Implementation of a SOME/IP Firewall with Deep Packet Inspection for automotive use-cases

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Cyber Security
May 2024
Eva Helena Zorman

Supervisors:
Jouni Isoaho
Tahir Mohammad

UNIVERSITY OF TURKU
Department of Computing

Eva Helena Zorman: Implementation of a SOME/IP Firewall with Deep Packet Inspection for automotive use-cases

Master of Science (Tech) Thesis, 87 p., 3 app. p.
Cyber Security
May 2024

As the in-vehicle network steadily aims at replacing the traditional and leading Controller Area Network (CAN) protocol, the Scalable service-Oriented MiddlewarE over IP (SOME/IP) comes into focus with its speed and reliability for transferring control-level messages. The SOME/IP protocol must be adequately protected else the whole system's safety can be endangered. However there are still known vulnerabilities present in SOME/IP, such as lack of authentication or encryption. This thesis aims at improving the security of the SOME/IP protocol by developing, implementing, and evaluating a SOME/IP firewall tailored for embedded automotive communication systems. The study involves implementing rule-based security measures utilizing SOME/IP header values to control and restrict access, alongside the development of a deep packet inspection mechanism for basic payload validation. Performance and resource requirements are evaluated on a Raspberry Pi 4 Model B to emulate smaller ECUs in a vehicular environment. This work bridges a gap in the literature by introducing an alternative approach to securing SOME/IP data transmission without the full CommonAPI framework, by proposing a novel SOME/IP firewall solution and providing insights into its performance and effectiveness for embedded system use. The basic packet parsing results, which introduce a delay in the range from $0.03ms$ to $0.15ms$, demonstrate the feasibility and potential of the proposed firewall implementation for enhancing security in automotive communication systems. Further research is suggested to validate and analyse the resource constraints of a fully-functional SOME/IP firewall with support for deep packet inspection of any data type in an actual production environment.

Keywords: SOME/IP, CAN, embedded, firewall, DPI, ECU, connected cars, cyberattacks, in-vehicle network, autonomous vehicles, CommonAPI

Contents

List of Figures

List of Tables

List of acronyms

AGL Automotive Grade Linux

API Application Programming Interface

ASLR Address Space Layout Randomization

AUTOSAR Automotive Open System Architecture

AVB/TSN Audio Video Bridging / Time Sensitive Networking

BOM Byte Order Mark

CAN Controller Area Network

COVESA Connected Vehicle Systems Alliance

DAS Distributed Applicaiton Subsystem

E/E Electrical and Electronic

ECU Electronic Control Unit

FDEPL Franca Deployment File

HIDS Host-based Intrusion Detection System

HSM Hardware Security Module

IDL Interface Definition Language

IDS Intrusion Detection System

IPC Interprocess and Network Communication

IPS Intrusion Prevention System

IVN In-Vehicle Network

LIN Local Interconnect Network

MITM Man-In-The-Middle

MOST Media Oriented Systems Transport

NIDS Network-based Intrusion Detection System

OEM Original Equipment Manager

OSI Open Systems Interconnection

PCAPNG PCAP Next Generation

PCAP Packet Capture

PoC Proof of Concept

RPC Remote Procedure Calls

RQ Research Question

SAE Society of Automotive Engineers

SOA Service-Oriented Architectur

SOME/IP Scalable service-Oriented MiddlewarE over IP

1 Introduction

In an era marked by the rapid evolution of automotive technology, from infotainment systems to autonomous driving capabilities, modern vehicles rely heavily on intricate networks to facilitate seamless communication between its units. However, with this technological advancement comes the pressing need for robust security measures not only for legacy protocols, but also newer ones. In particular, the SOME/IP protocol, widely adopted in Automotive Ethernet networks capable of transmitting control messages and thus being named a possible successor to the CAN protocol, presents challenges and vulnerabilities that could benefit from a specialized firewall solution.

1.1 Research Problem

Current developments in automotive connectivity are transitioning from traditional protocols like CAN to more intricate ones like FlexRay, Automotive Ethernet or SOME/IP. As this transition gains momentum, there is a critical gap in ensuring robust security for other communication protocols, as most effort is still focused on the CAN protocol. The existing SOME/IP protocol lacks intrinsic security mechanisms such as authentication and encryption, rendering automotive communication systems vulnerable to attacks. Specifically coupled with potential system crashes caused by poorly implemented packet decoders, the need for an effective security solution is apparent. The research problem presented in this thesis is to design, implement, and evaluate a dedicated SOME/IP firewall for an embedded system with

deep packet inspection to mitigate vulnerabilities and secure automotive communication systems against potential attacks due to malformed payloads or unauthorised access.

1.2 Research Objective

This thesis aims to design, implement, and evaluate a proof of concept (PoC) SOME/IP firewall implementation tailored for embedded automotive communication systems in C++. The study will focus on implementing rule-based security measures that utilize SOME/IP header values such as service, client and method ID to effectively control and restrict access to unauthorised clients, as well as deserializing and parsing the payload contents. In particular, the objectives of the thesis are as follows:

1. To design and implement a compact, robust, and modular firewall equipped with fundamental security features such as deep packet inspection. A deep packet inspection mechanism will be developed for deserialization of data based on the CommonAPI Framework, allowing inspection and validation of SOME/IP payloads with the additional benefit of being structure-aware by loading Franca IDL models locally.

2. To test the proposed firewall's resilience against potential attacks against known SOME/IP header attacks.

3. To analyse and evaluate the firewall's ability to handle malformed payloads to prevent ECU crashes or disruptions, ensuring system reliability under adverse conditions. Access restriction mechanisms will be demonstrated by the firewall's capability to restrict client access to specific interfaces on an ECU, as well as the strain the rule-set size introduces to the system.

4. To evaluate the firewall's performance and resource requirements on a Raspberry Pi 4 Model B to emulate smaller ECUs in a vehicular environment.

1.3 Contributions

The results of this evaluation will be based on established security and data transmission delay goals. To the author's current knowledge, there is no SOME/IP specific firewall implementation proposed or researched yet that is publicly available. As such, this work aims at bridging that gap and propose an alternative solution to securing SOME/IP data transmission using a rule-based approach with deep packet inspection that does not require the full CommonAPI framework present on an embedded system.

Currently there is a limited amount of research done on securing the SOME/IP protocol data transmission. Most of those works either provide a stand-alone solution, or increase the protocol security by modifying the existing protocol or adding-on additional security mechanisms such as authentication. This is generally not well-received or implemented in practice where maintaining functionality with already-running systems takes precedence, and is even more noticeable in the automotive world, where such changes lead to a number of call-back vehicles. This work instead focuses on including an additional layer of security to the protocol without modifying its structure or functionality.

Modern security practices and standards focus on a multi-layered approach, with simple and minimalistic toolchains that introduce a minimal additional security risk to the system they are trying to protect. As such, the SOME/IP firewall proposed within the scope of this thesis can be used in tandem with other measures that can secure further aspects of the SOME/IP protocol, such as authentication of clients and service providers, which cannot by design be covered by a firewall implementation. Should SOME/IP become a popular message transmission protocol

instead of the CAN protocol, a SOME/IP firewall could become a de-facto part of a security mechanism toolchain for any vehicle supporting SOME/IP, its output being used for other tools, e.g. such as an Intrusion Detection System.

1.4 Research Question

1. What are the key relevant SOME/IP attacks that can be prevented by an implementation following the principles of a firewall?

2. Can the security vulnerabilities of the SOME/IP protocol implementations be addressed through the application of such a firewall, incorporating a baseline set of rules for access restriction and deep packet inspection?

3. How effective is such a firewall solution in preventing general SOME/IP header and payload attacks on automotive communication systems, which are embedded devices?

1.5 Scope of the Thesis

In this thesis a new approach to securing SOME/IP protocol data transmission is proposed by introducing a rule-based firewall implementation that is additionally structure-aware of the services, methods and arguments being passed between client and service providers. This enables the firewall to perform basic deep packet inspection and introduce an additional layer of security in front of SOME/IP applications running on vehicles without the need for larger frameworks that could bottleneck smaller embedded systems.

This thesis produces a basic design and proof-of-concept implementation of a SOME/IP firewall which is capable of DPI in a similar fashion as the CommonAPI deserialization. This enables smaller systems to run a minimally viable parsing

solution for SOME/IP message payloads. This thesis aims at providing another solution to securing the SOME/IP protocol and evaluating the feasibility of such a solution for smaller embedded systems, as there is not many different solutions currently available for automotive use-cases. Further optimised implementation that supports all data-types and complex interface structures is out-of-scope for this work and would be an interesting future work.

1.6 Thesis Outline

The rest of the thesis is organized as follows. The second chapter reviews the history and current state of vehicular networks, its architecture style and how data is transferred within such networks. It further analyses how vehicle security is achieved against network attacks, what are its security mechanisms, as well as a general overview of firewalls and their application in automotive systems. The third chapter provides a deep-dive into the SOME/IP protocol, its features, structure and security, as well as discusses currently known vulnerabilities and attacks on SOME/IP. It also introduces the Franca framework which provides the structure-awareness to the firewall, and lastly the CommonAPI framework, off which the deserialization of data payloads is based. The fourth chapter presents the design process and decisions taken for the SOME/IP firewall implementation, its architecture and further hardening possibilities for the firewall. The fifth chapter goes over the testing and validation environment, datasets and methodology used to analyse and benchmark the provided implementation of the SOME/IP firewall. The next chapter further analyses the results obtained during testing and discusses and compares them to available constraints for embedded systems. The final chapter is the conclusion of the thesis.

2 Literature Review

During the last two decades, the automotive industry experienced a surge in technological innovations based on customer's demands. A once purely mechanical system now carefully harmonizes with an increasing amount of technological solutions to increase customer safety, comfort and satisfaction. This is achieved in current premium vehicles by using up to 100 Electronic Control Units (ECUs), which is leading to increased production costs for manufacturers per each additional unit. To ensure communication between these units, a number of protocols have been designed. The main security issues these protocols have is that they were designed at a point in time where secure-by-design was not widely implemented. As this is a profitable point-of-entry for malicious actors, many known security vulnerabilities have been discovered over time for a number of protocols [1].

The remainder of this chapter goes over vehicular networks, its architecture and the most common communication protocols used nowadays. It then inspects network safety in vehicles and presents some recent real-life attacks to showcase what lack of protection can lead to. Some network security mechanisms are discussed and a short mention of other mechanisms is present.Finally, there is a short examination of firewalls and its usage in vehicles, as well as the functionality of packet capturing.

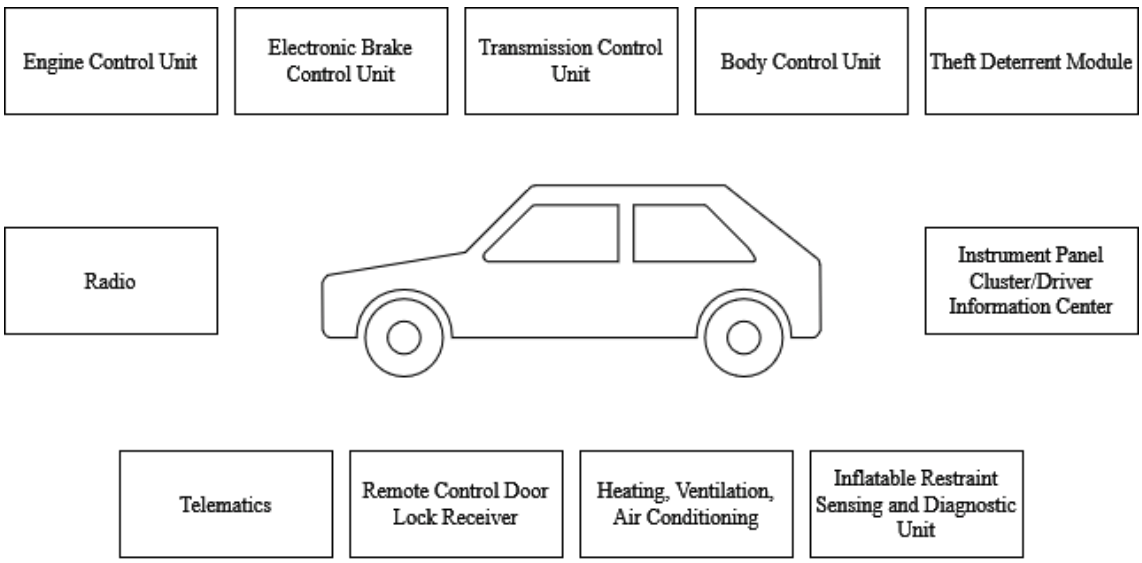

Figure 2.1: Typical ECUs found in a modern car.

2.1 Vehicle Networks

In 1978 General Motors introduced the first vehicle with an electronic system, after which their usage drastically increased, where the number of control units required for a premium car to function currently generally ranges from 50 units upwards [2]. This was largely due to the demands in increased driver and pedestrian safety, as well as country specific legislations, which started pushing for smaller emission levels from the automotive industry [3]. A modern vehicle contains several ECUs which control different aspects of its functionality. An example of some of the most important ECUs, extracted from Koscher, Czeskis, Roesner, *et al.* [4], can be found in Figure 2.1.

Due to such a large number of ECUs required for modern vehicles to operate, automotive manufacturers generally relay tasks of developing specific subsystems to outside suppliers. As stated in Obermaisser, El Salloum, Huber, *et al.* [5], this can lead to several different suppliers developing their respective systems. These systems to be developed are called Distributed Applicaiton Subsystems (DASs), and they can span multiple ECUs depending on their complexity. Considering the distributed

nature of the development on these systems, there comes a question of responsibility and liability in case of failure. As such, a federated architecture, where dedicated hardware units manage one single feature, had been implemented [5]. This however has a drawback, which is of interest to every company - the cost. With an increasing number of ECUs, so rises the cost necessary to produce such a car, as well as the sheer size of the threat landscape available to malicious actors. Should one unit be poorly designed or implemented in respect to security, the whole car can be compromised or recalled.

Modern ECUs have evolved to possess enough processing power to run dynamic operating systems such as custom Linux OS, e.g. Automotive Grade Linux (AGL) [6]. There are also other automotive efforts of standardisation such as The Connected Vehicle Systems Alliance (COVESA), which was initially known as GENIVI, and the Automotive Open System Architecture (AUTOSAR) framework [7]. It had been recently announced the COVESA standardisation is intent to align with AUTOSAR to ensure even better standardisation efforts. COVESA will focus on managing and standardising vehicle and cloud data and interactions, whereas AUTOSAR will focus on the overall system architecture and the In-Vehicle Network [8] (IVN). There is also a growing interest both from the automotive industry and other larger software development companies such as Google, Apple and Amazon, to combine their best efforts and deliver Service-Oriented Architecture (SOA), which may also be including the OS running on the ECU [9]. This is already the case for some ECUs, and there are already some available articles tackling the best implementation of using Android on (or in-relation to) an ECU, such as [10] or [11].

2.1.1 Vehicle Network Architectures

The traditional implementation of an automotive Electrical and Electronic (E/E) software architecture, which by now is only present in older vehicles still on the road,

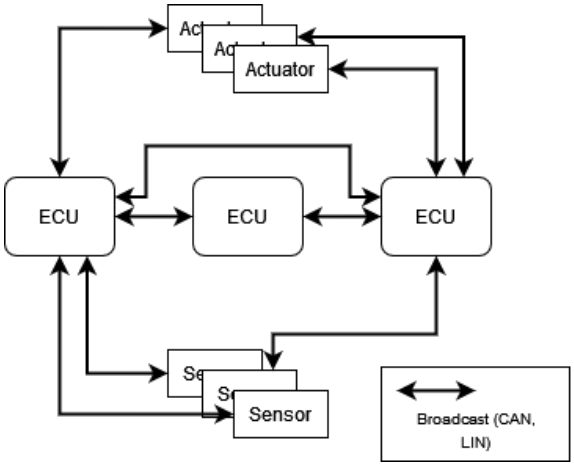

Figure 2.2: Distributed E/E architecture

consisted of a number of ECUs connected via a broadcast bus. This corresponds with the federated architecture that was used at the time. As ECUs in this architecture type rarely interact with each other, there is high modularity, and each ECU provides very specific and encapsulated functionality [12]. An example of such an architecture can be seen in Figure 2.2.

This state of the IVN however still poses some issues. Functionality had to be split based on availability of processing power within the car, which resulted in many of the different larger ECUs being designed separately or over-designed due to the mandated requirements of the automotive company (also called the Original Equipment Manager, or OEM) [9].

This led to the design of a new architecture type called integrated automotive architecture, the main difference being that one single ECU would run multiple different DAS functions irrespective of who developed them [5]. An example of such an architecture type is the Domain Centralised E/E architecture, which is currently present in Volvo [13], Volswagen [14] and Ford [15] vehicles currently on the road. The Domain Centralised E/E architecture not only includes faster data transfer protocols, but also brings security updates to the IVN with its domain gateways controlling communication between ECUs in different domains. An example of such

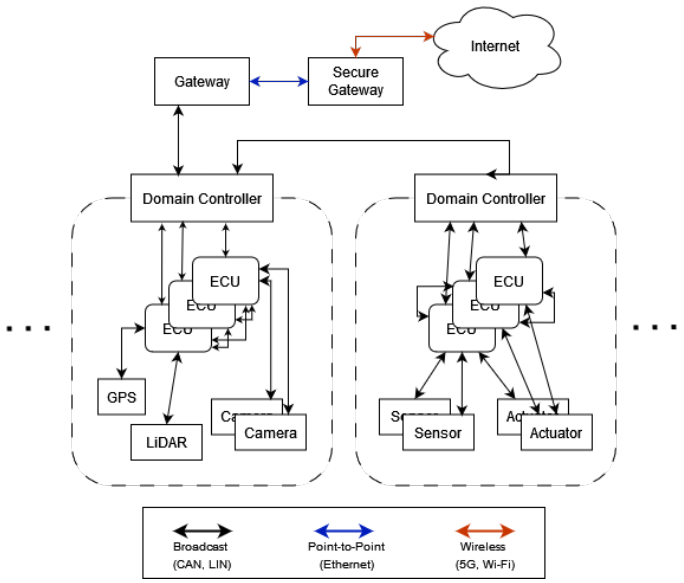

Figure 2.3: Domain Centralised E/E architecture

an architecture, extracted from Bucaioni and Pelliccione [12] can be found in Figure 2.3.

However, due to the fact that cars have a long on-road life span, averaging between 18 and 28 years in Western and Eastern Europe respectively [16], the automotive industry is faced with a difficult task of delivering future-proof vehicles with underdeveloped technology as compared to future solutions. As it is, the automotive industry is always carefully balancing cutting-edge innovation and safe, tested solutions that will stay secure over the next decade or longer. This is turn leads to the industry looking far ahead and predicting future trends, as well as researching and developing products and features that will be implemented in the upcoming years, vehicular networks being no exception.

Automotive companies such as BMW have already started designing and tackling the current architecture and infrastructure issues to implement in newer generations of their products. As stated in Traub, Maier, and Barbehön [9], BMW is focusing on implementing powerful integration platforms, which will enable a top-down hierarchical architecture for their ECUs, preventing over-designing, as well as local and

distributed implementations of their ECUs. They are also focusing on scalability and flexibility of future implementations, that will help in developing vehicle architectures from scratch quickly and cost-effectively. BOSCH is similarly developing new E/E architectures that instead focus on vehicle and zone-centralised architecture [17]. This would allow BOSCH to use a limited number of powerful processing units instead of many smaller individual ECUs, which is also presented in Bucaioni and Pelliccione [12] as a "tomorrow" architecture that will be de-facto standard for automotive companies in the foreseeable future.

Even further than "tomorrow" is the architecture "future", which even further consolidates the processing units present in a zonal-based E/E vehicle architecture. The number of processing ECUs is decreased even further, to a singular core processing ECU with a Service-Oriented Architecture (SOA) network access, which is connected to a limited number of other zonal gateway ECUs [18]. Automation in vehicles is also another reason for such future developments, not just minimising production costs, as an increased number of sensors introduces the demand for the wiring harness, which is a heavy and costly part of the vehicle, and should thus be as minimised as possible.

2.1.2 Communication Within the Vehicle

Since it is a well-established fact that there is numerous ECUs available in a modern car (even more so if it is a premium car), the question arises how do these singular processing units communicate and forward information between itself. To ensure smooth data transmission, a communication network is required. This network, which was traditionally implemented as a communication bus, connects multiple units together, and depending on the constraints or requirements of the vehicle, certain features must be assured, e.g. fault-tolerance or dependability. Due to

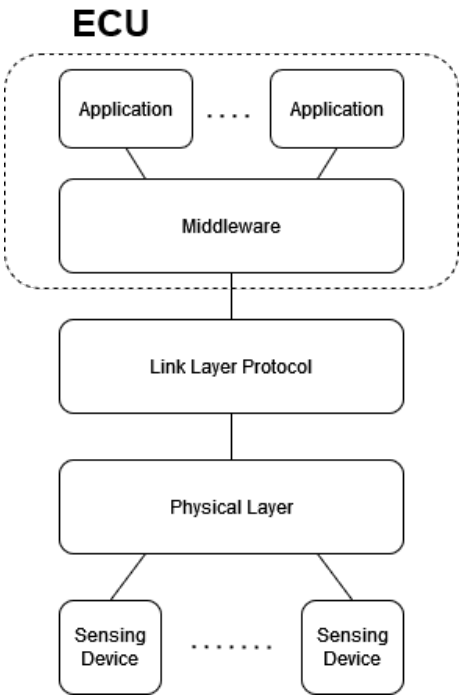

Figure 2.4: Wired network communication example architecture.

the unsafe nature of wireless protocols, which was initially raised by Rouf, Miller, Mustafa, *et al.* [19], many believe that inter-vehicle communication will predominantly be implemented in a wired way for the foreseeable future [20]. The general architecture of an ECU connected by a wired vehicle bus was presented in Tuohy, Glavin, Hughes, *et al.* [20], with data being transferred passing from a sensing device over the different Open Systems Interconnection (OSI) layers [21] to an application.

