Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Host Firewall on AUTOSAR Adaptive based Vehicle Computers & Domain ECUs

Eric Schneider

Course of Study:	Informatik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Tim Würtele, M.Sc.
	Oliver Bubeck, Dipl.-Ing. (FH) (Robert Bosch GmbH)
Commenced:	May 18, 2022
Completed:	November 17, 2022

Abstract

Setting up firewalls without additional tooling can be inefficient and complicated. In this paper a prototype will be presented that allows the configuration of an ECU host firewall based on a well defined configuration file. This firewall is designed to run on vehicle computers and smart components inside cars that run the AUTOSAR Adaptive platform. The goal is to simplify firewall setup to secure these components against malicious traffic in the network and to prevent attack vectors that try to exploit physical access to the system.

The presented prototype will be using nftables and the netfilter subsystem to set up both stateless and stateful filtering rules for both incoming and forwarded traffic. Packet inspection will also be evaluated in this context and approaches to filtering of the high level SOME/IP protocol will be presented.

Example rulesets for both regular ECUs that are running the AUTOSAR Adaptive platform as well as an example for network separation will be provided.

A short introduction to the AUTOSAR IAM concept will be given along with a comparison between it and the presented Firewall concept will be drawn.

Keywords: AUTOSAR Adaptive Platform, Firewall, IAM, WSL2

Acknowledgements

I want to take this chance to thank the people at the Robert Bosch GmbH that supported me during my thesis. Both, Oliver Bubeck and Arup Mukherji, were great help for technical details surrounding the AUTOSAR Adaptive platform. I want to thank Michael Schneider for his continued support with the Firewall concept, questions about the configuration file specification and general help as well.

Contents

1 Introduction **11**

2 The AUTOSAR Adaptive Platform **13**
2.1 Platform Characteristics 14
2.2 Architecture 16
2.3 SOME/IP 19

3 Firewalls **23**
3.1 Firewall Policies 24
3.2 Filtering types 27
3.3 Host Firewalls 30
3.4 netfilter 32

4 The proof-of-concept **37**
4.1 Specification 37
4.2 Implementation 44
4.3 Parsing rules 46
4.4 Ruleset example 50

5 IAM **57**
5.1 Proof-of-concept comparison 58

6 Related Work **61**

7 Conclusion and Outlook **63**

Bibliography **65**

List of Figures

2.1 Example system with multiple different platforms, including Adaptive and Classic AUTOSAR and non-AUTOSAR software [AUT21b]. 15
2.2 Adaptive Platform architecture showing existing and planned functional clusters [AUT21b]. 17
2.3 Standard SOME/IP header format including all required fields [AUT21e]. 19
2.4 Example padding of a message containing variables of different length. It contains five different members, first a UINT16 variable, then an array of variable size containing UINT8 elements. A UINT32 and UINT64 variable. And finally another array of UINT8 elements [AUT21e]. 20

3.1 The netfilter hook structure along with the packet flow for incoming and outgoing packets [net21d]. 34

4.1 The UML diagram shows the architecture of the firewall configuration file. The protocol and layer subrules of the FirewallRule each contain header fields of the related protocols. 38
4.2 Contents of datalink layer rules. 40
4.3 Contents of network layer rules. 41
4.4 Contents of transport layer rules. 42
4.5 Contents of the payload byte pattern rules. 43
4.6 Contents of the SOME/IP protocol rules. 43
4.7 Activity diagram of the POC that shows the general program flow. 44
4.8 Example ruleset for one IPv4 rule. 46
4.9 Abbreviated IPv4 header. 47
4.10 Standard SOME/IP header format including all required fields [AUT21e]. 48
4.11 Offset and length of the message ID field in the SOME/IP header. 49
4.12 An example payload byte pattern rule. 50
4.13 Example ruleset for an AUTOSAR Adaptive machine. 52
4.14 Resulting nftables configuration when the example ruleset is parsed by the program. 53
4.15 Each rectangle represents a single ECU. The ones in orange are domain gateways while the one in red is the central gateway that is also able to communicate with an outside server over the internet. 54
4.16 Example configuration for network separation. 55
4.17 Resulting nftables configuration for the network separation example ruleset. . . . 55

1 First part of the XML schema. Contains the top level elements and some rule types. 68
2 Second part of the XML schema. Contains most of the OSI layer rules and type definitions. 69
3 Third part of the XML schema. Contains the transport layer and SOME/IP rules and type definitions. 70

4 C script to send custom UDP packets. Adapted from Dalton’s example [Dal20]. . 71

Acronyms

AA Adaptive Application. 16

AP AUTOSAR Adaptive Platform. 13

ARA AUTOSAR Runtime for Adaptive Applications. 16

AUTOSAR AUTomotive Open System ARchitecture. 11

CLI command-line interface. 45

conntrack connection tracking. 36

CP AUTOSAR Classic Platform. 13

DPI Deep Packet Inspection. 29

E2E end-to-end encryption. 43

ECU Electronic Control Unit. 11

FC Functional Cluster. 16

IAM Identity and Access Management. 57

ICMP Internet Control Message Protocol. 24

IDPS Intrusion Detection and Prevention System. 30

IETF Internet Engineering Task Force. 29

IoT Internet of Things. 61

IPC Inter Process Communication. 17

IPv4 Internet protocol Version 4. 25

IPv6 Internet protocol Version 6. 25

NAC Network Access Control. 27

NAT Network Address Translation Protocol. 23

NIST National Institute of Standards and Technology. 23

POC proof-of-concept. 37

RPC Remote Procedure Call. 19

SOA Service-Oriented-Architecture. 14

SOME/IP Scalable Service-Oriented Middleware over IP. 19

SOME/IP-SD SOME/IP Service Discovery. 21

SOME/IP-TP SOME/IP Transport Protocol. 21

SSO single-sign on. 57

TCP Transmission Control Protocol. 21

TLS Transport Layer Security. 11

TP Transport Protocol. 20

UDP User Datagram Protocol. 20

1 Introduction

The ever-continuing computerization of the modern car has lead to the creation of complicated networks of Electronic Control Unit (ECU), sensors and other components. This has enabled the development of features that were unimaginable 20 years ago. Self-parking cars, cars that can update their software over the internet and assisted driving modes were all deemed for the distant future not too long ago. In the near future these systems will also allow the development of fully self-driving cars which are already being heavily tested. A functionality that needs the support of multiple sensors and ECUs all connected via high-bandwidth interfaces. However, the introduction of a heavily distributed and inter-connected network as well as the possibility of connecting it to the internet has created new problems that car manufacturers never had to deal with before.

