_scan_result_parse(scan_at-
tempt)
            return scan_result

        def tls_1_1_scan_results(self):
            """
            Description: SSLyze scan for TLS 1.1.
            Checks if TLS 1.1 is supported and what cipher suites it accepts
            Returns: Python dictionary about scan results
            """
            scan_results = self.run_scan_commands([ScanCommand.TLS_1_1_CI-
PHER_SUITES])
            scan_result = scan_results[0]
            scan_attempt = scan_result.scan_result.tls_1_1_cipher_suites
```

```
116             scan_result  =  self.cipher_suites_scan_result_parse(scan_at-
    tempt)
118             return scan_result

120         def tls_1_2_scan_results(self):
                """
122             Description: SSLyze scan for TLS 1.2.
                Checks if TLS 1.2 is supported and what cipher suites it accepts
124             Returns: Python dictionary about scan results
                """
126             scan_results = self.run_scan_commands([ScanCommand.TLS_1_2_CI-
    PHER_SUITES])
128             scan_result = scan_results[0]
                scan_attempt = scan_result.scan_result.tls_1_2_cipher_suites
130             scan_result  =  self.cipher_suites_scan_result_parse(scan_at-
    tempt)
132             return scan_result

134         def tls_1_3_scan_results(self):
                """
136             Description: SSLyze scan for TLS 1.3.
                Checks if TLS 1.3 is supported and what cipher suites it accepts
138             Returns: Python dictionary about scan results
                """
140             scan_results = self.run_scan_commands([ScanCommand.TLS_1_3_CI-
    PHER_SUITES])
142             scan_result = scan_results[0]
                scan_attempt = scan_result.scan_result.tls_1_3_cipher_suites
144             scan_result  =  self.cipher_suites_scan_result_parse(scan_at-
    tempt)
146             return scan_result

148         def supports_tls_compression(self):
                """
150             Description: SSLyze scan for TLS compression support
                Returns: Boolean value for if TLS compression is supported
152             """
                scan_results  =  self.run_scan_commands([ScanCommand.TLS_COM-
154 PRESSION])
                scan_result = scan_results[0]
156             scan_attempt = scan_result.scan_result.tls_compression
                if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
158                 logger.error(f"Scan  attempt  failed!  Reason:  {scan_at-
    tempt.error_reason}")
160                 return "Error"
                elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
162 PLETED:
                    return scan_attempt.result.supports_compression
164
        def supports_tls_1_3_early_data(self):
166             """
                Description: SSLyze scan for TLS 1.3 early data support
168             Returns: Boolean value for if TLS 1.3 early data is supported
                """
170             scan_results        =        self.run_scan_commands([ScanCom-
    mand.TLS_1_3_EARLY_DATA])
172             scan_result = scan_results[0]
                scan_attempt = scan_result.scan_result.tls_1_3_early_data
174             if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
```

```
                logger.error(f"Scan attempt failed! Reason: {scan_at-
176 tempt.error_reason}")
                return "Error"
178             elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
    PLETED:
180                 return scan_attempt.result.supports_early_data

182         def vulnerable_to_openssl_ccs_injection(self):
                """
184             Description: SSLyze scan for OpenSSL CCS injection
                Returns: Boolean value for if target is
186             vulnerable to OpenSSL CCS injection
                """
188             scan_results = self.run_scan_commands([ScanCom-
    mand.OPENSSL_CCS_INJECTION])
190             scan_result = scan_results[0]
                scan_attempt = scan_result.scan_result.openssl_ccs_injection
192             if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
                    logger.error(f"Scan attempt failed! Reason: {scan_at-
194 tempt.error_reason}")
                    return "Error"
196             elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
    PLETED:
198                 return scan_attempt.result.is_vulnerable_to_ccs_injection