Popular Communication Protocols Available

Table 2.1: SAE automotive network classification.

Classification	Data Speed	Applications
Class A	<10 kb/s	Convenience features
Class B	10 - 125 kb/s	General data transfer
Class C	125 kb/s - 1 Mb/s	Real-time control
Class D	>1 Mb/s	Multimedia applications and Hard Real-time critical functionality

Within a modern vehicle, a number of communication protocols are used depending on what is their main purpose. As there is many different functionalities within a vehicle that must be covered, a common and standardised recommendation and practices were published by the Society of Automotive Engineers (SAE). It classifies protocols in a range of class A, B, C and D, where the higher the order of the letter, the more strict and demanding requirements in bandwidth, latency and reliability must be met [22].

Table 2.2: Overview and comparison of popular protocols

Protocol	**Speed**	**Wiring**	**Drawback**	**SAE**
CAN [23]	125 Kb/s - 1 Mb/s	Twisted Pair	Less Bandwidth	B
LIN [7]	10 Kb/s - 20 Kb/s	Single Wire	Low Speed	A
FlexRay [24]	Up to 10 Mb/s	Twisted Pair/ Optical Fibre	High cost	D
Automotive Ethernet [25]	Up to 100 Mb/s	Twisted Pair	Must use AVB/TSN IEEE 802.1 standard	D
MOST [26]	Up to 150 Mb/s	Optical Fibre	High cost	D

CAN The Controller Area Network (CAN) bus protocol is generally known as the first available automotive protocol being used for communication within a car, which was developed by Robert Bosch GmbH [23]. Due to this, it is no surprise it has become the most commonly used protocol in the automotive industry, selling approximately 400 million CAN nodes in early 2000's [27]. While this number may have dwindled over the past few years due to the emerging newer and better communication protocols for the automotive industry, it is still considered the most popular protocol for IVN. Over the decades the CAN bus protocol has been used, a number of attacks have been uncovered, such as CAN fuzzing, bus frame falsifying or bus injection attacks [28], mainly due to the lack of authentication in its design. Many researches tackled the task to provide additional security mechanisms by-design, mainly message authentication, to CAN and CAN-FD [29]. The

main issue with such additions is that they add-on high over-head or latency to the protocol, introduce additional key distribution mechanisms to the protocol and/or completely change the way the protocol works Szilagyi and Koopman [30], Szilagyi and Koopman [31], Mundhenk, Steinhorst, Lukasiewycz, *et al.* [32], Nowdehi, Lautenbach, and Olovsson [33], Agrawal, Huang, Zhou, *et al.* [34], Marasco and Quaglia [35], and Groza, Murvay, Van Herrewege, *et al.* [36]. As such, it is difficult to apply these changes to cars using CAN which are already on the road.

LIN The Local Interconnect Network (LIN) protocol is a less expensive broadcast communication bus developed in conjunction of a number of automotive manufacturers with the wish to produce a UART-based protocol that can become a global standard for automotive manufacturers [37]. While the LIN bus protocol is not used for time or security sensitive data transmission, it can still cause lessened functionality or a shut-down of specific subnets if targeted by a malicious actor. The LIN bus protocol itself has not had many additional add-ons for the protocol, irrespective of the protocol specialisation that is being upgraded.

FlexRay The FlexRay network is a potential successor to CAN, with a faster communication rate. It is a more complex communication protocol which also brings along higher costs for implementation. Although it may be used for safety related systems, its own security is rather neglected [38]. This led to discovery of known attacks on the FlexRay protocol, which are eavesdropping [38] and static segment attacks, which also include masquerading, injection and replay attacks [39].

Automotive Ethernet While Ethernet is a widely used and popular protocol, it has been avoided in automotive settings due to its noise and high-latency restrictions [40]. Automotive Ethernet has been designed to cover all communication needs of an automotive system, thus minimizing the cost and complexity requirements during

development. Due to its known shortcomings, a new IEEE 802.1 standard was developed called Audio Video Bridging / Time Sensitive Networking (AVB/TSN), which provides a high-bandwidth network that can relay time-sensitive information if required [41]. The Automotive Ethernet data frame also contains the source and destination address, which means message authentication is possible by default. Due to its high data transfer and reliability, data for infotainment and autonomous driving are some examples of data being transferred using Automotive Ethernet. If compromised, results can be devastating not only for the whole system, but also for the passengers' safety. There are a number of known attacks on Automotive Ethernet, mainly being traditional network attacks, such as eavesdropping, spoofing, injection, DoS or TCP hijacking [1].

MOST The Media Oriented Systems Transport (MOST) protocol is one of the newer protocols used within vehicles out of the popular choices of protocols presented so far, as it is mainly used for transmission of audio, video, voice, and control data. It was developed for the needs of the ever-growing infotainment systems within modern cars. Its messages also include a sender and receiver address, which means authentication is possible by design [42]. Due to its novelty, its security analysis is not examined as in-depth as e.g. the CAN protocol. There are however some known attacks, that being synchronisation disruption [43], as well as jamming attacks [42].

2.2 Vehicle Security

Back in the 1970's, the cost of electronics in a vehicle was around 5%, whereas it is predicted that around 2025, that cost will near the 50% mark [2]. As such, there was no need to consider the security of IVNs which were isolated from the outside world and held little reason for attackers to focus on [44]. With the constant increase of software and innovation in modern vehicles, it was only a matter of time vehicles

became connected with outside networks, and its security flaws be exploited. This led to rapid modernisation of the network security, which is continuing still today. This section goes over a short summary of vehicle network based security mechanisms known, as well as other security mechanisms which work in tandem to provide security within vehicles. The section also discusses how firewalls, Intrusion Detection (IDS) and Intrusion Prevention Systems (IPS) are used in this environment.

2.2.1 Real-life Vehicle Network Attack Instances

Due to such a large number of ECUs and software present in modern cars, the attack surface grows exponentially with each addition to the IVN. In 2022, key-less remote entry thefts and break-ins reached almost a fifth of all incidents reported in the automotive industry that year [45]. One of the first instances of a real-life attack simulation was done by Hoppe, Kiltz, and Dittmann [46], successfully managing to inject malicious code into an ECU that allowed for eavesdropping, replay and injection attacks on the CAN bus, which could lead to disabling warning lights or simulating the presence of the airbag system which would not actually be available. Rouf, Miller, Mustafa, *et al.* [19] showcased the security vulnerabilities of the tire pressure monitoring system present in vehicles, as it had no authentication or encryption support, and was thus open to eavesdropping and even spoofing attacks. While spoofing attacks may not damage the actual system, it may lead to the vehicle completely disregarding real errors in the system. Miller and Valasek [47] demonstrated security vulnerabilities in the U-connect software present in Fiat Chrysler vehicles. They were able to remotely control a vehicle approximately 16km away and amongst other things, control the braking system of the vehicle, which ended up crashing off the road. They were also able to exploit the vulnerabilities of the CAN bus to remotely overwrite the firmware of a Renesas V850 processor. Similar attacks to that on the Fiat Chrysler vehicles were tested on the Mitsubishi Outlander Plug

in Lodge [48] by using a man-in-the-middle attack, which resulted in being able to turn the lights on and off and disable the theft alarm system of the vehicle. Similarly, security vulnerabilities in the Nissan Connect application caused it to be disabled and discontinued due to the possibility of remote hijacking air conditioning and the heating system of their Leaf electric vehicles, which led to a constant drainage of the vehicle's battery [49]. In April of 2023, a new vulnerability of the CAN bus was discovered, which allowed for CAN injection attacks in the Toyota RAV4 [50]. This was done by removing the headlight of the vehicle and physically connecting to the ECU that handled smart key functionality and was positioned close to the headlight.

It is clear there is an identifiable gap between solutions proposed by researchers and the implementation done by the automotive industry. Although there had been numerous disclosed vulnerabilities found in the CAN bus protocol, it is still largely used due to its low production costs and familiarity. The listed attacks only further incited researchers to find and propose more secure protocols that could become future standards for the entire automotive industry.

2.2.2 Vehicle Network Based Security Mechanisms

The newer E/E software architectures brought along with them a much needed modernisation of how the IVN was structured in regards to data transmission security. Wolf, Weimerskirch, and Paar [42] had already proposed general security features to be implemented in the early 2000's, such as message authentication and encryption, as well as firewalls and secure gateways. Nonetheless, more recent works such as El-Rewini, Sadatsharan, Selvaraj, *et al.* [51] and Rathore, Hewage, Kaiwartya, *et al.* [52] have been focusing on identifying, classifying and presenting current security risks existent in a vehicular network. While there have been various solutions proposed both by researchers and industry, most of them have a tendency of focusing

on hardening older and known-to-be vulnerable protocols such as CAN. This may also be due to the fact that several real-life attacks target the CAN protocol.

As mentioned in [42], vehicle network security can be achieved as a multi-layer approach to harden several aspects of the system. Such hardening can be achieved by using measures such as sub-nets, gateways and/or firewalls, Intrusion Detection and/or Prevention Systems (IDS/IPS), encryption and authentication. While gateways and firewalls are a standard mechanism used in a vehicle, Intrusion Detection Systems (IDS) have steadily been gaining popularity within the automotive industry, and are generally implemented monitoring the CAN protocol. They are customarily used in tandem with a firewall, conjointly working towards securing the vehicle network. While a firewall restricts access to malicious actors based on a pre-defined set of rules, an IDS surveys the network for any suspicious activity and can notify a responsible party in case an attack is detected. IDS can be categorised as a Host or Network-based IDS (HIDS or NIDS), and depending on their detection methodology, can generally be categorised as a signature or anomaly-based detection [53]. Recently there has also been a noticeable trend of simulating the human immune system approach to an IDS. The gap between research and industry practice is revealed due to IDS tendency to have a high false-negative rate and require expert-level knowledge to properly utilise the generated reports [54]. There is a large number of work and surveys already done on the topic, such as [28], [55]–[58], which all review existing IDS solutions for the CAN protocol and categorise these solutions based on their detection characteristics.

A point to note however is that in recent years, there has been steady progress in suggested solutions for newer protocols such as FlexRay or Ethernet, which are currently considered possible successors to the CAN protocol. [59] proposes a FlexRay/Ethernet gateway that can monitor and support protocol security due to increasing importance of the Ethernet protocol in future vehicles. [60] suggests

to improve the transfer rate on FlexRay/CAN gateways by utilising Intelligent Detection Method to minimise false detection and retain high data transmission rates. [61] proposes a gateway-based protocol that uses broadcasting to ensure the sender's authentication for automotive Ethernet. There are also less researched but equally interesting approaches to securing vehicle networks, such as [62], which proposes solving security vulnerabilities of automotive networks using a hybrid blockchain approach.

2.2.3 Other Vehicle Security Mechanisms

While the vehicle network remains one of the most vulnerable parts, other aspects of the vehicle must be appropriately secured as well to ensure no malicious entry to attackers. This can largely be split into physical security and SW/HW security of the units present in the vehicle. This section will ignore physical security aspects of vehicles as they are not as closely related to the hardware components in the vehicle, but they should not be ignored when building a "safe and secure" product.

Hardware and software of units in vehicles are secured by means of Hardware Security Modules (HSMs), Secure Boot, several cryptographic functionalities such as proper key management schemes, resistant cryptography algorithms used for encryption or random number generators [63]. Due to how long a car's on-road life span is, engineers must also make sure to provide future-proof solutions to these mechanics, e.g. by using post-quantum cryptography algorithms. Some automotive companies such as LG Electronics are already providing such services with their Car-pay technology [64].

2.3 Firewalls

Ever since the first commercial firewall produced by Ranum was introduced in 1992 [65], firewalls have been used in numerous industries to this day, automotive industry being no different. Wolf, Weimerskirch, and Paar [42] first proposed using a firewall or gateway within a vehicle network to increase its security, and they have been a staple ever since. The remainder of this section discusses how automotive firewalls differentiate from traditional ones and how packet capturing is performed.

2.3.1 Firewalls in Automotive Vehicles

Within a classical network, a firewall is usually positioned as an entry point into a network through which all traffic must flow. This allows for monitoring and filtering of traffic at a single choke-point by following pre-defined rule sets, and can prevent malicious actors from surveying what kind of systems are present in the network [66]. Firewalls in IVNs have been discussed in multiple works so far, and have a varying level of distribution within the vehicle. While the initial idea of placing a firewall at the vehicle gateway to the internet [42] may have been enough in the early stages of vehicular networks, it is currently not adequate enough to protect the entire network. As such, there is a number of diverse approaches to such a conundrum, ranging from more traditional to distributed solutions. NXP Semiconductor [67] proposed a centralised powerful gateway with a multitude of capabilities, such as firewall functionality, data and diagnostics routing, and even intrusion detection. There are also other works, such as [68], [69], which had similar propositions. Oppositely, Rizvi, Willett, Perino, *et al.* [70] suggests not to use a single instance of a firewall, but instead have a distributed system of firewalls for each ECU present within the vehicle. This means every packet headed towards an ECU would need to pass through such a firewall, which can introduce overhead if not implemented efficiently. While there is no correct answer to how a firewall can be implemented and used

within a vehicle network, there is also a danger to introducing additional components in the network. If it is not implemented securely, it can provide another entry point for malicious actors to gain access to the entire network from a perceived "trusted zone". Not only that, but firewalls may become a choke point of information if the message delay or overhead impact the data transmission. Since modern and smart vehicles process a lot of real-time data, it is imperative that packets sent are delivered with minimal delays. Ideas such as [67] would thus produce a bottleneck for the network, and would introduce high costs in the manufacturing phase, as the system running such a firewall would require relatively significant processing capabilities in terms of computational power. For vehicles using Automotive Ethernet, firewalls could in turn be placed on the switches connecting other ECUs within a car. This solution is however limited to the Automotive Ethernet protocol, which is not quite widely implemented yet.

The thesis's solution also takes a distributed approach, and would only be required for smaller units with limited resources that cannot run the full stack of the CommonAPI framework.

2.3.2 Packet Capturing

Irrelevant of if the firewall is implemented in a distributed or centralised manner, packets being transmitted must be captured and inspected before being dropped or forwarded. This can be done in several distinct ways on a UNIX based system. A popular way of storing and parsing information is by using a Packet Capture file, also called `PCAP`. Open source tools such as `tcpdump`, `libpcap`, `winpcap` and `Snort` are examples of programs that support the PCAP file format. There is also a newer format of PCAP files called PCAPNG (PCAP Next Generation), which has a more flexible file structure and is more portable. It has been popularized by

the Wireshark tool[1], and includes additional information about the packet and the interface used to capture the packet. The main downfall of this is that PCAP files are storing information that had already been transferred, and the firewall cannot drop malicious packets in real-time by only receiving a copy of the traffic that had already transpired. In turn, some tools can capture live traffic and store the data in a PCAP format, which can be processed, and then the firewall can decide on the action to take for each packet. This however means that packet processing may be limited by basic file and memory I/O processing, which can be a considerable overhead in processes that take mili- or nanoseconds.

Another way to process data traffic is to use `iptables`, a user-space utility program which can configure the IP packet filter rules of the Linux kernel firewall. This in-turn uses the well tested solution provided by the kernel itself, which should be preferred over a personal implementation in most cases. The packets can be intercepted and parsed by using libraries such as `libipq`, which uses the kernel provided mechanism of passing the captured packets in the stack to another process in the userspace.

Should the Linux kernel provided solution not be desirable, raw sockets may also be used to capture data in transmission. Depending on the socket type being bound, specific OSI model layers can be ignored and automatically parsed as well. This can be achieved by using the `socket` system call, which is useful when working with specific protocols or features not directly accessible in available interfaces. While this is quite a fast approach, it requires a deeper level of knowledge to process the data and the data processed is usually byte arrays of raw data.

As the implementation of the firewall for this thesis was done in C++, a decision to use PcapPlusPlus[2] to capture network data was selected. PcapPlusPlus is a C++ library for capturing, parsing and crafting network packets, which is achieved by us-

[1] `https://www.wireshark.org/`

[2] `https://pcapplusplus.github.io/`

ing wrapper classes for some of packet processing engines such as `libpcap`, `DPDK` and raw sockets. As such, it was quite easy to switch from capturing real-time data to parsing data from a PCAP file, which made it simple to switch from real-time network capturing by using the `PcapLiveDevice`, to testing the firewall's functionality by using larger datasets stored in PCAP files by using the `PcapFileReaderDevice`.

2.4 Summary

A considerable amount of effort in security vehicular network protocols has gone into the CAN protocol, and a limited amount to other protocols available. While this may be due to the sheer popularity of the CAN protocol, it is speculated that is will sooner rather than later be replaced by a more modern protocol such as FlexRay or Ethernet. There is also a number of other protocols not included in this comparison, such as SecOC [71], (D)TLS [72] or IPsec [73] which are modern protocols designed for vehicle networks with security in mind. This section of the thesis thus only shortly demonstrates the most popular and well-known protocols used in modern vehicles today, although their usage may be limited with time as stronger, faster and more secure data transfer protocols are required. The following chapter will talk more in-depth about the Scalable service-Oriented MiddlewarE over IP (SOME/IP) protocol, which is another solid contender for replacing the CAN protocol.

3 SOME/IP and CommonAPI

Within the scope of this thesis, the SOME/IP protocol and the CommonAPI C++ framework are used, which are further explained in this chapter. SOME/IP is a middleware transportation protocol used in automotive solutions to relay control messages. It was originally developed by the BMW automotive company, and is now part of the AUTOSAR standard. On a similar note, CommonAPI C++ is a framework initially designed for seamless interprocess and network communication (IPC), and is now maintained and provided by the COVESA alliance.

3.1 SOME/IP General Overview

The SOME/IP protocol was designed with limited resources and scalability in mind, and can be considered for communication which previously used protocols such as CAN or MOST. As such, it supports a number of middleware features such as data serialization, service discovery, publish/subscribe dynamics, other remote procedure calls (RPC) and messaging, as well as UDP message segmentation with the SOME/IP-TP module [74]. It can also use either the TCP or UDP protocol, and message transmission and configuration depends on the protocol used. An in-depth specification of the basic protocol can be found in the AUTOSAR SOME/IP specification [75].

Data serialization is achieved using minimal resource consumption due to a minimalistic and non-descriptive format without much metadata, although latter versions support tagging data structures to some degree. This in turn means the data during transportation is quite similar to data in-memory, where only differences can come from padding used after variable-length data types. SOME/IP can serialise a set of basic and complex datatypes, from boolean to union [74].

RPC and messaging supports a number of SOME/IP messages, which are request/response, fire & forget and notification events. Depending on the message type, the communication looks either like a traditional server-client model, a broadcasting or a publish/subscribe model. Each message type has specific constraints and values within its header structure which differs slightly based on the message type.

SOME/IP Service Discovery (also known as SOME/IP-SD) is used to locate service instances using so-called "find" messages, broadcasting service offers, detecting working services as well as handling of the publish/subscribe communication model. It is implemented using UDP and broadcasting messages on a designated port, and is an optional module on top of the basic SOME/IP protocol. A more detailed explanation of the SOME/IP-SD protocol can be found in the respective AUTOSAR specification [76].

Message segmentation is a bit more complex in terms of SOME/IP messages. If SOME/IP is used over UDP, only a message that can fit within an IP packet can be transmitted, else SOME/IP-TP must be used to transfer the fragmented the message. TCP on the other hand allows for sending much bigger SOME/IP messages, as it uses the robustness of TCP. It is however advised to use TCP only if large chunks of data need to be transported and there are no latency requirements

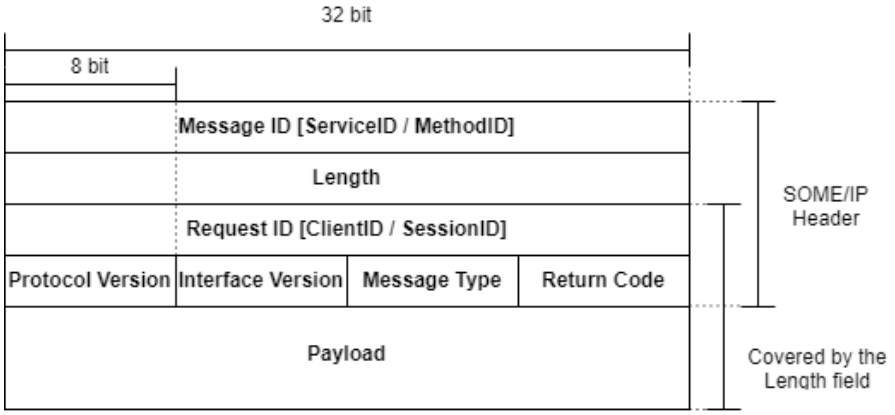

Figure 3.1: SOME/IP packet structure

for the data transmission. It is also good to note that a single packet can contain more than one SOME/IP message, which is identified by the length field present in the SOME/IP header.