The first problem was the different systems used by different manufacturers. There was no standardization for the individual components, and everything had to be created from scratch for every single manufacturer. ECUs back then had software embedded in them with no way to alter or update it. This works for simpler systems, however, as the complexity increases so does the need for patches and fixes down the line for bugs and errors that were not caught during initial testing. To solve this, AUTomotive Open System ARchitecture (AUTOSAR) was founded, a global partnership for the development of a standardized platform for automotive vehicles. They created both the AUTOSAR Classic and AUTOSAR Adaptive platforms [AUT22]. Both AUTOSAR platforms are meant to coexist and solve different problems, but they both provide a unified and standardized underlying architecture that allows updates to local ECUs over the internet and other means.

Another of these problems is securing communication between components while still assuring high throughput and bandwidth. Both qualities are immensely important in modern cars as even slight delays can have drastic consequences, such as airbags not deploying on time or the ESP kicking in too late. Additionally, the communication should be encrypted and authenticated such that third parties can not inject or read network messages. For this use-case the use of Transport Layer Security (TLS) to secure the communication between components running AUTOSAR Adaptive was proposed, which fulfilled the earlier named requirements in my last paper [Sch19]. TLS 1.2, which is currently widely used for accessing web pages over HTTPS, has been proven to be secure and relatively fast on embedded ECUs when the right encryption algorithms are used. However, since TLS 1.3 is even more secure and has better performance it will most likely establish itself as the standard TLS version in the near future [wol22].

Additionally, the packets that a component can receive also need to be monitored. Even if we do use TLS between components and when connecting to the internet this does not protect from malicious messages that are being sent from other sources or compromised components inside the network. Since cars are generally parked outside in accessible places for longer periods of time this creates another attack vector for physical tampering with the actual hardware. While ECUs are generally protected against physical access and intrusion this is not negligible as attacks that target hardware implementations exist. We propose the introduction of a Host Firewall to deal with this problem

and possible attacks over the internet. A host firewall protects the system it is attached to, therefore all ECUs inside the vehicle will be protected if they have their own designated host firewall, even if one or more ECUs are compromised. A firewall generally allows the policing of messages arriving at the protected destination. All packets that are sent to a system protected by a firewall will have to pass through a filter first. There, certain rules can be applied to filter out any potentially malicious packets. This is done by three main approaches:

- Stateless filtering
- Stateful filtering
- Deep packet inspection

Each of these approaches targets different layers of the OSI network model and can therefore be effectively used together. For a system where malicious packets most likely originate from outside sources an external firewall should be used. This kind of firewall is usually implemented on one machine, usually a router, and checks all incoming traffic that enters the network. This is not practical for our use case since components inside the network might get compromised, therefore an internal firewall, also called a host firewall, will be used instead. This kind of firewall is implemented on every ECU and can protect against all traffic inside the network, even if it originates from a local component.

In this paper we will present a proof of concept for a Host Firewall for AUTOSAR Adaptive based vehicle computers. Additionally, a working prototype for stateless and stateful filtering will be created and tested. In chapter 2 the AUTOSAR Adaptive Platform will be presented, along with some inner workings of the platform. The SOME/IP protocol will also be discussed briefly, as the firewall should be SOME/IP aware to some extent to be able to filter SOME/IP traffic. In depth information about firewalls and the different filter types will then be provided in chapter 3. In chapter 4 the actual implementation will be presented along with challenges that arose during development. Two example rulesets that can be used with the proof-of-concept will also be provided in this chapter. The first ruleset is meant for ECUs running the Adaptive platform, while the second is meant to give an example for network separation. In the fifth chapter we compare the firewall concept to a concept that achieves something similar, IAM, also known as Identity and Access Management. Finally, a conclusion is given and suggestions for future work will be provided.

2 The AUTOSAR Adaptive Platform

The Automotive Open System Architecture, or AUTOSAR for short, was initially developed in conjunction by people from different automotive companies, suppliers and tool-developers. It is a standardized software architecture that aims to offer a software foundation that is independent of the underlying architecture and vendor while remaining widely usable on different ECUs [Sch19].

In-vehicle systems used to contain basic ECUs designed to last the entire lifetime of the vehicle while performing specified tasks with low resource requirements in resource constrained environments. These ECUs would work hand in hand with the electro-mechanical systems present in the vehicle. The AUTOSAR Classic Platform provides a vendor independent solution for this use case. ECU software might be updated slightly but this is generally not expected [AUT21d].

As vehicles get equipped with more and more features, many of which require high computing power and very low latency between components, the Classic Platform is not able to keep up with the rising requirements. Use-cases like automated or assisted driving require several interconnected high-performance ECUs that can process data from many sensors and react to the given stimuli with hard latency constraints. To provide a platform that is suited for the ever evolving in-vehicle systems and ECUs the AUTOSAR Adaptive Platform (AP) was developed. It supports high-performance computing and provides low latency communication mechanics. Additionally, it implements mechanisms to update embedded ECUs over-the-air. Features of the AUTOSAR Classic Platform (CP) that are not part of the AP can still be integrated into the AP [AUT21b].

Legacy communication technologies like CAN that are supported by the CP cannot keep up with the growing bandwidth requirements. For this reason, Ethernet is supported and used in the Adaptive Platform. Ethernet has been proven to provide a high bandwidth, efficient transportation of long messages, point-to-point communication and other useful features. This makes Ethernet a good communication technology for the in-vehicle use. Along the increased bandwidth need, the increased need in processing power requires the use of many-core processors equipped with tens to hundreds of cores. New chip architectures like Network-On-Chip allow the communication between cores on a single chip to be magnitudes faster than the inter-ECU communication that used to be customary. Energy efficiency is another problem that needs to be tackled. To deal with this, a compound architecture that makes use of many-core processors, co-processors, GPUs and FPGAs is generally employed. This exceeds the scope of the Classic Platform and is another reason the Adaptive Platform was developed [AUT21b]. Along with these requirements the software of the components inside the vehicle needs to be updated regularly to account for changes in the external systems, add new functionality or fix security problems. Since new exploits can be discovered that leave outdated systems susceptible to attacks, the latter is an important reason to keep the software updated [AUT21d].

