UCLouvain

École polytechnique de Louvain

Extending a dynamic IoT firewall tool

Integration of TLS and MQTT Protocols for Smart Home Security

Author: **Mehdi Laurent**
Supervisor: **Ramin Sadre**
Readers: **Ramin Sadre, François De Keersmaeker, Jean-Michel Dricot**
Academic year 2024–2025
Master [120] in Cybersecurity

Acknowledgments

First and foremost, I would like to thank François De Keersmaeker, for his guidance and advice over the year, and for all the constructive remarks he provided throughout my work on this thesis. I would also like to thank my supervisor, Ramin Sadre. And I would like to thank Jean-Michel Dricot for the interest he took in my thesis and for being part of the jury.

Next I want to thank and say my sincere gratitude to my good friends Bruno Sylin, Romain Grimau, Vincenzo Cubeddu, Nicolas Jonas, and Nicolas Bruyere for the quality of their company and the support they gave me in carrying out this task, whether by working alongside me in the library or by allowing me to discuss my work with them at length.

After that I would also like to thank my psychologist Caroline Valentiny for the support she gave me at times when the task of this work seemed insurmountable, and helped me find the motivation to carry on.

Finally, I would like to thank my family, that have always supported me and given me the opportunity to get to this point.

Statement about generative AI

I should also mention the AI assistant tools I used to improve the quality of the writing in my manuscript and also to help me in the process of writing the code needed for my implementation.

In this respect, I should first mention OpenAI's very famous LLM, ChatGPT. When I was feeling insecure about any of the paragraphs I was writing, I'd ask it if it had any suggestions for improving the grammar and making the text clearer and smoother to read. More often than not, I found the text proposed by ChatGPT to be much better than the original, so my manuscript

often benefited from this tool.

As far as writing code is concerned, I've certainly tried using ChatGPT and Copilot, but this time with far less convincing results. On the other hand, at the end of my work, I benefited from the use of the Claude AI assistant by Anthropic, which greatly helped me to make faster progress in the code-writing process. I think it's worth pointing out that the results produced by this LLM exceeded my expectations.

Mention is made in relevant places in my manuscript when I have used these specific tools.

Contents

Chapter 1

Introduction

In recent years, the Internet of Things (IoT) has revolutionized the way we interact with our environment, bringing an unprecedented level of convenience and connectivity to everyday life. Among the most promising domains for the use of IoT technology is the concept of Smart Homes, in which everyday household objects now interact with the Internet or among themselves to provide services to their users. From lamps and washing machines to security cameras and voice detection systems, these devices offer a seamless and intelligent way to manage our homes.

However, the potential widespread integration of IoT devices into our daily routines comes with significant security risks. As these smart devices collect and transmit vast amounts of personal data, they become attractive targets for cybercriminals. In recent years, solutions were proposed by the research community to help us protect against threats to the IoT world, such as the IETF's MUD standard [1] that aims to simplify and automate the secure deployment of end devices in the networks. Though an improvement to the overall situation, it lacked some features of interest for which a new IoT firewall system has recently been proposed [2] to address these shortcomings and enhance the overall security of IoT ecosystems.

The purpose of this thesis is to extend this new firewall tool by adding support for protocols previously missing from it, such as the prevalent cryptographic protocol TLS as well as the lightweight IoT messaging protocol MQTT.

1.1 Overview

Our work is divided into chapters, which we outline below.

Chapter 2 : Background This chapter introduces the context of our work, focusing on IoT and Smart Home environments. It examines their unique characteristics, associated risks, and existing security solutions. In particular, we discuss the MUD standard, which inspired the development of the firewall tool that we extend in this work. We also provide an introduction to NFTables, which the firewall leverages, as well as the protocols—TLS and MQTT—that we incorporate into its implementation.

Chapter 3 : Design and implementation This chapter provides an overview of the existing firewall architecture and details how TLS and MQTT protocols were incorporated into it. The first section establishes the foundation by describing the system's design principles, components, and code structure, focusing on elements most relevant to protocol extensions. The subsequent sections explain the implementation of TLS support—addressing version identification complexities and encrypted messages handling—and MQTT integration, for which custom traffic simulations were created. Both protocol implementations required developing specialized parsing mechanisms, with MQTT additionally featuring payload validation through regular expressions. Throughout, the focus remains on technical decisions that enhance the firewall's capability to monitor and control IoT device communications.

Chapter 4 : Results This chapter details the evaluation of the extended firewall implementation. It begins by describing the experimental setup using a Vagrant virtual machine to simulate router operations. The methodology section explains the testing process, which involves generating NFTables scripts, compiling NFQueue code, and using Tcpreplay to assess functionality and efficiency. Results demonstrate that the firewall successfully handles both MQTT and TLS protocols—accepting legitimate packets while correctly dropping malicious ones. Performance analysis shows minimal processing overhead for both protocols, with some unexpected variations attributed to system-level factors rather than the parsing mechanisms themselves.

Chapter 2

Background

This chapter provides the necessary context for our work, covering IoT and Smart Home security challenges. We discuss the MUD standard, which inspired our firewall extension, along with NFTables, TLS, and MQTT—the key technologies used in our implementation.

2.1 Internet of Things (IoT)

Although there is no real consensus on its definition, it's generally agreed that the Internet of Things (IoT) refers to the integration of sensors and network interfaces into common household devices and/or industrial equipments [3]. This enables them to collect data, communicate, and be controlled remotely. The sensors embedded in these devices measure various parameters related to their environment or their operational status, such as temperature, motion, or energy consumption. Through a network interface, the data gathered by these devices can be transmitted to other systems for analysis or used to enable remote monitoring and control. This technological framework allows for the integration of physical objects into digital networks, fostering advanced automation and data-driven functionality.

The IoT finds applications across diverse domains, enhancing efficiency, convenience, and connectivity. In smart homes [4], IoT devices like thermostats, lighting systems, and security cameras enable automated control and remote access through mobile apps. In industrial settings [5], IoT solutions monitor machinery performance and optimize operations, forming the backbone of predictive maintenance. Smart grids [6] leverage IoT to manage energy distribution dynamically, reducing waste and improving reliability. From connected vehicles to healthcare wearables, IoT integrates

seamlessly into everyday life and industry, driving innovation and fostering interconnected ecosystems.

Despite its benefits, the IoT also raises ecological concerns. The production and disposal of IoT devices contribute to electronic waste, while the energy demands of constant connectivity and data processing can increase carbon footprints. These challenges underscore the need for sustainable design and practices in IoT development. Such principles have been discussed by Nižetić *et al.* [7] among others.

However many also argue that IoT technologies can actually play a significant role in combating climate change by enabling sustainability improvements [3]. Through automation, real-time monitoring, and data-driven insights, IoT systems can optimize energy usage, reduce waste, and enhance resource efficiency in various sectors, and consequently promote sustainability.

IoT statistics

In any case, the IoT is expected to play an increasingly crucial role in the future of our societies.

According to IoT Analytics, the number of connected devices was expected to grow to nearly 19 billion by the end of 2024 [8]. This expansion is transforming industries such as retail, healthcare, manufacturing, and smart homes by enabling data exchange and device connectivity. IoT is enhancing efficiency, improving decision-making, and providing real-time insights to better customer experiences. Additionally, Statista reports that the global IoT market is expected to surpass $900 billion by 2033, further solidifying its growing impact on automation, sustainability, and global connectivity [9].

Still according to Statista, the number of smart homes worldwide is expected to experience significant growth, with an increase of 424.5 million users (a 117.69% rise) forecasted between 2023 and 2028 [10]. By 2028, the global smart home market is projected to reach a new peak of 785.16 million users, marking nine consecutive years of growth. This continuous rise reflects the increasing adoption of smart home technologies, which offer enhanced convenience and energy efficiency. As smart homes become more integrated into daily life, they are also likely to play an increasingly central role in the broader IoT ecosystem.

2.1.1 Smart homes

Richard Harper [11] defines a smart home as a residence equipped with computing and information technologies designed to anticipate and fulfill the needs of its occupants. By integrating and managing technology within the home and connecting to the outside world, smart homes aim to enhance comfort, convenience, security, and entertainment.

The underlying computing and information technology primarily involve IoT devices connected to the home's Local Area Network (LAN). These IoT devices enable seamless communication among themselves, with general-purpose IT devices such as PCs and smartphones, and even with external systems via the Internet through a router. This interconnected framework allows for efficient automation and remote management of the smart home environment.

It is for this specific kind of environment that our new IoT firewall tool was developed.

2.1.2 IoT protocols

Certain protocols have been specifically developed to address the unique requirements of the Internet of Things (IoT), with some achieving significant prominence in the domain of smart home applications. Among these, Zigbee [12] is widely recognized for its reliability, scalability, and low energy consumption, attributes that render it particularly well-suited for smart home systems. It is a low-power, reliable wireless communication protocol designed for smart home devices. It uses a mesh networking architecture, allowing devices to communicate directly and relay data, ensuring robust coverage. Based on the IEEE 802.15.4 standard [13], Zigbee adds network and application layer specifications to support scalable, interoperable IoT applications, ideal for devices like smart lights, thermostats, sensors, locks, and plugs. Zigbee's energy-efficient design and mesh networking capabilities make it suitable for home automation.

Another well-known IoT-specific protocol, namely MQTT [14], is discussed in a fairly exhaustive manner in section 2.5. It is a publish/subscribe protocol widely used in IoT applications. Contrarily to Zigbee, it is an IP-based protocol. Its lightweight, efficient design makes it suitable for low-bandwidth, low-

power devices in environments like sensor networks and remote monitoring.

2.1.3 IoT features

The peculiar nature of IoT objects make them distinct from general-purpose computing devices like personal computers and smartphones. As a result, they have their own specific features that need to be considered at the time of building a firewall designed for their particular IoT ecosystem.

One of the most important aspects of IoT systems that distinguish them from their general-purpose computers counterparts is their constrained nature, as stated in the work by Wei Zhou *et al.* [15]. In contrast to mainframes and personal computing devices, IoT devices have much less computing ability, storage resources, and power supply.

Other features specific to IoT systems identified by Wei Zhou *et al.* [15] are the following :

- Interdependence : IoT devices exhibit increasingly complex interactions, often operating autonomously with minimal human intervention, as they are often implicitly influenced or controlled by the behavior of other devices or environmental conditions.
- Diversity : IoT devices are highly heterogeneous, each designed for specific tasks, and they interact dynamically with various physical environments.
- Mobile : some IoT devices can be described as being mobile such as wearable devices and smart cars as they can hop from one network environment to another and communicate with many unknown new devices.
- Ubiquitous: this feature highlights how IoT devices have seamlessly integrated into everyday life, becoming an essential part of people's routines.
- Myriad: this feature emphasizes the vast proliferation of IoT devices and the astronomical volume of data they generate and consume.
- Unattended : this features describes how IoT devices in some specific situations have to operate for long periods of time without physical access, and are thus left unattended.

- Intimacy: this feature highlights how some IoT devices collect sensitive personal data, including biological metrics like heart rate and blood pressure, daily activities, and even detailed housing information in the context of smart homes.

Interdependence and intimacy are particularly noteworthy IoT features to consider in the context of smart homes.

Intimacy is crucial, especially regarding privacy and security concerns. Extensive private data collection by IoT devices can lead to the potential misuse or unauthorized access to sensitive information, making it an essential aspect of any IoT solution.

Interdependence is equally significant, particularly in the design of our new IoT firewall for smart homes. The interactions between devices can indeed be leveraged to create dynamic firewall rules, enabling the profiling of IoT devices to whitelist legitimate packets. This aspect of IoT communication highlights a key advantage of the proposed firewall solution this thesis is about. Unlike previous firewall approaches, which often overlooked this dimension, this new IoT solution emphasizes device interactions as a central design consideration. This idea is explored in greater detail in the section 2.2, which discusses the MUD standard. This standard was designed to facilitate the creation of detailed network profiles for IoT devices, with further insights provided in the discussion about the limitations of this standard.

2.1.4 IoT security issues

Due to their constrained nature, difficulties arise when trying to run host-based IDS and other state-of-the-art solutions on IoT devices, as illustrated by the work of Martins *et al.* [16]. This is because IDS typically require significant computational and memory resources, which makes them challenging to deploy effectively on such devices.

Moreover, it should also be noted that the security of the IoT devices often depends on the goodwill of the manufacturers of the devices to apply security by following best practices and updating consciously their systems. Without such efforts, many devices risk being left with outdated security solutions, leaving them increasingly vulnerable to emerging threats.

Numerous types of attacks can target IoT systems, reflecting the growing complexity and diversity of threats in this domain. Sadhu *et al.* [17] provide a

comprehensive overview of potential attacks against IoT, offering an extensive catalog that highlights the multifaceted nature of these threats.

In another study, Al Hayajneh *et al.* [18] classified IoT attacks into three primary categories: denial of service (DoS), data manipulation, and data disclosure. DoS attacks jeopardize the availability of systems, data manipulation attacks undermine the integrity of data, and data disclosure attacks threaten confidentiality and privacy. These classifications help to better understand and address the specific risks posed to IoT systems.

It can be noted that in some critical applications, the consequences of certain attacks can be dire. For instance, both DoS and data manipulation attacks in the context of medical settings that use IoT systems could lead to life-threatening situations as operational failures in such situations can result in devastating outcomes.

Given the vast array of possible attacks against IoT systems, this thesis narrows its focus to those most relevant to our specific threat model, avoiding an exhaustive exploration of all potential threats. For instance, while physical attacks on IoT devices—requiring direct access—are an important concern, they fall outside the scope of this study. Instead, we concentrate on network-based attacks that align more closely with the objectives of our research.

The attacks considered in our threat model – identical to the one defined during the initial development of this new IoT firewall tool [2] – can be broadly categorized into two types. The first type involves attempts to compromise devices through network-based methods, such as denial-of-service (DoS) attacks, unauthorized device scanning, illicit remote access, or the injection of unauthorized control messages not initiated by the Smart Home resident. The second type encompasses the misuse of already compromised devices, either to launch further attacks on other devices or to exfiltrate sensitive data, thereby amplifying the scope and impact of the initial breach.

The second category of attacks, as described above, can be deployed on a large scale to carry out destructive operations. A notable example of such an attack is the Mirai botnet.

The Mirai botnet [19] is a significant case in cybersecurity, illustrating the vulnerabilities of Internet of Things (IoT) devices. Discovered in 2016, it targets IoT devices with weak or default credentials, exploiting their susceptibility to brute-force attacks. Once infected, these devices are co-opted into a botnet controlled by a command-and-control server, enabling large-scale cyberattacks, in particular Distributed Denial of Service (DDoS) attacks.

Mirai gained notoriety for attacks such as the 620 Gbps DDoS on Krebs on Security, a website run by cybersecurity expert Brian Krebs, and the 1.2 Tbps attack on Dyn DNS, which disrupted major online services like Twitter and Netflix. Its release as open-source code has led to numerous variants, such as the IoTroop botnet [20], thus extending its impact. These incidents have emphasized the critical need for improved security practices, such as enforcing strong credentials, regular firmware updates, and implementing network-level defenses. The legacy of Mirai highlights persistent risks in the IoT landscape, underscoring the importance of both user vigilance and manufacturer responsibility.

The widespread vulnerabilities highlighted by incidents like the Mirai botnet underscore the urgent need for robust security solutions in the IoT ecosystem. One promising approach to enhancing IoT security is the Manufacturer Usage Description (MUD), which provides a framework for enforcing security policies based on device behavior. The following section 2.2 will explore in more detail the MUD standard, which is an essential inspiration for the new IoT firewall tool our thesis is about.

2.2 The Manufacturer Usage Description

This section gives an introduction to the Manufacturer Usage Description (MUD) that is the standard which inspired the logic of the new IoT firewall tool that is the focus of this thesis.

2.2.1 Description of the standard

The Manufacturer Usage Description (MUD) is a standard introduced by the Internet Engineering Task Force (IETF) in RFC8520 [1], published in 2019. This standard is designed to enhance network security and functionality by enabling devices to communicate their specific network requirements to the environment in which they operate. According to the document:

> "The goal of MUD is to provide a means for end devices to signal to the network what sort of access and network functionality they require to properly function."

To achieve this, the MUD standard requires device manufacturers to supply a MUD file – a detailed specification outlining the device's permitted network behaviors, such as allowed IP ports and host addresses.

This machine-readable file can then be used to configure firewalls and switches to enforce these restrictions, ensuring that only the specified network activities are permitted while blocking unauthorized ones. This approach helps devices perform their intended functions securely and efficiently.

The rationale behind this standard stems from the understanding that smart objects are designed with specific, intended purposes, making all other uses unintended. As a result, the expected behavior of a smart object can be systematically defined. In this context, creating a MUD file is analogous to profiling the allowed behavior of an IoT device.

The concept of identifying patterns in the behavior of purpose-driven smart objects had already been explored in RFC7452, which discusses design patterns for smart objects [21]. This earlier work laid the groundwork for the structured approach that MUD adopts to define and enforce device-specific network behavior.

MUD file format

MUD files are written in JSON or XML format and structured using the YANG data modeling language [22], which facilitates the creation of access control lists (ACLs). These ACLs effectively serve as the device's behavioral profile, defining its permitted network interactions.

Below is an excerpt from a Manufacturer Usage Description (MUD) file, presented as Listing 2.1. This example highlights the typical structure and content of a MUD file, with a primary focus on access control lists (ACLs). The file has been compacted to conserve space, and part of metadata has been omitted to emphasize the ACLs. The full uncompacted version of the file is available in Section 9 of RFC8520 [1].

```
{ "ietf-mud:mud": {
    "mud-version": 1,
    "mud-url": "https://lighting.example.com/lightbulb2000",
    [...],
    "systeminfo": "The BMS Example Light Bulb",
    "from-device-policy": {
      "access-lists": {
        "access-list": [
          { "name": "mud-76100-v6fr" } ] } },
```

```
    "to-device-policy": {
      "access-lists": {
        "access-list": [
          { "name": "mud-76100-v6to" } ] } } },
    "ietf-access-control-list:acls": {
      "acl": [
      { "name": "mud-76100-v6to",
        "type": "ipv6-acl-type",
        "aces": {
          "ace": [
            { "name": "cl0-todev",
              "matches": {
                "ipv6": {
                  "ietf-acldns:src-dnsname": "test.example.com",
                  "protocol": 6 },
                "tcp": {
                  "ietf-mud:direction-initiated": "from-device",
                  "source-port": {
                    "operator": "eq",
                    "port": 443 } } },
              "actions": {
                "forwarding": "accept" } } ] } },
      { "name": "mud-76100-v6fr",
        "type": "ipv6-acl-type",
        "aces": {
          "ace": [
            { "name": "cl0-frdev",
              "matches": {
                "ipv6": {
                  "ietf-acldns:dst-dnsname": "test.example.com",
                  "protocol": 6 },
                "tcp": {
                  "ietf-mud:direction-initiated": "from-device",
                  "destination-port": {
                    "operator": "eq",
                    "port": 443 } } },
              "actions": {
                "forwarding": "accept" } } ] } } ] } }
```

Listing 2.1: MUD file example from RFC8520 (snippet) [1]

As discussed in the RFC [1], the MUD file in this example defines two policies: one governing traffic from the device and the other governing traffic to the device.