3.2 SOME/IP Structure

A SOME/IP packet's structure consists of a header and payload, each of which have their own format. The structure of a SOME/IP packet can be seen in Figure 3.1. A SOME/IP header starts with a MessageID, which is further split into a ServiceID and MethodID. The ServiceID value is set to the specific service the message is trying to communicate with, and the MethodID specifies the method being called, or identifies an event which it tracks. As the MessageID is split into two 16 bit values, SOME/IP can differentiate between 2^{16} services and methods/events respectively. The length value covers the following header values as well as the length of the payload of the message. With this field, it is possible to know if there are multiple messages in the packet or not. Similarly, the RequestID is split into two 16 bit values of ClientID and SessionID. The ClientID identifies the client contacting the service, and the Session ID is a unique identifier that can determine sequential messages or requests from the same sender. The Protocol Version distinguishes the SOME/IP

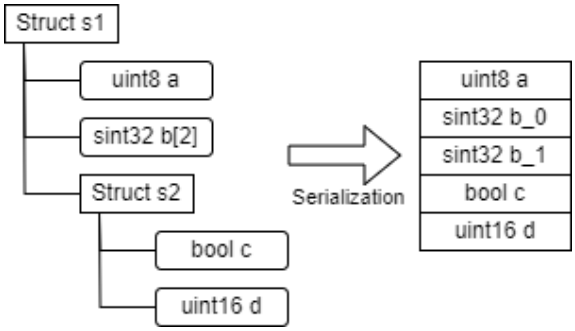

Figure 3.2: SOME/IP simple datatype serialization

version used for the header format, excluding the payload format. The Interface Version covers the major version of the service's interface it is trying to use, which also identifies the format of the packet payload. The Message Type field identifies the message type based on the values set by AUTOSAR, and the Return Code is used to signal the status of the response and weather an error occurred or not. As such, the set size of a correctly formed SOME/IP header is 128 bits or 16 bytes, which is also the minimal size a SOME/IP packet can have.

3.2.1 Payloads

SOME/IP boasts its high efficiency partially because of the simple and binary serialization of data when being transmitted. It supports both basic as well as complex data-types to be serialized. If only simple data-types with fixed length, strings or byte arrays are transmitted, there is no padding used and the transmission data is similar to that of the in-memory data. An example of such serialization can be found in Figure 3.2.

Due to time constraints, within the scope of this thesis only basic data-types with a fixed size as well as strings are under consideration. A list of these data-types can be found in Table 3.1. For a further list of supported data-types that can be transmitted by SOME/IP, the reader is forwarded to the SOME/IP AUTOSAR specification [75]. The solution presented supports deserialization of basic data

types as well as strings, to showcase how variable-length data can be deserialized and inspected.

Table 3.1: SOME/IP basic data-types under consideration

Type	Description	Size
UInt8	Unsigned integer	1 Byte
UInt16	Unsigned integer	2 Bytes
UInt32	Unsigned integer	4 Bytes
UInt64	Unsigned integer	8 Bytes
Int8	Signed integer	1 Byte
Int16	Signed integer	2 Bytes
Int32	Signed integer	4 Bytes
Int64	Signed integer	8 Bytes
Integer	Integer with optional boundaries	4 Bytes
Boolean	True/False value	1 Byte
Float	Floating point	4 Bytes
Double	Double precision floating point	8 Bytes
String	Null-terminated string	Fixed or Dynamic

As one may imagine, basic data-types that have a fixed size and no padding introduced are quite simple to serialize and deserialize. It becomes slightly more difficult once variable-length data in introduced, such as strings. SOME/IP supports deserialization of static and dynamic strings by introducing a specific format for strings which can be found in Figure 3.3. There is a mandatory length field that can be of size 1, 2 or 4 Bytes. The default value of the field is 4 Bytes if no other configuration is specified. After the length field comes the Byte Order Mark (BOM), which is a unicode character that specifies the encoding type used for the string. `UTF-8`, `UTF-16LE` and `UTF-16BE` encodings are currently supported by SOME/IP. The BOM field is either 2 or 3 Bytes in size, depending on the encoding BOM used (`UTF-8` uses 3 Bytes, `UTF-16` uses 2), and is included under the length field. After the BOM field comes the actual string contents which should fit the length

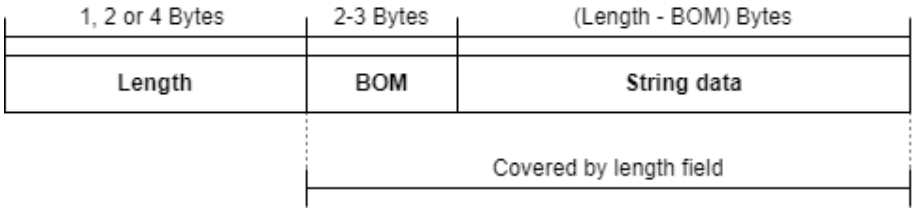

Figure 3.3: SOME/IP string structure

provided, otherwise the payload is considered malformed and may be dropped by the application based on its implementation.

Other complex data-types such as arrays have optional length fields as well, and its serialization is dependant on the configuration of the SOME/IP protocol being used. Based on the protocol configuration used, there may also be an optional length field of 8, 16 or 32 bits in front of any struct object. Within the AUTOSAR specification of SOME/IP, forward and backwards compatibility can be achieved using payload metadata in the form of IDs and tags, although this is an optional inclusion. This additional metadata relays the wire type, data identifier and the length field before the actual data being transmitted. As such, it is simple to identify the presence of optional method parameters and transmission of variables that were not relayed in the same order as that of the method signature. This metadata also helps transfer additional information on complex datatypes transmitted, and can make parsing and deserialization slightly simpler. Parsing tagged payloads is out of scope for this thesis, and should be looked further into in a separate work once an implementation that uses data tags is available.

3.2.2 Existing Implementations

An implementation of the SOME/IP protocol is also called a stack. Currently there is only one publicly available and fully implemented SOME/IP stack to the author's knowledge, which is implemented and maintained by COVESA called vsomeip[1].

[1] `https://github.com/COVESA/vsomeip`

While there are other implementations available as well, they are generally not fully implemented, such as a Rust[2] implementation which is missing some Service Discovery functionality, or a Python[3] one where SubscribeAck and SubscribeNack messages are ignored/not fully implemented yet. There had also been another implementation done by Vujanić, Trifunović, Kaštelan, *et al.* [77], however the implementation seems to not be publicly available yet. Work referencing the COVESA implementation of the SOME/IP and CommonAPI have also received "translation" efforts, such as [78], which created a Rust-based code generator, runtime and SOME/IP implementation aiming at interoperability with already existing implementations and providing higher code and memory safety, which Rust is known for. As vsomeip is thus the only fully-implemented and functional implementation currently available, it will be used as reference within the scope of this thesis, and was used to test real-time data transfer. The solution was also developed and tested with the vsomeip configuration in mind.

3.3 SOME/IP Security

SOME/IP was introduced and presented to the general public in 2012, and has been released as part of AUTOSAR since 2016. While there are no reported real-life attacks on the SOME/IP protocol that had been successful, during the BMW Private Bug Bounty program in 2022, a number of security vulnerabilities have been reported for applications using SOME/IP by Chung, Hanjun [79]. A number of other works have also reviewed the security of SOME/IP and its Service-Discovery protocol, since it is used for transferring control messages. As such, it is imperative that SOME/IP data transmission is protected to its utmost to assure passenger and vehicle safety. Works talking about its vulnerabilities can be found in the following

[2] `https://docs.rs/someip/latest/someip/`

[3] `https://github.com/afflux/pysomeip`

section, and SOME/IP recognised attack patterns are discussed afterwards.

3.3.1 Vulnerabilities

By design, SOME/IP lacks security in the sense there is no authentication or encryption information contained in a SOME/IP packet. Additionally, SOME/IP-SD which broadcasts Offer messages can be intercepted by malicious actors to determine the address an port of services running on a system which can then be directly attacked. [79] used SOME/IP-SD to identify the address, service ID and port of a running service and was then directly attacked with fuzzing. [80] similarly proposes a fuzzer for discovering vulnerabilities in the SOME/IP protocol. SOME/IP-SD messages also make Man-In-The-Middle (MITM) attacks possible [81]–[83]. Attacks such as sniffing and injection are also possible due to the lack of encryption and authentication. DoS attacks are also plausible if the service provided is flooded with too many requests at once. Lastly, since SOME/IP is implemented on top of the TCP or UDP protocol, there may be certain native vulnerabilities to the aforementioned protocols which could impact the transmission of SOME/IP messages.

Due to such vulnerabilities present in the base SOME/IP protocol, there had been works which introduced a more secure-by-design SOME/IP protocol, such as Ticket-based SOME/IP [81], secure SOME/IP [84], SESO-RC and SESO-AS [83].

3.3.2 Attack Patterns

The SOME/IP protocol by itself has a well-defined flow with a limited amount of possible exchanges. Most known attacks on the protocol interrupt that flow or abuse the fact there is no authentication information contained. Attacks that exploit undefined behaviour can affect the service or client running the SOME/IP application if not handled properly. Figure 3.4 showcases some known attacks, i.e. Request-No-Response, Response-No-Request, Response-On-Notification, extracted

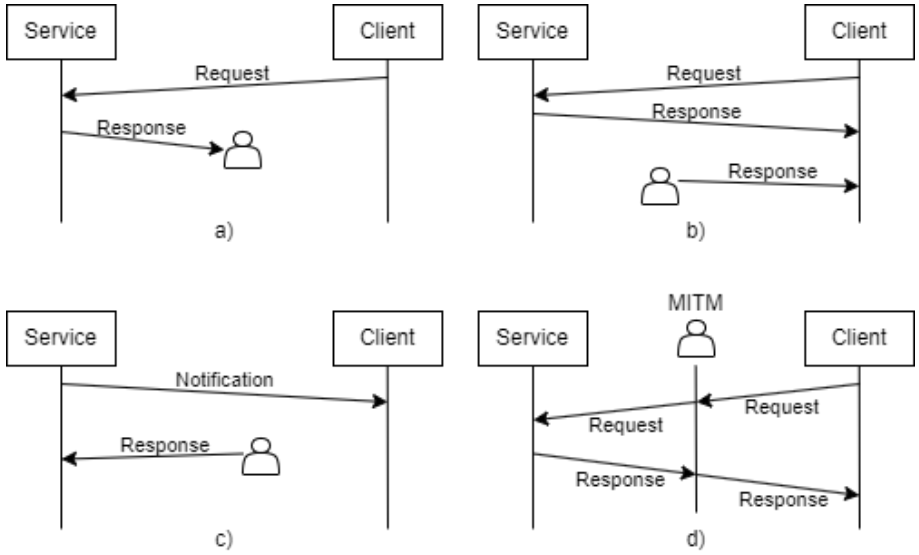

Figure 3.4: SOME/IP possible attack patterns

from [85]. Additionally, a MITM attack is presented in sub-figure `d)`.

There are also other known attack patterns with use the MITM positioning, which focus on the SOME/IP-SD implement. Known attacks are the copycat, de-association or publish/subscribe attacks presented by Zelle, Lauser, Kern, *et al.* [83]. The scope of this thesis is mainly the SOME/IP implementation, not its Servuce Discovery sub-protocol, which means such attacks will be omitted for further in-depth analysis.

For each SOME/IP request, either a response or error should be returned by the service provider. Should no response or error be returned to the client, there could be an attacker present in the network. The client application sending the request must handle the logic of hanging requests, or re-try to send the same request again after a specific time-out period. After a response or an error had been received by the client, no further error or response messages should be received based on the AUTOSAR standard specification. Should this occur, an attacker may be present. These attacks can be prevented easily with a "queue-aware" firewall which keeps track or sent and received SOME/IP packets. Similarly, no error or response should

be received on a notification message sent by a service. Should it occur, it indicates a malicious actor and can be blocked by the firewall. MITM attacks are more difficult for a firewall to block due to lack of authentication in the SOME/IP protocol, but could increase the security should the IP address be included in the firewall rule-set and its source and destination checks.

3.4 Franca framework

The Franca framework allows for integrating different software components from various suppliers using distinct runtime frameworks, platforms and IPC mechanisms. This is made possible with the Franca Interface Definition Language (IDL), with which Application Programming Interfaces (APIs) can be described in a purely textual way [86]. Together with a specified API derived by Franca IDL, Franca deployment files offer an enhanced interface specification with additional information the original API does not cover. All additional information is thus stored in a deployment model, which is mandatory for messaging protocols such as SOME/IP.

3.4.1 Franca IDL

The Franca IDL is a straight-forward, textually-based representation of an application's API, which does not require knowledge in any programming language to be able to read. This information is stored in so-called `FIDL` files), and an example of such a file is shown in Listing 1.

Every `FIDL` file corresponds to a Franca "Model", which can contain several interfaces and methods. There are also other sections which can be present in a `FIDL` file, such as imports, typeCollections, contracts, etc. Avid readers can find more information in the Franca User Guide found on the official GitHub page [86]. An interface corresponds to a provided service with its method signatures that are

Listing 1 Example of a basic FIDL file

```
interface HelloWorld {
    version { major 0 minor 1 }

    method sayHello {
        in {
            String name
        }
        out {
            String message
        }
    }
}
```

available for other clients to call. Each interface must also contain a version number. There are three different types of methods, i.e. normal methods with both in- and out-arguments, fire-and-forget methods with only in-arguments and broadcast methods with only out-arguments. Depending on the type of method, the "in" or "out" list of method arguments may or may not be present. With this simple example, an API of a service providing a method called `sayHello` is described, which takes in a string as an argument, and returns another string if successfully completed. The firewall searches and parses `FIDL` files to generate a hierarchy of models, interfaces and methods available on the system and knows which arguments need to be parsed based on the service and method ID in a SOME/IP message.

3.4.2 Deployment Files

As mentioned before, Franca deployment files in-turn provide additional information for the specified API. In case of SOME/IP, this corresponds to identification values for services and methods, encoding being used, transportation protocol used (TCP or UDP), etc.

An example of a SOME/IP Franca deployment file (FDEPL) is seen in section Listing 2, which is excerpted from the CommonAPI C++ in 10 minutes (with

Listing 2 Example of a basic FDEPL SOME/IP file

```
import "HelloWorld.fidl"

define org.genivi.commonapi.someip.deployment for interface
commonapi.examples.HelloWorld {
    SomeIpServiceID = 4660

    method sayHello {
        SomeIpMethodID = 30000
        SomeIpReliable = true

        in {
            name {
                SomeIpStringEncoding = utf16le
            }
...
        SomeIpInstanceID = 4660

        SomeIpUnicastAddress = "192.168.0.2"
        SomeIpReliableUnicastPort = 30499
        SomeIpUnreliableUnicastPort = 30499
    }
}
```

SOME/IP) guide hosted by COVESA[4]. One can notice the SOME/IP port assignment, unicast address used for broadcasting Service Discovery messages and transportation protocol choice used being TCP due to `SomeIpReliable` being set to true. Without this extra information, SOME/IP would not know what configuration or encoding to use. As such, a CommonAPI application is unable to be compiled without a corresponding FIDL and FDEPL files both present. The firewall implementation created in this thesis also extracts the ID information from each FDEPL files and and thus pinpoint which service/method is being requested in a SOME/IP message.

[4] `https://github.com/COVESA/capicxx-someip-tools/wiki/CommonAPI-C---SomeIP-in-10-minutes`

3.5 CommonAPI

CommonAPI C++ is a C++ framework for interprocess and network communication designed by BMW and published by the COVESA alliance. It provides a high-level C++ API which enables developers to easily implement lesser-known frameworks or protocols [87]. The CommonAPI C++ framework currently supports so called "bindings" for the dbus, SOME/IP and WAMP messaging protocols. The solution provided in this thesis follows the serialization and deserialization process for SOME/IP messages by the vsomeip implementation, and requires the same type of files the CommonAPI C++ SOME/IP runtime requires, FIDL and FDEPL files. The remainder of this section goes over a general overview of the CAPI framework and how it works, as well as how it handles data serialization and deserialization.

3.5.1 General Overview

To achieve seamless IPC, CommonAPI consists of a core and glue component depending on the type of application and messaging protocol is used. To specify the interface of the service, Franca IDL is used, which is them used by the CommonAPI core generator to generate proxy- and stub-code API which will be used by the actual CommonAPI application written by the developer. A similar step is done by the glue component based on the messaging protocol used, and for SOME/IP, FDEPL files are used as input. An example of how a CommonAPI application with SOME/IP messaging is created can be seen in Figure 3.5, extracted from [87].

3.5.2 Data Serialisation

While the SOME/IP specification defines how data should be serialized and deserialized, it by itself lacks any information of the services running and their API's. So while it defines how to deserialize data, it does not know the structure and sequence

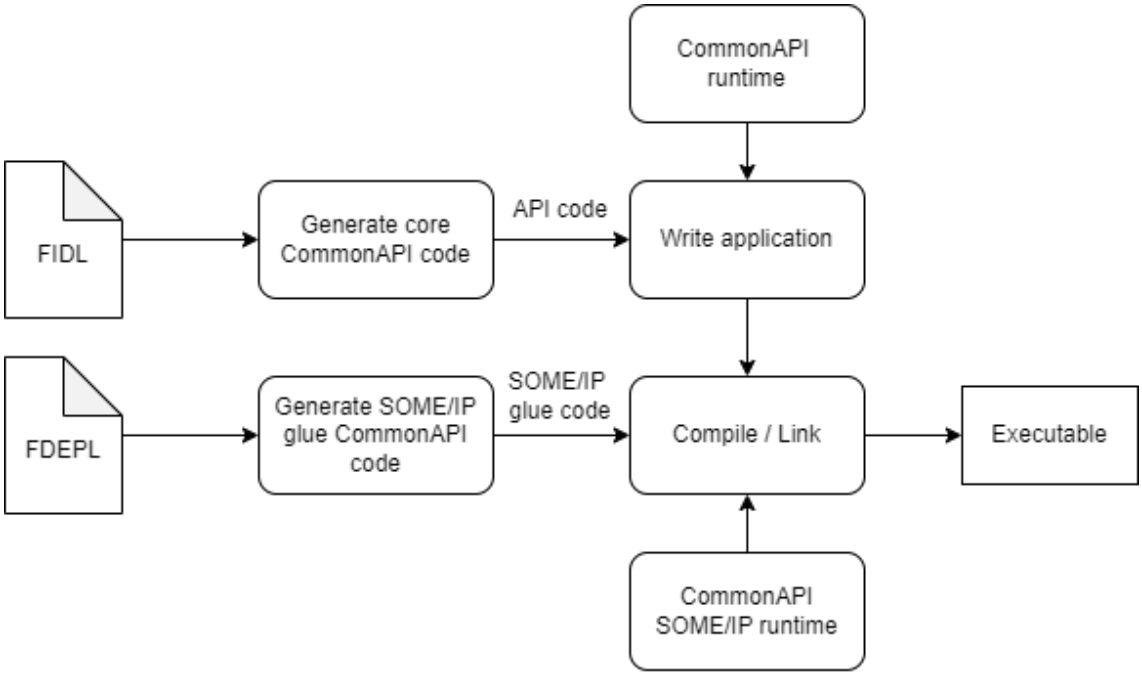

Figure 3.5: CommonAPI application creation process with SOME/IP

of arguments and types sent to a method via SOME/IP. This is where CommonAPI comes into play, as it contains deployment information for services and methods. This information is stored in deployment classes for different data types, such as `StringDeployment` or `ByteBufferDeployment`.

With this, the CommonAPI framework takes over the responsibility of deserializing and parsing the data being transferred by SOME/IP from the applications. Should a malformed payload or badly implemented SOME/IP message be received, it can be dropped before it reaches the application. If an application is poorly implemented, it could also cause a security risk or disrupt a service providing an interface to other ECUs in the vehicle. Extensive work and testing had been spent on ensuing the CommonAPI framework is hardened and well-tested, and eases the implementation efforts of developers. While this is already a solid solution to parsing SOME/IP messages and provides some kind payload parsing, there are two issues. Firstly, the parsing is done only to confirm the structure of the payload which should match the

method is it for, there are no value checks done on the contents of the payload. This means that basic attacks such as a signed integer overflow in C++ could cause undefined behaviour in the application if not properly caught. Secondly, CommonAPI is a large framework that has a relatively high resource footprint, which might not be needed in a smaller unit. The firewall solution provided within this thesis is thus a solution specific for small, resource-constrained processing units which do not require a full-stack implementation of the CommonAPI framework, but could still benefit from payload parsing as done by CommonAPI. The firewall's DPI process is largely similar to that of CommonAPI, and currently only supports the deserialization of basic data-types and strings.