200         def supports_tls_fallback_scsv(self):
                """
202             Description: SSLyze scan for TLS fallback SCSV
                Returns: Boolean value for if TLS fallback SCSV is supported
204             """
                scan_results = self.run_scan_commands([ScanCom-
206 mand.TLS_FALLBACK_SCSV])
                scan_result = scan_results[0]
208             scan_attempt = scan_result.scan_result.tls_fallback_scsv
                if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
210                 logger.error(f"Scan attempt failed! Reason: {scan_at-
    tempt.error_reason}")
212                 return "Error"
                elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
214 PLETED:
                    return scan_attempt.result.supports_fallback_scsv
216
        def vulnerable_to_heartbleed(self):
218             """
                Description: SSLyze scan for Heartbleed
220             Returns: Boolean value for if target
                is vulnerable to Heartbleed
222             """
                scan_results = self.run_scan_commands([ScanCom-
224 mand.HEARTBLEED])
                scan_result = scan_results[0]
226             scan_attempt = scan_result.scan_result.heartbleed
                if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
228                 logger.error(f"Scan attempt failed! Reason: {scan_at-
    tempt.error_reason}")
230                 return "Error"
                elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
232 PLETED:
                    return scan_attempt.result.is_vulnerable_to_heartbleed
```

```
234
        def vulnerable_to_robot(self):
236             """
                Description: SSLyze scan for ROBOT
238             Returns: Boolean value for if target is vulnerable to ROBOT
                """
240             scan_results = self.run_scan_commands([ScanCommand.ROBOT])
                scan_result = scan_results[0]
242             scan_attempt = scan_result.scan_result.robot
                if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
244                 logger.error(f"Scan  attempt  failed!  Reason:  {scan_at-
    tempt.error_reason}")
246                 return "Error"
                elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
248 PLETED:
                    robot_result = scan_attempt.result.robot_result
250                 if     robot_result     in     (_robot_tester.RobotScanRe-
    sultEnum.NOT_VULNERABLE_NO_ORACLE,
252                                             _robot_tester.RobotScanRe-
    sultEnum.NOT_VULNERABLE_RSA_NOT_SUPPORTED):
254                     return False
                    else:
256                     logger.info(f"ROBOT scan test failed: {robot_result}")
                        return True
258
        def session_renegotiation(self):
260             """
                Description: SSLyze scan for Session Renegotiation
262             Returns: Python dictionary that contains
                Session Renegotiation scan results
264             """
                scan_results = self.run_scan_commands([ScanCommand.SESSION_RE-
266 NEGOTIATION])
                scan_result = scan_results[0]
268             scan_attempt = scan_result.scan_result.session_renegotiation
                if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
270                 logger.error(f"Scan  attempt  failed!  Reason:  {scan_at-
    tempt.error_reason}")
272                 return "Error"
                elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
274 PLETED:
                    test_results = {}
276                 test_results["supports_secure_renegotiation"]  =  scan_at-
    tempt.result.supports_secure_renegotiation
278                 test_results["client_renegotiation_dos_vulnerability"]   =
    scan_attempt.result.is_vulnerable_to_client_renegotiation_dos
280                 test_results["client_renegotiations_success_count"]     =
    scan_attempt.result.client_renegotiations_success_count
282                 return test_results

284     def http_headers(self):
                """
286             Description: SSLyze scan for HTTP Headers
                Returns: Python dictionary that contains
288             HTTP Headers -scan results
                """
290             scan_results  =  self.run_scan_commands([ScanCommand.HTTP_HEAD-
    ERS])
292             scan_result = scan_results[0]
```

```
            scan_attempt = scan_result.scan_result.http_headers
294         if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
                logger.error(f"Scan attempt failed! Reason: {scan_at-
296 tempt.error_reason}")
                return "Error"
298         elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
    PLETED:
300             test_results = {}
                test_results["strict_transport_security_header"]       =
302 scan_attempt.result.strict_transport_security_header
                test_results["http_request_sent"]   =   scan_attempt.re-
304 sult.http_request_sent
                test_results["http_error_trace"]   =   scan_attempt.re-
306 sult.http_error_trace
                test_results["http_path_redirection"] = scan_attempt.re-
308 sult.http_path_redirected_to
                return test_results
310
        def elliptic_curves(self):
312         """
            Description: SSLyze scan for Elliptic Curves
314         Returns: Python dictionary that contains Elliptic Curves -scan
    results
316         """
            scan_results   =   self.run_scan_commands([ScanCommand.ELLIP-
318 TIC_CURVES])
            scan_result = scan_results[0]
320         scan_attempt = scan_result.scan_result.elliptic_curves
            if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
322             logger.error(f"Scan attempt failed! Reason: {scan_at-
    tempt.error_reason}")
324             return "Error"
            elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
326 PLETED:
                test_results = {}
328
                test_results["supports_ecdh_key_exchange"]   =   scan_at-
330 tempt.result.supports_ecdh_key_exchange