2.1 Platform Characteristics

To accommodate the need for higher computing power the Adaptive Platform was designed from the ground up to be able to cater to those needs. Design decisions taken early in development allow the platform to take advantage of technologies that are traditionally not used in ECUs. There are seven major design decisions that help achieve this [AUT21b].

The first design decision is the use of C++ when programming software for the Adaptive Platform. C++ has been shown to be suited for the development of resource-intensive algorithms. This allows the developers to adapt novel algorithms quicker and with better optimization than with other languages that were traditionally used in this environment like Ansi C [Sch19].

The second decision is that the Adaptive Platform is a Service-Oriented-Architecture (SOA). An SOA allows a platform to provide their functionality through different services that can be used by developers or even other services. This decouples the components of the platform and allows for more flexibility and scalability while also being able to run more complex programs with ease. It does not matter where a specific service is run, as long as it is reachable over the network. If the service runs on a remote ECU it can still be used as if it were a local call, the Adaptive Platforms communication infrastructure will handle the necessary calls to the remote ECU in such a case. High-bandwidth communication technologies like Ethernet, which has been widely used in recent years in high-bandwidth, low-latency environments, benefits this sort of distributed computing architecture as well [AUT21b].

The third design decision is the ubiquitous use of parallelization. Since distributed programming is parallel by default, different ECUs can obviously run software at the same time. The SOA of the Adaptive Platform is distributed as well, therefore it is also able to have some parallelization by default. Since different applications run on different ECUs they tend to rely on different services that can be run in parallel. Additionally, to the inherent parallelism provided by the architecture, the use of many-core processors and heterogeneous computing, the use of different kinds of processors in the same architecture, allows for efficient parallelization inside single ECUs as well. Automatic parallelization tools are also part of the AP standard and should be implemented by an Adaptive Platform provider to further help with parallelization of novel software [AUT21b].

The reuse of existing standards is another design decision. Standards that have already been established and proven to work in their respective fields should be reused and adapted to the Adaptive Platform as much as possible. This aims to reduce time used on defining new standards that could have easily been reused from somewhere else to increase the development speed over all. Existing functionality in general should be adapted to the AP as much as possible and new functionality should not be coded from scratch without consideration [AUT21b]. Instead of creating a new standard for a standard that is already used elsewhere it is advised to provide information on how to adapt this standard to the Adaptive Platform [Sch19].

Both safety and security play an important role in the Adaptive Platform. For this reason, another design decision is how to introduce new features, software and concepts in general to the AP. Since many systems that run the AP are security critical or need at least a basic level of security, the introduction of new concepts must be monitored carefully so that no new vulnerabilities are introduced. The service-oriented architecture of the platform already decreases the coupling of

components making interference less likely when changing a single component. Additionally, coding guidelines that increase security need to be adhered to when working on the platform, such as the official C++ coding guideline [AUT21b].

Another design decision was the introduction of planned dynamics. The platform is able to allocate required resources automatically as needed during runtime. Constraints can be imposed on the system via the execution manifest but remain highly configurable. This reduces the risk of unwanted interactions during runtime [AUT21b].

The last design decision deals with the development cycle of the Adaptive Platform. The platform aims to be compatible with different developments styles. However, the Agile [atl22] approach to development is currently favored. The whole platform is incrementally scalable, meaning that it will be able to be updated after it was already deployed [AUT21b].

If AUTOSAR Adaptive is implemented in a target vehicle it usually complements implementations of the Classic Platform and non-AUTOSAR systems. A composite system is created that uses the different platforms for different use cases. In this case the Adaptive Platform will work hand in hand with the Classic Platform, the non-AUTOSAR software and additional back-end systems to create an interconnected and integrated system [AUT21b].

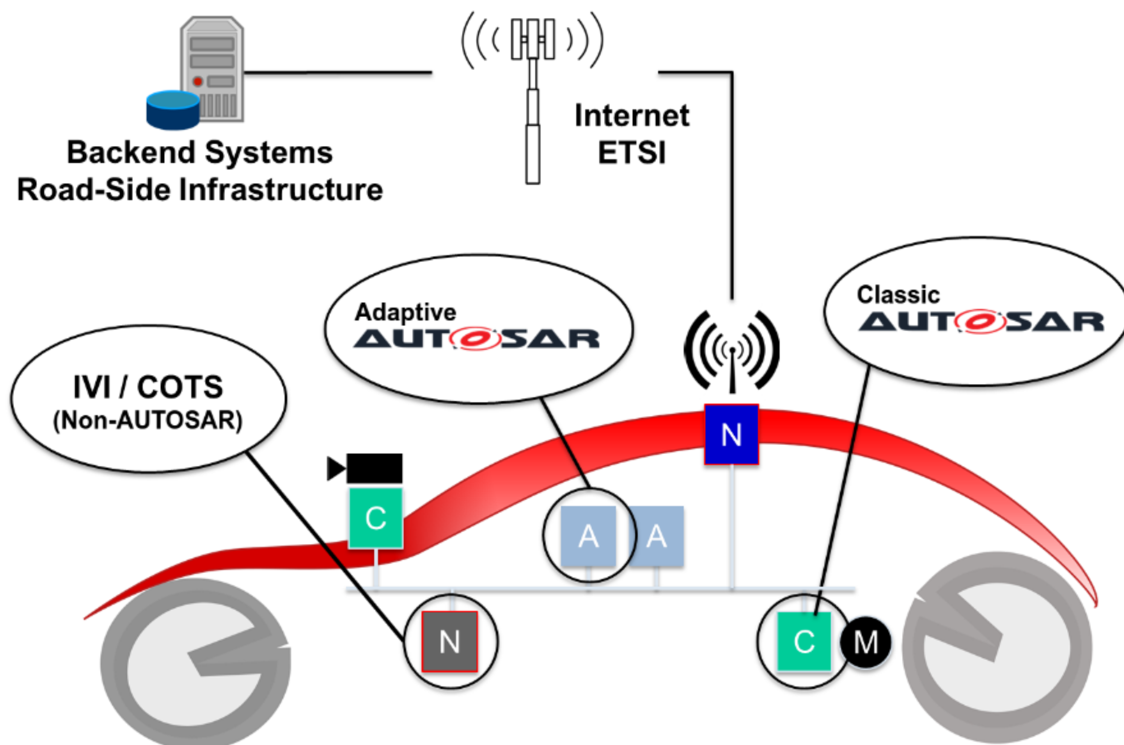

Figure 2.1: Example system with multiple different platforms, including Adaptive and Classic AUTOSAR and non-AUTOSAR software [AUT21b].