4 Implementation of a SOME/IP Firewall

This chapter presents the implementation of a SOME/IP based firewall that parses FIDL and FDEPL Franca files to generate an internal representation of available interfaces and methods together with their respective IDs, enforcing firewall rules and preforming coarse DPI on the SOME/IP messages received. Due to time constraints, the DPI is done solely on basic data-types and strings. This is shown to be a good starting points for more complex data-types being included later on, as they are build on the premise of basic data-types, such as `structs` or `typedef` variables. The analysis will focus on the important parts of the SOME/IP firewall, namely the firewall rule generation and enforcing, generation of an interface list and logical connections, and the DPI by deserializing and enforcing basic constraints on the values passed.

4.1 System Level Decisions

This section will go over any system-level decisions that were made while implementing the SOME/IP firewall. This is split into a detailed explanation of the implementation and development environment as well as its constraints and requirements for others to be able to re-implemented the firewall from scratch. Currently, this design is a Proof of Concept (PoC) and could benefit from further improvements

such as additional functionality for DPI or closer-to-hardware implementation that could further minimise required resources for running the SOME/IP based firewall.

4.1.1 Implementation Environment

As mentioned before, the idea behind the SOME/IP firewall was for it to present a small, modular and resistant security mechanism that would run easily on resource-constrained systems and provide an additional layer of security for the SOME/IP protocol. As a smaller code-base and lightweight implementations further increase security as compared to larger Frameworks such as CommonAPI, this is a good alternative solution.

As many companies such as Audi, Hyundai and Toyota use components in vehicles with operating systems based on Linux, such as AGL [88], the firewall design and implementation was similarly done within a Linux environment. To ensure the solution was easily testable on different machines, a containerised solution was implemented from the start. As such, the only requirement for successfully building the firewall is a running and working installation of Docker[1]. This was used to generate a Ubuntu-based container due to its simplicity to use an a large pool of developer tools readily available. The operating system image used for running the firewall was a Ubuntu 22.04 latest image, and the `gcc` compiler. Furthermore, to ensure the implementation was fit for a smaller embedded device, a RaspberryPi model 4 B with a `Linux raspberrypi 5.15.0-1053-raspi #56-Ubuntu aarch64`, which is a RaspberryPi specific Ubuntu 22.04 Server LTS distribution, and the compiler `gcc version 11.4.0`.

[1] `https://www.docker.com/`

4.1.2 Implementation Requirements and Constraints

The SOME/IP firewall provided is written in C++17 and relies on two external libraries, namely `PcapPlusPlus`[2] and `Boost regex`[3] library, when compiling and linking. The decision to use C++ was due to its speed and numerous libraries already available, however the basic ideas behind the firewall can be implemented in any language desired. The firewall can currently be successfully compiled with the `gcc` compiler with C++17 or higher. `Clang` was not tested but should in theory not pose additional trouble with compilation.

The PcapPlusPlus library is used due to its ease-of-use and a simple approach when switching between real-time packet capture and using PCAP files, however should a more hardware-close approach be required, or the size of the compiled binaries too large, it can be omitted, and a more native approach such as raw sockets can be used. The Boost regex library is used solely for the firewall rule generation steps, and had been chosen as it is well-tested and faster than the regular expressions library provided since the C++11 standard. Although regular expressions are quite slower than pure string comparison, it was chosen due to it being simpler to implement with less overhead for parsing rules in configuration files. Should a quicker or different approach be sought, this dependency can also be easily removed.

To further develop the current SOME/IP firewall PoC, which currently supports basic data-types, a fully-supported DPI of payload packets could be added. This is due to the fact that a minimalistic working solution is preferred which can be extended in the future with the correct building blocks put in-place beforehand. The basic functionality was achieved and remaining time was spent testing the current solution to analyse the feasibility of its usability for small, embedded and resource-restrained systems.

[2] `https://pcapplusplus.github.io/`

[3] `https://www.boost.org/doc/libs/1_85_0/libs/regex/doc/html/index.html`

4.2 SOME/IP Firewall Architecture

This section presents the architecture design of the SOME/IP firewall together with its relative subcomponents. It is split into the larger components, i.e. the firewall rule-set generation and restriction, the FIDL and FDEPL parsing and generation of a hierarchy of interfaces and methods, as well as the DPI, deserialization and basic value checking done on the SOME/IP message payloads received. All files created during this thesis can be found in Appendix A.

4.2.1 General Program Architecture

The SOME/IP firewall can be split into four larger logically separate units respectively, which will be called modules henceforth. Firstly, the `Interface Parser` logical module parses the FIDL and FDEPL files used to specify the running service's APIs. The module generates a class containing all the hierarchy and information required for enforcing the firewall rules which are generated by the `Rule Generator` module. The `Rule Generator` module is tasked with the actual parsing of the firewall rules, and creating a list of rules which can be compared against in the opposite order they were written in. The `Deserializer` logical module takes care of deserializing the payload data received from SOME/IP packets captured, and does basic value checking on the deserialized data. It is also in charge of deserializing the packet header. The final logical module, the `Packet Parser` takes case of capturing the packets and executing the firewall logic with the help of the other modules. It checks if the message type of the received packet has an outstanding request present in the queue and also makes the final decision to drop or forward the data.

The general flowchart of the SOME/IP firewall can be found in Figure 4.2. At start-up time of the firewall, the firewall rule-set is parsed and generated, as well as the interface list, which is generated from parsed FIDL and FDEPL files available. Once the firewall receives a message, the contents are parsed to determine if the

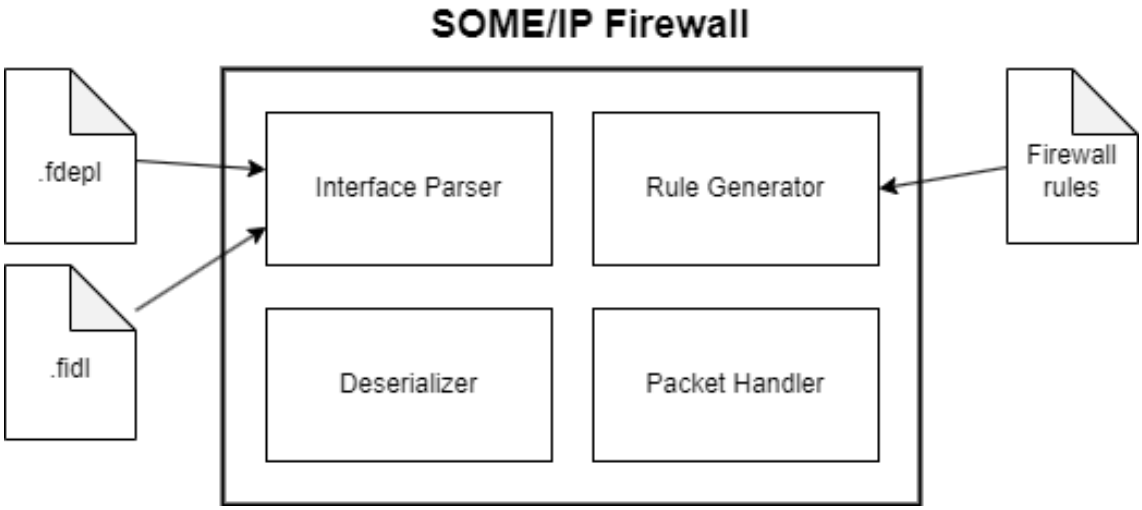

Figure 4.1: SOME/IP firewall general architecture.

message contents are SOME/IP, and if it contains a valid header structure. At this point in time, the respective data populates a C++ struct `SomeIpHeader`. If the header is valid and the `SomeIpHeader` struct is populated, the firewall proceeds to check the extracted values against the standing rule-set, else the packet is dropped. If there exists no rule in the firewall rule-set that allows forwarding for the service, method and client ID pair, the packet is once again dropped. If all previous checks pass, the firewall's internal queue is checked. This puts new requests on the queue and checks for existing requests on it if the message type if a response instead. If no previous identical request is found on the queue or if a request had been issued for the response, the message is finally forwarded to the deserializer. It searches for the method associated with the ID value extracted from the header and tries to parse the data based on the known structure. If any of the arguments cannot be parsed, the payload is flagged as malformed and the packet is dropped. Else, the packet is finally forwarded to the intended target.

Logging can additionally be added and used at every step of the way to have a clear trace of any malicious packets and their contents. Analysing the log records

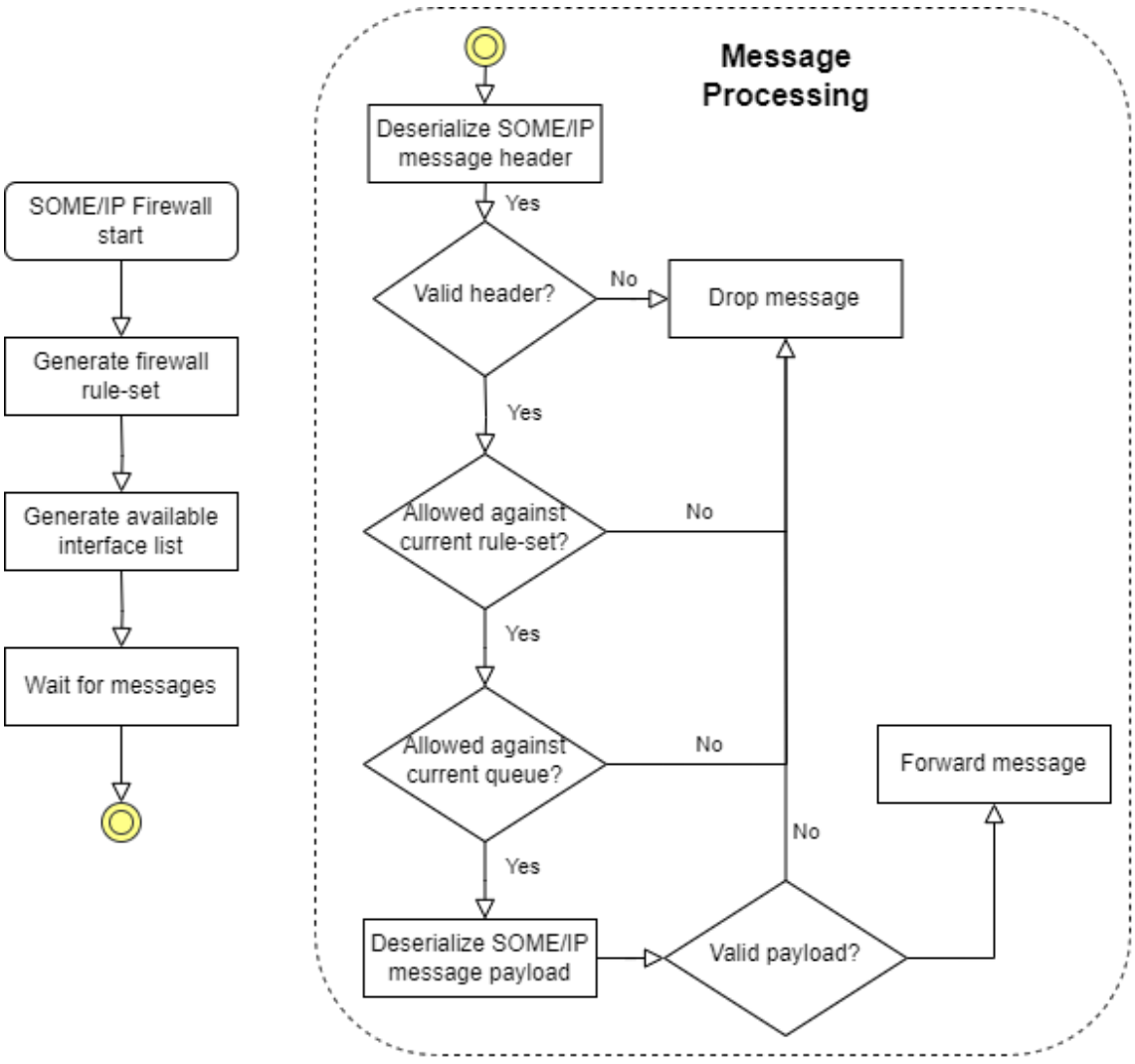

Figure 4.2: SOME/IP firewall general flowchart.

could also show the frequency of SOME/IP attacks in production settings, or be a source of information for other security mechanisms such as an IDS or IPS.

This was a quick general overview of the designed SOME/IP firewall architecture. The following subchapters will go into greater depth of the different logic modules of the firewall as well as their respective logic flow.

4.3 SOME/IP Firewall Interface Parser

One of the first two steps for the SOME/IP firewall is to generate a hierarchy of models, interfaces and methods present and running on the system which it is trying to protect. To prevent storing multiple instances of the same information which can very quickly be outdated, already present FIDL and FDEPL files are used to generate this information. The FIDL and FDEPL files are thus input to the Interface Parser, which must generate this information.

While the parsing done for the PoC SOME/IP firewall is done manually, the most future-proof solution is to use already provided API for parsing FIDL and FDEPL files from the official Franca repository, which is also what CommonAPI does. Unfortunately, the API is currently only available in Java or Xtend, which posed a problem for a pure C++ solution. While it is possible to use the Java Native Interface[4] (JNI) from the `<jni.h>` header file, this would inflate the final binary and introduce another external library dependency. Additionally, as only specific data needed to be extracted from the FIDL and FDEPL files, this was deemed unnecessary for the implementation, and the files were parsed manually. In the implementation provided, the `InterfaceParser` class follows the singleton design pattern and only has one instance.

4.3.1 FIDL and FDEPL Parsing

The constraint for successfully parsing both FIDL and FDEPL files is that the files must be well-formed, else the parsing may fail. The parsing algorithm searches for specific keywords such as *interface*, *method*, *in*, *out*, etc. Depending on the keyword found, the algorithm creates a container class. Each FIDL file corresponds to a single `FModel` class, which has a list of interfaces. Each interface corresponds to an `FInterface` class in the list, which contains its name, ID and a list of methods

[4] `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/`

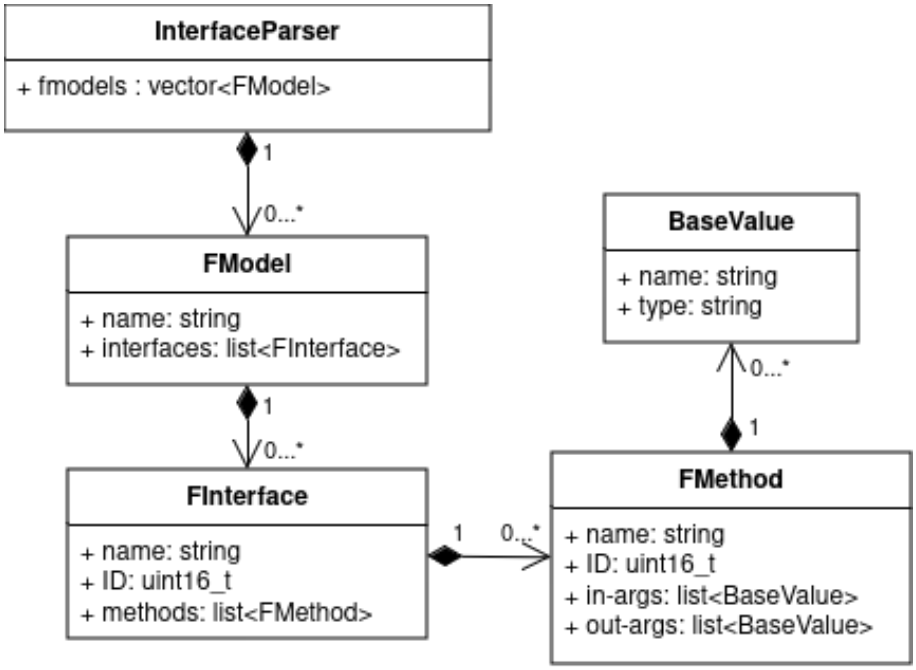

Figure 4.3: Interface Parser UML chart of classes generated.

available. Each method is stored in a separate `FMethod` class instance, with its name, ID, and list of in- and out-arguments.

Once the FIDL file had been parsed, a FDEPL file with the same name is searched for. If not found, the parsing of the FIDL file fails as well. From the FDEPL file, only specific information is extracted, i.e. the `SomeIpServiceID` and `SomeIpMethodID`. The identification values are parsed and added to the corresponding interface and method instances respectively. With this, the hierarchy of available interfaces, methods and their corresponding ID's is finalised.

4.4 SOME/IP Firewall Rule-set

Similarly as the `Interface Parser` module, the `RuleGenerator` module correlates to a C++ class that takes care of parsing and generating a rule-set for the firewall to use. The `RuleGenerator` class similarly follows the singleton pattern. While there was no appropriate existing firewall rule-set language to follow which would allow

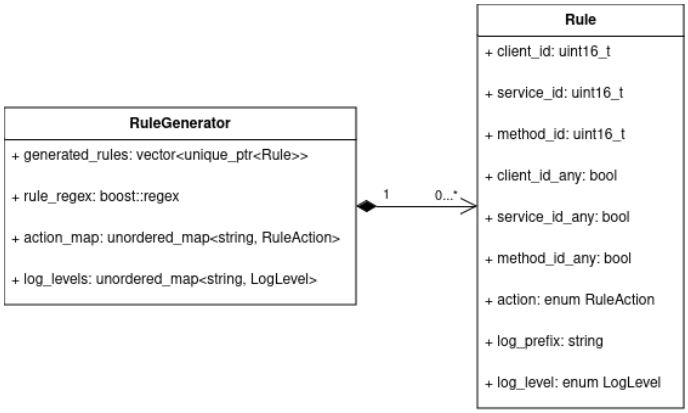

Figure 4.4: SOME/IP RuleGenerator and Rule class UML diagram.

for using ID's instead of source and destination IP addresses, an approximation to the "Rich Language" syntax[5] of `iptables` was used. This is due to the clear formatting of rules and their corresponding values of the Rich Language syntax, as well as its popularity. As it is highly used, it is easier for individuals who have worked with `iptables` and "Rich Language" syntax to read the rule-set generated for the SOME/IP firewall. An example of a custom firewall rule-set can be found in Figure 3.

4.4.1 Rule-set Structure and Generation

The SOME/IP firewall rules have a well-defined structure, which is shown in Listing 3. Each rule starts with the string "rule", followed by an arbitrary number of spaces. This makes it so the rules can be broken into multiple lines such as shown on the example for better readability. If desirable, they can also be displayed in a single line. The source section contains the `clientID` value, and can easily be extended to also include a source address following the traditional firewall rule syntax. Similarly, the destination section required a `serviceID` and `methodID`, and can be extended to include IP addresses as well. There can also be an optional section which describes the logging configuration used. Lastly, the action section's value can either be

[5] `https://firewalld.org/documentation/man-pages/firewalld.richlanguage.html`

`accept`, `deny` or `log`. Any of the values except for the action value can also be set to `any`, which matches against all values passed.

Listing 3 SOME/IP Firewall rule syntax example.

```
rule
    source clientID=2000
    destination serviceID=1000 methodID=31000
    action=accept
```

Multiple rules can be stored either in a plain singular file, or as a number of separate files in the same directory, which is taken as input by the `RuleGenerator` class. The files are parsed using regular expression matching, namely the Boost regex library, which supports multiline parsing and multiple match results. Once parsed, the rules are stored in a `Rule` class. Once the SOME/IP firewall application stops, the rules are deleted from the heap automatically. If the rule does not match the syntax shown, it will not be loaded or stored. The `Rule` class also has an overridden comparison operator, which compares the associated `clientID`, `serviceID`, `methodID` and `action`, and returns equal is all four as of the same value. The class can be found in Appendix A.

Once a packet is received and the header has been successfully deserialized and the `SomeIpHeader` struct has been populated, the firewall compares the header values extracted against the stored rule-set. The return value of the check is an action, which facilitates the message dropping, or forwarding it to next checks. If the value of any filed in the `Rule` class is equal to "any", the match is automatically true. Should no such match be found, the default response is to drop the message.

4.5 SOME/IP Firewall Packet Handler

The `Packet Handler` module can be described as the logical "brain" of the firewall. It holds a reference to both the `RuleGenerator` and `InterfaceParser`. It man-

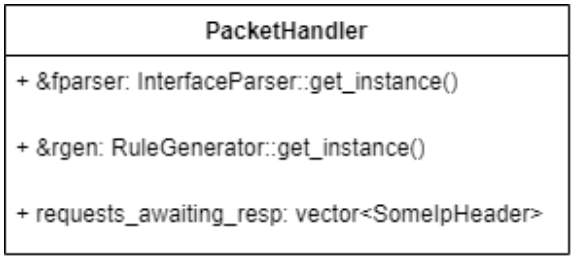

Figure 4.5: SOME/IP PacketHandler class UML diagram.

ages the logical flow of parsing the packet, from the header to the payload. If the payload has either the structure of a SOME/IP layer known to PcapPlusPlus or a general data layer with no recognisable protocol information, the `Packet Handler` instantiates a deserializer for the message and tries to parse the header. If it fails, the module drops the package. If successful, the `Rule Generator` instance is called to check against the current outstanding rule-set, and the returned action is taken. Should the packet pass the firewall checks, the `Packet Handler` then checks the state of the internal queue of processed messages.

4.5.1 Internal Message Queue

The class `PacketHandler` stores a vector of SOME/IP message headers, which have not yet received a valid response or error message. The choice for a vector container instead of a queue is due to the fact that if a number of clients sends a request at the same time, and a response is generated to a message that is stored somewhere in the middle of the queue (e.g. due to a short time for processing required to finish compared to the other requests), accessing and changing the state of the middle of the queue is a costly process, and can happen quite often. It is also not mandatory to queue responses to messages based on their arrival time, hence a vector was chosen.