332             supported_curves = {}
                for    elliptic_curve    in    scan_attempt.result.sup-
334 ported_curves:
                    supported_curves[elliptic_curve.name]    =    ellip-
336 tic_curve.openssl_nid
                test_results["supported_curves"] = supported_curves
338
                rejected_curves = {}
340             for elliptic_curve in scan_attempt.result.rejected_curves:
                    rejected_curves[elliptic_curve.name]     =     ellip-
342 tic_curve.openssl_nid
                test_results["rejected_curves"] = rejected_curves
344             return test_results

346     def tls_extended_master_secret(self):
            """
348         Description: SSLyze scan for TLS extended master secret
            Returns: Boolean value for if TLS extended
350         master secret is supported
            """
```

```
352             scan_results   =   self.run_scan_commands([ScanCommand.TLS_EX-
    TENDED_MASTER_SECRET])
354             scan_result = scan_results[0]
                scan_attempt = scan_result.scan_result.tls_extended_master_se-
356 cret
                if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
358                 logger.error(f"Scan  attempt  failed!  Reason:  {scan_at-
    tempt.error_reason}")
360                 return "Error"
                elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
362 PLETED:
                    return scan_attempt.result.supports_ems_extension
364
        def session_resumption(self):
366             """
                Description: SSLyze scan for Session Resumption
368             Returns: Python dictionary that contains
                Session Resumption -scan results
370             """
                scan_results = self.run_scan_commands([ScanCommand.SESSION_RE-
372 SUMPTION])
                scan_result = scan_results[0]
374             scan_attempt = scan_result.scan_result.session_resumption
                if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
376                 logger.error(f"Scan  attempt  failed!  Reason:  {scan_at-
    tempt.error_reason}")
378                 return "Error"
                elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
380 PLETED:
                    test_results = {}
382                 session_id_results = {}
                    session_id_results["resumption_result"] = scan_attempt.re-
384 sult.session_id_resumption_result
                    session_id_results["attempted_resumptions_count"]       =
386 scan_attempt.result.session_id_attempted_resumptions_count
                    session_id_results["successful_resumptions_count"]      =
388 scan_attempt.result.session_id_successful_resumptions_count
                    test_results["session_id_resumption"] = session_id_results
390
                    tls_ticket_results = {}
392                 tls_ticket_results["resumption_result"] = scan_attempt.re-
    sult.tls_ticket_resumption_result
394                 tls_ticket_results["attempted_resumptions_count"]       =
    scan_attempt.result.tls_ticket_attempted_resumptions_count
396                 tls_ticket_results["successful_resumptions_count"]      =
    scan_attempt.result.tls_ticket_successful_resumptions_count
398                 test_results["tls_ticket_resumption"] = tls_ticket_results
                    return test_results
400
        def certificate_info(self):
402             """
                Description: SSLyze scan for Certificate Info.
404                          Uses SSLyze's own JSON formatter instead
                             of python dict because of complex data structure.
406             Returns: JSON formatted string that contains
                Certificate Info -scan results
408             """
                start_time = datetime.now()
410
```

```
            scan_results = self.run_scan_commands([ScanCommand.CERTIFI-
412 CATE_INFO])
            json_output = SslyzeOutputAsJson(
414             server_scan_results=[ServerScanResultAsJson.model_vali-
    date(result) for result in scan_results],
416             invalid_server_strings=[],
                date_scans_started=start_time,
418             date_scans_completed=datetime.now()
            )
420         return json_output.model_dump_json()