The example system shown in figure 2.1 demonstrates how different systems can be deployed for different use cases while remaining highly interconnected. In this example the communication over the air will be handled by AUTOSAR independent software, which is normally the case. All

systems are connected over an internal bus system. A software implementation that could make use or even require such an environment would be autonomous driving of the vehicle. In this case, the Classic Platform would be used to collect data from the different sensors built into the vehicle. This data will be transferred to ECUs running AUTOSAR Adaptive which can handle and evaluate these amounts of data efficiently. With this, the current vehicle state can be determined as well as the state of the environment as measured by the sensors. This allows the system to determine if speed and direction of the vehicle need to be adapted or not. Necessary changes to both speed and direction of the vehicle will again be handled by the CP after the evaluated data has been passed back to systems running it. The IVI and COTS systems in the figure are non-AUTOSAR systems. Here, IVI stands for „In-vehicle Infotainment“ which can be used to provide both entertainment and information for the passengers. COTS stands for „Commercial-off-the-shelf-Products“ which is a line of products that is mass produced for all kinds of use-cases and can be integrated easily. In this figure, ETSI describes a series of communication standards used to communicate over-the-air. Additional information, such as the original equipment manufacturer or a map used for navigation, is provided via the backend systems [Sch19].

2.2 Architecture

The Adaptive Platform provides a custom runtime environment for programs implementing AUTOSAR Adaptive, so called Adaptive Applications (AAs). This runtime is called AUTOSAR Runtime for Adaptive Applications (ARA). It is generally implemented on a Unix based operating system such as QNX or Linux. Functional Clusters (FCs) are the basis of this runtime environment. They provide the necessary interfaces to access AUTOSAR features. These clusters are grouped into two distinct categories:

- Adaptive Platform Foundation
- Adaptive Platform Services

The Adaptive Platform Foundation provides fundamental features such as execution management, communication management and logging. Adaptive Platform Services provide functionality standardized by the platform like state and network management [Sch19]. Adaptive Applications can also expose their functionality to other Adaptive Applications as a non-platform service [AUT21b].

The interfaces provided by both Adaptive Platform Foundations and Services are implemented through C++ function calls. This allows both types of Functional Cluster interfaces to be called the same way, even though the underlying implementation might differ. Additionally, interfaces might be offered in other programming languages in the future as well [Sch19]. The Functional Clusters shown in figure 2.2 are meant to give an overview of the logical view of the AP architecture. Some of the clusters might not yet be part of the AP but will be added in future releases [AUT21b].

The environment AUTOSAR Adaptive is executed in is seen as a single machine by the platform. This creates an environment that is independent of the actual underlying hardware and creates a consistent execution environment. A machine can either be a real physical machine, a fully virtualized one or a combination of the two. This means that it is possible to run more than one instance of the AP on a single physical machine by using virtualization. However, only a single instance of the Adaptive Platform can run per virtualized machine [Sch19].

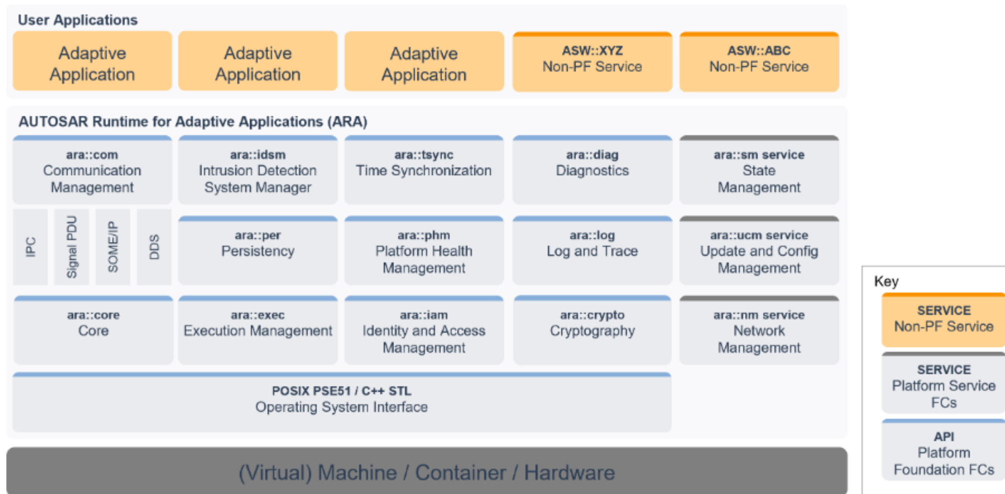

Figure 2.2: Adaptive Platform architecture showing existing and planned functional clusters [AUT21b].

Execution and termination of applications is handled by the highly configurable Execution Management cluster. Not only Adaptive Applications but also other Functional Clusters are treated as applications by the Execution Management for this purpose. It manages their lifecycle according to the information provided by the State Management FC [AUT21b]. Applications can also communicate with the State Management cluster during runtime to coordinate what applications need to be started or terminated [Sch19].

2.2.1 Communication

Communication between Adaptive Applications is handled exclusively by the platform provided communication interface. This is required since Inter Process Communication (IPC) between Adaptive Applications is not specified or standardized. The communication interface provided by the Communication Management cluster handles the implementation of the communication as well as communication in and between machines [AUT21b]. Local AA may interact with each other without using the designated communication interface by using their respective ARA interfaces. However, this kind of communication is not an explicit communication interface and rather a byproduct of the interaction between the applications through their ARA interfaces [AUT21b].

Implementing custom communication interfaces is supported by the Adaptive Platform. It should still be guaranteed that existing security requirements and general communication standards are not violated. Custom interfaces like this should be kept to a minimum and only be used if the existing one is not suitable for a certain use-case. This ensures that the portability of the platform remains as high as possible [Sch19].