The first policy, defined in the `mud-76100-v6to ACL`, permits the device to initiate outbound connections to the domain `test.example.com` using the TCP protocol on port 443. This rule ensures that the device can securely communicate with the specified server for functions such as updates or remote management.

The second policy, defined in `the mud-76100-v6fr` ACL, allows inbound traffic from `test.example.com` over TCP on port 443, enabling the server to respond to the device's requests.

Both policies assume that the device initiates the connection, and the enforcement point (such as a firewall or router) is expected to allow traffic that matches these rules while blocking any unauthorized traffic. The access is specifically tied to the domain `test.example.com`, ensuring that only interactions with this server are allowed, which helps maintain the device's secure and intended operation.

2.2.2 Limitations of the MUD standard

The MUD standard serves as the foundational inspiration for the new IoT Smart Home firewall tool presented in this thesis. While the MUD standard represented a significant advancement in enhancing IoT security, it has notable limitations that the new IoT firewall tool aims to address. Specifically, the MUD standard struggles to account for complex communication patterns and to effectively describe these behaviors in device profiles. The IoT firewall tool discussed in this thesis was developed to overcome these shortcomings and extend the capabilities of the MUD standard.

The specific MUD weaknesses raised in the Smart Home Firewall article [2] are the following :

- The MUD standard only supports matching capabilities for commonly used protocols defined by the ACL YANG model such as Ethernet, IPv4 and v6, TCP and UDP [22]. Our new firewall tool allows to match on many more higher network layer protocols, such as DHCP, DNS and HTTP for examples.

- MUD standard lacks traffic statistics matching capabilities such as packet rate and count or flow size (in bytes).
- The MUD standard can only take into consideration individual communications, with no capacity to consider the complex interactions between the devices.

The lack of accuracy of MUD to describe patterns emerging from IoT communication leaves systems vulnerable to malicious activities that deviate from those patterns, since such activities cannot be blocked by the firewall. This vulnerability enables unauthorized device control and data exfiltration attacks on IoT networks. Data exfiltration refers to the unauthorized transfer of information from an information system.

For example, Ronen *et al.* [23] discuss attacks that exploit the designed functionality of IoT devices to achieve entirely different outcomes – attacks they call extended functionality attacks. One such example involves smart lights, where attackers manipulate light color and intensity to covertly transmit data, effectively using the lights to exfiltrate information.

Similarly, Uroz *et al.* [24] examine the suitability of IoT protocols for data exfiltration. They developed a tool to encapsulate and exfiltrate data via IoT protocols, notably MQTT, demonstrating another avenue of attack even when MUD is applied.

These examples highlight the limitations of the MUD standard in securing IoT systems and underscore the need for a more robust framework to effectively profile and protect IoT behaviors.

It is important to note that some limitations of the MUD standard still apply to our new IoT firewall tool, as they stem from the inherent logic of the system. As discussed earlier in Section 2.1.4, the security state of an IoT device often depends on the manufacturer's commitment to incorporating robust security measures.

In some cases, the legitimate behavior of a device may inherently involve insecure practices, such as using deprecated protocol versions or omitting essential security features. This leaves the device vulnerable and limits the security profile that can be defined for it, as it ultimately reflects the manufacturer's approach to security.

Blocking traffic using insecure protocols, while seemingly protective, may not always be feasible, as it could render the device non-functional. This underscores the challenges of balancing security with usability in such scenarios.

2.3 NFTables

NFTables [25] is the modern Linux kernel packet classification framework developed by the NetFilter Project. It provides simple, efficient rule-based packet header matching for key protocols of the transport, network, and data link layers of the OSI model. As the successor to the well-known iptables [26], NFTables was introduced to address the complexities and limitations of its predecessor.

Iptables, a cornerstone of Linux network security for many years, was a robust yet often cumbersome framework for managing firewall rules and packet filtering. Its rule management was linear, which could lead to performance bottlenecks and complex configurations as rule sets grew in size. NFTables streamlines this process with a more modern, efficient approach, replacing multiple legacy tools (e.g., iptables, arptables, and ip6tables) with a unified framework. Key features include a unified syntax for IPv4, IPv6, ARP, and bridge filtering, reduced duplication of rules, faster rule evaluation using a set-based architecture, and advanced matching capabilities based on packet rate and size.

This makes NFTables not only a powerful upgrade for firewall management but also a versatile tool for network security in modern Linux systems.

The following NFTables script (Listing 2.2), for which the creation has benefited from the use of ChatGPT [27], illustrates the configuration of a simple firewall for an IPv4 network. This example demonstrates how NFTables provides a streamlined approach to defining and applying rules for packet filtering and network security. The script defines a rule set that allows specific types of incoming traffic while blocking all other traffic by default, a common baseline for securing network communications.

```
#!/usr/sbin/nft -f

flush ruleset # flush any existing rules to ensure a clean slate

# define a new table in the 'inet' family
table inet my_table {
    # define the 'input' chain for managing incoming packets
    chain input {
        type filter hook input priority 0; policy drop;

        # accept traffic on the loopback interface
```

```
12          iif lo accept
13          # accept packets from established and related connections
14          ct state established,related accept
15          # allow incoming SSH traffic (TCP port 22)
16          tcp dport 22 accept
17          # allow incoming HTTP (port 80) and HTTPS (port 443) traffic
18          tcp dport { 80, 443 } accept
19
20          log prefix "Dropped packet: " flags all level info # Log all
                dropped packets
21
22          drop # default rule: drop all other traffic
23      }
24
25      # define the 'output' chain for managing outgoing packets
26      chain output {
27          type filter hook output priority 0; policy accept;
28      }
29
30      # define the 'forward' chain for managing forwarded packets
31      chain forward {
32          type filter hook forward priority 0; policy drop;
33      }
34  }
```

Listing 2.2: NFTables script for basic firewall configuration

This script begins by flushing any existing rules from the system using the `flush ruleset` command to ensure a clean slate for the new configuration. It then creates a table named `my_table` in the `inet` family, which supports both IPv4 and IPv6 protocols, consolidating the rule management for these protocols into a single framework. Within this table, three chains are defined: `input`, `output`, and `forward`. These chains correspond to incoming, outgoing, and forwarded packets, respectively. Each chain is associated with a specific hook, which determines at what point in the packet processing flow the chain is applied.

The `input` chain is configured with a default policy to drop all packets. This chain includes rules to explicitly allow specific types of traffic, beginning with traffic on the loopback interface, which is essential for internal system communication. It also permits packets associated with established or related

connections, ensuring the continuity of ongoing communications without requiring repeated rule matches. Specific rules are added to allow incoming SSH traffic on port 22 for remote management and HTTP/HTTPS traffic on ports 80 and 443 for web services. These ports are grouped concisely using a set-based notation, demonstrating one of NFTables' efficient features.

To enhance security monitoring, the script includes a rule to log dropped packets with a custom prefix for identification. Any packet that does not match the preceding rules is dropped by default, adhering to the secure principle of least privilege. The `output` chain is configured with a default policy to accept all outgoing traffic, simplifying egress management, while the `forward` chain drops all forwarded packets, reflecting a typical configuration for non-routing systems.

The NFQueue library

NFTables' native matching and filtering capabilities are relatively limited, focusing on efficient classification of packets at lower OSI layers. However, its flexibility is greatly enhanced through the `libnetfilter_queue` library (hereafter referred to as NFQueue) [28]. The IoT firewall tool this thesis is about makes use of it.

NFQueue allows NFTables rules to offload packets that meet specific criteria to a "queue", where they can be processed by a user-space program. This queue is implemented as an arbitrary callback function, enabling detailed inspection and treatment of packets, including analysis of higher OSI layers. This integration makes NFTables a powerful tool for advanced network filtering and application-specific processing.

```
1    # HTTPS/TLS matching rule
2    meta l4proto tcp tcp dport 443 ip saddr 192.168.1.150 ip daddr
        192.168.1.135 limit rate 134/second queue num 4070
```

Listing 2.3: NFTables rule enabling further processing using NFQueue

Listing 2.3 provides an example of an NFTables rule that utilizes NFQueue. This example originates from a rule implemented during the addition of TLS protocol support to the firewall. It demonstrates the utility of the NFQueue library by illustrating how packets are redirected to a user-space program, written in C, for advanced matching of specific fields in the packet.

This NFTables rule is designed to match and process specific TCP traffic under certain conditions. It begins by specifying that it applies to packets

using the TCP protocol (`meta l4proto tcp`). It further narrows its scope to traffic destined for port 443, which is commonly used for TLS/HTTPS (`tcp dport 443`). The rule is also specific to packets originating from the source IP address 192.168.1.150 (`ip saddr 192.168.1.150`) and destined for the IP address 192.168.1.135 (`ip daddr 192.168.1.135`), which refer, on the one hand, to the client device and, on the other, to the server device.

Once the traffic matches these criteria, the rule enforces a rate limit of 134 packets per second (`limit rate 134/second`). If the traffic exceeds this rate, additional packets are not matched by the rule and are therefore excluded from further processing. For traffic within the rate limit, the rule redirects the packets to NFQueue number 4070 (`queue num 4070`), allowing user-space applications to perform additional custom processing, such as packet inspection or modification. Specifically, it compares the values of certain packet fields against the values written in the profile of its corresponding device to determine whether the packet is legitimate. Based on this comparison, the application decides whether to drop or accept the packet.

2.4 The TLS protocol

The Transport Layer Security (TLS) protocol was previously not implemented in the IoT firewall tool. The integration of TLS support into the tool is one of the key objectives of this thesis.

2.4.1 Introduction

TLS is a cryptographic protocol designed by the IETF to provide secure communication over a computer network, in particular over the Internet.

It is the successor of a previous widely used encryption protocol called Secure Sockets Layer (SSL), which was back then developed by the company Netscape [29]. The IETF afterwards published as a historical document the 1996 draft of SSL 3.0 in RFC 6101 [30].

TLS was first defined in 1999 by the IETF as a revision of SSL 3.0 in the RFC 2246 [31].

This SSL 3.0 heritage is still visible today in TLS payloads as the version value “3.1” serves to indicate TLS 1.0 rather than the value “1.0” in the record header of the payload, as TLS 1.0 actually corresponds to SSL 3.1.

Since then, many new versions of the protocol have been released. At the time of writing this thesis, the current version of the TLS protocol was 1.3.

TLS usage

It is worth noting that even though versions prior to 1.3 are now considered deprecated, and even unsecure for the oldest ones, they are still largely used to this day, especially the previous version 1.2. According to SSL Labs, a research and service platform for SSL and TLS protocols managed by Qualys, at the time of the SSL Pulse May 2024 scan, almost all the websites surveyed supported TLS version 1.2 (+99%), while only 70% of them supported TLS version 1.3. [32] The websites that support the newest version of TLS usually also support many older versions of the protocol.

2.4.2 Protocol description

All versions of the TLS protocol (as specified in RFC 2246 [31], RFC 4346 [33], RFC 5246 [34], and RFC 8446 [35]) describe how TLS enables secure communication between client and server applications over the Internet. TLS is designed to prevent eavesdropping, tampering, and message forgery.

This security is achieved by allowing the client and server to exchange parameters used for encrypting their communication. The TLS handshake begins when a connection is initiated, during which the client and server exchange information to establish a secure connection. This process includes negotiating the highest level of security both parties support, exchanging digital certificates for authentication, and generating session keys for encryption.

Once the handshake is complete, data can be transmitted securely. TLS encrypts the data to prevent eavesdropping and tampering. Additionally, TLS ensures data integrity through message authentication codes (MACs), safeguarding against loss or alteration during transmission. Throughout the session, these mechanisms work together to provide a secure and reliable communication channel.

The network layer at which level TLS operates

TLS is often considered to operate at Session Layer (Layer 5) but this can be considered arbitrary since the OSI model is an overgeneralization that does not fit well for TLS.

When using HTTPS, a more accurate conceptual model is that TLS operates between TCP and HTTP, though this representation is a simplification of its actual technical implementation. When rather considering the TCP/IP model, an appropriate description would be to say that TLS operates between the Transport Layer and the Application Layer by encrypting Application Layer traffic during transport.

Differences between TLS versions

TLS has evolved significantly over the years, with key improvements in security and performance. TLS 1.0 [31] and 1.1 [33], based on SSL, are now deprecated due to weaknesses like insecure cipher suites and vulnerabilities to attacks like BEAST [36]. TLS 1.2 (RFC 5246 [34]) introduced stronger cryptographic algorithms such as SHA-256 and added support for elliptic curve cryptography, making it more secure and flexible.

TLS 1.3 (RFC 8446 [35]) further streamlined the protocol by simplifying the handshake and removing obsolete features, such as the RSA key exchange and weaker ciphers like RC4. It also mandates forward secrecy for all key exchanges, ensuring better security. Additionally, TLS 1.3 supports zero round-trip time (0-RTT) data, which speeds up connections for clients that have previously connected to the server.

TLS version 1.2 description

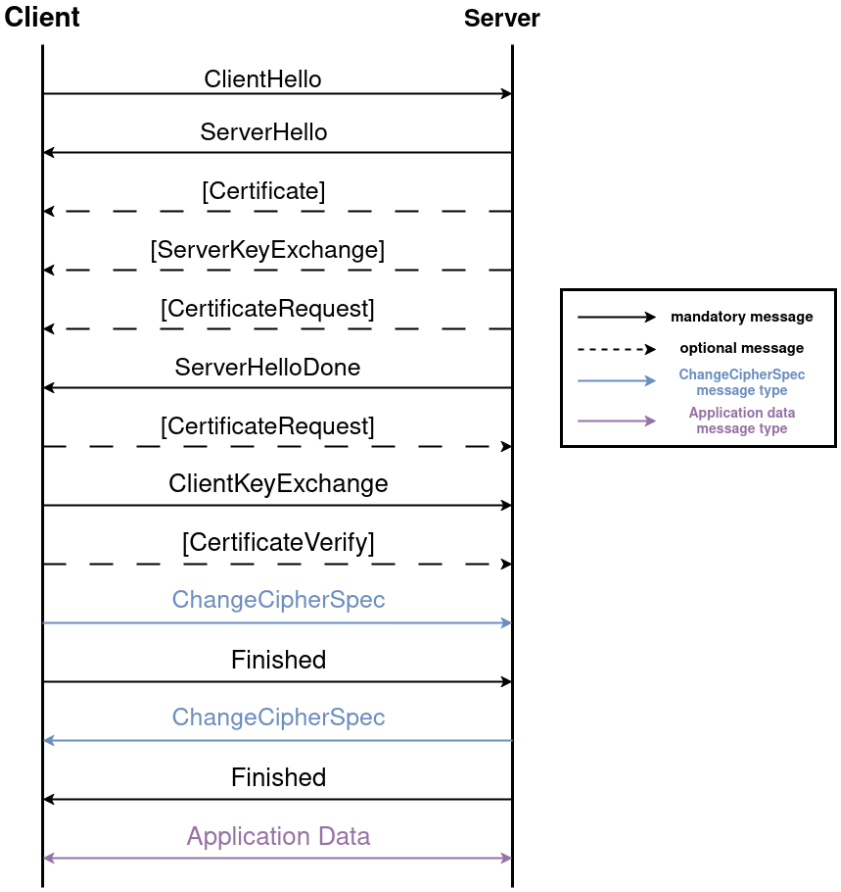

Figure 2.1: TLS 1.2 message flow for a full handshake

TLS 1.2 [34] significantly improves upon earlier versions, particularly in terms of cryptographic strength. The handshake process in TLS 1.2, illustrated by Figure 2.1, while similar to earlier versions, includes more robust cryptographic mechanisms and flexibility in cipher suite selection. The key exchange and certificate verification steps are outlined in the handshake schema, which ensures secure communication through multiple round trips. The client and server exchange their "ClientHello" and "ServerHello" messages, followed by authentication and key exchange, and then both parties exchange "ChangeCipherSpec" and "Finished" messages to finalize the handshake.

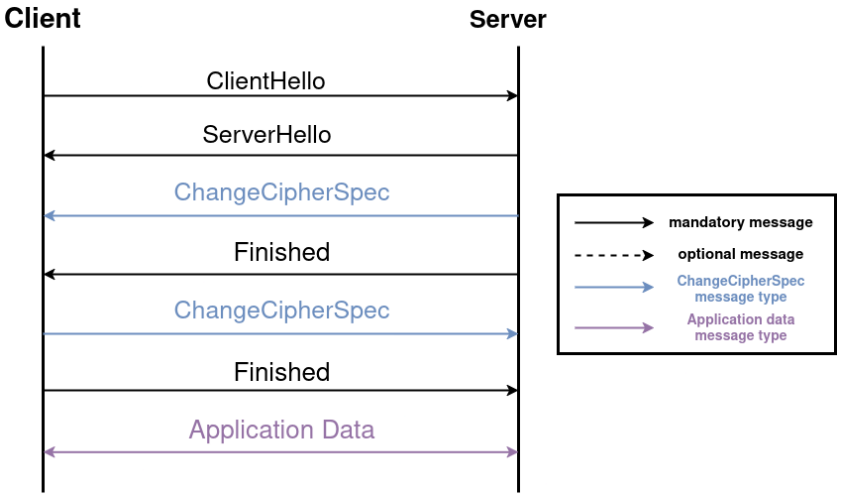

Figure 2.2: TLS 1.2 message flow for an abbreviated handshake

In an abbreviated handshake, illustrated by Figure 2.2, the client can simply provide the session ID to resume a previous connection, skipping much of the key exchange and certificate steps.

TLS version 1.3

TLS 1.3 [35] focuses on simplifying the handshake and enhancing security. The handshake in TLS 1.3, shown by Figure 2.3, reduces the number of round trips compared to TLS 1.2. This is achieved by removing outdated algorithms and ensuring all key exchanges are ephemeral for forward secrecy. The client and server still exchange "ClientHello" and "ServerHello" messages, followed by the server's certificate. The session is then secured with "ChangeCipherSpec" and "Finished" messages. TLS 1.3 also introduces 0-RTT data, allowing clients to send data immediately after the "ClientHello" message, provided they have previously established a session, though this comes with trade-offs such as potential susceptibility to replay attacks and the lack of forward secrecy for the transmitted data.