422
        """Support functions"""
424     def run_scan_commands(self, scan_commands):
            """
426         Helper function to run given scan commands.
            Parameters: scan_commands(list): List of SSLyze's ScanCommand-
428 objects.
            Returns: List of ServerScanResult-objects
430         """
            scan_request = ServerScanRequest(
432             server_location=self.server_location,
                network_configuration=self.network_configuration,
434             scan_commands=scan_commands
            )
436         scanner = Scanner()
            scanner.queue_scans([scan_request])
438         scan_results = list(scanner.get_results())
            return scan_results
440
    def cipher_suites_scan_result_parse(self, scan_attempt: Cipher-
442 SuitesScanAttempt):
            """
444         Helper function to parse cipher suite scans for
            different TLS versions.
446         Parameters: scan_attempt(CipherSuitesScanAttempt):
            Cipher suite scan attempt.
448         Returns: Python dictionary that contains parsed scan results.
            """
450         scan_test_results = {}
            if scan_attempt.status == ScanCommandAttemptStatusEnum.ERROR:
452             logger.error(f"Scan attempt failed! Reason: {scan_at-
    tempt.error_reason}")
454         elif scan_attempt.status == ScanCommandAttemptStatusEnum.COM-
    PLETED:
456             scan_result = scan_attempt.result
                assert scan_result
458
                scan_test_results["tls_version"] = scan_result.tls_ver-
460 sion_used.name
                scan_test_results["tls_version_support"] = scan_re-
462 sult.is_tls_version_supported

464             accepted_cipher_suites = {}
                for cipher_suite in scan_result.accepted_cipher_suites:
466                 cipher_suite_data = {}
                    cipher_suite_data["key_size"] = cipher_suite.ci-
468 pher_suite.key_size
```

```
470                     cipher_suite_data["is_anonymous"]  =  cipher_suite.ci-
    pher_suite.is_anonymous
472                     accepted_cipher_suites[cipher_suite.ci-
    pher_suite.openssl_name] = cipher_suite_data
474                 scan_test_results["accepted_cipher_suites"] = accepted_ci-
    pher_suites
476
                    rejected_cipher_suites = {}
478                 for cipher_suite in scan_result.rejected_cipher_suites:
                        cipher_suite_data = {}
480                     cipher_suite_data["key_size"]    =    cipher_suite.ci-
    pher_suite.key_size
482                     cipher_suite_data["is_anonymous"]  =  cipher_suite.ci-
    pher_suite.is_anonymous
484                     rejected_cipher_suites[cipher_suite.ci-
    pher_suite.openssl_name] = cipher_suite_data
486                 scan_test_results["rejected_cipher_suites"] = rejected_ci-
    pher_suites
488
            return scan_test_results
```

Program 20. *Python class to implement SSLyze scan keywords for Robot.*

APPENDIX D: QUIC TESTER PROGRAM

```
import asyncio
from aioquic.asyncio import client
from aioquic.asyncio.protocol import QuicConnectionProtocol
from aioquic.quic.configuration import QuicConfiguration
from aioquic.quic.events import QuicEvent, HandshakeCompleted,
StreamDataReceived
from importlib import resources
import ssl

class QuicClient(QuicConnectionProtocol):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def quic_event_received(self, event: QuicEvent) -> None:
        if isinstance(event, HandshakeCompleted):
            print("Handshake completed!")
        elif isinstance(event, StreamDataReceived):
            print(event.data)

class QuicTester:
    def __init__(self, address, port, cert_file=None, alpn=None):
        self.address = address
        self.port = port
        alpn_protocols = alpn or ["real-alpn-field"]
        self.conf = QuicConfiguration(
            is_client=True,
            idle_timeout=1,
            alpn_protocols=alpn_protocols,
            verify_mode=ssl.CERT_REQUIRED
        )

        if not cert_file:
            self.cert_file = resources.files("penetra-
tion_tester").joinpath("certs", "real_cert.pem")
        elif cert_file == "fake":
            self.cert_file = resources.files("penetra-
tion_tester").joinpath("certs", "untrusted_cert.pem")
        else:
            self.cert_file = cert_file

        self.conf.load_cert_chain(self.cert_file)
        self.conf.load_verify_locations(self.cert_file)

    async def connect_once(self, hold_time=0):
        try:
            async with client.connect(
                self.address, self.port,
                configuration=self.conf,
                create_protocol=QuicClient
            ):
                print(f"Successfully connected to device with address
{self.address}")
                if hold_time:
                    await asyncio.sleep(hold_time)
                return True
        except Exception as e:
```