Both processes from Adaptive Platform Foundation modules and Adaptive Platform Services will have to use IPC to interact with Adaptive Applications. There are two approaches this can be implemented. These are:

- **Library-based approach:** The interface library of the Functional Cluster trying to communicate with an Adaptive Application calls the IPC directly. The interface library needs to be linked to the respective AA in this scenario. This method should be used if the Functional Cluster is only used locally. This approach is generally easier to understand and can be more efficient than the service-based approach.
- **Service-based approach:** The Functional Cluster uses the Communication Management interface. In this scenario a server proxy library needs to be linked to the AA that will call the Communication Management interface to perform IPC between the AA and the proxy server. This method should be used if the FC is distributed over multiple instances of the AP as it handles communication across machines transparently and regardless of location.

A combination of both methods can also be used in certain cases. The AP Foundation clusters are generally Library-based while the AP Services are, as the name implies, service-based [AUT21b].

2.2.2 Multi-Processing

To provide proper multi-threading capabilities the use of a designated AP Operating System is required. Adaptive Applications are then generally implemented as a set of one or many processes with its own memory- and namespace. Processes always belong to exactly one application. If the application consists of multiple processes, they may also be distributed over multiple instances of the AP. The Functional Clusters, as well as the Adaptive Platform service and non-platform services are also implemented as a set of processes and sub processes. Therefore, the Adaptive Platform itself and the Adaptive applications are just a set of processes. These processes may contain sub-processes or threads and can interact with each other through Inter-Process-Communication, if they belong to the AP. Processes belonging to Adaptive Applications will have to communicate through the communication interface as stated earlier [AUT21b].

2.2.3 Configuration

The configuration of an AP product is handled via manifests. A manifest is an AUTOSAR model description piece. Since every AUTOSAR deployment aspect needs to be configured this way, multiple smaller manifests are used instead of a single one. This allows more readable manifests that are easier to understand. Different manifests are also important at different times of the deployment process. The most important manifests are:

- **Execution Manifest:** This part of the manifest is bundled with the executable code that specifies the integration of the general executable code and specifies information related to its deployment.
- **Service Instance Manifest:** This manifest configures the requirements of the used communication protocols. It is bundled with the code related to the implementation of the service-oriented communication.
- **Machine Manifest:** This is bundled with the software needed to establish an AP instance and specifies the content related to the configuration of the underlying machine.

- **Application Manifest:** This is bundled with individual Adaptive Applications. Can be used to configure an AA.

Additionally, the general application design also needs to be provided. It is not deployed with the AP but instead specifies how application software should be created for the platform [AUT21b]. The format manifests are written in is standardized as ARXML, an adaptation of XML for the AUTOSAR platform [AUT17].

2.3 SOME/IP

Scalable Service-Oriented Middleware over IP (SOME/IP) is a Remote Procedure Call (RPC) protocol for AUTOSAR that is designed to be used in embedded environments with several resource constraints. While generic RPC protocols exist already, SOME/IP aims to solve a few issues that other commonly used RPC protocols have. The main aim for SOME/IP is to provide a protocol that excels in embedded environments while still being highly scalable and compatible with the AUTOSAR standard out of the box. Additionally, SOME/IP is designed to provide features needed in automotive contexts by design while remaining highly compatible with many communication partners and standards. It is designed to be implemented on multiple different operating systems like AUTOSAR, GENIVI or embedded systems that do not run an operating system at all. The main use case for the protocol is the serialization of Client/Server communication between ECUs [AUT21e].

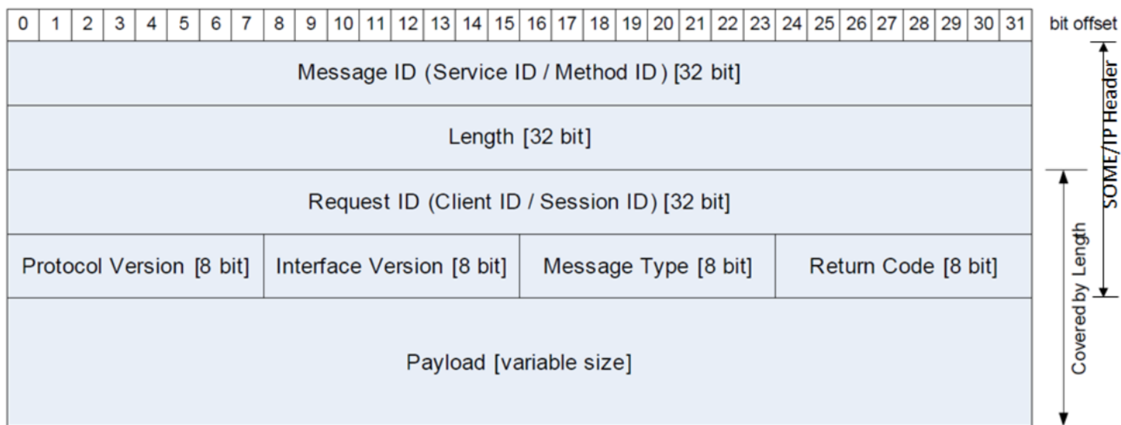

Figure 2.3: Standard SOME/IP header format including all required fields [AUT21e].

The header looks slightly different if E2E encryption is used. In this case there is another E2E header added before the payload [AUT21e].

The header fields, as seen in figure 2.3, of a SOME/IP message are used for:

- Message ID: This is a 32 bit identifier that specifies the used RPC or to specify a certain event. It contains either a Service/Method ID or an Event ID depending on whether an RPC call or an event is specified. This ID should be unique for the entire system.
- Length: This field specifies the length of the SOME/IP message starting from the Request ID to the end of the message.

- Request ID: The Request ID is used to uniquely identify a provider/subscriber pair to handle multiple parallel accesses. This is important when multiple subscribers want to access the same method or event at the same time. A provider will copy the Request ID in his response to the Subscriber and add it to the response. For a client the Request ID is the unique Client ID assigned to it. The Session ID is meant to help differentiate between multiple messages from the same client. The Session ID is also a unique identifier and is generally incremented after each call. Session handling can be deactivated in which case the Session ID will be set to 0x00 in every message.
- Protocol Version: This field specifies the version of the SOME/IP protocol and what kind of header format is used.
- Interface Version: The interface version field specifies what major version of the Service Interface is used.
- Message Type: Specifies what kind of message is being sent. This can either be a request, a response, a notification, an error or the Transport Protocol (TP) version of these messages. The TP message types are only used when the SOME/IP-TP protocol is used to send large messages over the User Datagram Protocol (UDP). If the message is a TP message it signifies that the packet is a segment.
- Return code: The return code shows whether an RPC has been completed successfully. For messages that are not response messages or error messages this field is always 0x00.
- Payload: This field carries a serialized set of parameters supplied for a remote procedure call.