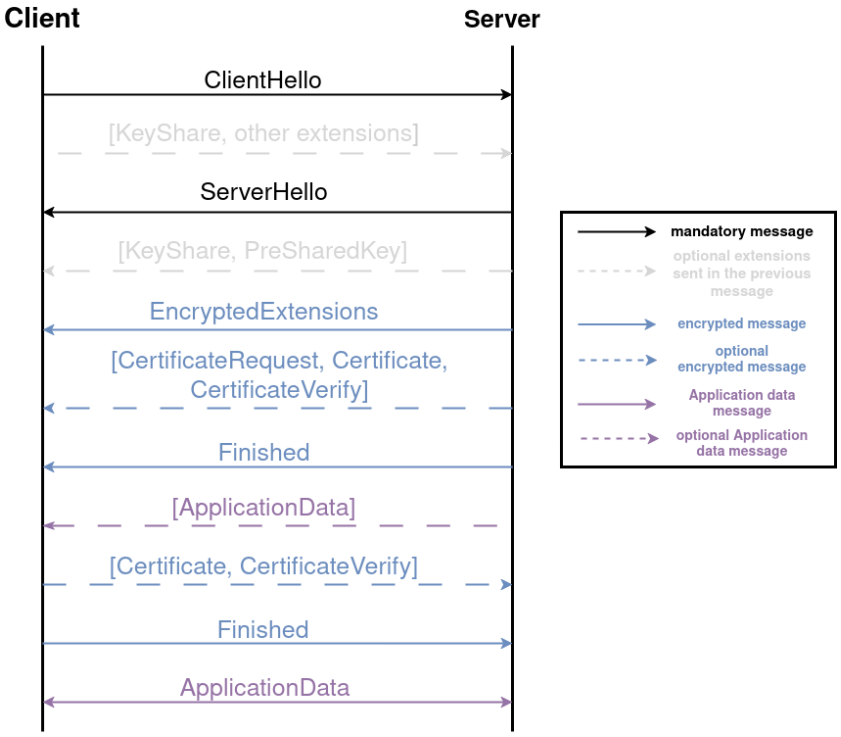

Figure 2.3: TLS 1.3 message flow for a full handshake

2.5 The MQTT protocol

This section introduces the MQTT protocol, which was integrated into our firewall as part of the work conducted for this thesis. Prior to this, the firewall did not support MQTT, and its functionality was extended to address this gap.

2.5.1 Introduction

MQTT (Message Queuing Telemetry Transport) [14] is a lightweight, publish/-subscribe messaging protocol designed for low-bandwidth, high-latency, or unreliable networks. It was originally developed by IBM in the late 1990s and has since become an open standard (ISO/IEC 20922:2016) in 2013.

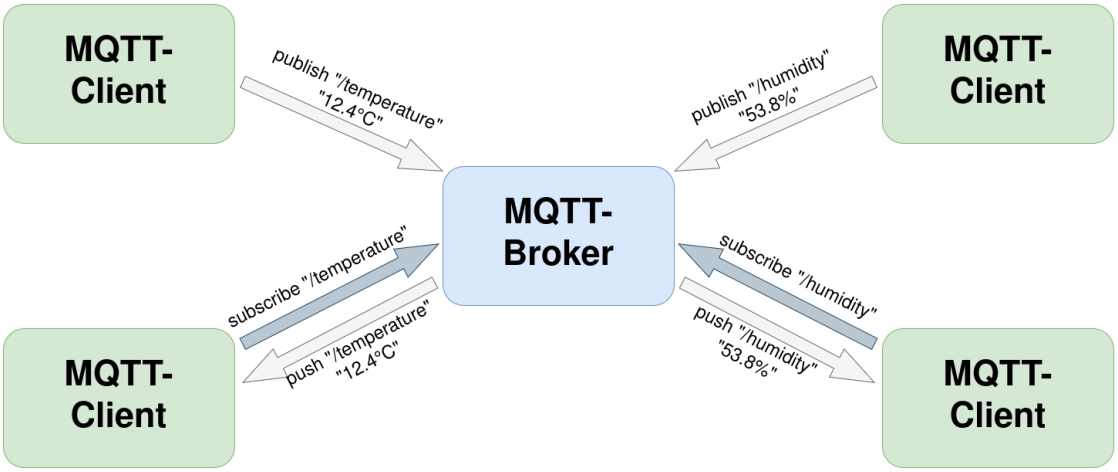

Figure 2.4: MQTT minimal network example

In a MQTT network, devices (clients) communicate by publishing messages to specific topics or subscribing to them. A central broker manages these interactions, distributing messages from publishers to relevant subscribers. Figure 2.4 illustrates this concept, showing two MQTT clients publishing messages and two others subscribing to topics, demonstrating the protocol's publish/subscribe model.

2.5.2 Protocol description

MQTT typically operates on port 1883 over TCP/IP.

Messages consist of a compact fixed header containing the message type, flags, and length, along with optional headers and payloads, depending on the message type.

Topic names are UTF-8 encoded strings with no specific format, allowing them to include text, binary data, or encrypted information, providing flexibility in message content and structure.

Topics

MQTT topics are organized hierarchically in a tree structure, similar to filesystems. For example, there could be the following MQTT topics :

- `hospital/room01/temperature`
- `hospital/room02/temperature`

- `hospital/room02/humidity`
- `office/room01/temperature`

Clients can use wildcards to subscribe to multiple topics in a flexible manner. The `+` wildcard matches exactly one level of the topic hierarchy, as in `hospital/+/temperature`, which matches both `hospital/room01/temperature` and `hospital/room02/temperature`. The `#` wildcard, on the other hand, matches one or more levels. For instance, `hospital/#` will match any topic under `hospital`, such as `hospital/room01/temperature` or `hospital/room02/humidity`.

Type of messages

In MQTT, `CONNECT`, `PUBLISH`, and `SUBSCRIBE` are packet types defined in the protocol payload to facilitate communication between clients and brokers. The `CONNECT` packet initializes the connection, carrying details like client credentials and keep-alive settings. The `PUBLISH` packet delivers messages to a specific topic, enabling the distribution of data. The `SUBSCRIBE` packet registers a client's interest in a topic, ensuring it receives relevant published messages. These packet types form the foundation of MQTT's efficient publish/subscribe messaging architecture.

Retained messages

By default, MQTT brokers forward messages to subscribers only if they are connected at the time of publication. New subscribers generally do not receive messages published before they joined. However, messages can be marked as "retained," allowing the broker to store them and deliver the latest retained message to new subscribers immediately upon their subscription. This ensures critical information is available to late-joining clients.

Last Will

In MQTT, publishers can configure a Last Will message that the broker sends to a specified topic if the client disconnects unexpectedly. If the broker does not receive a message from the client within 1.5 times the keep-alive duration, it assumes the client is offline and triggers the Last Will message. This feature provides a mechanism to notify subscribers of abrupt disconnections, ensuring system resilience and timely awareness of network issues.

Quality of Service

MQTT offers three levels of Quality of Service (QoS) to manage message delivery reliability. At QoS 0, messages are delivered at most once without acknowledgment, suitable for non-critical data. QoS 1 ensures at least one delivery, with potential duplicates due to retransmissions. QoS 2 guarantees exactly one delivery through a handshake process, making it the most reliable but resource-intensive option. These levels allow developers to balance reliability and overhead based on application needs.

Chapter 3

Design and implementation

We outline the firewall's architecture and explain how TLS and MQTT were integrated. The chapter covers system design, protocol-specific parsing mechanisms, and key technical decisions that enhance IoT traffic filtering.

3.1 Overview and design principles

As the work for this thesis only consisted in the enhancement of an already existing firewall, most of the architecture has remained unchanged. An exhaustive description of the latter is already provided in the article detailing the creation of this firewall, along with an explanation of the new language developed to describe and supervise interactions with smart home devices [2].

The following sections of this thesis therefore mainly describe design principles that are already discussed in the above-mentioned article. The purpose there is actually to provide a basis for understanding how the inclusion of missing protocols was made possible without the need for the reader to look at other documents beside this one. The greatest attention is therefore paid to the points that were essential to enable our own implementation.

3.1.1 System overview

The firewall system was designed based on comprehensive traffic profiles that specify the allowed communication patterns for each IoT device in the Smart Home. This approach allows the system to analyze packets as part of broader interaction patterns, ensuring more precise and context-aware traffic

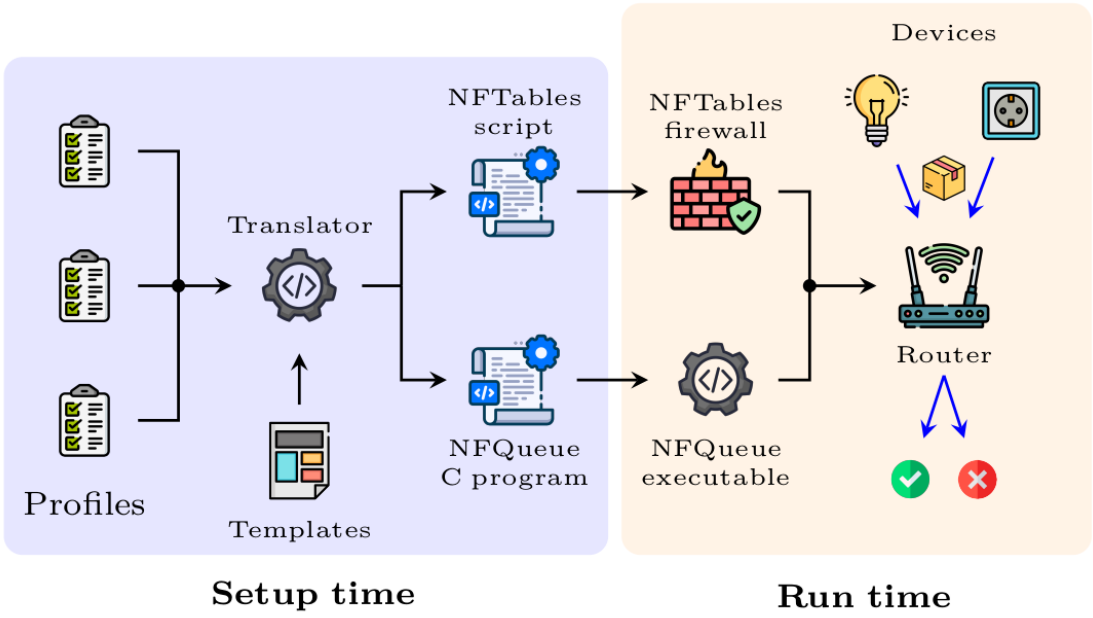

Figure 3.1: System components of the firewall, from [2, Fig. 3]

management. This capability is enabled by a newly developed language, created specifically for this innovative firewall project.

The information given in the traffic profile of a device can then be used to defend against attacks by identifying and blocking unwanted or unexpected communications, as any traffic that deviates from the patterns identified in the profile can be dropped by the firewall.

Under the assumption that the smart home to be protected is based on a (wireless) LAN infrastructure, the most obvious place on which the firewall should be running would be on the central home router or access point. This is justified by the fact that this is where communications between devices and the outside world can be monitored.

System components

Figure 3.1 illustrates the different components of the system.

We can distinguish between the components that are required for the setup of the firewall system and those used when actually running the firewall.

During run time, the firewall, which is built on NFTables [25] (see section

2.3), runs on the router to filter traffic to and from the smart home devices. Where necessary, the matching packets are offloaded to a user-space program using the NFQueue infrastructure. This is required since NFTables' matching and filtering capabilities are limited by themselves to the lower-layer Internet protocols, and consequently the higher-layer protocols behavior can only be considered by using the well-known NFQueue library [28] from which NFTables takes its flexibility.

The components required for the setup include the devices' profiles and the templates used to guide the writing of the NFTables rules and their associated NFQueue user-space program code.

For efficiency reasons, the firewall does not load the devices' profiles and templates during runtime. Instead, these are utilized by another program, developed as part of this IoT firewall project, that automatically generates the appropriate NFTables configuration scripts along with the user-space NFQueue program codes.

This program, referred to as a translator, automates the conversion of the Smart Home devices' profiles directly into NFTables rules and their associated NFQueue user-space program code that will run on the router. It achieves this by also utilizing Jinja templates [37] developed as part of this project.

The motivation behind this design choice is that the firewall is intended to be used on consumer-grade routers and access points, which are inherently less powerful and resource-intensive than enterprise-grade hardware.

3.1.2 Code overview

This section provides a high-level overview of the code, explaining its main components and how they interact. The goal is to give a clear understanding of the project's structure and the relationships between its various parts, offering insight into how the overall system functions cohesively.

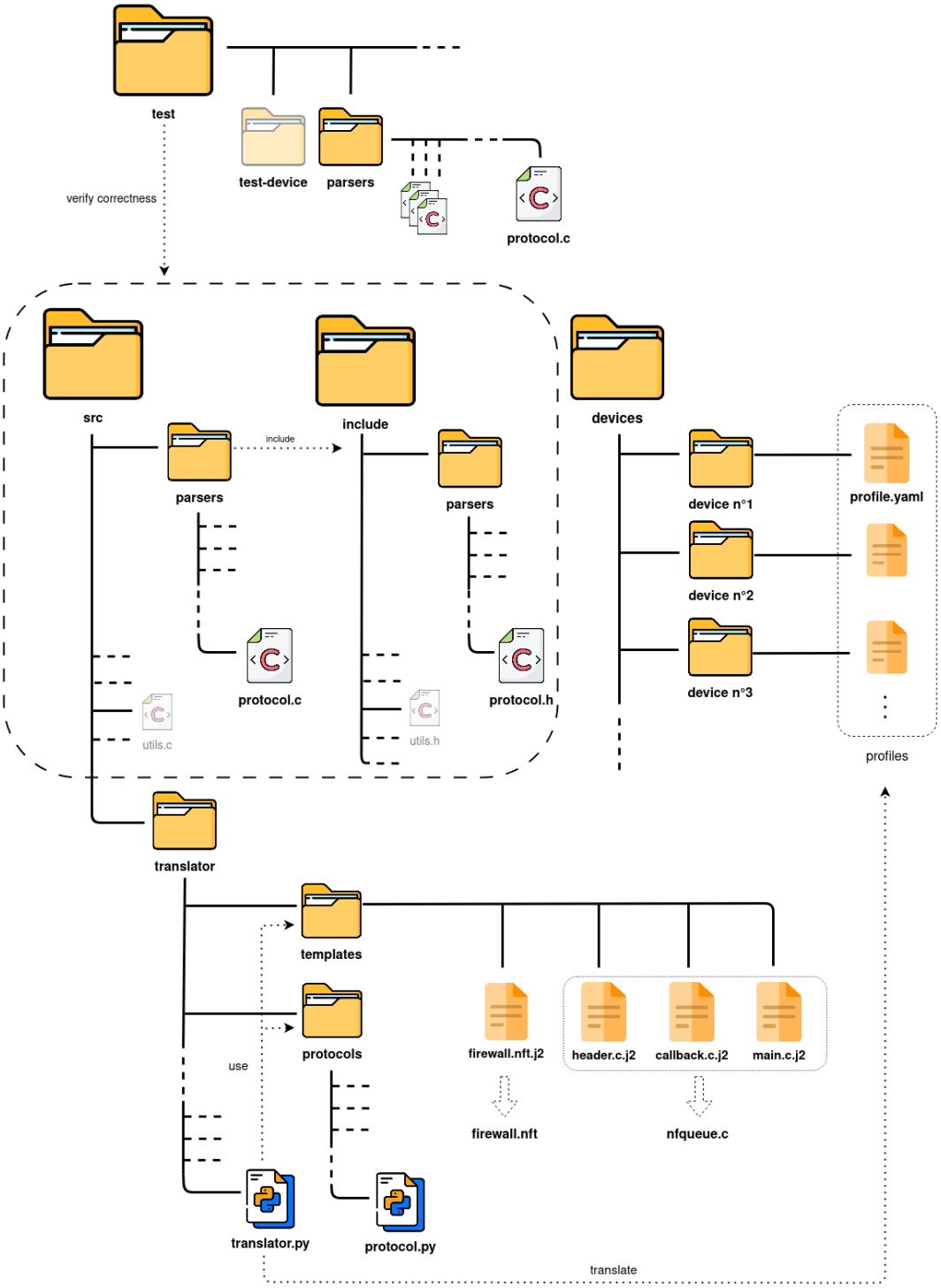

Figure 3.2: General code structure

Translator program

The translator program for this firewall is written in the Python language. It is run by executing the Python script `translator.py` (located in the ‘`src/translator/`‘ folder). It requires the device’s profile as an argument which is a file written in the YAML language [38]. The script then translates the profile into an NFTables configuration script, and when necessary an associated NFQueue user-space code, written in the C programming language.

The translation is guided by template files that leverage the Jinja templating engine [37], located in the `src/translator/templates/` folder. These templates serve distinct purposes in generating the necessary firewall components: `firewall.nft.j2` generates the NFTables configuration script, while the remaining templates (`header.c.j2`, `callback.c.j2`, and `main.c.j2`) collectively produce the corresponding NFQueue C code - with `header.c.j2` generating declarations and includes, `callback.c.j2` creating the packet processing logic, and `main.c.j2` providing the program entry point and initialization code. These templates have undergone minor modifications during our own implementation to enable new protocols to be taken into consideration by the firewall.

The translator also leverages Python code files, located in the ‘`src/translator/protocols/`‘ folder, that implement classes for each protocol supported by the firewall. Each protocol has its own dedicated Python class, allowing the appropriate code lines for that specific protocol to be seamlessly injected into templates during the translation process.

The object-oriented programming (OOP) capabilities of Python are utilized to structure these classes effectively. All protocol classes inherit from a general Protocol class. For protocols requiring a custom parser, their classes first inherit from a `Custom` class, which itself is a subclass of `Protocol`. This hierarchy is illustrated in Figure 3.3, showing the organization of the Python classes used in the program.

This figure also visually distinguishes the protocols based on their corresponding OSI layers, highlighting how protocols from OSI layers 3 and 4 primarily inherit directly from the `Protocol` class, while higher-layer protocols inherit from the `Custom` class.