The internal queue keeps track of all SOME/IP messages received by the firewall. If the message type is equal to `REQUEST_NO_RETURN` or `NOTIFICATION`, the message is skipped successfully, as those two messages require no further action from the

Listing 4 PacketHandler internal queue logic.

```
void PacketHandler::manage_req_queue:
    if (header.msg_type == REQUEST_NO_RETURN or NOTIFICATION)
        return;

    if (header.msg_type == REQUEST)
        if QUEUE is empty:
            add header to QUEUE

        match = find header in QUEUE
        if match found:
            throw error
        else
          add header to QUEUE

    else if header.msg_type == RESPONSE or ERROR
        if QUEUE is empty:
            throw std::logic_error();

        match = find header in QUEUE
        if match found:
            remove request from QUEUE
        else
            throw error
```

receiver. If the message type equals `REQUEST`, the queue is searched for an already present request that is identical. This is to protect against replay attacks and possibly injection attacks, if the attacker tries using past captured messages to impersonate an actual ECU, while the request is still present in the queue or being processed by the service. Lastly, if the message is of type `RESPONSE` or `ERROR`, the queue is checked if a corresponding request had been sent previously. This helps prevent Error-on-error SOME/IP attacks, and also helps maintain the designed structure of data flow between the service and clients. If no errors are thrown, the `Packet Handler` modifies the internal queue and continues with the payload deserialization.

The queue could still benefit from further improvements, such as checking the

state of the `sessionID` and if it was incrementing as should be. Also, if there are any messages received with a `sessionID` lower than the previously received message from the same client or service, it could be dropped as well. This would again enforce the transfer pattern of SOME/IP as is specified in the AUTOSAR specification. Currently there is also no time-constraints or queue clean-up processes specified, which would ensure that requests which were not answered after some time get removed from the queue. Should an attacker try to intentionally send a number of messages to introduce an overflow from the queue, there needs to be a form of protection such as a limit of messages from a single client IP and a time configuration after which older messages are thrown away.

4.6 SOME/IP Firewall Deep Packet Inspection

The `Deserializer` or `Deep Packet Inspection` module takes care of deserializing and parsing is handled by a `Deserializer` class, which parses the message data firstly when the message is received, to extract the header information. During this deserialization, the `SomeIpHeader` struct is populated with all the necessary information for further checks.

The `#pragma pack` calls are compiler instructions which ensure there is no artificial padding inserted between the struct members. As the `SomeIpHeader` has a very specific structure which is aligned to appropriate addresses in memory, this ensures the header is stored in memory the exact same was as the struct is declared, and each member variable can be addressed at a specific address based on its size. The Deserializer class is then called once again when the payload data is matched to a known model recognised by the firewall. This requires an already initialised instance of the class that is aware of the message length extracted from the header value.

Listing 5 SomeIpHeader struct definition.

```
#pragma pack(push, 1)
struct SomeIpHeader
{
    uint16_t serviceID = 0;
    uint16_t methodID = 0;
    uint32_t length = 0;
    uint16_t clientID = 0;
    uint16_t sessionID = 0;
    uint8_t protocol_version = 0;
    uint8_t interface_version = 0;
    uint8_t msg_type = 0;
    uint8_t return_code = 0;
};
#pragma pack(pop)
```

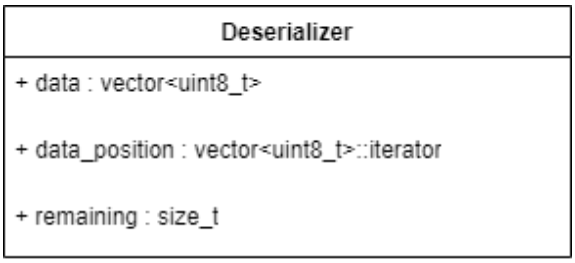

Figure 4.6: SOME/IP Deserializer class UML diagram.

4.6.1 Data Deserialization

The `Deserializer` class also hold an internal pointer to where the end of the header is in the message, and the length of the remaining data based on the length field from the header. Once the firewall rule-set and queue checks have been finalised, the `Deserializer` tries receives a variable reference to try and deserialize the data into for every argument of the method called. The deserialization is done by bit shifting the specific bytes into the appropriate positions, and assumes both the transmission data as well as the CPU running the code to be little-endian type. This is due to the fact that `vsomeip` transmits data little-endian, as well as that newer CPUs all use the same byte order, whereas big-endian is used for most network-based and older protocols, as well as older CPUs. Should a more portable solution be required,

automatic checking and if-else include's can automatically detect and use the correct byte order for the CPU running the SOME/IP firewall. The deserialization fails if the data is longer than the specified length field in the header, or if the value deserialized fails basic value checks. The value checks are done for every data type separately, and currently only check if the values are within the range of available values, i.e. if the value deserialized is larger than the minimum or maximum available value, the parsing fails. This is enforced by using the `<limits>` header, and checks such as The DPI could always be expanded for use-cases specified by the systems or services

Listing 6 DPI basic value bound checks.

```
struct SomeIpHeader
    if (value < std::numeric_limits<uint32_t>::min() ||
        value > std::numeric_limits<uint32_t>::max())
        return false;
```

running on the ECU, and could be further improved by more rigid checks.

4.7 Further Hardening

Since each additional line of code introduces the possibility of additional bugs or vulnerabilities being exploitable, each new component should be additionally hardened to ensure the continued security of the system. In the example of the SOME/IP firewall, several additional security mechanisms can be implemented to ensure its robustness.

4.7.1 Securing Input Files

The SOME/IP firewall requires input files to generate an internal rule-set and API hierarchy, hence there is an obvious need to ensure such files cannot be tempered with before the firewall is running. Should such a vulnerability exist, an attacker can easily modify the input files to change, delete or add arbitrary rules, which would

nullify its security extension of the system. An attacker could similarly modify the FIDL or FDEPL files used as input for the firewall to launch a form of DoS attack, by making the firewall unintentionally drop valid SOME/IP packets. Such important files should thus be stored on partitions of the system which are read-only and integrity-protected, and appropriate permission management should be implemented to prevent malicious actors from modifying the files. Kernel-enforced tools such as SELinux [89] could also be used to ensure only authorised applications have access to such files.

4.7.2 Memory Safety

Once the input files are loaded, parsed and its contents have generated an internal rule-set or available API hierarchy, those contents are stored in memory. The rule-set of the firewall is stored on the heap, and the API hierarchy generated from FIDL files is stored on the stack. As such, it is once again important that the input files are loaded from a controlled partition of the system which cannot easily be modified. The environment variable which allows for the firewall to load files from the path given should also either be checked or removed. Stack canaries are generally used together with Address Space Layout Randomization (ASLR). This can be used with the `gcc` or `Clang` compiler using the `-fstack-protector` and `-fPIE` flag, however they can introduce an overhead. The heap memory can also be additionally protected using `mprotect`, such as setting it to read-only once all the rules have been loaded by using the `PROT_READ` flag.

4.7.3 Containerisation and Fail-Safety

As the SOME/IP protocol transfers control-level data through the vehicle, it is important that the SOME/IP firewall does not impact or block the data transmission should it fail or crash. On reasonably-sized systems that can support deploying

applications in containers such as Docker, the SOME/IP firewall can be deployed on such a container. Should it at any point crash, the application can easily be re-deployed by re-composing the container. In such a case, it should be able to detect the environment in which it is deployed and configure packet capturing at startup time. Should no dropping of packets ever be desirable action in fear of important control-level data being lost, the firewall can also be configured to never drop packets, but instead log any suspicious activity and regularly send reports to an authorised person or server, mimicking something of an IDS instead.

If a large number of data is being transmitted, the firewall could also contain a degree of load-balancing, which would distribute the workload across multiple instances of the firewall, so that no single instance is overburdened and no bottleneck in introduced due to the firewall. Additionally, by containerising the firewall, its functionality is isolated from the outside system, should an attacker successfully gain access to the SOME/IP firewall application.

5 Testing and Validation

The main validation of the designed and presented SOME/IP firewall was done by compiling and running a benchmarking application, which took a number of generated PCAP files as input, and took action based on the environment set-up and what was being tested. The base classes and functionality of the firewall was thus manually tested during the benchmarking process. This chapter presents the testing environment in which the benchmarks were ran, and goes over the testing and validation of the different logical modules of the SOME/IP firewall.

5.1 Testing

To benchmark and test the implementation of the SOME/IP firewall, four different datasets were created, which were stored in PCAP files. There are some existing datasets for SOME/IP header attacks such as the dataset generated by Alkhatib, Ghauch, and Danger [85] or Alkhatib, Mushtaq, Ghauch, *et al.* [90], however they do not contain payload data for the SOME/IP message, and could thus not be used for testing the DPI module of the firewall. They were however usable for testing the `PacketHandler` internal queue implementation. The dataset provided by [85] used the SOME/IP Generator[1] tooling, which is publicly available on GitHub. The tooling generates a PCAP file based on the configuration used. The tooling also provides a way to test malicious attacks, as it implements a MitM attacker which

[1] `https://github.com/Egomania/SOME-IP_Generator`

modifies messages caught between the service and clients. There is a number of attacks implemented, and custom ones can easily be added to the tooling.

The attacks taken under consideration, which were first introduced in Figure 3.4, are the error-on-error and error-on-event attacks, which are logical attacks that target the SOME/IP data transmission specification by responding to messages that should in theory not receive any further response, such as the `NOTIFICATION` or `ERROR` message type. Additionally, a form of MitM attacks are taken into consideration, where an attacked tempers with the data payload or intentionally generates a malformed payload, which are tested with the malformed payload dataset.

5.1.1 Datasets

The error-on-error and error-on-event datasets have been extracted by using the `erroronerror2.csv` and `erroronevent2.csv` files[2] respectfully, which are part of the test data used on the IDS solution provided by [85]. They were generated from the CSV files provided by a small Python script, which can be found in A. The normal and malformed payload datasets however had to be generated from scratch, as they had to represent and generate the correct payload structures and ID's based on the API's defined in FIDL files. This was done with the same tooling, the SOME/IP-Generator, with a few changes.

The tooling had several issues in its base implementation. Firstly, it wrongly sent `NOTIFICATION` SOME/IP messages via client. As stated in the AUTOSAR specification, `NOTIFICATION` messages are sent by the service provider when there is an event or notification notice. The `clientID` of such a message should also be set to `0`, which was not the case for the SOME/IP Generator implementation written for Python3. Clients arbitrarily sent `NOTIFICATION` messages to the service, which thus had to be removed from the tooling for proper validation. As such, no testing

[2] `https://github.com/Alkhatibnatasha/SOMEIP_IDS/tree/main/Data/Test`

or support for `NOTIFICATION` messages is currently available both in the normal and malformed-payload generated dataset as well as the SOME/IP Firewall. Secondly, the payloads originally generated by the tooling were always randomly generated from a set of strings, which made the payload a single string with no proper encoding or structure. As such, a forked version of the tooling was used, which was aware of the interface and methods' structure. The tooling would thus generate random payloads based on the interface and method structures used for testing and verifying the benchmarks. An example of this set-up can be found in Figure 5.1. Additionally, the forked version of the tooling added a custom attack pattern, which intentionally generated a malformed payload. There was a bit of randomness involved in the type of malformed payload would be generated, as a number of different payloads being tested provided a more realistic verification. Note that the SOME/IP Generator also does not produce time-ordered PCAP files, and can be properly ordered using the `reordercap` tooling provided by the wireshark-cli package. All files were processed to be properly time-ordered.

During testing, it was also noticed that a there are instances where `serviceID` values in client messages are repeated, which does not follow the AUTOSAR specification. This excludes the cases where an `ERROR` message was received by the client, and the client tries re-send the same message with an identical `serviceID` again. As such, there might be a higher number of packets dropped due to the internal queue of the firewall, since it expects incrementing values that do not repeat multiple times.

The number of successful attacks done can be found in Table 5.1. The normal dataset is a "clean" dataset with no malicious packets sent. It was used for testing the firewall rule-set restrictions, and has been generated with the same configuration values as those in [85]. The error-on-error dataset is configured to send error messages when a client receives an error message from the service, meaning an error occurred when processing the original request. This is a known attack that can dis-

Table 5.1: SOME/IP datasets generated for testing.

Attacks	No. of packets	No. of attacks	Notes
Normal	5284	0	No attacker is present
Error-on-error	1810	90	Attacker sends error messages after an error was received
Error-on-event	1524	54	Attacker sends error messages after a notification was received
Malformed payload	6163	537	Random malformed payloads generated

turb the service's working if not properly handled, and can quite easily be stopped with the internal queue implementation of the firewall. The same can be said for the error-on-event dataset, the main difference being an error message is sent by an attacker after a service sent an event message. The malformed payload dataset is the custom crafted attack which contains packets with malformed payloads. This dataset was used to benchmark and test the robustness of the DPI module of the SOME/IP firewall.

5.1.2 Environment

To ensure the PoC implementation is tested in an approximation of a realistic embedded environment, a RaspberryPi 4 model B was used. The CPU is a Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64- bit SoC running at 1.5GHz. The memory is a 4GB LPDDR4-3200 SDRAM, and its operating system is `Linux raspberrypi 5.15.0-1053-raspi aarch64 GNU/Linux`. This is an approximation of an average sized ECU in a modern vehicle. The version of `gcc` used to compile the firewall is `gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1 22.04)`, its target architecture `aarch64-linux-gnu`.

The process of benchmarking the implementation was done by using PCAP files to have a reproducible input and output, which could be tested on different devices

later on. These files were generated from a configuration of two different environments. The configuration of the `error-on-error` and `error-on-event` environment is the same as presented by Alkhatib, Ghauch, and Danger [85], its excerpt can be found in Table 5.2.

Table 5.2: Testing environment configuration based on [85].

Parameters	Description	Value
Servers	No. of servers configured	8
Clients	No. of clients configured	8
Attackers	No. of attackers configured	1
Services	No. of services offered	3
No. of packets	No. of packets generated per client per service	50
No. of attacks	Rate of attacker	10
Min. response time of attackers	Min. response time in ms	1
Max. response time of attackers	Max. response time in ms	3
Implemented Attacks	Which attacks were used when generating the dataset	Error-on-error Error-on-event

The environment for testing the DPI and firewall rule-set can be found in Figure 5.1. There is one service provider and three clients, together with one attacker. The server provides 17 methods, whose input and output arguments were specified in a FIDL file. Each basic data type corresponds to one method, to ensure testing coverage of every basic data-type. The remaining methods are a mixture of data-types in a random order.

5.1.3 Methodology

To ensure the SOME/IP firewall can run on embedded, resource-limited systems, the most important measurements done were the time it took on average to parse a message, i.e. how long is the message delayed for delivery, and what is the memory footprint of such an application on the system. While this implementation is only a

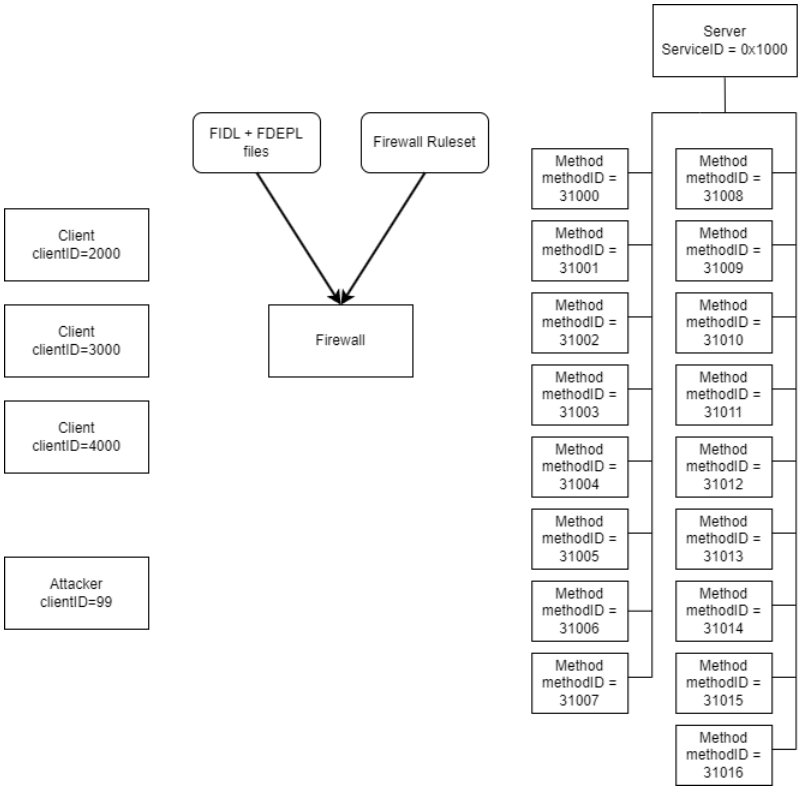

Figure 5.1: Testing service and client configuration for testing DPI.

PoC and could always be more optimised, rough measurements of the unoptimised implementation can be viewed as guidelines and an insight of how viable such a firewall is for embedded systems. As such, the benchmark binary itself was compiled with the minimum optimisations available, i.e. with the `-O0 -g` flags set. This compiled the binary with debug information present, which was necessary when running the `valgrind`[3] tooling against the binary.

The time measurements were done using the `<time.h>` header file together with the `clock()` call. The time ticks reported by the `clock()` function are using the `CLOCK_PROCESS_CPUTIME_ID` kernel clock implementation available since `Linux kernel v2.6.12`. The memory measurements were done using valgrind's `massif`[4] tool to measure heap and stack allocations for the binary.

[3] `https://valgrind.org/`

[4] `https://valgrind.org/docs/manual/ms-manual.html`

5.2 SOME/IP Firewall Benchmark Implementation

The SOME/IP firewall benchmark implementation can be configured to print debug information using the `SOMEIP_FIREWALL_DEBUG` environment variable. By default, printing debug information to the console is turned off. An input PCAP file can be specified with the `SOMEIP_FIREWALL_IN_FILE`, and an output file which logs the results of the benchmark can be specified with the `SOMEIP_FIREWALL_OUT_FILE` variable. The benchmark program takes a PCAP file as input and parses all available packets found in it. Each parsing of a packet is timed and stored in a vector for later, as can be seen in Listing 7. After all the packets in the PCAP file have been parsed and appropriate action has been taken by the SOME/IP firewall, an average time and average clicks are computed and reported.

Listing 7 Timing packet parsing function

```
...
clock_t t1 = clock();
p_handler.consume_packet(parsedPacket, stats, print_debug_info);
clock_t t2 = clock();
...
```

The most important pieces of the SOME/IP firewall implementation are the internal queue and the DPI parsing of the message payloads, as they present a new approach to securing SOME/IP message transferring, and are the additional knowledge contribution gained within this thesis. As such, these validations took most effort and analysis time. The compiled benchmark binary was $3.8MB$ in size, and was linked against several libraries.

5.2.1 Packet Handler Queue Validation

The `PacketHandler` class and its internal queue was tested by running the benchmark program against the error-on-error and error-on-event dataset. The error-on-error and error-on-event datasets validated the queue implementation and tested its

efficiency. While this was being validated, there was a single firewall rule in the rule-set, which allowed all connections. As such, all packets were being forwarded when checked against the outstanding rule-set.

5.2.2 DPI Validation

The `Deserializer` class was tested against the normal and malformed-payload dataset. The normal dataset was used to analyse the average packet parsing resource and time-wise would be, while the malformed payload dataset validated the implementation of the DPI and its efficiency for detecting malformed payloads, which would prevent applications to spend time and processing power on messages that cannot be successfully parsed either way. Similarly to the `PacketHandler` module validation, there was a single firewall rule enforced at the time, which allowed all connections.

5.2.3 Other Validations

Parsing a SOME/IP header and enforcing a firewall's rule-set against the parsed values or the parsing of the FIDL and FDEPL files is not the main contribution of this thesis, and as such the testing of these components was minimal. The normal dataset was used to measure the time taken to drop or forward packets based on the configured rule-set available, and the `InterfaceParser` was tested against the normal dataset by ensuring that every packet and its payload was parsable by the SOME/IP firewall, since that meant the `InterfaceParser` generated an internal hierarchy representation which corresponded to the payloads generated by the forked SOME/IP Generator tooling.

The more interesting measurements for the `RuleGenerator` module of the firewall are instead a comparison of the prolonged parsing for an increased number of rules each packet must check against. This means that there is an optimal number of

rules, after which the slow-down of parsing and checking the packet against a rule-set becomes too large for embedded and real-time systems and their requirements in regards to data delay. This was thus additionally measured and analysed in Chapter 6.

6 Results and Analysis

This chapter goes over the testing and validation results of the SOME/IP firewall benchmark implementation and its logical modules. The average time and resources spent on parsing a SOME/IP packet is analysed, the results of the error-on-error and error-on-event datasets are compared to that of the work done by Alkhatib, Ghauch, and Danger [85], as the same datasets were used and can thus be compared. The DPI functionality of the SOME/IP firewall is discussed, the heap and stack usage of the PoC is examined and a breakdown of an optimal rule-set is considered. The chapter concludes with a discussion of possible further improvements to the SOME/IP firewall proposed in this thesis and its usability in the world of embedded systems for automotive use-cases.