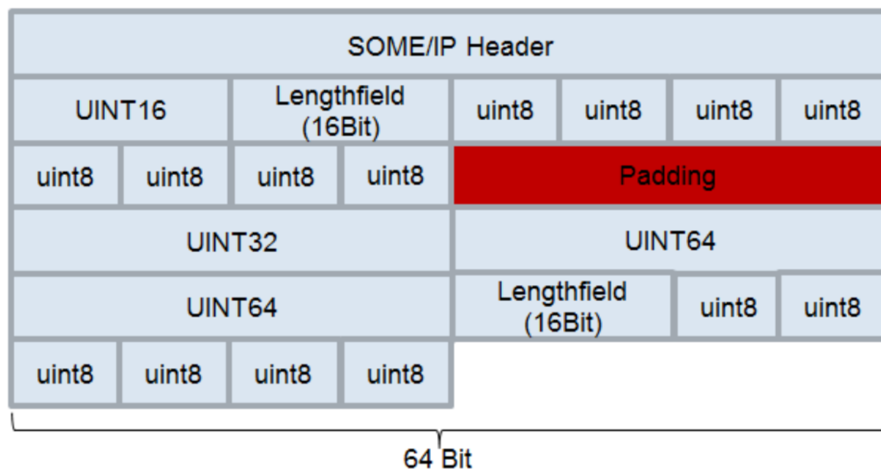

Figure 2.4: Example padding of a message containing variables of different length. It contains five different members, first a UINT16 variable, then an array of variable size containing UINT8 elements. A UINT32 and UINT64 variable. And finally another array of UINT8 elements [AUT21e].

Serialization of data is an important part of the protocol and depends mostly on the number of parameters passed and the data structures used. The goal of serialization is to be able to send a data structure over a message and to be able to deserialize the sent data to get the original data structure.

This allows the transmission of complex data structures such as arrays or structs via the protocol. The position of all data structures in the message is given in the specification. This also includes necessary padding elements to help align the data for performance reasons [AUT21e].

Supported basic data types for the protocol include signed and unsigned integers with 8, 16, 32 and 64 bits length, boolean variables and floats with 32 or 64 bits length. Supported advanced data types include structs, fixed length and dynamic length strings and fixed length and dynamic length arrays. The exact specification on how those data types are serialized can be found in the official SOME/IP documentation [AUT21e].

The SOME/IP protocol supports the two basic transport layer protocols for message transmission, UDP and the Transmission Control Protocol (TCP). It is also possible to send more than one SOME/IP message in one UDP packet or TCP segment. To support this SOME/IP needs to be able to receive unaligned messages since alignment of SOME/IP messages inside one transport layer segment cannot be guaranteed. Additionally, UDP multi-casting is also supported [AUT21e].

When transporting large messages over UDP that exceed the max length of a UDP packet, the SOME/IP Transport Protocol (SOME/IP-TP) should be used instead. It allows splitting the original SOME/IP message into smaller segments that can then be transported over UDP individually. The header of a SOME/IP-TP segment includes an offset field, a field of three bit for reserved flags not to be checked by the receiver and a one-bit flag that is set to zero for the last segment and to one for every other segment. TCP should only be used for long messages if there are no hard latency requirements as TCP can result in high latency when errors occur [AUT21e].

There are four types of messages that can be sent over SOME/IP. They are:

- Request/Response messages: These messages are generally used for remote procedure calls and are widely used. A request message indicates that it expects a response after the request has been processed.
- Fire&Forget messages: This type of message works like a request message but does not expect a response.
- Notification messages: Notifications can be used in a publish/subscriber context where every subscriber receives the notification message from the related publisher. The basic SOME/IP protocol does not support the publish/subscribe concept itself and can only be used to transport the actual message. The publish/subscribe functionality is instead implemented in the SOME/IP Service Discovery (SOME/IP-SD) protocol.
- Error messages: Error messages are used to return errors in the case of a failed request. Instead of sending a response message with an error code, an error message containing the exception and error information is sent instead. Having a designated error message format allows for greater flexibility when handling errors.

The SOME/IP specification [AUT21e] also provides information about when to use which transport protocol. Generally, the UDP protocol is favored since it is light weight and has low latency which is desired in automotive contexts. The applications receiving the UDP packets is generally also able to deal with communication errors. If a SOME/IP message is too long for the default UDP protocol and there are still hard latency constraints (below 100ms), then the SOME/IP-TP protocol should be used with UDP as described above. However, if there are no latency constraints in the case of errors and the SOME/IP message is longer than the maximum UDP length TCP should be used instead.

If external transport mechanisms such as the Network File System or APIX link are better suited than SOME/IP they should be used instead. SOME/IP can still be used in conjunction with those to transport e.g., a file handle [AUT21e].

Before any communication can happen via SOME/IP the sender needs to be aware of what kind of services are available on the network. This can be done by using the SOME/IP Service Discovery protocol. This protocol uses publish/subscribe messages to communicate available service instances and to show which service instances are currently accepting messages [Aut17].

3 Firewalls

In general, a firewall is used to protect a host or a separate network from malicious actors. This is especially important when the network has higher security requirements or contains sensitive data or vulnerable machines [SH09]. Companies that expose their own infrastructure to the internet or regular home networks will both need protection from malicious traffic. Firewalls have been around since the early 90s to tackle this exact use-case. Back then, the emergence of malicious software, such as the Morris Worm [IF02] marked the end of an open and trusting internet community. Nowadays, most networks are protected by a firewall that is preinstalled and set up by default on many commercially available routers [AVM22]. Host Firewalls are installed on a lot of end-user machines by default as well, such as the Windows Defender Firewall for Windows machines [Mic22].

However, creating a custom firewall, and along with it a custom policy, is not trivial. Many aspects need to be considered when setting up a policy for a specific network or machine. While this paper focuses on Host Firewall technology the policies discussed in the next chapter will be applicable for firewalls in general and not just host firewalls.