```
                print(f"Failed to connect to device at {self.address}:
58  {e}")
                return False
60
        async def flood_connections(self, concurrency, duration,
62  hold_time=0):
            semaphore = asyncio.Semaphore(concurrency)
64          success_connections = 0
            total_connections = 0
66          end_time = asyncio.get_event_loop().time() + duration

68          async def limited_connect():
                nonlocal success_connections, total_connections
70              async with semaphore:
                    if await self.connect_once(hold_time):
72                      success_connections += 1
                    total_connections += 1
74
            async def flood_loop():
76              while asyncio.get_event_loop().time() < end_time:
                    await limited_connect()
78
            # Spawn concurrency number of loops
80          tasks = [asyncio.create_task(flood_loop()) for _ in range(con-
    currency)]
82          await asyncio.gather(*tasks)

84          print(f"Flood test completed: {success_connections}/{to-
    tal_connections} successful")
```

Program 21. *Python program for QUIC testing.*

APPENDIX E: WEBSOCKET TESTER PROGRAM

```
from importlib import resources
import ssl
import threading
import time
import websocket

class WebSocketTester:
    """
    Class to test different WebSocket message variations
    as part of security testing
    """
    def __init__(self, address:str, certificate=None):
        """
        address: Target ip-address and port in websocket format
        certificate: premade certificate name or path to certificate
        """

        self.address = address
        self.certificate_name = certificate

        if self.certificate_name == "default":
            self.cert_file = resources.files("penetration_tester").joinpath("certs", "default_cert.pem")
        elif self.certificate_name == "fake":
            self.cert_file = resources.files("penetration_tester").joinpath("certs", "untrusted_cert.pem")
        elif self.certificate_name == "zap":
            self.cert_file = resources.files("penetration_tester").joinpath("certs", "zap_root_ca.pem")
            self.ssl_context = ssl.create_default_context(cafile=self.cert_file)
        else:
            self.cert_file = certificate
            self.ssl_context = ssl.create_default_context(cafile=self.cert_file)

        if self.certificate_name in ("fake", "default"):
            # Create an SSL context for mTLS.
            self.ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
            self.ssl_context.load_verify_locations(cafile=self.cert_file)
            self.ssl_context.load_cert_chain(certfile=self.cert_file, keyfile=self.cert_file)
            self.ssl_context.check_hostname = False

        self.sslopt = {"context": self.ssl_context}


    def single_connection(self, alive_time=0.1):
        """
        Creates single WebSocket connection and keeps it alive for given time (seconds).
        """
        try:
            ws = self.connection_creation()
```

```
                print("Connection successful!")
                time.sleep(alive_time)
                ws.close()
            except Exception as e:
                print(f"Connection failed: {e}")

    def flood_connections(self, duration, concurrent_connections,
connection_alive_time):
        """
        Creates multiple connections concurrently and keeps them
alive for a short period.
        """
        def duration_loop():
            end = time.time() + duration
            while time.time() < end:
                self.single_connection(alive_time=connec-
tion_alive_time)

        threads = [
            threading.Thread(target=duration_loop)
            for _ in range(concurrent_connections)
        ]

        for t in threads:
            t.start()
        for t in threads:
            t.join()

    def send_message(self, message):
        """
        Function to send single WebSocket message
        and receive response
        """
        try:
            ws = self.connection_creation()
            ws.send(message)
            print(ws.recv())
            ws.close()
        except Exception as e:
            print(f"Message send failed: {e}")

    def dos_client(self, client_id, message, duration, sleep=0):
        """
        DoS client that sends a high volume of messages.
        """
        try:
            ws = self.connection_creation()
            count = 0
            end_time = time.time() + duration
            while time.time() < end_time:
                try:
                    ws.send(message)
                    #response = ws.recv() # If user wants to print
responses
                    print(f"[Client {client_id}] Response {count}
successful.")
                except websocket.WebSocketTimeoutException:
                    print(f"[Client {client_id}] Timeout for message
{count}")
```