This distinction is logical, as the requirements of lower-layer protocols can typically be handled exclusively by the NFTables configuration script, which only requires the addition of NFTables rules. However, for higher-layer

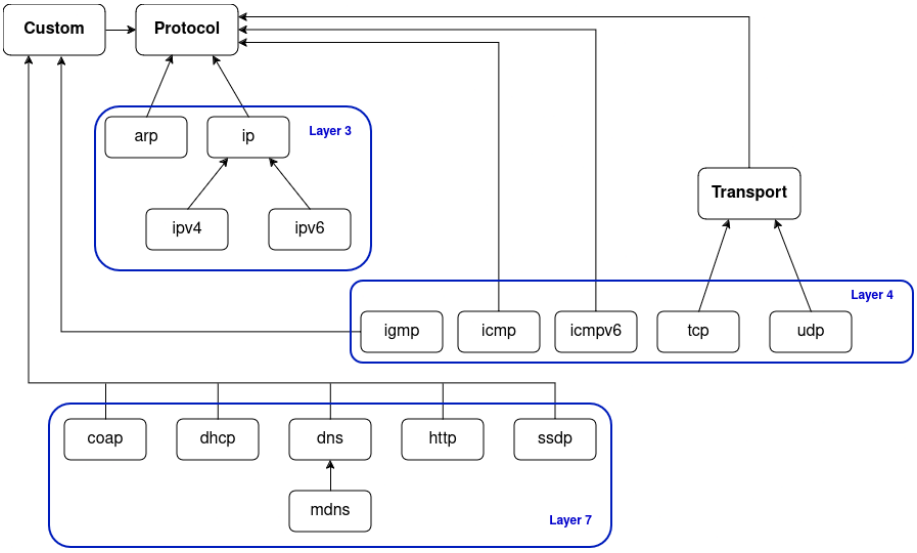

Figure 3.3: Hierarchy of the Python classes for the network protocols

protocols, NFTables alone are insufficient to address the protocol's device requirements, and their packets must be offloaded to NFQueue user-space code for processing.

NFQueue programs leverage other code files written in C that serve to parse the payload and evaluate whether it aligns with the expected behavior of the device or not. If the payload conforms to the expected behavior, it is accepted; otherwise, it is flagged as unexpected or undesirable and subsequently dropped.

Protocol-specific parsers

The parsers for the payloads of the protocols handled by the firewall, which are implemented in C, are used to parse the payloads of high-layer protocols. Their source files are located in the '`src/parsers/`' directory, while the corresponding header files are located in the '`include/parsers/`' directory.

These parsers are designed to analyze protocol payloads by extracting the contents of specific fields deemed relevant during the profiling of the appropriate network behavior for IoT devices. These extracted values can then be compared to the values specified in the profile of a device for the traffic of the corresponding protocol, allowing verification of whether the traffic payload is legitimate or not (e.g., ensuring the TLS version in the payload matches the value indicated in the profile for a specific policy).

Figure 3.4: Minimal device profile example, from [2], Fig. 4

The functions implemented for the parsers for the payloads of the protocols can then be injected into the Jinja code templates for the NFQueue user-space program, as needed, to perform the matching process of the firewall.

Devices profiles

The profile of the different devices can be found in their respective directories, each of them located in a ‘`devices`‘ parent directory.

The content of the profile is typically structured by first giving metadata, then the single policies for the device, and at last its interactions. Figure 3.4, taken directly from the Smart Home Firewall article [2], gives an example of minimal device profile that illustrates the abilities of the new profiling language. Key elements of a typical profile are shown, with custom fields highlighted in green.

Two main sections are shown, namely `device-info` and `interactions`.

The metadata of the device are specified into the `device-info` field. It contains information about the device such as its name, its MAC address, and its IP address (wheter IPv4 or IPv6). The keyword `self` can then be used in the rest of the profile to reference the device. The device can even also be referenced in another device’s profile by using the new `include` directive implemented in the context of the development of this firewall.

The `include` directive is also illustrated in the minimal device profile example through the use of the identifier `patterns.dns-p` that refers to the field `dns-p` under the field `patterns`.

The `interactions` section of the example shows one interaction called `dns-https-server` that contains two policies. A policy describes the packet features that the firewall would need to match on to let it through. Packets for which the features are not matched are automatically dropped by default.

Policies can be defined within an interaction when describing complex network activities, as shown in this example. Alternatively, they can be specified under the `single-policies` field in the profile for cases where they are not part of an interaction.

In an interaction, policies are not active simultaneously. An interaction begins when the first policy matches an incoming packet, making it the active policy of the interaction. The interaction will eventually deactivate the current active policy and transition to the next one. How it does so depends on the type of the currently active policy, which will be explained below.

In the minimal device profile example, the interaction consists of two policies. The first policy, named `dns-server`, refers to the identifier `patterns.dns-p`. The second policy, `https-server`, defines its own unique characteristics for matching new packets that the firewall must process.

Such an interaction cannot be expressed using the MUD standard profiling format [1], presented in a previous section 2.2. This underscores the advantages of the new profiling language developed for the IoT firewall tool presented in this thesis.

The currently active policy can be of different types, which determine how the interaction will eventually deactivate it and transition to the next policy. Below are the possible policy types:

- *One-off* policy: This is the default policy type. When it matches a packet, it is immediately deactivated, and the next policy is activated. A `bidirectional` property can be set, requiring the policy to match a packet in both directions – first based on the specified fields, and then with relevant fields (such as source and destination addresses and ports) inverted.
- *Transient* policy: This policy has a maximum duration or packet count defined in its `stats` section. The interaction transitions to the next

policy when the transient policy reaches its specified duration or packet limit, or when a packet matching the next policy is observed. If the `bidirectional` property is set, the policy matches packets in both directions while it remains active.

- *Periodic* policy: This policy includes a packet rate specification in the `stats` section. It remains active until a packet matching the next policy is observed. Any packets exceeding the specified rate are dropped.

  The `rate` can also be set to 0 to indicate an infinite rate.

3.2 Common implementation details

The following section discusses in which way the new IoT firewall tool was modified to extend it to new protocols. Since certain implementation procedures were common between both TLS and MQTT, this section aims to give a preamble to what had to be done in a similar way for both protocols. The document then goes in further details when deemed necessary in the sections that regard each specific protocol.

3.2.1 Translator modification content

Figure 3.5 illustrates the Python class hierarchy, now including the newly added TLS and MQTT protocols.

As high-layer OSI protocols, both TLS and MQTT inherit from the `Custom` class, as they require specific parsers to handle their payloads. These parsers enable the analysis of payload contents, allowing comparisons between the values of specific fields and those defined in the device profile. Since NFTables alone cannot match on high-layer OSI protocols, NFQueue is essential for processing these requirements effectively.

It might be of interest to note that TLS is indicated on the diagram as a Layer 5 OSI protocol since it is commonly agreed as such, but such choice is quite arbitrary as the OSI model does not apply well for TLS. This was discussed in more detail in section 2.4.

The Python classes for both TLS and MQTT can be utilized by the script `translator.py` to expand the NFTables script and its associated NFQueue user-space program. This process is carried out for each device whose profile

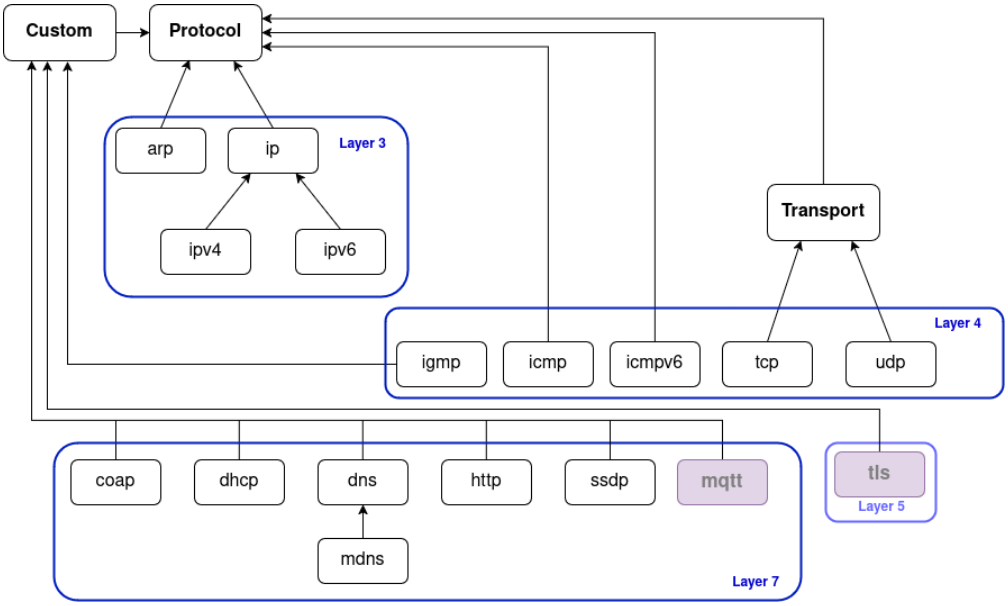

Figure 3.5: Python classes grouped by OSI layer membership with addition of the newly-added protocols TLS and MQTT

includes policies that account for these protocols, as they represent legitimate network behavior.

3.2.2 Parsers implementation and profiles writing process

The method for implementing the parser was very similar for both protocols. It required the writing of C code files to treat the protocols payloads and extract their relevant values.

The relevant values correspond to those of the fields of the payload that are identified as relevant at the time of creating or extending profiles for devices that use either new implemented protocols in their traffic.

The writing process regarding the profiles however differed between the two protocols. This process is thus discussed exhaustively into respective sections (3.3.1, 3.4.1) for each of the protocols.

The integration of new protocols into device profiles has led to the introduction of additional YAML elements, which are handled by the corresponding Python classes for each protocol.

For TLS, the new YAML elements are presented in the TLS profiling templates found in the appendix. Further details are provided in subsection

3.3.1.

Regarding MQTT, as discussed in section 3.4, the C parsing code already supports all fields within MQTT packets. However, the associated Python class currently processes only a subset of these fields. As a result, only the following YAML elements have been introduced: `packet-type`, `client-id`, `client-id-length`, `clean-session`, `keep-alive`, `topic-name`, `payload-regex`, and `payload-length`. These elements correspond to MQTT packet fields that were considered relevant for profiling examples—except for `payload-regex`, which is a specialized element discussed in subsection 3.4.2.

3.2.3 Unit testing

Unit tests for both protocol parsers were written during their implementation and are located in the '`test/`' directory at the root of the project.

These tests are designed to verify the proper functionality of the parser functions. Each function is tested by providing it with hexstring representations of packet examples. The test then compares the actual results returned by the functions when parsing those packets with their expected values, ensuring correctness.

The packet examples were obtained either from actual traffic traces or generated using Scapy [39].

During the execution of tests, assert statements are used to verify that the values returned by the parsers functions actually correspond to the expected values for each field deemed relevant for the packets. If the all the field values found by the parser match the expected ones, then it means the parser works as intended.

For example, to verify the correctness of the TLS parsing function, we use unit tests with various TLS messages, such as Client Hello, Server Hello, and Client Key Exchange. Each test provides a hexstring representation of a TLS packet and defines expected values for relevant fields, such as TLS content type, TLS version, and handshake type. The parser processes the packet, and assert statements compare its output with the expected values, ensuring accuracy.

3.3 TLS protocol addition

This section outlines the modifications implemented in the firewall code to enable support for the TLS protocol.

3.3.1 Profiles extension

Methodology

As part of the work for this thesis, we were provided with PCAP traces of actual traffic from and to the devices of the experimental setup used at the time of the publication of the Smart Home Firewall article [2].

By using the Wireshark tool [40], it is possible to filter TLS traffic from the traces of the devices that use this protocol, and even to isolate specific TLS conversations. From them it was then possible to manually write into the profiles of the concerned devices the content of those TLS exchanges as interactions (series of policies).

Interaction details

The appendix of this document provides YAML templates that describe possible TLS interactions. They illustrate how a TLS interaction could typically look like into the profile of a device. They are given in the appendix rather than directly here because of how much place they take due to the huge numbers of messages exchanged during a TLS communication. Note that only TLS messages are included in the presented TLS interactions. In reality, the TCP messages exchanged in between should also be part of the device's profiles for the firewall to correctly handle actual traffic. However, these TCP messages were ignored in our templates, as we only want to illustrate how the different steps of TLS handshake and application data exchange messages would be written into devices' profiles. The demonstration of the firewall's handling of TCP messages is not required either—this was already covered in the previous firewall implementation. That said, it is worth highlighting how much space TLS handling occupies in device profiles due to the large number of messages exchanged during TLS communication.

These templates cover three possible scenarios: the full handshake for TLS 1.2 and earlier versions, the abbreviated handshake, and the TLS 1.3 handshake. We emphasize that these YAML templates are not actual files used by the program, but only serve as structural examples of how an TLS

interaction would look like into a profile. As it will be discussed in more details below, those scenarios represent ideal (or even "idealized") TLS interactions, and are actually quite optimistic in regards to actual TLS interactions.

Listing 3.1 provides an excerpt of the interaction template for a TLS 1.2 exchange, showcasing the beginning of a full handshake between two devices. The device of the profile can be referenced using the keyword `self`, while relevant information about the other device can be specified using the `include` directive.

```
tls-device1-device2:
  tls-device1-device2-client-hello:
    protocols:
      tls:
        handshake-type: 1 # handshake type is CLIENT HELLO (1)
        tls-version: 1.2
      tcp:
        dst-port: 443
      ipv4:
        # typically 'self' or a server's hostname
        src: ... # (device1)
        dst: ... # (device2)

  tls-device2-device1-server-hello:
    protocols:
      tls:
        handshake-type: 2 # handshake type is SERVER HELLO (2)
        tls-version: 1.2
      tcp:
        src-port: 443
      ipv4:
        src : ...
        dst : ...

  [...]
```

Listing 3.1: TLS 1.2 full handshake - interaction beginning

Optional messages within the TLS handshake can be marked as such in the interaction by setting the packet `rate` of their policy to 0, which indicates an infinite rate, in the `stats` field.

This makes the policy periodic. As such the policy stays active until a packet matching the next policy is observed. If the optional message does not appear during the exchange, the policy deactivates and transitions to the

next one as soon as the next packet in the exchange is received and matched.

Listing 3.2 provides an example of an optional TLS message, showing an excerpt of the interaction that includes the policy for the optional Certificate message sent by a TLS client.

```
tls-device1-device2:
  [...]

  tls-device2-device1-certificate:
    protocols:
      tls:
        # handshake type is CERTIFICATE (11)
        handshake-type: 11
        tls-version: 1.2
      tcp:
        src-port: 443
      ipv4:
        dst: ...
        src: ...
    stats:
        rate: 0 # Certificate is optional

    [...]
```

Listing 3.2: TLS 1.2 full handshake - optional certificate field

The policy corresponding to the last step of a TLS interaction must be indicated as being bidirectional since the messages goes both way during the application data exchange.

A rate can also be specified for this policy to indicate a limit to the number of packets transmitted per second. An conservative value would typically be 1.2 times the highest rate observed in actual traffic, providing a margin without being excessive.

Listing 3.3 below gives an excerpt of the last policy of the TLS 1.2 interaction template.

```
tls-device1-device2:
    [...]

    tls-device1-device2-encrypted-data-exchange:
    protocols:
      tls:
        # content type is APPLICATION DATA (23)
        handshake-type: 23
        tls-version: "1.2"
```

```
      tcp:
        src-port: 443
      ipv4:
        src: ...
        dst: ...
    bidirectionnal: true
    stats:
      # highest rate in the pcap trace * 1.2
      rate: ...

    [...]
```

Listing 3.3: TLS 1.2 full handshake - application data field

It may be worth revisiting the selection of TLS fields considered by the firewall. Although the number of fields may seem small — and it is — limiting the number of variables to match on was deemed appropriate for this work. This approach ensures a focus on general TLS interactions that can be easily transferred across different device profiles.

However, in a different context, where it would be up to the device manufacturer to profile its devices, it would be easy to imagine choosing to consider more fields, since the manufacturer would have more concrete knowledge of how its devices communicate.

For example, we decided not to match on Cipher Suites and Compression Methods data exchanged during the TLS handshakes between devices. These values are considered too volatile and subject to change for various reasons, such as device updates. However, if we assume the manufacturer takes responsibility for sharing its device profiles, it is reasonable to expect that they would update these profiles alongside updates to the devices themselves, and thus it could become appropriate to consider such fields during the filtering.

One last thing to say about TLS profiling : it happens sometimes that some TLS records are sent into the same network packet. This must also be considered into the writing of TLS interactions in the profiles of devices. This can be done by mentioning a list of handshake types in the profile. An example is shown on Listing 3.4.

```
tls-device1-device2:
    [...]

    tls-device1-device2-certificate--server-key-exchange--server-hello-done:
      protocols:
        tls:
```

```
          handshake-type: 11, 12, 14 # certificate, server key exchange,
                                     server hello done
          tls-version: ...
        tcp: ...
        ipv4: ...
```

Listing 3.4: TLS combined messages

Note also that when a Change Cipher Spec message is sent in a record along other messages, we have to write the value of its content type as an handshake type in the device's profile. The tool has been extended to handle this special case and treats it as content type when converting profiles into NFQueue codes.

To simply specify a `content-type` field along one or more `handshake-type` fields would not be a proper solution as the firewall requires to know in which order the messages are received in order to handle them correctly.

One might assume that Change Cipher Spec messages could be confused with TLS Finished messages, given that both share the same numerical value (20) for their respective content type and handshake type fields. However, this is not the case, as the handshake headers of TLS Finished messages are encrypted. Consequently, the actual value recorded as the handshake type in the profile is effectively random for Finished messages. This distinction prevents any ambiguity when processing these messages. A more detailed analysis of this issue is provided in Section 3.3.2, particularly in the discussion on encrypted handshake messages.

3.3.2 TLS payload parsing

Following the same approach as for the previously protocols implemented for this firewall, the code for the TLS parser was written so as to parse TLS payloads in order to extract their most relevant information. It is obtained by looking at the right offset on the payloads. It can then be put into a C structure in order to hold it for later use by the firewall during the filtering. The code for this C structure is given by the Listing 3.5 just below.

```
/**
 * Abstraction of a TLS message
 *
 * Only relevant fields for the IoT firewall are included.
*/
typedef struct tls_message {
```

```
7     tls_version_t tls_version;
8     uint16_t length;
9
10    /* RECORD HEADER */
11    tls_content_type_t content_type;
12
13    /* HANDSHAKE HEADER */
14    tls_handshake_type_t handshake_type;
15    bool session_id_present;
16 } tls_message_t;
```

Listing 3.5: tls_message struct definition

Each field of the structure corresponds to a relevant field of the TLS payload that needs to be checked to know whether or not the analyzed packet matches the behavior described in the profile. Consequently some optional fields must be present in order to handle the many possible forms that a TLS message can take.