6.1 Dataset Validation Results

As mentioned in previous sections, certain datasets were used to validate different modules of the SOME/IP firewall. The normal dataset was used to validate the basic implementation of the PoC firewall, as well as to analyse the average time and resources required for checking received packets against the SOME/IP firewall rules and doing DPI on the payload data. The error-on-error and error-on-event datasets validate the implementation of the `PacketHandler` queue which keeps track of sent and received requests and responses, and drops any packets not conforming to the AUTOSAR data-flow specifications. The final dataset containing malformed

payloads validated the implementation of the `Deserializer` module of the firewall and analyses the efficiency of doing DPI on SOME/IP packets.

6.1.1 Normal

As mentioned in subsection 5.1.1, the SOME/IP Generator tool has a tendency to generate SOME/IP traffic to sometimes does not correspond to the AUTOSAR specification of the SOME/IP protocol. While this may be due to a naive implementation of the SOME/IP protocol, it causes there to be dropped packets by the SOME/IP firewall implementation against the normal dataset with no attacker present. The results of the benchmark can be found in Table 6.1.

Table 6.1: Benchmark results ran against the normal dataset.

Value Name	**Description**	**Value**
No. of packets	No. of all packets in the PCAP file	5284
No. of malicious packets	No. of packets generated by an attacker	0
No. of parsed packets	No. of packets that were successfully parsed	5262
No. of firewall rules	No. of firewall rules packets were checked against	1
No. of dropped packets	No. of packets that were dropped by the firewall	22
Firewall rules	No. of packets dropped due to the firewall rule-set	0
SessionID queue	No. of packets dropped due to the sessionID internal queue	22
DPI	No. of packets dropped due to unsuccessful payload parsing	0

After manually inspecting the normal dataset to analyse if the internal queue is faulty, a trend for the dataset generated by the SOME/IP Generator tool emerges. As visible in Figure 6.1, the first message with values of `serviceID = 0x1000`, `methodID = 0x7921`, `clientID = 0x07d0` and `sessionID = 0x1` is sent from the client at IP address 10.1.0.2 to the service provider at 10.1.0.1.

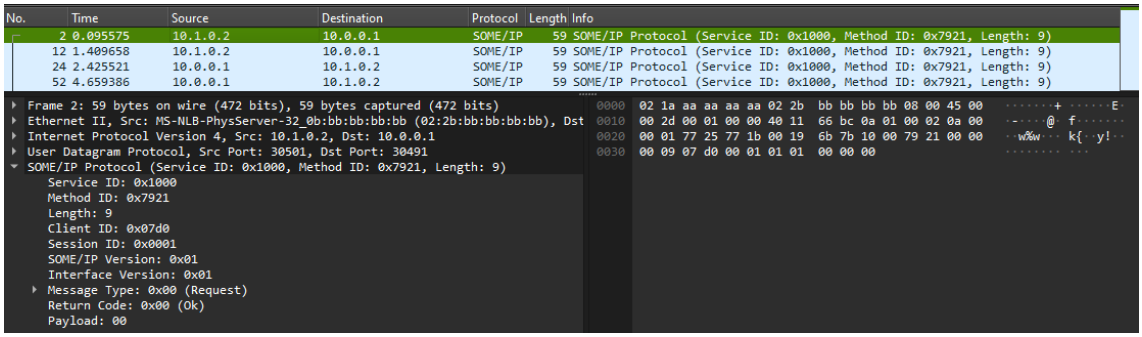

Figure 6.1: First request message by client from IP 10.1.0.2

Some time afterwards, the same message with identical values except for the payload is sent from the client to the service provider once again, without waiting for a response to the first request. This message, seen in Figure 6.2 is thus blocked by the internal queue in the `PacketHandler` class, as such a request already exists and has not yet been answered.

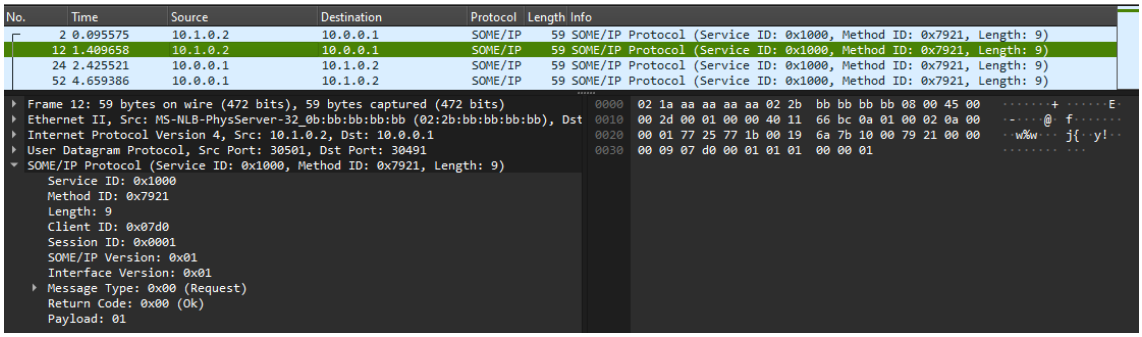

Figure 6.2: Second request message by client from IP 10.1.0.2

The service provider then responds to the first message sent by the client, as shown in Figure 6.3, which is allowed by the firewall, and the initial request is removed from the internal queue, considered to be answered by the firewall.

However, as this dataset was generated at a different point in time with no firewall in between, the server still received the second request with the identical values, its implementation apparently did not check for such cases. As such, the server generated a second response to the requests it received, and thus it back to

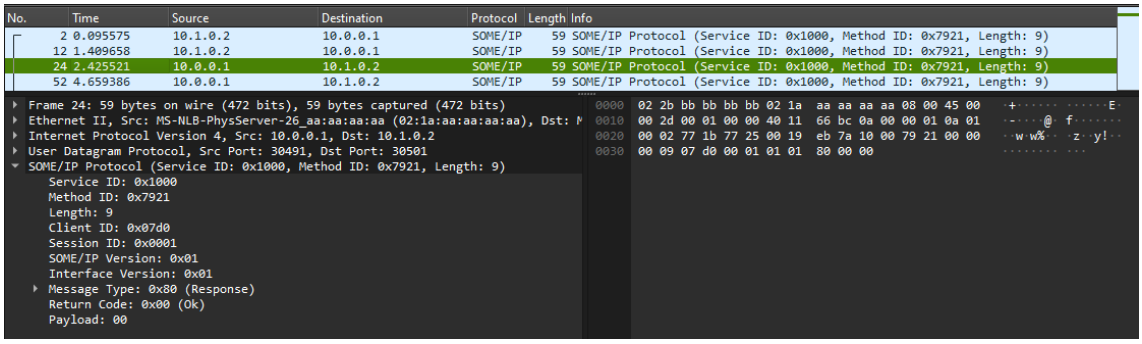

Figure 6.3: First response message by server from IP 10.1.0.1

the client, as seen in Figure 6.4. Because the initial request is considered answered by the firewall, the second response comes across as a response with no request, which is blocked by the SOME/IP firewall implementation.

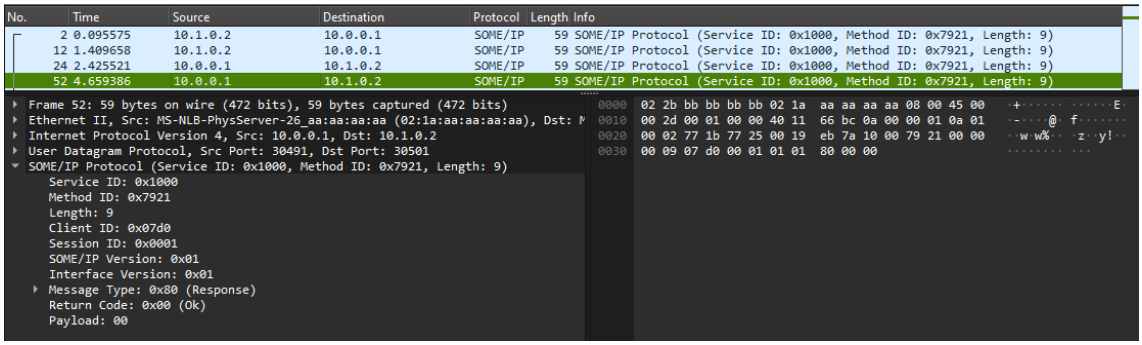

Figure 6.4: Second response message by server from IP 10.1.0.1

Due to this, an inflated number of packets in the normal, error-on-error, error-on-event and malicious payload datasets are dropped by the SOME/IP firewall implementation. This should hence be taken into consideration when going over the results produces by the firewall benchmarks.

The normal dataset was also used to establish a baseline of how long payload parsing and DPI take for each packet. The value returned by the `clock()` call is of type `clock_t`, and represents the number of clock ticks that have passed since the beginning of the packet parsing function until it has finished. The average clock tick was 32.3935, which can be translated into seconds by dividing the value by

`CLOCKS_PER_SEC`, which is equal to $3.23935e^{-05}s$ or $0.032395ms$. The median clock tick value was 29, which equals to $0.029ms$. These values can be influenced by a number of reasons due to the short duration, such as logging or writing messages out to `stdout` or `stderr`. Should the benchmark be ran without writing the logging information into a file, there will be a noticeable slowdown to the parsing process.

6.1.2 Error-on-Error and Error-on-Event

The error-on-error and error-on-event datasets have both been generated with a different environment configuration than the other datasets, and as such do not fit the FIDL file specifications. These datasets also contained 8 clients and 8 servers each, which means a larger number of duplicate requests and responses are contained in the dataset.

Error-on-Error

The error-on-error dataset had been labelled and thus the number of attacks generated and labelled as malicious can easily be found. Looking at the `erroronerror2.csv` file and filtering the labels, we can see that 90 entries have a different label than the rest, meaning there are 90 malicious packets in the dataset. Table 6.2 contains the results of the firewall benchmark running against the error-on-error dataset.

As we can see, the firewall dropped 220 packets due to internal queue mismatching. After manually digging through the PCAP and log files of the firewall, we know that 73 messages were dropped due to the implementation decisions of a client not waiting for a response to send an identical request, as described in the normal dataset analysis. This means that 147 packets are error messages that have been dropped due to no outstanding request being present on the internal queue. That is still 57 messages more than what the dataset labelling shows.

Out of the 1810 packets contained in the error-on-error dataset, 678 were request

Table 6.2: Benchmark results ran against the error-on-error dataset.

Value Name	Description	Value
No. of packets	No. of all packets in the PCAP file	1810
No. of malicious packets	No. of packets generated by an attacker	90
No. of parsed packets	No. of packets that were successfully parsed	0
No. of firewall rules	No. of firewall rules packets were checked against	1
No. of dropped packets	No. of packets that were dropped by the firewall	220
Firewall rules	No. of packets dropped due to the firewall rule-set	0
SessionID queue	No. of packets dropped due to the sessionID internal queue	220
DPI	No. of packets dropped due to unsuccessful payload parsing	1590

messages, 562 were response messages and 320 were error messages. The reason is similar to that of the normal dataset. An example of packets that are not marked as malicious, but have still been flagged and dropped by the firewall can be observed in Figure 6.5 below. The first client request passes normally, and is stored on

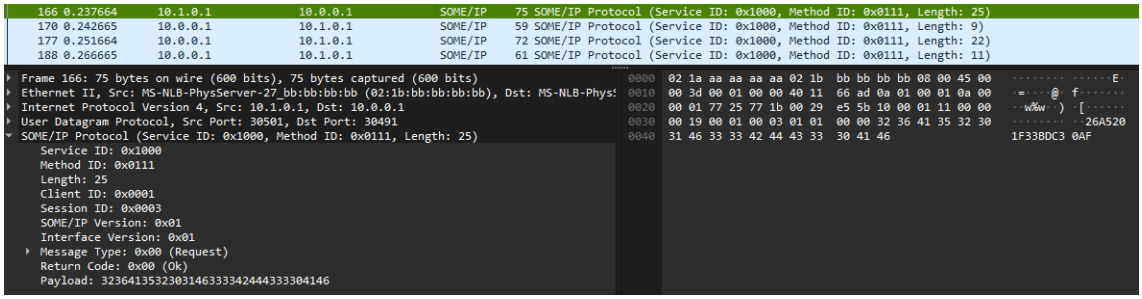

Figure 6.5: Request message by client from IP 10.1.0.1

the internal queue. The response also successfully passes and the initial request is removed from the queue, as can be traced in the PCAP file and seen in Figure 6.6. The server then sends another identical response as seen in Figure 6.7, which is dropped by the SOME/IP firewall, as there is no more request that can be answered.

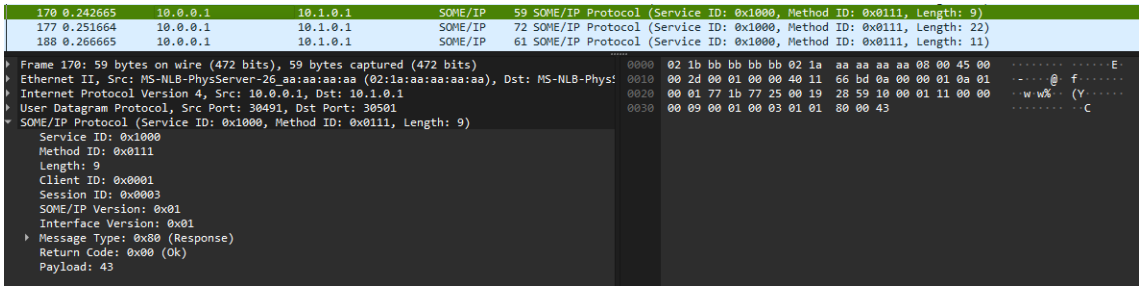

Figure 6.6: Response message by server from IP 10.0.0.1

Lastly, the server sends yet another error message with the same sessionID to the

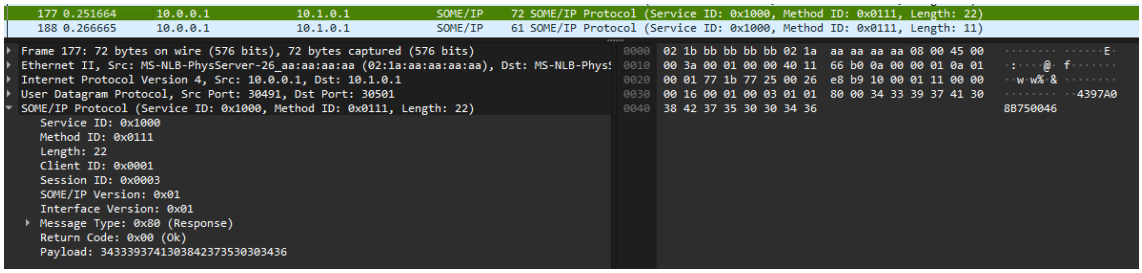

Figure 6.7: Second response message by server from IP 10.0.0.1

client at 10.1.0.1, but again the packet is flagged and dropped by the firewall due to a missing request to match the error message. It is also important to note that this packet was not marked as malicious in the `erroronerror2.csv` file. Due to this, the

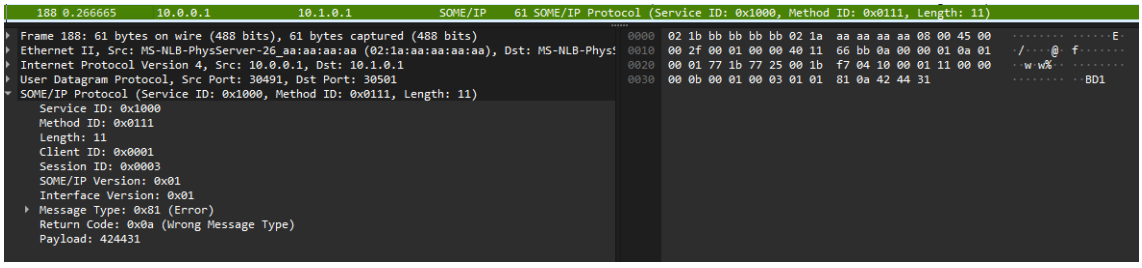

Figure 6.8: Third error message by server from IP 10.0.0.1

number of packets dropped by the internal queue is inflated, however the firewall implementation is enforcing the AUTOSAR specification of the SOME/IP protocol and its data transfer. As such, we can say the firewall has flagged and dropped

100% of malicious error-on-error packets that aim at disrupting the flow of the service providers, be it by design or unintentionally. This is an increased precision compared to the fluctuation range of 61-91% obtained by the IDS implementation introduced in [85].

The error-on-error dataset took a significantly longer time than the baseline measurements, which may raise questions. The average clock tick was 173.12, which is equal to $0.17312ms$. The median clock tick value was 153, which equals to $0.153ms$. Since the same amount of firewall rules were enforced both for the normal and error-on-error dataset, the measurement was not impacted due to rule parsing on matching. Because the datasets did not conform to the FIDL API's the firewall had loaded, the parsing failed, which introduced a delay due to the fact that every model stored by the firewall was checked, and once a match in service and method ID was found, parsing was unsuccessful. As such, it is safe to assume that the slowdown was introduced by both the internal queue handling as well as the failed matching and DPI. This could also be due to the fact there was an increased amount of clients and servers which inflated the number of entries in the queue. The slowdown introduced by the internal queue was mainly the searching and removing items from a vector, as pushing back data is relatively quick in comparison.

Error-on-Event

The error-on-event dataset had been labelled similarly to the error-on-error dataset, which means we can once again easily extract the number of malicious attacks labelled as 54 entries. Table 6.3 contains the results of the firewall benchmark running against the error-on-event dataset.

Similarly to the error-on-error dataset, the number of packets dropped by the SOME/IP firewall implementation is higher than the number of packets labelled as malicious. The reasoning stays the same as for the error-on-error dataset. The

Table 6.3: Benchmark results ran against the error-on-event dataset.

Value Name	Description	Value
No. of packets	No. of all packets in the PCAP file	1524
No. of malicious packets	No. of packets generated by an attacker	54
No. of parsed packets	No. of packets that were successfully parsed	0
No. of firewall rules	No. of firewall rules packets were checked against	1
No. of dropped packets	No. of packets that were dropped by the firewall	66
Firewall rules	No. of packets dropped due to the firewall rule-set	0
SessionID queue	No. of packets dropped due to the sessionID internal queue	66
DPI	No. of packets dropped due to unsuccessful payload parsing	1458

firewall logged and dropped 56 error and 4 response messages that did not have a matching request message in the internal queue. The remaining 6 messages were duplicate request messages. The IDS implementation of [85] reached a solid 87-100% range of successfully recognised and categorised malicious packets, however the firewall implementation has the additional benefit of being able to not only recognise but also protect from such attacks. As such, the firewall solution introduces an additional benefit over that of a passive IDS solution.

The error-on-event dataset has similar average and median measurements, the average clock tick being 163.209, which is equal to $0.163209ms$, and the median clock tick value being identical to the error-on-error measurements, i.e. 153, which equals to $0.153ms$. Since the same number of servers, clients and remaining environment configurations were used, the similar measurements are not too surprising. Similarly to the error-on-error measurements, there were no firewall rules enforced, and the service and method ID's did not match that of the API provided by the FIDL file, which introduced an additional delay to the results.

6.1.3 Malformed Payload

The malformed payload dataset had been generated using the same SOME/IP Generator tooling, however with slight modification to make it aware of the methods and its arguments defined in the FIDL file. The dataset was then implemented similarly to any other attack pattern provided by the tool, and the SOME/IP packet generated would either have an incorrect data type as some arguments, or would generate a random value outside of the allowed bounds for the data-type. While fuzzing may be more beneficial at actually uncovering the implementation vulnerabilities or issues with the DPI of malicious packets, the time constraints did not allow for it. This could however become an interesting future research or implementation for the SOME/IP firewall PoC implementation. The results of the benchmark can be found in Table 6.4.

Table 6.4: Benchmark results ran against the malformed payload dataset.

Value Name	Description	Value
No. of packets	No. of all packets in the PCAP file	6163
No. of malicious packets	No. of packets generated by an attacker	537
No. of parsed packets	No. of packets that were successfully parsed	5358
No. of firewall rules	No. of firewall rules packets were checked against	1
No. of dropped packets	No. of packets that were dropped by the firewall	805
Firewall rules	No. of packets dropped due to the firewall rule-set	0
SessionID queue	No. of packets dropped due to the sessionID internal queue	712
DPI	No. of packets dropped due to unsuccessful payload parsing	93

Similarly to the normal dataset, there are a number of duplicate or missing request/response/error messages that had been dropped by the SOME/IP firewall. There had been 469 packets logged and dropped due to no matching request being

found on the queue. 235 of those were response messages, and 234 were error messages. 243 messages were also dropped due to an identical request already being present in the queue, which means a duplicate message was sent without waiting for a response, or a replay attack is ongoing. While the SOME/IP Generator tool reported 537 malicious packets being successfully injected, only 79 packets were dropped by the SOME/IP firewall due to failed payload serialization.