```
                    except Exception as e:
                        print(f"[Client {client_id}] Send/recv error on
msg {count}: {e}")
                        break
                    count += 1
                    if sleep != 0:
                        time.sleep(sleep)
                ws.close()
            except Exception as e:
                print(f"[Client {client_id}] Connection failed: {e}")

        def dos_with_multiple_clients(self, num_clients, message, dura-
tion, sleep=0):
            """
            Launches multiple DoS clients in parallel threads.
            """
            threads = []
            for client_id in range(num_clients):
                thread = threading.Thread(
                    target=self.dos_client,
                    args=(client_id, message, duration, sleep)
                )
                threads.append(thread)
                thread.start()

            for t in threads:
                t.join()
            print("DoS test completed.")

        def connection_creation(self):
            """
            Helper function to create right WebSocket connection
            """
            ws = None
            if self.certificate_name == "zap":
                ws = websocket.create_connection(
                    self.address,
                    sslopt=self.sslopt,
                    http_proxy_host="127.0.0.1",
                    http_proxy_port=8080
                )
            else:
                ws = websocket.create_connection(
                    self.address,
                    sslopt=self.sslopt,
                )
            return ws
```

Program 22. *WebSocket API testing program.*

APPENDIX F: REST API TESTER PROGRAM

```
import httpx
from importlib import resources
import threading
import time
import ssl

class RestApiTester:
    """
    Class to test different REST API request variations as part of security testing
    """
    def __init__(self, base_url:str, certificate=None, use_jwt_token=True, signin_endpoint=None, username=None, password=None):
        """
        base_url: Target ip-address and port
                  ("https://ip-address:port")
        certificate: premade certificate name or path to certificate
        use_jwt_token(Bool): Boolean value to tell if JWT token is
                             requested and used.
        username(str): Username for sign in
        password(str): Password for sign in
        """

        self.base_url = base_url
        self.certificate_name = certificate
        self.signin_endpoint = signin_endpoint
        self.username = username
        self.password = password

        if self.certificate_name == "default":
            self.cert_file = resources.files("penetration_tester").joinpath("certs", "default_cert.pem")
        elif self.certificate_name == "fake":
            self.cert_file = resources.files("penetration_tester").joinpath("certs", "untrusted_cert.pem")
        elif self.certificate_name == "zap":
            self.cert_file = resources.files("penetration_tester").joinpath("certs", "default_cert.pem")
            self.cert_file_zap = resources.files("penetration_tester").joinpath("certs", "zap_root_ca.pem")
            self.ssl_context_zap = ssl.create_default_context(cafile=self.cert_file_zap)
        else:
            self.cert_file = certificate
            self.ssl_context = ssl.create_default_context(cafile=self.cert_file)

        # SSL/TLS settings
        if self.certificate_name in ("fake", "default", "zap"):
            # Create an SSL context for mTLS.
            self.ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
            self.ssl_context.load_verify_locations(cafile=self.cert_file)
            self.ssl_context.load_cert_chain(certfile=self.cert_file, keyfile=self.cert_file)
```

```
                self.ssl_context.check_hostname = False

            self.jwt_token = None
            if use_jwt_token:
                self.get_jwt_token()

        def single_request(self, endpoint="/", method="GET", data=None,
headers=None):
            """
            Sends a single HTTP request.
            """
            try:
                url = self.base_url + endpoint
                proxy = None
                if self.certificate_name == "zap":
                    verify = self.ssl_context_zap
                    # Replace proxy value with one configured in ZAP
                    proxy="http://127.0.0.1:8080"
                else:
                    verify = self.ssl_context

                with httpx.Client(verify=verify, proxy=proxy, timeout=1)
as client:
                    response = client.request(method=method, url=url,
data=data, headers=headers)
                    print(f"\n=== Response from {url} ===")
                    print(f"Status Code: {response.status_code}")
                    print(f"Headers:\n{response.headers}")
                    print(f"Body:\n{response.text}")
                    print("============================\n")
                    return response
            except Exception as e:
                print(f"Request failed: {e}")