In particular, some fields are required for when the TLS payload corresponds to a TLS handshake message because the handshake type needs to be systematically considered during the handshake part of a TLS interaction: indeed, for any device the value of the handshake type is always to be provided in its profile in the event of a TLS interaction. This is an essential requirement because the handshake type has to be taken into account to be sure that the packets are arriving in the expected order.

At the start of a TLS handshake, it is also necessary to look at the content of the session ID field of the Handshake Header since a session ID can be provided to resume a previous session : this shortens the handshake by omitting certain steps that would otherwise be necessary. Thus when the client and the server provide an identical Session ID, the interaction differs because an abbreviated handshake will take place instead of a full one. If the server does not have the given Session ID in its cache, a full handshake is performed as if no session had been established.

It is also necessary to save the payload length to avoid issues during parsing, especially when identifying the actual TLS version used in the payload. Indeed, the way TLS was implemented makes it such that the parsing of the TLS version is not so direct and trivial. This is explained more exhaustively in the following subsections that discuss the TLS version identification and the parsing of the encrypted handshake Finished message.

An easy way to determine the payload length is by extracting the content of its fourth and fifth bytes, which correspond to the length field in the TLS Record Header. However, this field only indicates the number of bytes following the Record Header, not the total payload length. To obtain the full payload length, we thus need to add the size of the Record Layer (which is five bytes) to the extracted value.

TLS version identification

It might be naively assumed that the TLS version would simply be indicated in the header of the TLS message, particularly in its version field. Alas, this is not the case and a few things actually have to be taken into consideration.

First of all, for TLS version 1.1 and above, the TLS version indicated in the version field of the Record Header for Client Hello messages does not indicate the actual TLS version used during the handshake. Instead it is always set to TLS 1.0 for implementation reasons.

The Go programming language's crypto/tls library includes a comment explaining that some TLS servers fail if the version in the record header is set higher than TLS 1.0 for the initial ClientHello message, which is why the version is defaulted to TLS 1.0 in such cases. [41]

```
if vers == 0 {
    // Some TLS servers fail if the record version is
    // greater than TLS 1.0 for the initial ClientHello.
    vers = VersionTLS10
}
```

The interoperability of newer TLS versions with servers using older versions is also addressed in the RFCs for each TLS version, particularly in the appendices, starting from TLS 1.1 [33, Appendix E] [34, Appendix E] [35, Appendix D].

In the RFC for TLS 1.2 [34, Appendix E] it is written :

> TLS clients that wish to negotiate with older servers MAY send any value {03,XX} as the record layer version number. Typical values would be {03,00}, the lowest version number supported by the client, and the value of ClientHello.client_version.

This practice is a backward compatibility measure designed to ensure that older servers that only support earlier versions of TLS (or SSL) can correctly

interpret and respond to Client Hello messages from modern clients.

When implementing the parser, we therefore must consider the TLS version indicated in the version field of the Handshake Header for each TLS Client Hello message, rather than the version in the Record Header. This field specifies the actual version used for communication in both TLS 1.1 and TLS 1.2.

Figure 3.6 illustrates the structure of the first part of a TLS Client Hello, highlighting the concept just discussed.

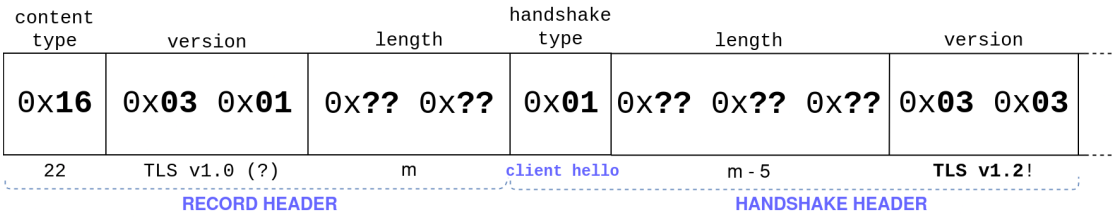

Figure 3.6: TLS 1.2 Client Hello payload first bytes

However, our approach also needs to be revised for TLS 1.3. In TLS 1.3 communication, the version field in the Handshake Header is always set to 1.2 to maintain compatibility with systems that haven't been updated to support TLS 1.3. The version negotiation mechanism allows the party using the latest version to fall back to an older version if the other party only supports older versions. Of course, this requires the party supporting the newest version to also support older ones, which is typically the case, as discussed in a previous section (see 2.4.1).

To indicate that the communication is actually using TLS 1.3 (rather than the TLS 1.2 version specified in the handshake header), the TLS 1.3 version must be indicated through a protocol extension called the 'supported versions' extension, introduced specifically for TLS 1.3. If the communication uses TLS 1.3, this will be reflected in that extension. Parsing this information requires additional steps, as we need to loop through the various extensions to locate the 'supported versions' extension specifically.

To summarize the process of determining the actual TLS version used during a TLS exchange :

- we verify the version field in the Handshake Header of the payload,

rather than in the Record Header, whenever the TLS message is a Client Hello message.

- we verify whether a 'supported versions' extension is present to indicate a newer TLS version when the Handshake Header version is set to TLS 1.2.

TLS application data payload

It must be noted that to know whether TLS 1.3 is the protocol version used during the application data exchange, it would be required to maintain a connection state during the whole exchange.

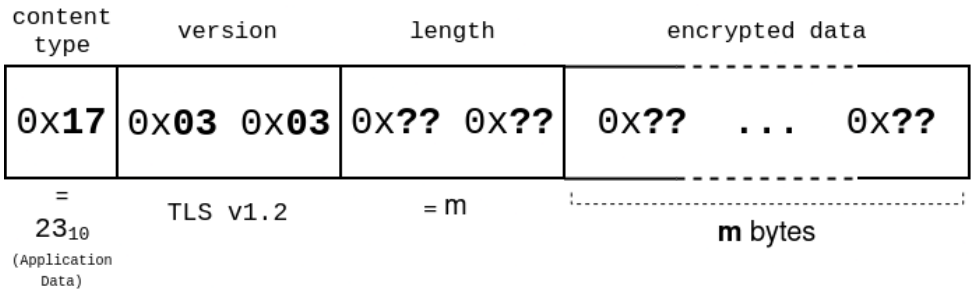

Figure 3.7: TLS 1.{2-3} application data payload

Indeed, when using TLS 1.3 the version number in the Record Header is always set to 1.2 for backward compatibility; for this reason, it cannot be relied on to know the actual TLS version of the payload as a TLS 1.2 application data payload is indistinguishable from a TLS 1.3 application data payload. Figure 3.7 shows the structure of a TLS data application payload that corresponds both to TLS 1.2 and TLS 1.3 application data payloads.

In such situation, the only way to determine with certainty that the application data payload is using TLS 1.3 rather than TLS 1.2 would be to track the handshake process : if the handshake established a TLS 1.3 session, then all subsequent application data records could be assumed to be using TLS 1.3.

As the IoT firewall tool was built to handle every payload passing through the network independently from each other, implementing a new tracking feature will require too much modification to the tool and is out of scope in the context of this thesis.

Fortunately, such feature is not required for the good quality of the filtering as we can simply ignore the real version of TLS used during this step of TLS interactions and indicate that we rather want to match on the value "1.2" for the `tls-version` field at the data application exchange step of the interaction for any TLS 1.3 interaction that we write in the profile of our devices. This, however, has the drawback of making the profiling of TLS 1.3 communications inconsistent with the actual protocol exchange.

Algorithms description

The following code corresponds to the function called to parse a TLS payload. It creates a `tls_message_t` structure, populates it, and then eventually returns it.

It takes as an argument a pointer to the start of the TLS message's payload. When this function is called in a user-space NFQueue code, an offset is applied the payload to skip its layer 3 and 4 headers.

```
1  tls_message_t tls_parse_message(uint8_t *data) {
2      tls_message_t message;
3
4      message.content_type = get_tls_content_type(data);
5      message.tls_version = get_tls_version(data);
6      message.length = get_record_length(data) + 5;
7
8      if (message.content_type == TLS_HANDSHAKE) {
9          message.handshake_type = get_handshake_type(data);
10         if (message.handshake_type == TLS_CLIENT_HELLO
               || message.handshake_type == TLS_SERVER_HELLO)
11             message.session_id_present = is_session_id_present(data);
12         else
13             message.session_id_present = NULL;
14
15         if (message.handshake_type == TLS_CLIENT_HELLO)
16            message.tls_version = get_tls_version_from_handshake_header(data);
17
18         if (message.tls_version == TLS_VERSION_1_2) {
19             if (message.handshake_type == TLS_CLIENT_HELLO
                 || message.handshake_type == TLS_SERVER_HELLO)
20                 message.tls_version = parse_tls_1_3(data,
                                                       message.handshake_type,
                                                       message.length); }
21     } else {
22         message.handshake_type = -1;
```

```
23          message.session_id_present = NULL; }
24      return message; }
```

Listing 3.6: TLS parsing function

The function starts by extracting the values for the content type and the TLS version in the Record Header. This is straightforwardly achieved by looking at the appropriate bytes of the payload. To do so, all that have to be done is to apply the correct offset to point to the appropriate field of the payload.

The length of the payload is acquired also similarly but since the length field of the Record Header only indicates the number of bytes that follows the Record Header, the length of the Record Header itself must be added to it to get the full length of the payload.

If the TLS message corresponds to a TLS handshake, further parsing is required. Otherwise, default values need to populate the fields of the Handshake Header, as they cannot be left empty due to the syntactical requirements of the C language.

When dealing with a message of the `handshake` content type, the first step is to determine the handshake type. Again, it is pretty straightforward to get it by simply applying an offset to the payload to retrieve the appropriate value at the right byte.

After what, the next step is to get the value of the Session ID field of the Handshake Header in case where the message is either a Client Hello or a Server Hello.

When the message is a Client Hello, it is also required to update the value for the TLS version by extracting the TLS version in the Handshake Header as the TLS version in the Record Header is hardcoded to TLS 1.0.

When the version obtained is TLS 1.2 it is also necessary to check whether this given version is genuine or not since TLS 1.3 disguises itself as a TLS 1.2. In such situation, we call a function specially written for this task called `parse_tls_1_3`. It obviously takes as argument the payload to parse but also the handshake type because the offset to apply differs from a Client Hello message to a Server Hello message.

The message's length has also to be taken in argument in order to prevent issues. This is especially required in the context of parsing the last message handshake exchanged between the client and the server just before they

start exchanging application data. This matter is discussed in depth in the following subsection 3.3.2.

It is assumed that this function `parse_tls_1_3` is only called when a payload is identified as supporting at least version 1.2 of TLS. This does not follow best coding practices but it is most convenient within the framework of this research project.

The TLS 1.3 parsing function starts by applying offsets to the payload until it reaches the extensions part of the payload. Its first instructions are given in the snippet 3.7.

Client Hello payloads and Server Hello payloads share similar parts that need to be skipped during the parsing. They are the following :

- the Record Header
- the Handshake Header
- the {Client/Server} Version Header
- the {Client/Server} Random data
- the Session ID

In a Client Hello message, the Cipher Suites and Compression Methods data have variable lengths, depending on the client's supported options, and must be skipped accordingly.
If the client provides a Compression Method that is not "null", we can know for sure that the payload does not use TLS 1.3 since TLS 1.3 no longer allows compression.

In contrast, the server only provides during its Server Hello two bytes of data for its Cipher Suite and only one byte for its Compression Method.

```
1 tls_version_t parse_tls_1_3(uint8_t* data,
                              tls_handshake_type_t message_type,
2                             uint16_t length) {
3     size_t offset = 43; // skip initial part
4
5     u_int8_t session_id_length = data[offset];
6     offset += 1 + session_id_length;
7
```

```
8     if (offset + 1 >= length) return TLS_VERSION_1_2; // guardrail
9
10    if (message_type == TLS_CLIENT_HELLO) {
11
12        uint16_t cipher_suites_length = (uint16_t)(data[offset] << 8)
                                                  | data[offset + 1];
13        offset += 2 + cipher_suites_length;
14
15        uint8_t compression_methods_length = data[offset];
16        offset += 1;
17
18        // check that the only compression method is "null" (0x00)
19        if (compression_methods_length != 1 || data[offset] != 0x00)
20            return TLS_VERSION_1_2;
21        offset += 1;
22
23    } else if (message_type == TLS_SERVER_HELLO)
24        offset += 3; // Cipher Suite + Compression Method
25
26    if (offset + 1 >= length) return TLS_VERSION_1_2; // guardrail
27
28    const uint16_t extensions_length = (uint16_t) (data[offset] << 8)
                                                | data[offset + 1];
29    offset += 2;
30
31    // Parse each extension
32    const uint8_t *extensions = data + offset;
33    uint16_t extensions_remaining = extensions_length;
34    if (extensions_remaining >= length) return TLS_VERSION_1_2; // guardrail
35    while (extensions_remaining >= 4)
36        [...] /* loop over the different extensions */
37
38    return TLS_VERSION_1_2; }
```

Listing 3.7: TLS 1.3 parsing function snipped (initial offset)

Some conditional instructions are also written as guardrails to prevent any offset overflow : if somehow the offset exceeds the length of the payload, we can assume it does not use TLS 1.3 and return TLS 1.2 as the supported protocol version. These guardrails are present on lines 8, 26, and 34 of Listing 3.7.

When the extensions part of the payload is reached, we only have to iterate over the different extensions provided by the payload in search for the "supported versions" payload. This is made easy by the fact that every

extensions follow the same data structure, which is illustrated by Figure 3.8: first there are two bytes to identify the extension, then again two bytes to indicate the numbers of bytes of extension data that will follow, and at last the extension data themselves.

Code snippet 3.8 shows the code for this part of the parsing.

If the "supported versions" extension cannot be found, it means that we are not in the presence of a TLS 1.3 payload, thus we know the version to actually be 1.2.

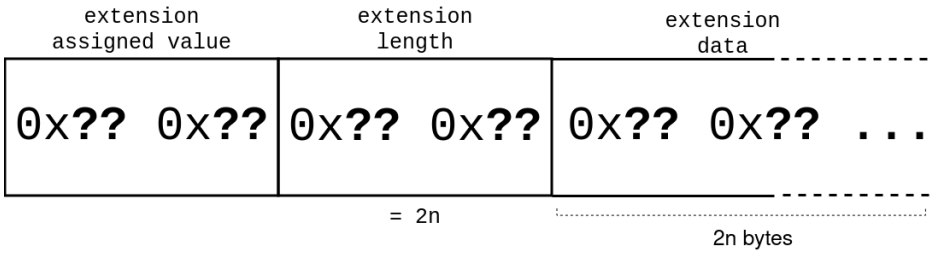

Figure 3.8: TLS extension data structure

```
tls_version_t parse_tls_1_3(...) {
    [...] /* initial offset */

    const uint8_t *extensions = data + offset;
    uint16_t extensions_remaining = extensions_length; [...]
    while (extensions_remaining >= 4) {
        uint16_t extension_type = (uint16_t) (extensions[0] << 8)
                                            | extensions[1];
        uint16_t extension_length = (uint16_t) (extensions[2] << 8)
                                            | extensions[3];

        if (extensions_remaining < 4 + extension_length)
            return TLS_VERSION_1_2; // TLS_SUPPORTED_VERSIONS not found

        // Check if this is the supported_versions extension
        if (extension_type == TLS_SUPPORTED_VERSIONS) {
            uint8_t supported_versions_length = (uint8_t) extension_length;
            if (message_type == TLS_CLIENT_HELLO) {
                for (int i = 0; i < supported_versions_length; i += 2) {
                    uint16_t version = (uint16_t) (extensions[5 + i << 8)
                                                | extensions[6 + i];
                    if (version == TLS_VERSION_1_3) return TLS_VERSION_1_3; }
            } else { // TLS_SERVER_HELLO
```

```
22                for (int i = 0; i < supported_versions_length; i += 2) {
23                    uint16_t version = (uint16_t) (extensions[4 + i] << 8)
                                                  | extensions[5 + i];
24                    if (version == TLS_VERSION_1_3) return TLS_VERSION_1_3;}}
25            return TLS_VERSION_1_2; } // TLS 1.3 not found
26        extensions += 4 + extension_length; // move to the next extension
27        extensions_remaining -= 4 + extension_length; }
28    return TLS_VERSION_1_2; } // TLS 1.3 not found
```

Listing 3.8: TLS 1.3 parsing function - iterating over the extensions

If the `supported versions` extension is found, it could be assumed that the protocol supports TLS 1.3 and immediately already returns it, but for the sake of rigor, we still iterate over the different supported versions included in the extension data just in case. This could also be useful in the future, in the potentiality of a new 1.4 version or higher.

Note however that one byte of offset must be added when dealing with a Client Hello with respect to a Server Hello because the client uses one more byte than the server to indicate the length of its `supported versions` extension. The reason for this is that flexibility, extensibility, and backward compatibility are the priority for the Client Hello message, while simplicity and efficiency prevail in the case of TLS server [35].

This is illustrated by the structures of their respective `supported versions` extension that are given by Figure 3.9 and Figure 3.10 .

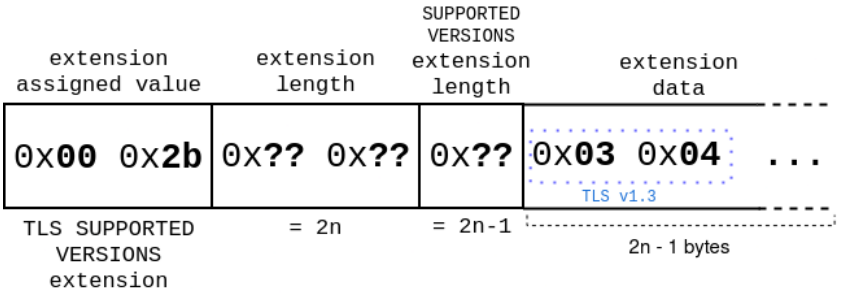

Figure 3.9: Client Hello "supported versions" extension

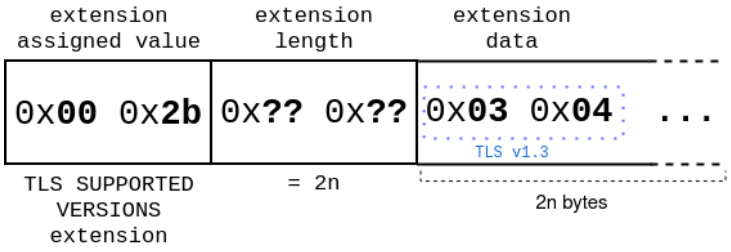

Figure 3.10: Server Hello "supported versions" extension

It must also be noted that when a client that does not support TLS 1.3 sends a TLS Client Hello message to a server that supports it, the server will not respond to it with a Server Hello that contains a "supported versions" extension and instead immediately includes the version of TLS that will be used for this connection.