All traffic that passes the firewall can be checked by its filter, in the use-case discussed in this paper only inbound traffic will have to be checked. Outbound traffic from a compromised ECU cannot be trusted since the Host Firewall itself will also be compromised in such a case. Generally, outbound traffic is checked in company networks to prevent certain connections and specific traffic from leaving the network. To check traffic passing through a firewall, all messages are evaluated against a policy. Each rule in the policy that applies to the message in question is evaluated sequentially. Rules can route the message to a different destination or require the message to be dropped.

Main points to consider for firewall design according to the National Institute of Standards and Technology (NIST) [SH09] are:

- The Network Address Translation Protocol (NAT) is not to be considered a type of firewall. Instead it is merely to be used and seen as a form of routing. It should not be configured to mimic the functionality of a firewall.
- For outbound traffic it should be guaranteed that the source IP is the IP of a machine inside the organization.
- If compliance checking is to be used it needs to be ensured that the firewall can actually block harmful communication. Messages that could be harmful to currently protected and sensitive systems need to be detectable by the firewall.
- It is important to consider what kind of firewall is needed. Host and remote firewalls can fulfill different roles in a system. For this paper we will mainly focus on Host firewalls.
- Deployment and configuration of Host Firewalls should be centralized. This helps with policy upkeep as well as initial setup and configuration.

3.1 Firewall Policies

Firewall policies use a set of rules and information found in messages, such as source IP, destination IP, content types and many more, to decide on an action to take. The default action should be to drop the message, if no rule is found that specifies what to do with it. This concept of a whitelist is more robust than the opposite, a blacklist that only flags certain messages as dangerous, because of the ever changing attacks that are found [SH09].

To correctly and accurately identify what kind of traffic should be allowed to enter the system an initial risk analysis needs to be performed. This analysis helps find attack vectors and what kind of network traffic needs to be explicitly allowed. Furthermore some traffic might only be allowed under certain circumstances which also needs to be taken into account [SH09].

Along with this, content can be filtered according to the protocol used. Only protocols that are frequently used when communicating with outside networks should be permitted to pass through the firewall. Some IP protocols that are usually used in such a context include:

ICMP: The Internet Control Message Protocol (ICMP) is used for exchanging status and error information via IPv4 [se22]. It is generally unfeasible to block all ICMP traffic since it will lead to performance issues and problems with diagnosing errors over the network. Not blocking any ICMP traffic will allow attackers to potentially manipulate the flow of network traffic. This is problematic because it can be used to avoid security checks inside the network. As a solution, only certain parts of the ICMP protocol should be permitted, such as important network diagnostic messages. The ping command can be blocked on incoming messages to prevent attackers from learning about the inner workings of the network [SH09].

TCP/UDP: The Transmission Control Protocol and the User Datagram Protocol are among the most used communication protocols used on the internet. They are both able to send data packets over the internet, however while TCP is stateful, UDP is stateless [Sec22]. Both TCP and UDP can be used by higher-level protocols. In such a case a single application can use either UDP, TCP or a combination of both. Limiting the outgoing UDP and TCP traffic is usually not viable since the majority of applications make use of the protocols. The firewall should be able to block malformed UDP and TCP messages. If required it should also be able to report that malformed messages have been received. Such messages can be used by attackers to scan the system for potential vulnerabilities [SH09].

ESP/AH: ESP and AH are both used in combination with IPsec. The Encapsulating Security Payload (ESP) protocol is used for authentication in the IPsec protocol. This can be used to authenticate the user when using a VPN [ele22]. The Authentication Header (AH) protocol can be used to achieve both message integrity and authentication of the data source [Sri22]. If an organization does not allow either of the two protocols then IPsec VPNs that start or end in the network will not work. When IPsec VPNs are allowed then both ESP and AH traffic should still be restricted to certain addresses only. These addresses function as IPsec VPN endpoints.

In some contexts, even one of the more frequently used protocols will not be needed at all. In this case prohibiting and blocking that protocol with the firewall will greatly reduce attack vectors on the system. All other IP protocols, unless required, should be disabled by default. Some other commonly used protocols might only be viable for inter network communication. In such a case

the protocol can also be blocked by an external firewall, since all necessary traffic using it will not traverse that firewall at all, while being permitted by the host firewall. One protocol like that would be the IGMP protocol for broadcasting messages. It is rarely ever used over the internet [SH09].

Besides certain IP protocols, it is also possible to block certain IP addresses. Some IP addresses should be blocked in general, since they won't ever be a valid source for messages anyways, such as the localhost addresses 127.0.0.0 to 127.255.255.255 as well as the address 0.0.0.0. The latter will be interpreted as a localhost or broadcast address depending on the operating system used. The link-local addresses 169.254.0.0 to 169.253.255.255 should be blocked as well as stated by the National Institute for technology [SH09]. If the source address of an incoming message or the destination address of an outgoing message is invalid, the message should be blocked. Messages like these will usually be caused by either a malicious actor or misconfigured hardware, both of which should be handled and usually blocked by the firewall. In cases where there are no hardware problems the messages might be caused via malware or spoofing. If outgoing messages use an invalid source address the component sending the message might be compromised. In such a case it is important that the outgoing message gets filtered by the firewall. Otherwise a compromised system can be used to send malware or malicious messages to other systems. Messages directly addressed to the firewall should generally be blocked as well, unless the firewall acts as an application proxy or provides other services [SH09].

Depending on the security requirements of the system, additional firewall rules might be appropriate. For security critical systems, such as company internal networks, it is recommended to block messages that use IP source routing information. This information would allow the message to take a certain path through the network. This might allow an attacker to circumvent network security controls meant to protect the network. Another potential attack that needs to be mitigated in such networks are inbound messages that contain broadcasting addresses. If those addresses are inside the network, an attacker will be able to direct the responses to a certain address that does not have to be the original source address. With this, a lot of traffic can be generated and pointed to a specific address with very little effort from the attacker, effectively resulting in a denial of service attack. However it might not be possible for all networks to block all broadcasting traffic across the firewall, because the functionality is specifically needed for certain services. In such a case different precautions must be taken [SH09].

General internal addresses that should not be accessible from the outside should be blocked by default. However, it is important to keep in mind that IP addresses might change over time. In cases like this the policy needs to be adjusted accordingly [SH09].