This could mean that the success rate of the firewall's DPI is only 14%, however that does not account for all the packets already dropped by the queue, nor that the randomly generated value actually ended up being within acceptable bounds of the data-type it was generated for. Queue checks are done before the payload deserialization as the latter on average takes more time and requires more memory to deserialize and check the value of each argument of the method. The logic order was kept to keep the consistency of the time measurements as precise as possible, however if we remove the queue checks before the DPI process, the firewall flagged and dropped 426 packets. This equals to 79% of the reported malicious attacks, which is a significant improvement, it is difficult to approximate if this is an actual representation of the firewall's accuracy due to the naive implementation of the random malicious payloads generated. For a more accurate approximation, the malicious payloads should not be randomised, however that means the coverage size decreases for actual tested values.

The malformed payload dataset's measurements are closer in value to the normal dataset as compared to the error-on-error or error-on-event dataset. This is once again probably due to the smaller number of servers, clients and other environment configurations. It is however reasonable to say that if each smaller ECU in a vehicle has its own firewall, there would be only one server/service provider the firewall should care about, and the environment would resemble the current set-up more so than the one used by the error-on-error or error-on-event datasets, which resembles

a centralised firewall instead. While its median clock ticks is quite close to that of the normal dataset, with its value of 32 or $0.032ms$, the average value is double that, at 65.6821 clicks or $0.06568ms$. This is most probably due to the variable size of payloads that the `Deserializer` module must parse, and can average from one value to up to 4 different strings. This value can also grow based on the number of arguments a methods requires and the size of the payload.

6.2 Firewall Rule-set Analysis

For validating the firewall rule-set enforcement, the normal dataset was used, which produced some surprising results. The different rule-sets that have been tested can be found in Table 6.5. A different number of rules were tested to find an optimal number of rules after which the firewall parsing is slowed down considerably. This was done by running against the normal dataset using 1, 10, 25 and 50 different rules, which were all allowing traffic through based on the `clientID`, `serviceID` and `methodID` matching.

Table 6.5: Benchmark results ran against the malformed payload dataset.

No. of rules	Avg. time in ms	Med. time in ms
1	0.153935	0.149
10	0.129985	0.153
25	0.096765	0.050
50	0.063899	0.035

As one may notice from Table 6.5, the more rules there are, the quicker the firewall seems to operate. That is a flawed explanation of what is actually happening, but it can help point to a reasonable conclusion. The firewall's largest slowdown and bottleneck is introduced by logging. As more and more packets are accepted and not dropped, the number of logging entries the firewall must process decreases. As such, the I/O slowdown of writing to a file is shrinking with the number of forwarded packets. Additionally, since the firewall rule-set checks are the first step

of the firewall's inspection of the packet, the other more time consuming tasks are skipped should a packet not have to be inspected. However it is important to note that all rules introduced for this measurement have been "positive" rules that accept the packet for a specific `clientID`, `sessionID` and `methodID`.

An additional note to point out here is that for every rule-set used in this testing, the last rule was always a deny-all rule, which matched with every `clientID`, `serviceID` or `methodID`, set. This is due to best-practices of firewall rules, but this final rule could provide some type of speed-up, as the firewall is designed to by default drop any packet that does not match any of the rules in the rule-set. As such, the rule-set does not require a deny-by-default rule.

6.3 Memory Profiling Results

The memory profiling was done with the firewall benchmark binary running against the normal dataset. The `valgrind` tool was used to profile the heap and stack memory usage during its runtime separately. The `valgrind` tool outputs a file generally called `massif.out.<PID of process>`, and can be printed on the console using the `ms_print` tool. The size of the normal dataset which was loaded at the start of the benchmark is $423KB$.

6.3.1 Heap Profiling

The graph of heap memory usage provided by the `ms_print` command can be found in Figure 6.9. The heap profiling was measured by running the following command:

The peak visible on Figure 6.9, which is denoted with a line of "#" characters, reaches $335,768B$. This may not be the actual heap usage at the time, however valgrind limits the accuracy to 1% offset. Approximately 30% of the heap allocation

Listing 8 Valgrind command for heap profiling.

```
SOMEIP_FIREWALL_IN_FILE=test/normal_dataset.pcap \
                                          valgrind \
                                          --tool=massif \
                                          --time-unit=B \
                                          ./build/firewall-benchmark
```

at this time was for pushing data to a vector, which means the internal queue uses up a rough third of the memory the firewall requires. If the number of clients is large, this could become an issue for smaller embedded devices with limited memory, which means the number of clients and traffic frequency can impact the performance of the firewall significantly.

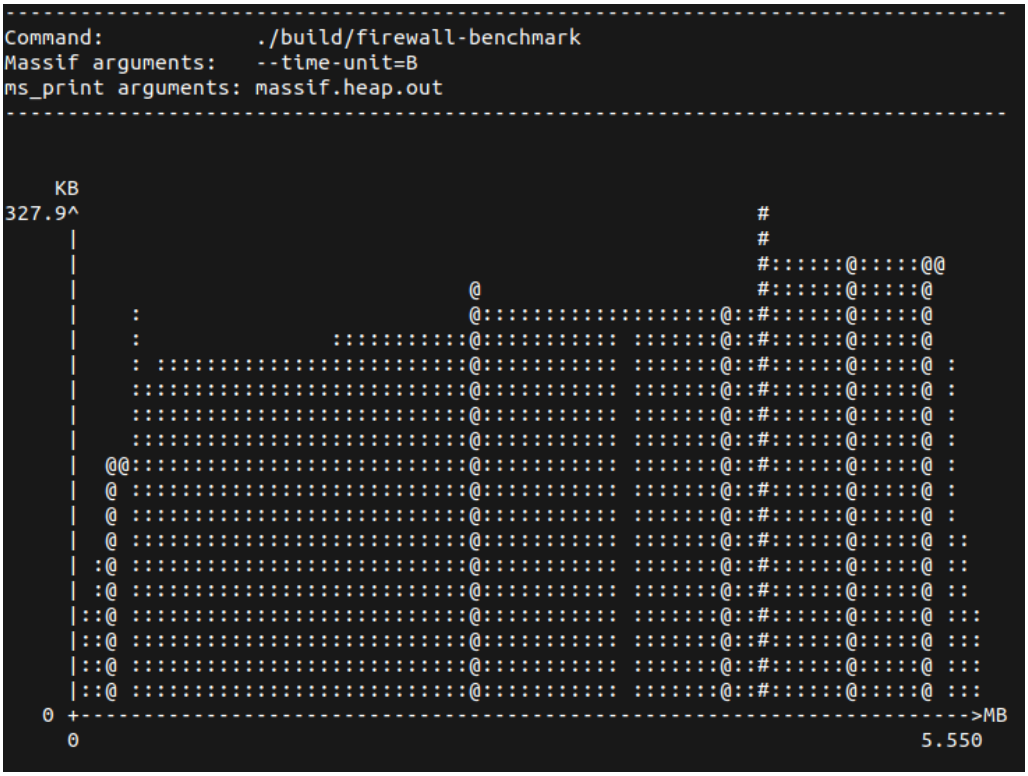

Figure 6.9: Massif output graph with heap profiling enabled.

6.3.2 Stack Profiling

Similarly to the heap profiling done by `massif`, the stack profiling can be measured by using the following command:

Listing 9 Valgrind command for stack profiling.

```
SOMEIP_FIREWALL_IN_FILE=test/normal_dataset.pcap \
                                        valgrind \
                                        --tool=massif \
                                        --stacks=yes  \
                                        --heap=no     \
                                        --time-unit=B \
                                        ./build/firewall-benchmark
```

The graph of heap memory usage provided by the `ms_print` command can be found in Figure 6.9. Unfortunately the stacks-only graph `massif` output does not provide any detailed snapshots to further investigate where the main usage of the stack originates from, however we can see that the maximum amount of stack usage for the SOME/IP firewall benchmark amounts to $9,568B$, which is a reasonably small stack usage for embedded systems. This further solidifies the possibility of implementing such a firewall for smaller embedded devices and not overload the system running on it.

6.4 Results Discussion

There had been a few interesting and surprising results achieved during the making of this thesis. After a closer analysis of each component of the SOME/IP firewall PoC implementation, it is safe to say that even the current unoptimised solution can run on an embedded device. Even for smaller units with very limited processing power, the only drawback would be the heap usage of the program. While it is difficult to precisely access the accuracy of the firewall solution proposed in this work due to the inconsistencies in the datasets used, as well as the randomness of

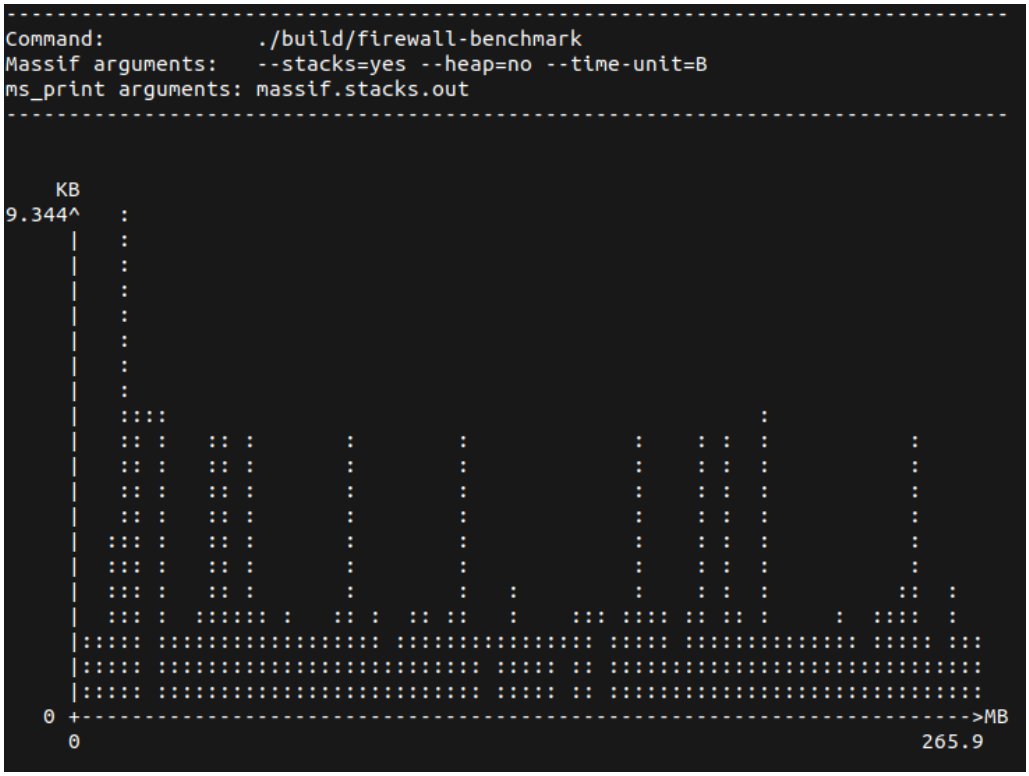

Figure 6.10: Massif output graph with stack profiling enabled.

the malformed payloads generated by the SOME/IP Generator tooling, there is clear success in affirming the AUTOSAR SOME/IP data transmission specifications as compared to some other implementations such as [85], which did not mark duplicate requests or responses.

It is reasonable to argument that any attacks which do not follow the AUTOSAR specification of the SOME/IP protocol, such as error-on-error or error-on-event attacks do not pose a threat to SOME/IP applications should the firewall be properly implemented, due to its internal queue keeping track of outgoing messages. Attacks such as a missing request or response however cannot be stopped by the firewall implementation, and should be taken into consideration when developing the queue, as it could be abused by an attacker to block and drop a valid message from a client, and instead send their own malicious message, which would pass the firewall due to the valid one being dropped. However, this is again a type of MITM attack

which cannot easily be defended against by the SOME/IP protocol due to its lack of authentication mechanisms.

Another interesting observation was that the number of rules does not impact the actual matching against packet header values noticeably in the range of 50 rules. This however raises another interesting question of well-formed and written rule-sets and how to keep them maintainable. The number of rules only grows with more clients and service providers available, which means that a large number of said rules introduces the issue of maintenance of said rules, as well as loading all of the rules into memory. It also showcases the need of implementing the firewall solution as a multi-threaded solution which transfers the task of logging important information in a different thread and does not block the parsing of further packets.

Depending on the importance and time-sensitivity of data being transferred, any delays longer than a specific value can cause great damage. Depending on the transmission speed, transmission deadlines can span from $33ms$ to $10ms$ [91]. Based on [92], control delay deadlines for the CAN protocol span from $50ms$ to $5ms$, and critical-control delay deadlines cannot be higher than $0.1ms$. Since SOME/IP also transfers control messages such as CAN, we can approximate the same delay deadlines for it, meaning the data cannot be delayed for more than $50ms$. This means that the delay introduced by parsing SOME/IP packets falls within the acceptable delay values, and in best-case scenarios with limited rules, clients and servers, can even be used for critical-control data. In the worst case scenario where a large number of clients and servers needs to be configured, the firewall can be used for control and normal messages without worrying about introducing large delay to the messages.

6.5 Limitations

As mentioned previously in Section 6.1, there had been a certain amount of limitations introduced by the datasets generated with the SOME/IP Generator tool. This resulted in a number of packets being dropped which were not labelled as malicious by [85] or where there was no attacker present. This was due to a missing time-delay configuration both in the SOME/IP Generator tool as well as the PoC implementation of the firewall. The time-delay configuration could be used for a client to send an identical message only if a specific time period has passed, and the firewall could keep track and drop identical messages that did not respect the delay configuration.

While this impacted the precision of the testing and benchmark measurements for the firewall implementation, it showcases that the internal queue enforces messages to follow the AUTOSAR specification, which requires incremental sessionID values for each following message sent from the same client. There had also been no known firewall implementation focused on a SOME/IP implementation, which limited the amount of comparison results able to be produced and tested against. While the implementation was compared against a previous IDS solution as best it could, there is a noticeable difference in the tasks done by a firewall and an IDS solution as well as which types of attacks can be prevented. As such, should another SOME/IP firewall solution be presented in the future, it would be interesting to compare the results of both implementations on a shared dataset.

6.6 Future Implementation Forecast

As mentioned a few times throughout this thesis, there are numerous improvements that can be taken into consideration for a future implementation of the proposed SOME/IP firewall.

SessionID tracking can be used to ensure that the packets sent from clients cannot randomly decrement or repeat the `sessionID` values in SOME/IP headers. Should a value be received that is not an increment of the last received message, the message may be dropped.

IP address could be introduced as an additional means of authentication or check to be done for firewall rules, which could add an additional layer of message security. It could also be analysed and tested to see if it is possible to integrate the SOME/IP firewall rule-set with an already existing and kernel-supported firewall tool such as e.g. `iptables` for additional security.

Multi-threaded approach can be used to ensure that logging packet content and reasons for dropping messages does not impact the processing time of parsing a packet. Additionally, the steps of checking the header values against the available rule-set, managing the internal queue status and performing DPI can all be performed parallelly, further minimising the time required for parsing a packet.

Complex data-type support could be achieved as an additional build-on of the current DPI performed by the firewall. While this could not be implemented in the scope of the thesis due to time constraints, it can provide a beneficial upgrade that could be tested in actual production environment.

Logging could be more detailed and optimised for lessening the strain on the firewall, which could ensure analysts could easily and quickly discover traces of attacks or could help with debugging.

FIDL and FDEPL parsing could be further improved to allow for a larger number of FIDL types and information to be used within the scope of the firewall. Currently, only basic data-types can be parsed, and more complex structures such

as e.g. TypeCollection's are ignored. This could be relatively easily implemented by using the `<jni.h>` library.

Real-life testing should be performed on actual data-sets from the field that could show points for further improvements of the thesis or its current lacking points. It would also prove as a good way to analyse and calculate the approximate precision of the firewall implementation.

7 Conclusion

In this thesis, an implementation, benchmarking and analysis of a SOME/IP firewall was introduced, as well as its applicability for embedded systems running in the automotive world. To simulate an embedded system, a RaspberryPi 4 model B was used with ARM64 architecture, and the benchmark and results analysis was conducted on binaries compiled and run on the RaspberryPi. The SOME/IP firewall was logically split into four different modules, each responsible for a dedicated task, i.e. rule generation, FIDL and FDEPL file parsing, DPI and packet handling.

During the testing phase of the SOME/IP firewall implementation, a number of known attack patterns were tested against the firewall to evaluate Research Question (RQ) 1. The error-on-error and error-on-event attacks were all successfully prevented due to the internal queue which stored the relevant `REQUEST` and `RESPONSE` message types. Malformed payloads were also recognised and dropped if the payload data could not be parsed successfully based on the available stored models and the AUTOSAR specification of how the payload should be structured, however this was not as precise as that of known logic-flow attacks such as error-on-error. It should also be mentioned that the firewall implementation cannot defend against MitM attacks that intercept a request or response and drop it.

While basic security vulnerabilities of the SOME/IP protocol cannot be directly addressed as speculated in RQ 2, as a lack of authentication and encryption cannot be solved with a firewall. What can be improved however is to mitigate the risk of

relying on a poorly-implemented client or server application which may not handle malicious data or logic-flow of messages properly and could introduce a critical crash of a system within a vehicle. The SOME/IP firewall can enforce and respect only valid AUTOSAR-specific data transmission and drop anything that is non-conforming.

The testing confirmed the SOME/IP firewall implementation to be reasonably fast with a normal and malformed payload dataset average's being in the span of $0.032395ms$ to $0.06568ms$. The error-on-error and error-on-event datasets had a slightly higher average run time of $0.17312ms$ and $0.163209ms$ respectively, which is still well within bounds of acceptable delay for control messages. As such, based on RQ 3, the implementation is effective for embedded systems with a limited amount of processing power and storage, and an additional layer of DPI does not introduce a delay that would make control-level messages too slow.

Finding an optimal number of firewall rules led to the realisation that the current implementation is bottlenecked most by I/O operations of writing log messaging to a file, and that within the range of 50 rules, there is no noticeable slowdown introduced by a higher number of rules present in the rule-set. Additionally, a section was dedicated to discussing further hardening of the firewall implementation, which could be a good starting point for testing the accuracy and strain on a real-life dataset or environment. Nonetheless, a point must be made that this implementation is a PoC with a limited scope such as only supporting basic data-types and string parsing, and there is much room for further improvement. As the number of data-types support increases and real-life FIDL configurations are used, there will be an understandably increased time delay introduced by each packet parsing. Hence it is monumental to test the implementation in an actual representation of a production environment or a much closer simulation that presented in this thesis to accurately assess what kind of delay will be introduced as a production-grade solution.

While the automotive world slowly shifts from the CAN protocol to newer, more secure and flexible data transmission protocols, there is an avid need for additional security measures that can support those protocols. As it is plausible for SOME/IP to relay control messages, it is a good alternative for the CAN protocol and had been under consideration in this thesis. The solution provided in this thesis introduces the idea of a firewall specifically for the SOME/IP protocol, which has not been researched before to the author's current knowledge, and is proven considerably fast even with introducing something of a DPI for each packet. This solution could thus be used in embedded systems for automotive use-cases where a short delay is mandatory, but a desire for DPI and data transmission control is required. Firewalls also generally introduce a much lower chance of false-positives if properly configured, which is desired for such systems. An interesting future study could validate and analyse the resource and time constraints introduced by a fully-functional SOME/IP firewall with support for DPI of any data-type.

References

[1] A. Anwar, A. Anwar, L. Moukahal, and M. Zulkernine, "Security assessment of in-vehicle communication protocols", *Vehicular Communications*, vol. 44, p. 100 639, 2023.

[2] R. N. Charette, *This Car Runs on Code*, Feb. 2009. [Online]. Available: `https://spectrum.ieee.org/this-car-runs-on-code`.

[3] Vaibhav, *Electronic control unit is at the core of all automotive innovations: Know how the story unfolded*, Jul. 2017. [Online]. Available: `https://www.embitel.com/blog/embedded-blog/automotive-control-units-development-innovations-mechanical-to-electronics`.

[4] K. Koscher, A. Czeskis, F. Roesner, *et al.*, "Experimental security analysis of a modern automobile", in *2010 IEEE symposium on security and privacy*, IEEE, 2010, pp. 447–462.

[5] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz, "From a federated to an integrated automotive architecture", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 956–965, 2009.

[6] P. Sivakumar, R. S. Devi, A. N. Lakshmi, B. VinothKumar, and B. Vinod, "Automotive grade linux software architecture for automotive infotainment system", in *2020 International Conference on Inventive Computation Technologies (ICICT)*, IEEE, Coimbatore, India, 2020, pp. 391–395.

[7] S. Fürst, J. Mössinger, S. Bunzel, *et al.*, "Autosar–a worldwide standard is on the road", in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, Citeseer, vol. 62, 2009, p. 5.

[8] COVESA, *COVESA and AUTOSAR Align on Leading Automotive Standards for the Future*, Jan. 2023. [Online]. Available: `https://www.covesa.global/sites/default/files/COVESA-AUTOSAR-Alignment-202301.pdf`.

[9] M. Traub, A. Maier, and K. L. Barbehön, "Future automotive architecture and the impact of it trends", *IEEE Software*, vol. 34, no. 3, pp. 27–32, 2017.

[10] J.-W. Yoo, Y. Lee, D. Kim, and K. Park, "An android-based automotive middleware architecture for plug-and-play of applications", in *2012 IEEE Conference on Open Systems*, IEEE, Daejeon, Korea, 2012, pp. 1–6.

[11] S. Osswald and M. Lienkamp, "An automotive hmi architecture based on a mobile operating system", in *Adjunct Proceedings of the 6th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, ser. AutomotiveUI '14, Seattle, WA, USA: Association for Computing Machinery, 2014, pp. 1–6, ISBN: 9781450307253.