Discussion about the encrypted handshake message case

We need to take a closer look at the very last handshake message sent just before the encrypted application data are exchanged between the client and the server. It is always sent immediately after a Change Cipher Spec message, on both client and server sides.

This message can be referred to as the encrypted handshake message or as the Finished message. Its purpose is to verify that the handshake was successful and not tampered with.

What forces us to take a closer look at this type of message is that they are constructed in such a way that they could be confused with other types of handshake messages.

Indeed, as it was the case when trying to figure out whether the version of a TLS application data payload really was 1.2 or on the contrary a TLS 1.3 application data payload under disguise like explained in a previous subsection 3.3.2, the only way to be certain that a Finished handshake message really is one would be to track the TLS exchange and assume that the handshake messages following the Change Cipher Spec messages are Finished messages. In absence of handshake process tracking, it can never be absolutely sure that we are in presence of a Finished handshake message when treating the payloads independently from each other.

The structure of a TLS Finished message payload is given on Figure 3.11.

As we can see, the content type of its Record Header has the value attributed for the handshake type.

However, contrarily to the other messages of handshake type, the byte that follows the bytes for the length field does not serve to indicate the type of the handshake message. Instead, it is a random byte just as the other bytes that follow and that are part of the encrypted data that are exchanged.

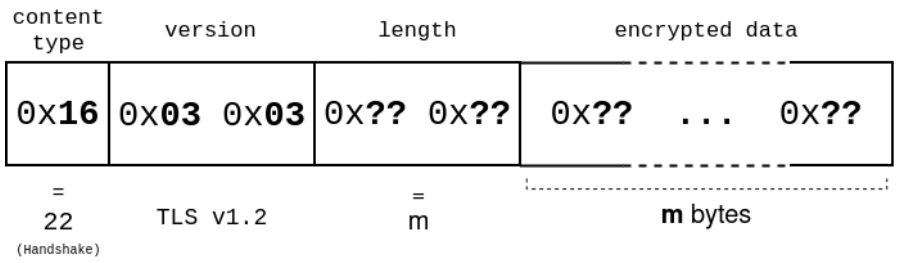

Figure 3.11: TLS encrypted handshake message payload structure

Such a structure can lead to troubles when the messages exchanged during the handshake are not tracked because such an encrypted handshake message can easily be taken for another type of handshake message. For example, let's imagine the byte at the beginning of the encrypted payload is equal to 1, it could be assumed that we are looking at a Client Hello message. This logic can be extended to every possible value for the handshake type field.

This could pose a problem when performing parsing operations, as each payload is processed separately from the others, without taking in consideration any tracking of any sort.

Thus, by applying the parsing algorithms presented in the previous subsection 3.3.2, some undefined behaviors could happen.

Let's take the example cited above of the Finished message that mimics a Client Hello message. In such case, the parsing function will assume the payload to be a TLS Client Hello message, and thus will call the function to retrieve the TLS version in the handshake header of the message a few bytes forward. This process is illustrated by Figure 3.12.

As a result, our parsing will have populated our structure with incorrect values that do not correspond to the reality of the considered payload.

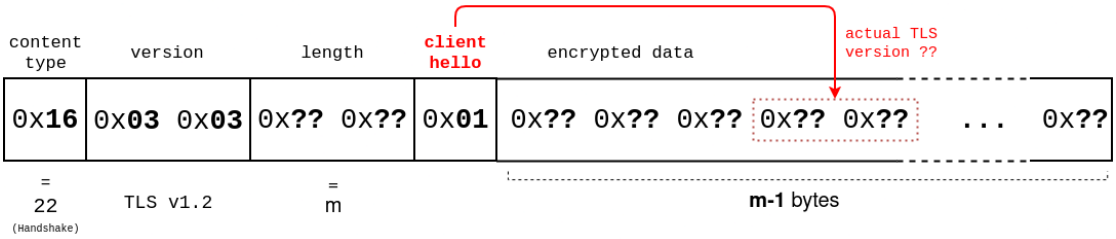

Figure 3.12: TLS encrypted message edge case (Client Hello mimicry)

The situation that deserves the most attention is when the TLS version in the Record Header is set to 1.2 and the first byte of the encrypted data is either 1 or 2, as the payload could pass as a Client Hello or a Server Hello respectively, and thus precipitates the call to the TLS 1.3 parsing function.

The probability of this function being called when the version in the header is 1.2 is quite high as there is approximately 2 chances on 255 for the first byte of the encrypted data to be either 1 or 2. However, the probability for the encrypted random bytes to take a configuration such that this function identifies the payload as being TLS 1.3 is so little that this possibility can just be ignored.

This function was also written as to prevent an offset overflow on the payload : the many conditional instructions in the function serve as railguards to stop the parsing process as soon as a value aberrantly large is located at sensible locations.

For example, they can prevent the function from running a loop for a long delay in the instance where large values end up at the position that indicates the Extensions Length. Such an edge case is illustrated by Figure 3.13.

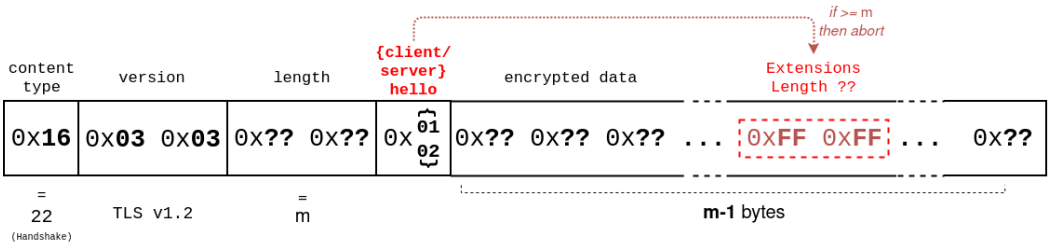

Figure 3.13: TLS Finished message overflow edge case

Obviously, such guardrails cannot prevent the parser from saving values that don't make sense in the context of a handshake into our TLS structure. Fortunately, such follies produced by the parser at this stage of the handshake

will have no impact on the filtering quality of the IoT firewall since the values of those fields can simply be ignored at this step of any TLS interaction.

Handling combined TLS messages

Instead of relying solely on the previously described parsing function, which is designed for handling individual TLS messages, a more comprehensive approach is required. Since multiple TLS messages may be transmitted within the same network packet, a packet-level parsing function must be employed.

To achieve this, the entire packet must be parsed to extract all embedded TLS messages. A linked list can be used to store these messages while maintaining the order in which they were sent. This structure allows for the sequential processing of TLS messages using the dedicated TLS message parsing function. The structure definition is provided in Listing 3.9.

```
typedef struct tls_message_list {
    tls_message_t message;
    struct tls_message_list* next;
} tls_message_list_t;
```

Listing 3.9: tls_message_list struct definition

Furthermore, another structure, shown in Listing 3.10, incorporates the TLS message linked list. This additional structure is necessary because network packets may be fragmented. To handle such cases, buffers are used during packet reception to ensure proper message reconstruction and processing.

```
typedef struct tls_packet {
    tls_message_list_t* messages;
    size_t record_count;
    size_t total_length;
} tls_packet_t;
```

Listing 3.10: tls_packet struct definition

3.4 MQTT protocol addition

This section describes the changes made to the firewall code to support the MQTT protocol. As an IoT-specific protocol, integrating MQTT into the firewall system was relatively straightforward, especially when compared to the TLS protocol. The process did not involve handling complex special cases or intricate design challenges.

3.4.1 Profiles creation and traffic simulation

We were not provided with PCAP traces of actual MQTT traffic, as none of the devices in the experimental setup for the Smart Home Firewall article [2] utilized this protocol. Consequently, we simulated MQTT traffic using a custom program and saved the output as new, original PCAP traces.

For this simulation, we used the Scapy library [39], an interactive Python-based toolkit for packet manipulation. Scapy allows users to create, decode, and analyze packets for various network protocols. Its features include packet generation, network traffic capture, and matching requests with responses, making it an invaluable tool for network analysis and testing.

The development of Python scripts with Scapy to generate MQTT-specific PCAP traces was further supported by the Claude AI assistant tool [42].

Their source code is available on a GitHub repository specifically created for this MQTT traffic simulation code.

To serve as the basis for creating profiles of fictional devices using MQTT, we first generated a traffic trace representing legitimate MQTT communications between four clients interacting through a broker.

Two clients act as publishers, identified by the Client IDs `temp_sensor` and `humidity_sensor`. They publish data to the topics `temperature` and `humidity`, respectively. The other two clients, with Client IDs `temp_monitor` and `humidity_monitor`, function as subscribers. Each subscriber is subscribed to the topic that corresponds to its Client ID.

These devices form a network similar to that depicted in Figure 2.4, which was presented in the MQTT introduction in section 2.5.1.

We then created concise profiles for the client publishers, as these devices were of primary interest for controlling behavior. The profiles for the MQTT temperature and humidity sensors are located in the ’`temperature-sensor`’ and ’`humidity-sensor`’ folders, respectively, inside the ’`devices`’ folder at the root of our project source code. An extract of the `temperature-sensor`’s profile is given by Listing 3.11.

```
device-info:
  name: temperature-sensor
  [...]

single-policies:
  connect-command:
    protocols:
```

```
      mqtt:
        packet-type: 1
        client-id-length: 11
        client-id: "temp_sensor"
        [...]
      tcp:
        [...]
      ipv4:
        [...]

  temperature-publishing:
    protocols:
      mqtt:
        packet-type: 3
        topic-name: "temperature"
        payload-length: 7
      tcp:
        [...]
      ipv4:
        [...]
```

Listing 3.11: temperature-sensor device's profile (extract)

This approach was motivated, in part, by the work of Uroz *et al.* [24], which highlighted how MQTT publish messages can be exploited for data exfiltration attacks, even in the presence of the MUD standard. Our objective is thus to demonstrate how integrating MQTT into our new firewall tool can effectively prevent such attacks, unlike the MUD standard. Indeed MUD can only perform matching on lower-layer network protocols and thus cannot analyze the content of MQTT packets.

While we could have explored more imaginative scenarios to consider specific features of MQTT or leverage all the new features of our firewall – such as devising particular interactions between devices for the firewall to analyze – we chose to focus on the approach outlined above, as our primary goal is to demonstrate the proper functioning of the firewall with MQTT. For the same reasons, we also do not consider TCP packets exchanged between MQTT packets in our profiling.

3.4.2 MQTT payload parsing

The development of the MQTT parser followed the same approach as for TLS and other previously implemented protocols. MQTT payloads are parsed to extract relevant values based on their offsets, which are then stored in a C

data structure for later use by the firewall during filtering.

Listing 3.12 gives an excerpt of the code for the MQTT structure in C.

```c
/**
 * Abstraction of an MQTT message
 */
typedef struct mqtt_message {
    mqtt_version_t mqtt_version;
    uint32_t remaining_length;

    /* Fixed Header */
    mqtt_packet_type_t packet_type;
    bool dup_flag;
    uint8_t qos_level;
    bool retain_flag;

    /* Variable Header */ [...]
    /* CONNECT specific fields */ [...]
    /* SUBSCRIBE specific fields */ [...]

    /* PUBLISH specific fields */
    uint16_t topic_length;
    char topic_name[MQTT_MAX_TOPIC_LENGTH];
    uint16_t payload_length;
    uint8_t payload[MQTT_MAX_PAYLOAD_LENGTH];
    uint16_t packet_identifier;  // Only present for QoS > 0

} mqtt_message_t;
```

Listing 3.12: MQTT_message struct definition

Although our profiling is limited to MQTT publish messages, and only certain fields are included in the profiles of our examples of devices, the structure allows for the inclusion of many other fields. This flexibility could enable the extension of the parser to support more complex profiles without requiring additional code modifications.

This comprehensive implementation was made possible not only by the ease of parsing MQTT messages but also by the use of the Claude AI tool [42], which significantly simplified the code-writing process.

The parsing function is straightforward: it scans the payload and extracts field values based on their respective offsets. Sub-functions process the payload according to the message type. It is also possible to encode range of values instead of an exact value for certain relevant fields in the profile.

Unlike for the integration of TLS to the project, there were no complex design aspects to consider.

Payload regex

Device profiles now support encoding regular expressions (regex) to validate MQTT payloads. This is achieved by including a `payload-regex` field within the corresponding section of the YAML profile. This capability is especially useful for MQTT Publish messages, where payload content needs to match a specific format.

For example, a regex like `-?[0-9]?[0-9]\\\\.[0-9]°[CF]` can be used to ensure temperature values follow the expected `{floating number}°C/F` format. Any payload that deviates from this format can then be discarded accordingly. Multiple backslashes are necessary due to how the tool handles escaping across various code files.

This feature enhances security by preventing malformed or malicious payloads, such as those used for data exfiltration.

For the two fictional devices used in our implementation, the tool has been extended to automatically apply regex validation to temperature and humidity topics. This functionality could potentially be expanded to other topics as well. Related extract of this function is shown on Listing

Implementing this feature required modifying the `parse` function of the MQTT Python class to handle the `temperature` and `humidity` topics specifically, by embedding the appropriate regular expression checks. These modifications ensure that the generated NFQueue source code includes the necessary logic to validate payload formats accordingly. Listing 3.13 illustrates the relevant code extract.

```
def parse(self, is_backward: bool = False, initiator: str = "src") -> dict:
    [...]
    # Handle MQTT topic name
    topic_name = self.protocol_data.get("topic-name", None)
    if topic_name is not None:
        string = 'strcmp(mqtt_message.topic_name, "{}") == 0'
        if topic_name == "temperature":
            string += "\n \t \t&& \n \t \tcheck_payload_regex(mqtt_message.
    payload, strlen((char *)mqtt_message.payload),\
\"-?[0-9]?[0-9]\\\\.[0-9]°[CF]\") == 1" # floating point number with 1
    decimal place and degree celsius or farenheit
        elif topic_name == "humidity":
```

```
11                 string += "\n \t \t&& \n \t \tcheck_payload_regex(
    mqtt_message.payload, strlen((char *)mqtt_message.payload),\
12 \"[0-9]?[0-9]\\\\.[0-9]%\") == 1" # positive floating point number with 1
    decimal place and %
13         rule = {"forward": string}
14         self.add_field("topic-name", rule, is_backward)
15     [...]
```

Listing 3.13: MQTT Python class - parse function (extract)

Chapter 4

Evaluation

This chapter evaluates the extended firewall using a Vagrant-based test setup. We analyze its ability to process MQTT and TLS traffic, ensuring correct packet handling with minimal performance overhead.

4.1 Experimental setup

The tests conducted in this thesis to verify both the functionality and efficiency of our implementation were performed in a simulated environment rather than under real-life conditions.

We simulated the operation of our firewall using a Vagrant virtual machine simulating the operation of a router, on which we ran our NFTables scripts and launched our NFQueue executables.

Source code

We made the source code for this project available on a GitHub repository.

Vagrant file

The specifications of the Vagrant virtual machine are detailed in Table 4.1.

Host hardware specifications

As the capabilities of the host machine can have an influence on the results obtained and might impact the performance and environment of the tests, they are also shared in this section.

Configuration	Details
Base OS	Ubuntu 22.04 (`ubuntu/jammy64`)
Memory	4096 MB
CPUs	2
Network	Private network with DHCP (`vboxnet0`)
	Interface 1 : NAT, Interface 2 : Host-only `vboxnet0`
Provisioning Script	Installs dependencies for IoT firewall testing

Table 4.1: Vagrant configurations setup details

The tests were carried out on a small refurbished laptop a few years old.

The host machine is equipped with an Intel(R) Core(TM) i5-7200U processor with x86_64 architecture. The CPU has 2 physical cores and 4 threads, running at frequencies between 400 MHz and 3.10 GHz. It features L1, L2, and L3 cache sizes of 64 KiB, 512 KiB, and 3 MiB, respectively. Virtualization is supported via VT-x, and the machine operates on a single NUMA node for memory access.

These specifications are summarized in Table 4.2.

Component	Specification
CPU Architecture	x86_64
CPU Model	Intel(R) Core(TM) i5-7200U @ 2.50GHz
Cores/Threads	2 cores, 4 threads
CPU Frequency	400 MHz (min), 3.10 GHz (max)
L1 Cache	64 KiB (Data) + 64 KiB (Instruction) per core
L2 Cache	512 KiB per core
L3 Cache	3 MiB shared
Virtualization	VT-x supported
NUMA Nodes	1 (CPUs 0-3 on node0)

Table 4.2: Host computer specifications

4.2 Methodology

This section explains how we assessed the performance of our extended firewall.

4.2.1 Vagrant test set-up

To test our extended firewall, we first generate NFTables scripts and NFQueue C source code for the relevant devices. The devices used to test the firewall's extension to the MQTT protocol were the fictional `temperature-sensor` and `humidity-sensor`. Additionally, a plug [43] referenced in the *Supervising Smart Home Interaction* article [2] was used to test the extension to the TLS protocol.

This is done by running the script `translator.py` with the appropriate arguments. This task has been automated for all devices thanks to the bash script `translate_profiles.sh` that we just need to run. In the context of tests performed on the Vagrant machine, we must set the flag `'-t'`. The flag `'-l CSV'` can also be used to produce log files.

We then need to compile the NFQueue source code. We can do so by running the bash script `build.sh` at the root of the project code.

At this point, we are ready to conduct tests on our Vagrant machine.

4.2.2 Tests execution

Afterward, we test our extended firewall by first activating the NFTables scripts, then executing the NFQueue binaries, and finally replaying relevant PCAP traces using Tcpreplay [44] toward the Vagrant machine's interface while observing the results. The traces used for testing the MQTT protocol were simulated with Scapy [39], as explained in Section 3.4.1. For the TLS protocol, we used a trace captured from the TP-Link Plug device's traffic [43], which was recorded during the writing of the *Supervising Smart Home Interaction* article [2].

With the logging flag set, we can also redirect the output of the NFQueue executable to log files to analyze the results later on.

To evaluate efficiency, we modify the NFQueue code by removing the content of the callback functions related to the targeted protocols, ensuring that the packets are always accepted without processing. We then replay the traces as described above and compare execution times.

4.3 Results

We present the results we got after implementing both protocols.

4.3.1 MQTT results

We tested our extended firewall by replaying the different PCAP traces generated through our simulation. These traces, along with the simulation source code, are available in the MQTT simulation GitHub repository. To focus exclusively on MQTT traffic, we filtered out TCP packets, as they were not considered in our profiling.