        def dos_client(self, client_id, endpoint, method, duration,
data=None, headers=None, sleep=0):
            """
            Client that sends a high volume of requests.
            """
            try:
                url = self.base_url + endpoint
                with httpx.Client(verify=self.ssl_context, prox-
ies=self.proxies, timeout=5) as client:
                    count = 1
                    end_time = time.time() + duration
                    while time.time() < end_time:
                        try:
                            response = client.request(method=method,
url=url, data=data, headers=headers)
                            print(f"[Client {client_id}] Request {count}
got {response.status_code}")
                        except httpx.RequestError as e:
                            print(f"[Client {client_id}] Request {count}
failed: {e}")
                            break
                        count += 1
                        if sleep != 0:
                            time.sleep(sleep)
            except Exception as e:
```

```
116                 print(f"[Client {client_id}] Setup failed: {e}")

118         def dos_with_multiple_clients(self, num_clients, endpoint,
    method="GET", duration=10, data=None, headers=None, sleep=0):
120             """
                Launches multiple DoS clients in parallel threads.
122             """
                threads = []
124             for client_id in range(num_clients):
                    thread = threading.Thread(
126                     target=self.dos_client,
                        args=(client_id, endpoint, method, duration, data,
128 headers, sleep)
                    )
130                 threads.append(thread)
                    thread.start()
132
                for t in threads:
134                 t.join()
                print("DoS test completed.")
136
        def get_jwt_token(self):
138             """
                Example function to receive JWT-token from sign-in endpoint.
140             """

142             if not (self.base_url and self.signin_endpoint):
                    print("No base-url or signin-endpoint provided to get
144 jwt-token.")
                    return
146
                url = self.base_url + self.signin_endpoint
148             data = {"username": self.username, "password": self.password}
                try:
150                 with httpx.Client(verify=self.ssl_context) as client:
                        response = client.post(url, json=data)
152                     if response.status_code == 200:
                            jwt_token = response.json().get("jwt")
154                         if jwt_token:
                                print("JWT-token received successfully!")
156                         self.jwt_token = jwt_token
                        else:
158                         print("Signin failed with status:", response.sta-
    tus_code)
160             except Exception as e:
                    print("Error during signin:", e)
```

Program 23. *REST API testing program.*

```
import os
import time
from zapv2 import ZAPv2

class ZAPScanner:
    """
    Tool to run ZAP security scans against given device and API.
    Init parameters:
    zap_proxy: Address where ZAP proxy application is running
    api_key: Secret API key to access ZAP API (Found from ZAP app GUI "Tools/Options/API")
    target_address: Address of device and API to scan ("https://<device-ip>:<port>")
    target_name: Name of API we are testing. Used as identifier for generated scan data.
    """
    def __init__(self, zap_proxy, api_key, target_address, target_name):
        """
        Initializes the ZAPScanner with required configuration
        """
        self.zap_proxy = zap_proxy
        self.api_key = api_key
        self.target_address = target_address
        self.target_name = target_name
        self.zap = ZAPv2(apikey=self.api_key, proxies={'http': self.zap_proxy, 'https': self.zap_proxy})

        # Only needed if you want to authenticate
        self.context_name = "MyContext"
        self.context_id = None
        self.user_id = None

    def configure_context_and_auth(self):
        """
        !EXAMPLE, CONFIGURE TO MATCH YOUR OWN AUTHENTICATION MECHANISM!