[12] A. Bucaioni and P. Pelliccione, "Technical architectures for automotive systems", in *2020 IEEE International Conference on Software Architecture (ICSA)*, IEEE, Salvador, Brazil, 2020, pp. 46–57.

[13] Volvo, *All-new XC90 the first Volvo built on the company's new Scalable Product Architecture*, Aug. 2014. [Online]. Available: `https://www.media.volvocars.com/global/en-gb/media/pressreleases/148966/all-new-xc90-the-first-volvo-built-on-the-companys-new-scalable-product-architecture`.

[14] Volkswagen, *Modular electric drive matrix (MEB).* [Online]. Available: https://www.volkswagen-newsroom.com/en/modular-electric-drive-matrix-meb-3677.

[15] A. Sergeev, *Ford-VW Partnership Expands, Blue Oval Getting MEB Platform For EVs*, Jul. 2019. [Online]. Available: https://www.motor1.com/news/359495/ford-vw-tech-alliance-expands/.

[16] M. Held, N. Rosat, G. Georges, H. Pengg, and K. Boulouchos, "Lifespans of passenger cars in europe: Empirical modelling of fleet turnover dynamics", *European Transport Research Review*, vol. 13, pp. 1–13, 2021.

[17] BOSCH, *Vehicle-centralized, zone-oriented E/E architecture with vehicle computers.* [Online]. Available: https://www.bosch-mobility.com/en/mobility-topics/ee-architecture/.

[18] J. Klaus-Wagenbrenner, "Zonal EE architecture: Towards a fully automotive Ethernet–based vehicle infrastructure", in *Proc. Automotive E/E Architecture Technology Innovation Conference*, Shanghai, China, 2019.

[19] I. Rouf, R. Miller, H. Mustafa, *et al.*, "Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study", in *19th USENIX Security Symposium (USENIX Security 10)*, Santa Clara, CA, USA, 2010.

[20] S. Tuohy, M. Glavin, C. Hughes, E. Jones, M. Trivedi, and L. Kilmartin, "Intra-vehicle networks: A review", *IEEE transactions on intelligent transportation systems*, vol. 16, no. 2, pp. 534–545, 2014.

[21] S. Kumar, S. Dalal, and V. Dixit, "The OSI model: Overview on the seven layers of computer networks", *International Journal of Computer Science and Information Technology Research*, vol. 2, no. 3, pp. 461–466, 2014.

[22] G. Leen, D. Heffernan, and A. Dunne, "Digital networks in the automotive vehicle", *Computing and control engineering journal*, vol. 10, no. 6, pp. 257–66, 1999.

[23] C. P. Szydlowski, "Can specification 2.0: Protocol and implementations", SAE Technical Paper, Tech. Rep., 1992.

[24] R. Makowitz and C. Temple, "Flexray-a communication network for automotive control systems", in *2006 IEEE International Workshop on Factory Communication Systems*, IEEE, 2006, pp. 207–212.

[25] P. Hank, S. Müller, O. Vermesan, and J. Van Den Keybus, "Automotive Ethernet: In-vehicle networking and smart mobility", in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, Grenoble, France, 2013, pp. 1735–1739.

[26] I. A. Grzemba, *MOST: the automotive multimedia network*. Franzis Verlag, 2012.

[27] L. Fang, "Handbook of networked and embedded control systems [book review; dh hristu-varsakelis and ws levine]", *IEEE Transactions on Automatic Control*, vol. 52, no. 1, pp. 145–148, 2007.

[28] E. Aliwa, O. Rana, C. Perera, and P. Burnap, "Cyberattacks and countermeasures for in-vehicle networks", *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–37, 2021.

[29] F. Hartwich *et al.*, "Can with flexible data-rate", in *Proc. iCC*, Citeseer, 2012, pp. 1–9.

[30] C. Szilagyi and P. Koopman, "Flexible multicast authentication for time-triggered embedded control network applications", in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, IEEE, Lisbon, Portugal, 2009, pp. 165–174.

[31] C. Szilagyi and P. Koopman, "Low cost multicast authentication via validity voting in time-triggered embedded control networks", in *Proceedings of the 5th Workshop on Embedded Systems Security*, 2010, pp. 1–10.

[32] P. Mundhenk, S. Steinhorst, M. Lukasiewycz, S. A. Fahmy, and S. Chakraborty, "Lightweight authentication for secure automotive networks", in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, Giza, Egypt, 2015, pp. 285–288.

[33] N. Nowdehi, A. Lautenbach, and T. Olovsson, "In-vehicle can message authentication: An evaluation based on industrial criteria", in *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)*, IEEE, Toronto, ON, Canada, 2017, pp. 1–7.

[34] M. Agrawal, T. Huang, J. Zhou, and D. Chang, "Can-fd-sec: Improving security of can-fd protocol", in *Security and Safety Interplay of Intelligent Software Systems: ESORICS 2018 International Workshops, ISSA 2018 and CSITS 2018, Barcelona, Spain, September 6–7, 2018, Revised Selected Papers*, Springer, 2019, pp. 77–93.

[35] E. O. Marasco and F. Quaglia, "AuthentiCAN: a Protocol for Improved Security over CAN", in *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*, IEEE, London, UK, 2020, pp. 533–538.

[36] B. Groza, S. Murvay, A. Van Herrewege, and I. Verbauwhede, "Libra-can: A lightweight broadcast authentication protocol for controller area networks", in *Cryptology and Network Security: 11th International Conference, CANS 2012, Darmstadt, Germany, December 12-14, 2012. Proceedings 11*, Springer, 2012, pp. 185–200.

[37] M. Ruff, "Evolution of local interconnect network (LIN) solutions", in *2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall (IEEE Cat. No. 03CH37484)*, IEEE, vol. 5, Orlando, FL, USA, 2003, pp. 3382–3389.

[38] A. R. Mousa, P. NourElDeen, M. Azer, and M. Allam, "Lightweight authentication protocol deployment over FlexRay", in *INFOS '16: Proceedings of the 10th International Conference on Informatics and Systems*, Giza, Egypt, 2016, pp. 233–239.

[39] Z. Gu, G. Han, H. Zeng, and Q. Zhao, "Security-aware mapping and scheduling with hardware co-processors for flexray-based distributed embedded systems", *IEEE Transactions on parallel and distributed systems*, vol. 27, no. 10, pp. 3044–3057, 2016.

[40] Ixia, "Automotive Ethernet: an overview", Ixia, White Paper, 2014. [Online]. Available: `https://support.ixiacom.com/sites/default/files/resources/whitepaper/ixia-automotive-ethernet-primer-whitepaper_1.pdf`.

[41] M. Ashjaei, L. L. Bello, M. Daneshtalab, G. Patti, S. Saponara, and S. Mubeen, "Time-Sensitive Networking in automotive embedded systems: State of the art and research opportunities", *Journal of Systems Architecture*, vol. 117, p. 102 137, 2021, ISSN: 1383-7621.

[42] M. Wolf, A. Weimerskirch, and C. Paar, "Security in automotive bus systems", in *Workshop on Embedded Security in Cars*, Bochum, 2004, pp. 1–13.

[43] J. Deng, L. Yu, Y. Fu, O. Hambolu, and R. R. Brooks, "Chapter 6 - Security and Data Privacy of Modern Automobiles", in *Data Analytics for Intelligent Transportation Systems*, M. Chowdhury, A. Apon, and K. Dey, Eds., Elsevier, 2017, pp. 131–163, ISBN: 978-0-12-809715-1.

[44] Q. Hu and F. Luo, "Review of secure communication approaches for in-vehicle network", *International Journal of Automotive Technology*, vol. 19, pp. 879–894, 2018.

[45] U. S. Ltd., "2023 global automotive cybersecurity report", Tech. Rep., 2023.

[46] T. Hoppe, S. Kiltz, and J. Dittmann, "Security threats to automotive CAN networks–practical examples and selected short-term countermeasures", in *Computer Safety, Reliability, and Security: 27th International Conference, SAFECOMP 2008 Newcastle upon Tyne, UK, September 22-25, 2008 Proceedings 27*, Springer, 2008, pp. 235–248.

[47] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle", *Black Hat USA*, vol. 2015, no. S 91, pp. 1–91, 2015.

[48] D. Lodge, *Hacking the Mitsubishi Outlander PHEV hybrid*, 2016. [Online]. Available: `https://www.pentestpartners.com/security-blog/hacking-the-mitsubishi-outlander-phev-hybrid-suv/`.

[49] R. Hull, *Nissan disables leaf electric car app after revelation that hackers can switch on the heater to drain the battery*, 2016. [Online]. Available: `https://www.thisismoney.co.uk/money/cars/article-3465459/Nissan-disables-Leaf-electric-car-app-hacker-revelation.html`.

[50] E. Kovacs, *Thieves Use CAN Injection Hack to Steal Cars*, 2023. [Online]. Available: `https://www.securityweek.com/thieves-use-can-injection-hack-to-steal-cars/`.

[51] Z. El-Rewini, K. Sadatsharan, D. F. Selvaraj, S. J. Plathottam, and P. Ranganathan, "Cybersecurity challenges in vehicular communications", *Vehicular Communications*, vol. 23, p. 100 214, 2020.

[52] R. S. Rathore, C. Hewage, O. Kaiwartya, and J. Lloret, "In-vehicle communication cyber security: Challenges and solutions", *Sensors*, vol. 22, no. 17, p. 6679, 2022.

[53] T. Hoppe, S. Kiltz, and J. Dittmann, "Applying intrusion detection to automotive it-early insights and remaining challenges", *Journal of Information Assurance and Security (JIAS)*, vol. 4, no. 6, pp. 226–235, 2009.

[54] N. J. Ojo and O. U. Solomon, "A Review of Technical Issues on IDS and Alerts", *Global Journal of Computer Science and Technology: E Network, Web and Security*, vol. 17, no. 5, 2017.

[55] Y. Xie, Y. Zhou, J. Xu, J. Zhou, X. Chen, and F. Xiao, "Cybersecurity protection on in-vehicle networks for distributed automotive cyber-physical systems: State-of-the-art and future challenges", *Software: Practice and Experience*, vol. 51, no. 11, pp. 2108–2127, 2021.

[56] A. Hafeez, K. Rehman, and H. Malik, "State of the art survey on comparison of physical fingerprinting-based intrusion detection techniques for in-vehicle security", SAE Technical Paper, Tech. Rep., 2020.

[57] S. Jadhav and D. Kshirsagar, "A Survey on Security in Automotive Networks", in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, IEEE, Pune, India, 2018, pp. 1–6.

[58] N. Mohammad, S. Muhammad, and E. Shaikh, "Analysis of in-vehicle security system of smart vehicles", in *Future Network Systems and Security: 5th International Conference, FNSS 2019, Melbourne, VIC, Australia, November 27–29, 2019, Proceedings 5*, Springer, 2019, pp. 198–211.

[59] T.-Y. Lee, I.-A. Lin, and R.-H. Liao, "Design of a FlexRay/Ethernet gateway and security mechanism for in-vehicle networks", *Sensors*, vol. 20, no. 3, p. 641, 2020.

[60] J. Wei, K. Ma, and C. Kong, "Research on Intelligent Detection Method of Automotive Network Data Security Based on FlexRay/CAN Gateway", in *International Conference on Machine Learning for Cyber Security*, Springer, Guangzhou, China, 2022, pp. 394–408.

[61] S. Hu, Q. Zhang, A. Weimerskirch, and Z. M. Mao, "Gatekeeper: A gateway-based broadcast authentication protocol for the in-vehicle Ethernet", in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, Nagasaki, Japan, 2022, pp. 494–507.

[62] N. Khatri, R. Shrestha, and S. Y. Nam, "Security issues with in-vehicle networks, and enhanced countermeasures based on blockchain", *Electronics*, vol. 10, no. 8, p. 893, 2021.

[63] Moriyama, Daisuke, "HW/SW Security Mechanisms for Future Automotive Society", Renesas, White Paper, 2023. [Online]. Available: `https://www.renesas.com/us/en/document/whp/hwsw-security-mechanisms-future-automotive-society`.

[64] Chung, Soonin, *Part 32. [Mobility Inside] Car as a Wallet with Post-Quantum Cryptography Technology*, 2023. [Online]. Available: `https://www.lg.com/global/mobility/more-stories/32-car-as-a-wallet-with-post-quantum-cryptography-technology`.

[65] K. J. Higginga, *Who Invented the Firewall?*, Jan. 2008. [Online]. Available: `https://www.darkreading.com/cybersecurity-analytics/who-invented-the-firewall-`.

[66] W. R. Cheswick, *Firewalls And Internet Security: Repelling The Wily Hacker, 2/E*. Pearson Education India, 2003.

[67] NXP Semiconductor, *Automotive gateway: A key component to securing the connected car*, 2018. [Online]. Available: `https://www.nxp.com/docs/en/white-paper/AUTOGWDEVWPUS.pdf`.

[68] F. Luo and Q. Hu, "Security mechanisms design for in-vehicle network gateway", SAE Technical Paper, Tech. Rep., 2018.

[69] F. Luo and S. Hou, "Security mechanisms design of automotive gateway firewall", SAE Technical Paper, Tech. Rep., 2019.

[70] S. Rizvi, J. Willett, D. Perino, T. Vasbinder, and S. Marasco, "Protecting an automobile network using distributed firewall system", in *ICC '17: Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing*, Cambridge, United Kingdom, 2017, pp. 1–6.

[71] AUTOSAR, "Specification of secure onboard communication", *AUTOSAR CP Release*, 2020.

[72] E. Rescorla and N. Modadugu, "Datagram transport layer security version 1.2", Tech. Rep., 2012.

[73] S. Frankel and S. Krishnan, "Ip security (ipsec) and internet key exchange (ike) document roadmap", Tech. Rep., 2011.

[74] Völker, Lars, *Scalable service-Oriented MiddlewarE over IP (SOME/IP)*, 2024. [Online]. Available: `https://some-ip.com/details.shtml`.

[75] AUTOSAR, "SOME/IP Protocol Specification", *AUTOSAR CP Release*, 2019.

[76] AUTOSAR, "SOME/IP Service Discovery Protocol Specification", *AUTOSAR*, 2022.

[77] M. Vujanić, N. Trifunović, I. Kaštelan, and B. Kovačević, "Bitroute SOME/IP: Implementation of a Scalable and Service Oriented Communication Middleware", in *2022 45th Jubilee International Convention on Information, Commu-*

nication and Electronic Technology (MIPRO), IEEE, Opatija, Croatia, 2022, pp. 1426–1429.

[78] J. F. B. de Almeida, "Rust-based SOME/IP implementation for robust automotive software", Ph.D. dissertation, Universidade do Porto (Portugal), 2021.

[79] Chung, Hanjun, *How to fuzz SOME/IP protocol - How did I get my name on the BMW Hall of Fame (Part 1)*, 2023. [Online]. Available: `https://onepwnman.com/SOMEIP-Fuzzing/`.

[80] Y. Li, H. Chen, C. Zhang, S. Xiong, C. Liu, and Y. Wang, "Ori: A greybox fuzzer for SOME/IP protocols in automotive Ethernet", in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, Singapore, Singapore, 2020, pp. 495–499.

[81] S. Lee, W. Choi, and D. H. Lee, "Protecting SOME/IP Communication via Authentication Ticket", *Sensors*, vol. 23, no. 14, p. 6293, 2023.

[82] Ran Ben Tzvi, *Hijacking SOME/IP Protocol with Man in the Middle Attack*, 2024. [Online]. Available: `https://argus-sec.com/blog/cyber-security-blog/some-ip-protocol-man-in-the-middle-attack/`.

[83] D. Zelle, T. Lauser, D. Kern, and C. Krauß, "Analyzing and securing SOME/IP automotive services with formal and practical methods", in *ARES '21: Proceedings of the 16th International Conference on Availability, Reliability and Security*, Vienna, Austria, 2021, pp. 1–20.

[84] M. Iorio, A. Buttiglieri, M. Reineri, F. Risso, R. Sisto, and F. Valenza, "Protecting in-vehicle services: Security-enabled SOME/IP middleware", *IEEE Vehicular Technology Magazine*, vol. 15, no. 3, pp. 77–85, 2020.

[85] N. Alkhatib, H. Ghauch, and J.-L. Danger, "SOME/IP intrusion detection using deep learning-based sequential models in automotive ethernet networks", in *2021 IEEE 12th Annual Information Technology, Electronics and Mobile*

Communication Conference (IEMCON), IEEE, Vancouver, BC, Canada, 2021, pp. 0954–0962.

[86] itemis AG, *Welcome to Franca*, 2018. [Online]. Available: `https://github.com/franca/franca`.

[87] Bayerische Motoren Werke Aktiengesellschaft (BMW AG), *CommonAPI C++*, 2024. [Online]. Available: `https://covesa.github.io/capicxx-core-tools/`.

[88] Vaughan-Nichols, Steven, *It's a Linux-powered car world*, 2019. [Online]. Available: `https://www.zdnet.com/article/its-a-linux-powered-car-world/`.

[89] S. Smalley, C. Vance, and W. Salamon, "Implementing SELinux as a Linux security module", *NAI Labs Report*, vol. 1, no. 43, p. 139, 2001.

[90] N. Alkhatib, M. Mushtaq, H. Ghauch, and J.-L. Danger, "Here comes SAID: A SOME/IP Attention-based mechanism for Intrusion Detection", in *2023 Fourteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, IEEE, Paris, France, 2023, pp. 462–467.

[91] G. Vasta and L. L. Bello, "An innovative traffic management scheme for deterministic/event-based communications in automotive applications with a focus on Automated Driving Applications", *IEEE Standards Association Ethernet & IP@ Automotive Technology Day, London, UK*, 2018.

[92] E. Choi, H. Song, S. Kang, and J.-W. Choi, "High-speed, low-latency in-vehicle network based on the bus topology for autonomous vehicles: Automotive networking and applications", *IEEE Vehicular Technology Magazine*, vol. 17, no. 1, pp. 74–84, 2021.

Appendix A Generated Code

The code designed, implemented and tested can be found at `https://gitlab.utu.fi/ehzorm/thesis-implementation`. The final commit within the scope of this thesis was `9854c393528c4df12b27d41f9ed92cc1ae8f3288`. The error-on-error and error-on-event datasets were generated from the files erroronerror2.csv[1] and erroronevent2.csv[2] respectfully. The code used to generate a PCAP file from the CSV files can be found below.

[1] `https://github.com/Alkhatibnatasha/SOMEIP_IDS/blob/main/Data/Test/erroronerror2.csv`

[2] `https://github.com/Alkhatibnatasha/SOMEIP_IDS/blob/main/Data/Test/erroroneventt2.csv`

Listing 10 Code snippet used to generate PCAP files from CSV files.

```
import pandas as pd
from scapy.all import *


if __name__ == "__main__":
    error_on_error_data = pd.read_csv("erroronerror2.csv")
    error_on_event_data = pd.read_csv("erroronevent2.csv")

    for index, row in error_on_error_data.iterrows():
        packet = (
            Ether(src=row[12], dst=row[13])
            / IP(src=row[7], dst=row[8])
            / UDP(dport=row[11], sport=row[10])
            / SomeIP(
                serviceID=row[14],
                methodID=row[15],
                dataLength=row[4],
                clientID=row[25],
                sessionID=row[2],
                messageType=row[1],
                returnCode=row[6],
                someIpPayload=createPayload(row[4] - 8),
            )
        )
        wrpcap("error_on_error_dataset.pcap", packet, append=True)

    for index, row in error_on_event_data.iterrows():
        packet = (
            Ether(src=row[12], dst=row[13])
            / IP(src=row[7], dst=row[8])
            / UDP(dport=row[11], sport=row[10])
            / SomeIP(
                serviceID=row[14],
                methodID=row[15],
                dataLength=row[4],
                clientID=row[25],
                sessionID=row[2],
                messageType=row[1],
                returnCode=row[6],
                someIpPayload=createPayload(row[4] - 8),
            )
        )
        wrpcap("error_on_event_dataset.pcap", packet, append=True)
```

Listing 11 SOME/IP packet class for scapy.

```
import pandas as pd
from scapy.all import *
# METAINFO
VERSION = 0x01
INTERFACE = 0x01


def createPayload(length):
    """Payloads are required for any semblance of testing"""
    values = "0123456789ABCDEF"
    data = "".join(
        [values[random.randint(0, len(values) - 1)] for _ in range(length)]
    ).encode("utf-8")
    return data


class SomeIP(Packet):
    """Generate a SOME/IP packet layer"""

    global VERSION
    global INTERFACE

    name = "SOMEIP"
    fields_desc = [
        XShortField("serviceID", None),
        XShortField("methodID", None),
        FieldLenField(
            "dataLength",
            None,
            length_of="someIpPayload",
            adjust=lambda pkt, x: x + 8,
            fmt="I",
        ),
        XShortField("clientID", None),
        XShortField("sessionID", None),
        XByteField("protocolVersion", VERSION),
        XByteField("interfaceVersion", INTERFACE),
        XByteField("messageType", None),
        XByteField("returnCode", None),
        StrLenField(
            "someIpPayload", "", length_from=lambda pkt: pkt.dataLength
            )
    ]
```