Three different test cases were examined: (1) a trace containing only legitimate MQTT packets, (2) a trace containing only maliciously crafted packets, and (3) a trace combining both legitimate and malicious traffic. The malicious packets were designed to simulate data exfiltration attacks and were assigned the same IP addresses as the temperature sensor device to ensure they were processed by NFQueue.

The results of these tests confirmed the effectiveness of our firewall. In every instance, legitimate packets were systematically accepted, while malicious packets were correctly dropped.

Additionally, the correct handling of regex-based payload name filtering was verified.

Efficiency analysis

The efficiency of our firewall was assessed using the methodology described in Section 4.2.2. To ensure statistical significance, we repeated the experiments ten times and recorded the results in log files, which are available in a dedicated GitHub repository. The test cases included evaluations for both the `temperature-sensor` and `humidity-sensor` devices. For each device, we conducted two runs: one with parsing activated and one without it. The processing times obtained across these test cases are summarized in Figure 4.1.

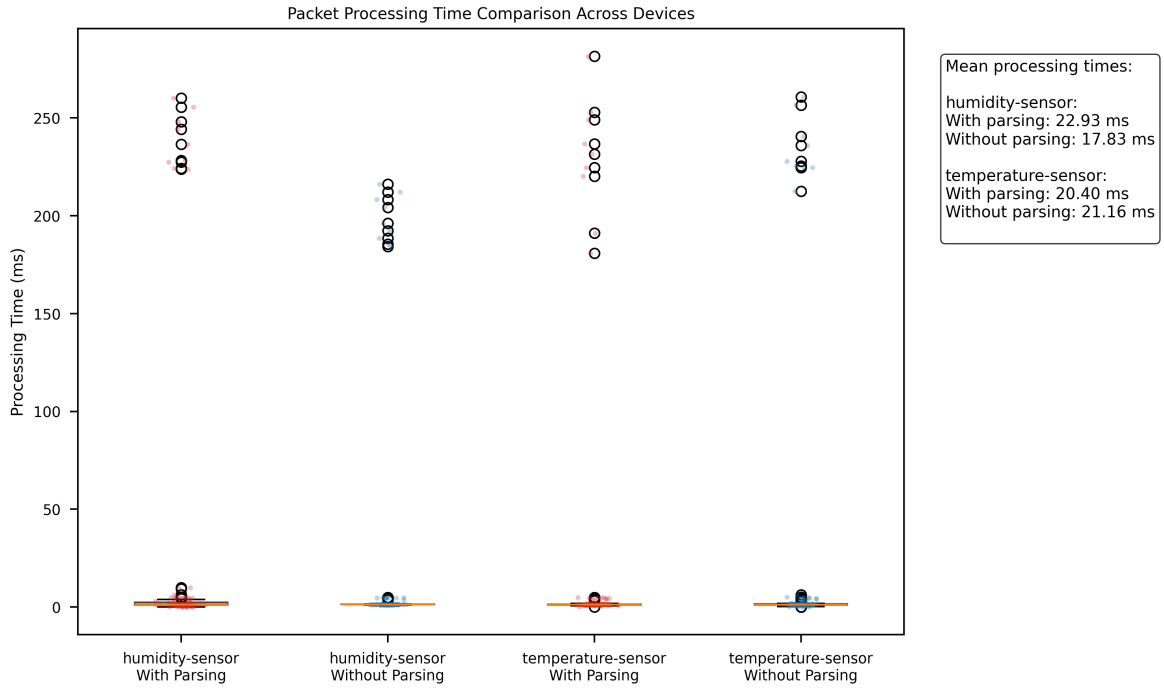

Figure 4.1: MQTT processing time comparison

As expected, the processing time for the humidity sensor was slightly higher when parsing was enabled, indicating a small computational overhead. However, an unexpected result was observed for the temperature sensor, where processing time appeared to be lower when parsing was enabled. Several factors could explain this anomaly, including system scheduling variations, memory management optimizations, or network stack behavior.

Despite these variations, the overall results indicate that the additional processing introduced by MQTT parsing does not impose a significant overhead on system performance.

In general, the processing time remained close to zero for most packets, although several outliers were observed in the boxplot. These outliers are unlikely to be attributed to the internal processing mechanisms of the firewall. Specifically, the packets with higher processing times occurred at regular intervals and consistently corresponded to `connect-command` packets. This suggests that the increased latency is not due to the firewall's packet processing but rather to the re-execution of the TcpReplay tool [44]. Given that the `connect-command` packet is the first packet in the replayed trace, the observed delay is likely attributable to the overhead introduced by the replay mechanism itself. The regularity of these spikes is well illustrated in

Figure C.1 in Appendix C, which presents a packet processing timeline.

4.3.2 Results for TLS

We demonstrated the firewall's ability to handle TLS packets by isolating a specific TLS communication and filtering out TCP packets to only consider the TLS protocol. This isolated communication originates from real-traffic captured from the TPLink-Plug device (TP-Link HS110 model) [43], as reported in the Supervising Smart Home article [2]. The isolated trace is available on a dedicated GitHub repository.

The TLS interaction was incorporated into the TPLink-Plug profile to assess the firewall's ability to process such traffic effectively. This evaluation focused on a TLS 1.2 handshake and encrypted data exchange between the plug and a physical interface. The complete interaction, as written in the extended profile, is provided in Appendix B. The test results confirmed that the firewall successfully processes TLS packets and correctly discards duplicate ones.

However, the need to manually "cherry-pick" a specific TLS interaction presents a significant drawback. Because TLS encrypts data, defining device profiles requires incorporating random values into the profiles, which contradicts the intended user-friendly design of the firewall tool. Additionally, this approach results in significantly larger profiles to accommodate TLS communications. Despite this limitation, our results demonstrate that extending the firewall to handle TLS traffic is feasible but cumbersome.

Future work could explore the development of an automated tool capable of generating profiles directly from legitimate traffic traces, similar to previous work done for the MUD standard [45].

Efficiency Analysis

Efficiency was evaluated using a methodology similar to that employed for MQTT tests, with the experiments repeated multiple times for statistical significance. Log files documenting the test results can be accessed in the dedicated GitHub repository.

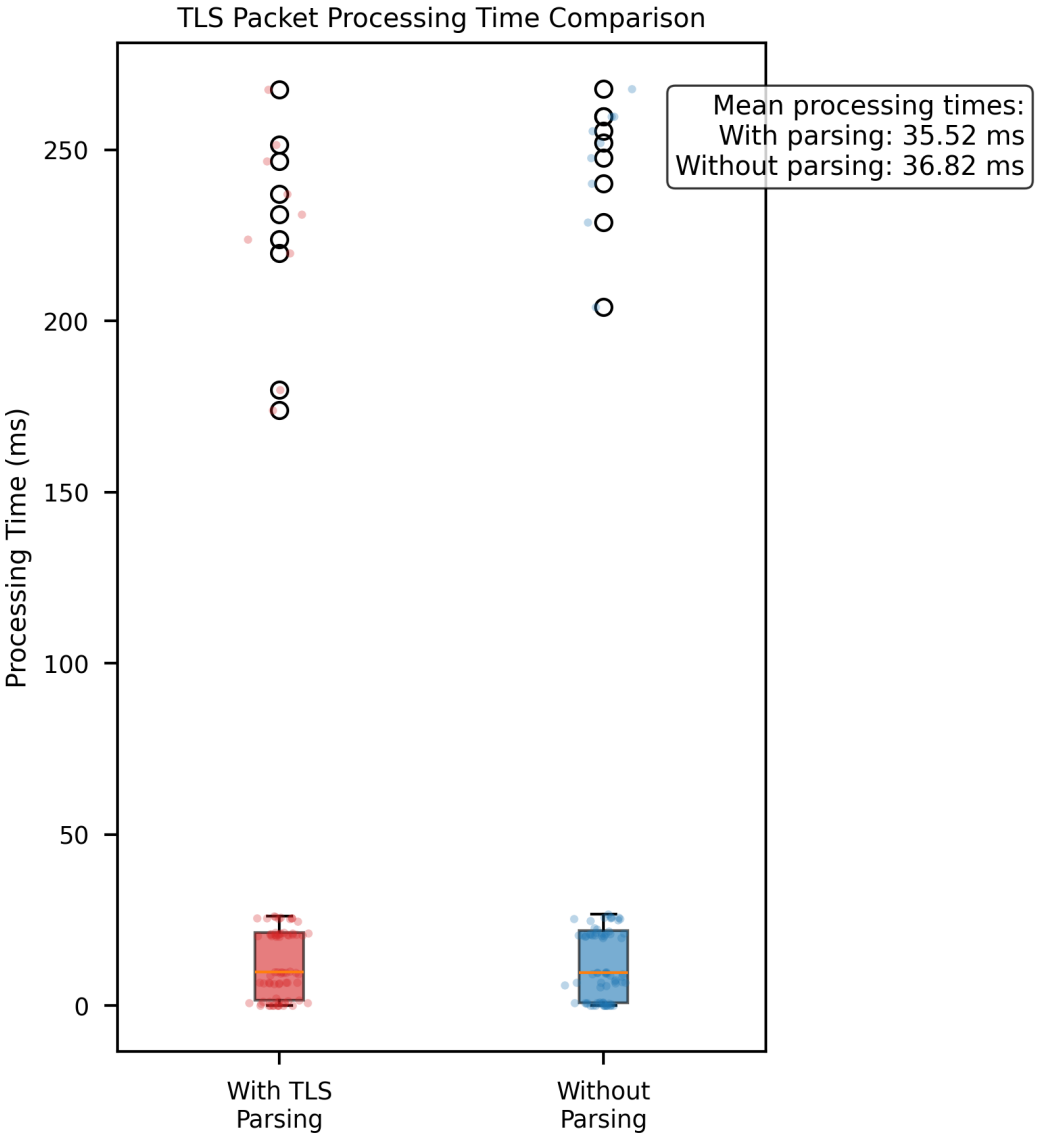

Figure 4.2: TLS processing time comparison

Interestingly, as observed with the temperature sensor in the MQTT tests, the processing time for TLS traffic was lower when parsing was enabled compared to when it was disabled. This counterintuitive result may be attributed to factors such as system scheduling, memory management, or inherent characteristics of the network stack. Despite these variations, the overall impact of TLS parsing on processing time remains negligible.

In general, the processing time for most TLS packets remained close to

zero, with a few notable outliers visible in the boxplot. As with the MQTT tests, these outliers are unlikely to stem from the firewall's internal processing but rather from the test execution methodology. Specifically, the packets exhibiting higher processing times occurred at regular intervals and consistently corresponded to `ClientHello` packets. This suggests that the observed delay is not due to the firewall's handling of TLS traffic but rather to the overhead introduced by the re-execution of the TcpReplay tool. Since the `ClientHello` packet is the first packet in the replayed trace, the increased processing time is likely a consequence of initialization overhead rather than an inherent inefficiency in the firewall. The regularity of these spikes is well illustrated in Figure C.2 in Appendix C, which presents a packet processing timeline.

Chapter 5

Conclusion

The extension of the new firewall tool to the MQTT protocol has proven to be a significant success. It has been demonstrated that the tool can effectively profile MQTT traffic within a network, thereby preventing attacks that previous solutions, such as the MUD standard alone, were unable to block. The practical utility of the tool in the context of smart homes using the MQTT protocol is thus well established.

In contrast, the results are significantly less convincing when extending the new firewall tool to the TLS protocol. Several constraints hinder the tool's functionality for TLS, particularly the inability to track TLS connections, which complicates both the identification of encrypted packets and the profiling of TLS traffic. In the tool's current state, manual encoding of random values for certain fields in the profiles is required, severely limiting its generalization for the TLS protocol. A thorough revision of the tool would likely be necessary to enable rigorous processing of TLS traffic, allowing for coherent packet parsing and facilitating traffic profiling. However, we can highlight that even in its current state, the tool has been shown to function effectively on specific packets, even in the presence of combined or fragmented packets. This demonstrates the potential viability of such firewall tools for managing TLS traffic.

When comparing the challenges posed by the implementation of these two protocols, it becomes evident that the firewall tool, having been designed for IoT, was significantly more adaptable to a protocol specifically intended for IoT, such as MQTT, than to a more general-purpose protocol like TLS, which was not designed with IoT in mind.

Bibliography

[1] Lear E, Droms R, Romascanu D. Manufacturer Usage Description Specification. RFC Editor; 2019. RFC 8520. Available from: https://www.rfc-editor.org/info/rfc8520.

[2] De Keersmaeker F, Sadre R, Pelsser C. Supervising Smart Home Device Interactions: A Profile-Based Firewall Approach. In: 2024 IFIP Networking Conference (IFIP Networking). IEEE; 2024. p. 413-22.

[3] Yasar K, Gillis AS. What is the internet of things (IoT)?; 2021. Accessed: 2024-12-17. Available from: https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT.

[4] Chakraborty A, Islam M, Shahriyar F, Islam S, Zaman HU, Hasan M. Smart home system: a comprehensive review. Journal of Electrical and Computer Engineering. 2023;2023(1):7616683.

[5] Boyes H, Hallaq B, Cunningham J, Watson T. The industrial internet of things (IIoT): An analysis framework. Computers in industry. 2018;101:1-12.

[6] Fang X, Misra S, Xue G, Yang D. Smart grid—The new and improved power grid: A survey. IEEE communications surveys & tutorials. 2011;14(4):944-80.

[7] Nižetić S, Šolić P, Gonzalez-De DLdI, Patrono L, et al. Internet of Things (IoT): Opportunities, issues and challenges towards a smart and sustainable future. Journal of cleaner production. 2020;274:122877.

[8] Analytics I. Number of Connected IoT Devices;. Available from: https://iot-analytics.com/number-connected-iot-devices/.

[9] Vailshery LS. IoT revenue worldwide; 2024. Accessed: 2024-12-02. Available from: https://www.statista.com/statistics/1194709/iot-revenue-worldwide/.

[10] Statista Research Department. Number of smart homes in the smart home market worldwide; 2024. Accessed: 2024-12-02. Available from: https://www.statista.com/forecasts/887613/number-of-smart-homes-in-the-smart-home-market-in-the-world.

[11] Harper R. Inside the smart home. Springer Science & Business Media; 2006.

[12] Ergen SC. ZigBee/IEEE 802.15. 4 Summary. UC Berkeley, September. 2004;10(17):11.

[13] IEEE Computer Society. IEEE Standard for Low-Rate Wireless Networks (IEEE 802.15.4); 2020. Available from: https://standards.ieee.org/ieee/802.15.4/7029/.

[14] OASIS. MQTT: The Standard for IoT Messaging; 2024. Accessed: 2025-01-05. Available from: https://mqtt.org/.

[15] Zhou W, Jia Y, Peng A, Zhang Y, Liu P. The effect of IoT new features on security and privacy: New threats, existing solutions, and challenges yet to be solved. IEEE Internet of things Journal. 2018;6(2):1606-16.

[16] Martins I, Resende JS, Sousa PR, Silva S, Antunes L, Gama J. Host-based IDS: A review and open issues of an anomaly detection system in IoT. Future Generation Computer Systems. 2022;133:95-113.

[17] Sadhu PK, Yanambaka VP, Abdelgawad A. Internet of things: Security and solutions survey. Sensors. 2022;22(19):7433.

[18] Al Hayajneh A, Bhuiyan MZA, McAndrew I. Improving internet of things (IoT) security with software-defined networking (SDN). Computers. 2020;9(1):8.

[19] Antonakakis M, April T, Bailey M, Bernhard M, Bursztein E, Cochran J, et al. Understanding the mirai botnet. In: 26th USENIX security symposium (USENIX Security 17); 2017. p. 1093-110.

[20] Team CPR. IoTroop Botnet: A Full Investigation. 2017. Accessed: 2025-01-03. Available from: https://research.checkpoint.com/2017/iotroop-botnet-full-investigation/.

[21] Tschofenig H, Arkko J, Thaler D, McPherson DR. Architectural Considerations in Smart Object Networking. RFC Editor; 2015. RFC 7452. Available from: https://www.rfc-editor.org/info/rfc7452.

[22] Jethanandani M, Agarwal S, Huang L, Blair D. YANG Data Model for Network Access Control Lists (ACLs). RFC Editor; 2019. RFC 8519. Available from: https://www.rfc-editor.org/info/rfc8519.

[23] Ronen E, Shamir A. Extended functionality attacks on IoT devices: The case of smart lights. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE; 2016. p. 3-12.

[24] Uroz D, Rodríguez RJ. Characterization and evaluation of IoT protocols for data exfiltration. IEEE Internet of Things Journal. 2022;9(19):19062-72.

[25] The netfilter.org "nftables" project;. Accessed: 2024-07-25. Available from: https://netfilter.org/projects/nftables/index.html.

[26] Purdy GN. Linux iptables Pocket Reference: Firewalls, NAT & Accounting. " O'Reilly Media, Inc."; 2004.

[27] OpenAI. ChatGPT; 2025. https://openai.com/chatgpt.

[28] The netfilter.org "libnetfilter_queue" project;. Accessed: 2024-07-25. Available from: https://netfilter.org/projects/libnetfilter_queue/index.html.

[29] Freier AO. The SSL protocol version 3.0. 1996. Available from: http://www.netscape.com/eng/ssl3/draft302.txt.

[30] Freier AO, Karlton P, Kocher PC. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC Editor; 2011. RFC 6101. Available from: https://www.rfc-editor.org/info/rfc6101.

[31] Allen C, Dierks T. The TLS Protocol Version 1.0. RFC Editor; 1999. RFC 2246. Available from: https://www.rfc-editor.org/info/rfc2246.

[32] Qualys I. SSL Pulse; 2024. Accessed: 2024-08-09. Available at: https://www.ssllabs.com/ssl-pulse/.

[33] Dierks T, Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC Editor; 2006. RFC 4346. Available from: https://www.rfc-editor.org/info/rfc4346.

[34] Rescorla E, Dierks T. The Transport Layer Security (TLS) Protocol Version 1.2. RFC Editor; 2008. RFC 5246. Available from: https://www.rfc-editor.org/info/rfc5246.

[35] Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC Editor; 2018. RFC 8446. Available from: https://www.rfc-editor.org/info/rfc8446.

[36] Duong T, Rizzo J. The BEAST Attack: Breaking SSL/TLS with Blockwise Chosen-Plaintext Attack. 2011. Demonstrated at ekoparty Security Conference. Available from: https://www.nccgroup.com/media/1nqor5lm/_ssl_attacks_survey.pdf.

[37] Projects P. Jinja Documentation; 2024. Accessed: 2024-11-25. Available from: https://jinja.palletsprojects.com/en/stable/.

[38] Ben-Kiki O, Evans C, döt Net I. YAML Ain't Markup Language (YAML™) Version 1.2.2; 2021. Accessed: 2024-12-27. Available from: https://yaml.org/spec/1.2.2/.