        Sets up a ZAP context for the target and configures JSON-based authentication
        matching the POST /api/signin request with JSON body you described.
        """
        # 1) Create a new context
        self.context_id = self.zap.context.new_context(self.context_name)

        # 2) Include all URLs under target into that context
        #    Note: the pattern below means "anything starting with self.target_address"
        self.zap.context.include_in_context(self.context_name, self.target_address + ".*")

        print(f"Created context '{self.context_name}' with ID: {self.context_id}")

        # 3) Configure JSON-based authentication
        #    We'll replicate the POST /api/signin with JSON body {username, password}.
```

```
60            #    (ZAP requires placeholders: {%username%} and {%pass-
    word%}.)
62            login_url = f"{self.target_address}/api/signin"

64            # The JSON with placeholders:
              #   {"username":"{%username%}","password":"{%password%}"}
66            # We also set the method to POST and specify requestHead-
    ers=Content-Type: application/json
68            login_request_body = '{"username":"{%username%}","pass-
    word":"{%password%}"}'
70            auth_method = 'jsonBasedAuthentication'
              auth_config = (
72                f"loginUrl={login_url}"
                  f"&loginRequestBody={login_request_body}"
74                "&method=POST"
                  "&requestHeaders=Content-Type: application/json"
76            )

78            # Call ZAP to set up JSON-based authentication on the context
              self.zap.authentication.set_authentication_method(self.con-
80  text_id, auth_method, auth_config)

82            print("Configured JSON-based authentication with login URL:",
    login_url)
84
              # 4) Create a user and set credentials
86            self.user_id = self.zap.users.new_user(self.context_name,
    "User")
88            # Even though it's JSON-based, ZAP expects form-like creden-
    tials in the format username=...&password=...
90            user_credentials = "username=myusername&password=mypassword"
              self.zap.users.set_authentication_credentials(self.con-
92  text_name, self.user_id, user_credentials)

94            # Enable the user in the context
              self.zap.users.set_user_enabled(self.context_id,
96  self.user_id, True)
              print(f"Created and enabled user with ID: {self.user_id}")
98
        def prepare_target(self):
100           """
              Prepares the target by opening the URL, which allows ZAP to
102 add it to its sites tree.
              """
104           print("Accessing target:", self.target_address)
              self.zap.urlopen(self.target_address)
106           time.sleep(2)  # Allow time for ZAP to update its sites tree
              # Optionally run the spider scan to discover endpoints
108           spider_scan_id = self.zap.spider.scan(self.target_address)
              while int(self.zap.spider.status(spider_scan_id)) < 100:
110               print("Spider progress: {}%".format(self.zap.spider.sta-
    tus(spider_scan_id)))
112               time.sleep(1)
              time.sleep(2)
114           print("Target preparation complete.")

116       def start_active_scan(self):
              """
118
```

```
            Initiates the active scan as the authenticated user (rather
120 than an unauthenticated scan).
            This ensures ZAP logs in via the JSON-based auth first.
122         """
            if not self.context_id or not self.user_id:
124             print("Starting ACTIVE SCAN as the unauthenticated
    user...")
126             scan_id = self.zap.ascan.scan(self.target_address, re-
    curse=True)
128         else:
                print("Starting ACTIVE SCAN as the authenticated
130 user...")
                scan_id = self.zap.ascan.scan_as_user(
132             contextid=self.context_id,
                userid=self.user_id,
134             url=self.target_address,
                recurse=True
136         )

138         print("Active scan started with Scan ID:", scan_id)

140         # Monitor progress
            while int(self.zap.ascan.status(scan_id)) < 100:
142             progress = self.zap.ascan.status(scan_id)
                print(f"Scan progress: {progress}%")
144             time.sleep(1)
            print(f"Active scan on {self.target_address} (as user) com-
146 pleted.")

148     def generate_zap_report(self):
            """
150         Generates an HTML report from the scan and saves it to the
    current working directory.
152         """
            report_html = self.zap.core.htmlreport()
154         report_file = os.path.join(os.getcwd(), "zap_report_" +
    self.target_name + ".html")
156         with open(report_file, 'w', encoding='utf-8') as f:
                f.write(report_html)
158         print("Report generated at:", report_file)

160     def scan(self):
            """
162         High-level method to run all steps:
              1) Configure context & JSON-based authentication (If you
164 want to authenticate,
                  needs modification to specified authentication case.
166 This is just example.)
              2) Prepare the target (spider, etc.)
168           3) Start the active scan as user
              4) Generate report
170         """
            # self.configure_context_and_auth()
172         self.prepare_target()
            self.start_active_scan()
174         self.generate_zap_report()
```

Program 24. *ZAP scan program with ZAP API.*