[39] Valadon G, Biondi P. Scapy: Packet Manipulation Tool; 2025. https://scapy.net/.

[40] Wireshark Foundation. Wireshark - Network Protocol Analyzer; 2024. Accessed: 2024-12-29. https://www.wireshark.org.

[41] Authors TG. conn.go; 2024. Accessed: 2024-08-02. https://github.com/golang/go/blob/master/src/crypto/tls/conn.go.

[42] Anthropic. Claude AI; 2025. https://www.anthropic.com/.

[43] TP-Link. HS110 Smart Wi-Fi Plug with Energy Monitoring; 2025. Accessed: 2025-03-02. Available from: https://www.tp-link.com/en/home-networking/smart-plug/hs110/#specifications.

[44] Turner A, et al.. Tcpreplay: Pcap Editing and Replaying Utilities;. Accessed: 2025-02-26. https://tcpreplay.appneta.com/.

[45] Hamza A, Ranathunga D, Gharakheili HH, Roughan M, Sivaraman V. Clear as MUD: Generating, validating and applying IoT behavioral profiles. In: Proceedings of the 2018 Workshop on IoT Security and Privacy; 2018. p. 8-14.

Appendix A

YAML profile templates for TLS

This appendix provides YAML profile templates for modeling TLS communications within the firewall's profiling system. These templates illustrate idealized TLS interactions, covering key scenarios such as the full TLS 1.2 handshake, the abbreviated handshake, and the TLS 1.3 handshake.

Since TLS communication involves numerous message exchanges, these templates are included here rather than in the main text for readability. Only TLS messages are considered—intervening TCP messages are omitted, as their handling was already addressed in the firewall's prior implementation. While these templates are not directly used by the firewall, they serve as structured examples of how TLS interactions are represented in device profiles.

A.1 TLS 1.2 full handshake interaction

```yaml
tls-device1-device2:
  tls-device1-device2-client-hello:
    protocols:
      tls:
        # handshake type is CLIENT HELLO (1)
        handshake-type: 1
        # version could be TLS 1.[0-2]
        # here TLS 1.2 is considered
        tls-version: 1.2
      tcp:
        dst-port: 443
      ipv4:
        # typically 'self' or a server\'s hostname
        src: ... # (device1 IP)
```

```
      dst: ... # (device2 IP)

tls-device2-device1-server-hello:
  protocols:
    tls:
      # handshake type is SERVER HELLO (2)
      handshake-type: 2
      tls-version: 1.2
    tcp:
      src-port: 443
    ipv4:
      ...

tls-device2-device1-certificate:
  protocols:
    tls:
      # handshake type is CERTIFICATE (11)
      handshake-type: 11
      tls-version: 1.2
    tcp:
      src-port: 443
    ipv4:
      ...
  stats:
    rate: 0 # Certificate is optional

tls-device2-device1-server-key-exchange:
  protocols:
    tls:
      # handshake type is SERVER_KEY_EXCHANGE (12)
      handshake-type: 12
      tls-version: 1.2
    tcp:
      src-port: 443
    ipv4:
      ...
  stats:
    rate: 0 # ServerKeyExchange is optional

tls-device2-device1-certificate-request:
  protocols:
    tls:
      # handshake type is CERTIFICATE_REQUEST (13)
      handshake-type: 13
      tls-version: 1.2
    tcp:
```

```
      src-port: 443
    ipv4:
      ...
  stats:
    rate: 0 # CertificateRequest is optional

tls-device2-device1-server-hello-done:
  protocols:
    tls:
      # handshake type is SERVER HELLO DONE (14)
      handshake-type: 14
      tls-version: 1.2
    tcp:
      src-port: 443
    ipv4:
      ...

tls-device1-device2-certificate:
  protocols:
    tls:
      # handshake type is CERTIFICATE (11)
      handshake-type: 11
      tls-version: 1.2
    tcp:
      dst-port: 443
    ipv4:
      ...
  stats:
    rate: 0 # optional

tls-device1-device2-client-key-exchange:
  protocols:
    tls:
      # handshake type is CLIENT KEY EXCHANGE (16)
      handshake-type: 16
      tls-version: 1.2
    tcp:
      dst-port: 443
    ipv4:
      ...

tls-device1-device2-certificate-verify:
  protocols:
    tls:
      # handshake type is CERTIFICATE VERIFY (15)
      handshake-type: 15
```

```
      tls-version: 1.2
    tcp:
      dst-port: 443
    ipv4:
      ...
  stats:
      rate: 0 # optional

tls-device1-device2-change-cipher-spec:
  protocols:
    tls:
      # content type is CHANGE CIPHER SPEC (20)
      content-type: 20
      tls-version: 1.2
    tcp:
      dst-port: 443
    ipv4:
      ...

tls-device1-device2-encrypted-handshake-message:
  protocols:
    tls:
      # FINISHED MESSAGE
      # handshake type is FINISHED MESSAGE (20) but value is encrypted
      handshake-type : ... # random value
      tls-version: 1.2
    tcp:
      dst-port: 443
    ipv4:
      ...

tls-device2-device1-change-cipher-spec:
  protocols:
    tls:
      # content type is CHANGE CIPHER SPEC (20)
      content-type: 20
      tls-version: 1.2
    tcp:
      src-port: 443
    ipv4:
      ...

tls-device2-device1-encrypted-handshake-message:
  protocols:
    tls:
      # FINISHED MESSAGE
```

```
        handshake-type: ... # random value
        tls-version: 1.2
      tcp:
        src-port: 443
      ipv4:
        ...

  tls-device1-device2-encrypted-data-exchange:
    protocols:
      tls:
        # content type is APPLICATION DATA (23)
        handshake-type: 23
        tls-version: 1.2
      tcp:
        src-port: 443
      ipv4:
        ...
    bidirectionnal: true
    stats:
      # highest rate in the pcap trace * 1.2
      rate: ...
```

Listing A.1: TLS 1.2 full handshake profile interaction example

A.2 TLS 1.2 abbreviated handshake interaction

```
tls-device1-device2-abbreviated:
    tls-device1-device2-client-hello:
      protocols:
        tls:
          handshake-type: 1 # client hello
          tls-version: 1.2
          session-id: true
        tcp:
          dst-port: 443
        ipv4:
          src: ... (device1 IP)
          dst: ... (device2 IP)

    tls-device2-device1-server-hello:
      protocols:
        tls:
          handshake-type: 2 # server hello
          tls-version: 1.2
```

```
        session-id: true
      tcp:
        src-port: 443
      ipv4:
        ...

tls-device2-device1-change-cipher-spec:
  protocols:
    tls:
      # content type is CHANGE CIPHER SPEC (20)
      content-type: 20
      tls-version: 1.2
    tcp:
      src-port: 443
    ipv4:
      ...

tls-device2-device1-encrypted-handshake-message:
  protocols:
    tls:
      # FINISHED MESSAGE
      handshake-type: ... # random value
      tls-version: 1.2
    tcp:
      src-port: 443
    ipv4:
      ...

tls-device1-device2-change-cipher-spec:
  protocols:
    tls:
      # content type is CHANGE CIPHER SPEC (20)
      content-type: 20
      tls-version: 1.2
    tcp:
      dst-port: 443
    ipv4:
      ...

  tls-device1-device2-encrypted-handshake-message:
  protocols:
    tls:
      # FINISHED MESSAGE
      handshake-type: ... # random value
      tls-version: 1.2
    tcp:
```

```
      dst-port: 443
    ipv4:
      ...

  tls-device1-device2-encrypted-data-exchange:
  protocols:
    tls:
      # content type is APPLICATION DATA (23)
      handshake-type: 23
      tls-version: 1.2
    tcp:
      src-port: 443
    ipv4:
      ...
  bidirectionnal: true
  stats:
    # highest rate in the pcap trace * 1.2
    rate: ...
```

Listing A.2: TLS 1.2 abbreviated handshake profile interaction example

A.3 TLS 1.3 handshake

```
tls-device1-device2:
    tls-device1-device2-client-hello:
        protocols:
            tls:
                # handshake type is CLIENT HELLO (1)
                handshake-type: 1
                tls-version: 1.3
            tcp:
                dst-port: 443
            ipv4:
                src: ... # (device1 IP)
                dst: ... # (device2 IP)

    tls-device2-device1-server-hello:
        protocols:
            tls:
                # handshake type is SERVER HELLO (2)
                handshake-type: 2
                tls-version: 1.3
            tcp:
                src-port: 443
```

```
            ipv4:
                ...

    tls-device2-device1-encrypted-extensions:
        protocols:
            tls:
                # handshake type is ENCRYPTED EXTENSIONS (8)
                handshake-type: 8
                tls-version: 1.3
            tcp:
                src-port: 443
            ipv4:
                ...

    tls-device2-device1-certificate:
        protocols:
            tls:
                # handshake type is CERTIFICATE (11)
                handshake-type: 11
                tls-version: 1.3
            tcp:
                src-port: 443
            ipv4:
                ...
        rate: 0  # Certificate is optional

    tls-device2-device1-certificate-verify:
        protocols:
            tls:
                # handshake type is CERTIFICATE VERIFY (15)
                handshake-type: 15
                tls-version: 1.3
            tcp:
                src-port: 443
            ipv4:
                ...
        rate: 0  # CertificateVerify is optional

    tls-device2-device1-finished:
        protocols:
            tls:
                # handshake type is FINISHED (20)
                handshake-type: ... # random value !!
                tls-version: 1.2 # !! true version number is encrypted
            tcp:
                src-port: 443
```

```
            ipv4:
                ...

    tls-device1-device2-finished:
        protocols:
            tls:
                handshake-type: ... # random value !!
                tls-version: 1.2 # !!
            tcp:
                dst-port: 443
            ipv4:
                src: ...
                dst: ...

    tls-device1-device2-encrypted-data-exchange:
        protocols:
            tls:
                # content type is APPLICATION DATA (23)
                handshake-type: 23
                tls-version: 1.2 # !!
            tcp:
                dst-port: 443
            ipv4:
                src: ...
                dst: ...
        bidirectionnal: true
        stats:
            rate: ...
```

Listing A.3: TLS 1.3 handshake profile interaction example

Appendix B

TPLink-Plug device profile extension to TLS

This appendix presents the complete TLS interaction included in the YAML profile of the TPLink-Plug device [43], used to test the proper functionality of the firewall tool's extension to the TLS protocol.

```yaml
# Initial TP-Link smart plug profile.

---
device-info:
  name: tplink-plug
  mac: 50:c7:bf:ed:0a:54
  ipv4: 192.168.1.135
  network: wireless


single-policies:
  [...]

interactions:

  ### BOOT TRAFFIC ###
  [...]


  ### INTERACTION TRAFFIC ###
  [...]

  tls-phys-plug:
    tls-phys-plug-client-hello:
```

```
    protocols:
      tls:
        handshake-type: 1 # client hello
        tls-version: 1.2
        session-id: false
      tcp:
        src-port: 45840
        dst-port: 443
      ipv4:
        src: 192.168.1.147
        dst: 54.76.18.183

  tls-plug-phys-server-hello:
    protocols:
      tls:
        handshake-type: 2 # server hello
        tls-version: "1.2"
      tcp:
        src-port: 443
        dst-port: 45840
      ipv4:
        src: 54.76.18.183
        dst: 192.168.1.147

  tls-plug-phys-tcp-msg:
    protocols:
      tcp:
        src-port: 443
        dst-port: 45840
      ipv4:
        src: 54.76.18.183
        dst: 192.168.1.147

  tls-plug-phys-certificate--server-key-exchange--server-hello-done:
    protocols:
      tls:
        handshake-type: 11, 12, 14 # certificate, server key exchange,
server hello done
        tls-version: "1.2"
      tcp:
        src-port: 443
        dst-port: 45840
      ipv4:
        src: 54.76.18.183
        dst: 192.168.1.147

```

```
    tls-phys-plug-client-key-exchange:
      protocols:
        tls:
          handshake-type: 16 # client key exchange
          tls-version: 1.2
        tcp:
          src-port: 45840
          dst-port: 443
        ipv4:
          src: 192.168.1.147
          dst: 54.76.18.183

    tls-phys-plug-change-cipher-spec:
      protocols:
        tls:
          content-type: 20 # change cipher spec
          tls-version: "1.2"
        tcp:
          src-port: 45840
          dst-port: 443
        ipv4:
          src: 192.168.1.147
          dst: 54.76.18.183

    tls-phys-plug-finished:
      protocols:
        tls:
          handshake-type: 0x00 # fifth byte
          tls-version: "1.2"
        tcp:
          src-port: 45840
          dst-port: 443
        ipv4:
          src: 192.168.1.147
          dst: 54.76.18.183

    tls-plug-phys-change-cipher-spec--finished:
      protocols:
        tls:
          handshake-type: 20, 0xc7 # change cipher spec, finished
          tls-version: "1.2"
        tcp:
          src-port: 443
          dst-port: 45840
        ipv4:
          src: 54.76.18.183
```

```
          dst: 192.168.1.147

    tls-phys-plug-application-data:
      protocols:
        tls:
          content-type: 23 # application data
          tls-version: "1.2"
        tcp:
          src-port: 45840
          dst-port: 443
        ipv4:
          src: 192.168.1.147
          dst: 54.76.18.183
      bidirectional: true
      stats:
        rate: 69/second # 57 * 1.2

[...]
```

Listing B.1: TPLink-Plug device profile – extension to TLS

Appendix C

Protocols packet processing time timelines

This appendix presents the packet processing timeline results for both MQTT and TLS protocols, obtained during the testing phase of the firewall's extension to the new protocols.

C.1 MQTT packet processing timeline

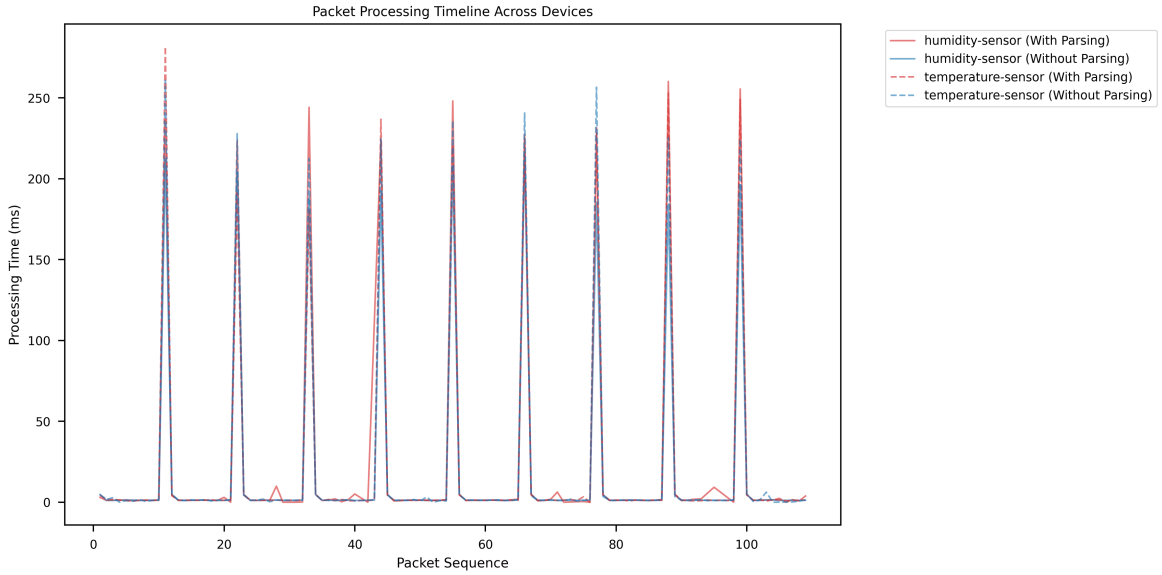

Figure C.1

C.2 TLS packet processing timeline

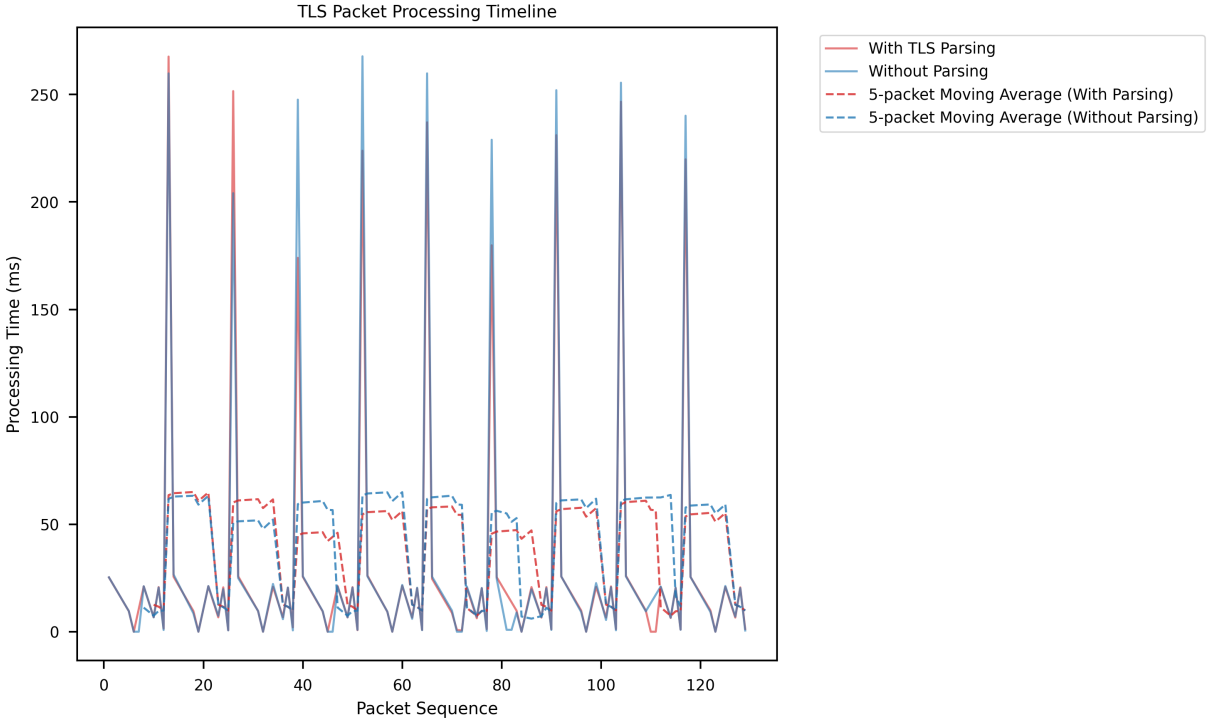

Figure C.2

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**