With the increasing importance and relevance of Internet protocol Version 6 (IPv6) [Rod12] it is necessary for firewalls to be able to deal with IPv6 addresses as well. Even if the internal network is not equipped to deal with IPv6 addresses it is important that the firewall is able to filter such content without issue. Since IPv6 supports some features that are the same as in Internet protocol Version 4 (IPv4) the firewall should be able to filter on those features regardless of the IP version used. For firewall setup it should be possible to adapt firewall rules from IPv4 to IPv6 easily. IPv6 specific protocols should also be filtered by the firewall, along with the ability to deal with IPv6 packets that are being tunneled in IPv4 packets. This is currently used because transit routers usually support IPv4 rather than IPv6. Since IPv6 is still not widely adopted, the aforementioned points are guidelines that should be followed for a firewall that needs to be explicitly IPv6 aware [SH09].

Another approach to implementing a firewall, as opposed to basic port based firewalls, are application layer firewalls. This kind of firewall can inspect traffic up to the application layer. Requests that are addressed to a server in the network will be allowed to pass through the firewall but will then be caught in a different server that inspects the messages thoroughly for any malicious code or general vulnerabilities, like possible SQL injections [f522]. The servers that catch the requests contain additional security layers to provide a well secured environment compared to the original destination server [SH09].

There are both active and passive approaches to application firewalls. The difference is that passive application firewalls do not block traffic that contains vulnerabilities. The active version will only allow safe requests [f522]. In general, application firewalls should be used if a server is not protected enough by a conventional firewall. This is the case when the server itself is missing critical security features that can not easily be added to it. However, one of the drawbacks of using such a firewall setup is that there will be an increase in latency since the request gets evaluated by the application proxy first. It is also important to take into consideration how laborious it would be to update the filter rules for the server and the application firewall in the case of IP address changes or newly discovered attack vectors [SH09].

Application proxies can also be used to monitor outbound traffic. Most commonly this is used to monitor connections over the HTTP protocol. With this, network traffic can be checked and tested for malicious connections before a response is sent to the remote server. Additionally all traffic passing the proxy can be logged this way, allowing network administrators to better understand what traffic users are sending. Additionally users can be made aware of malicious requests by the proxy server this way. The use of proxy servers for HTTP is generally advised not just for security reasons but because they allow the caching of frequently used web pages to drastically improve connection speed and latency while also decreasing the amount of bandwidth required for the connection [SH09].

Other metrics the firewall can reject connections by are both user activity and user identity. For user activity, existing connections will be blocked, if there has not been any incoming traffic from the user for a certain amount of time. This way, a new connection has to be established before another request can be made by the user. While this can be bothersome for users who do not use their established connection frequently it does add more security. If people forget to log out of the system for example they will be logged out automatically after the set time has elapsed, making the window for a possible attacker much smaller. Alternatively, the rate of traffic can also be throttled if needed with a network activity based policy. In such a case, traffic can be rerouted or throttled if a certain threshold is crossed. This can be difficult to implement since redirecting traffic to take a slower route or throttling in general can cause communication failures [SH09].

With services like VPNs the use of user identity based policies becomes feasible. As VPNs generally support ways to authenticate users, they allow a firewall to deny connections based on this authentication. However, basic firewalls might lack the necessary functionality to authenticate users at all, even when VPNs are used and proper authentication was provided. Ways to authenticate via conventional VPNs include:

- Multi-factor authentication with a secondary device
- Personalized secrets provided to individual users
- Digital certificates

Firewalls can enforce these rules and limit what parts of the network a user can access by using Network Access Control (NAC) and network separation. Application firewalls can also be used to reject requests from unauthorized users within the application. The same goes for proxies as well. Additionally, firewalls with user authentication based policies should be logging the users' identity as just logging the IP address will not be sufficient in determining why a certain connection or request was denied [SH09].

3.2 Filtering types

Filtering types for firewalls can generally be divided into three tiers:

- Stateless filtering
- Stateful filtering
- Deep Packet Inspection

The most basic form of filtering is stateless packet filtering. Initial firewalls and very basic ones rely on stateless filtering only. Stateless filtering allows traffic that passes through the firewall to be routed in a certain way or dropped according to a ruleset. Rulesets are a collection of certain actions to be taken, once a certain sender or receiver information or a combination of the two is encountered. A stateless filter will not keep track of the state of the traffic that passes through it, as the name implies. Additionally, it does not check the contents of messages or requests at all, it just checks information available on the routes based on this information, according to the ruleset. This includes the source and destination IP addresses, the protocol that is being used for communication, addressed ports, if any specific port is addressed and the interface the message has to pass through and whether it is an inbound or outbound message. Depending on what direction the message is being sent in, either into or out of the network, we talk about ingress and egress filtering respectively. Ingress filtering deals with messages that try to enter the system, while egress filtering handles any packet that tries to leave it [SH09].

Depending on whether ingress or egress filtering is used, different rules need to be enforced. Generally, for an organization a lot of outbound traffic can and should be blocked to prevent, among others, denial-of-service attacks that originate from within the organization network. It can also be used to prevent the use of protocols like the file transfer protocol that are hosted on foreign servers making sure no sensitive data can be sent over them. If there is traffic that needs to leave the organization network, the egress filter should check whether the source address actually belongs to a machine inside the network to prevent messages with spoofed or misconfigured source addresses from leaving the local network [SH09].

While stateless filters can protect against some attacks, they still remain very basic and more advanced approaches can easily get past such a filter. Attacks that specifically target the implementation of certain protocols, such as TCP or IP and abuse weaknesses in their specification will not be detected by such a filter. Since packet filters only deal with routing, they are not aware of information designated for higher layers. Specifically, information that can be provided but is usually used in malicious contexts, such as source routing, is not going to be discovered by a packet filter.

Additionally, spoofing of information in packet headers will also not be recognized by the filter and will therefore not be blocked. To deal with these kinds of attacks a firewall that operates on a higher layer is required [SH09].

Some packet filters have been equipped with more functionality to be able to correctly determine and handle fragmented packets. Attacks using packet fragmentation include:

- Masking other attacks by distributing them over several packets making them harder to detect.
- Attacking by sending spoofed fragmented packets, such as not sending the initial fragment of a packet or sending overlapping fragments.

Packet filters that are aware of packet fragmentation can be configured to not allow any fragmented packets to pass through. While this does protect against the aforementioned attacks, it also blocks legitimate sources that use packet fragmentation. A commonly used technology that usually uses fragmented packets are VPNs. They encapsulate packets within other packets to hide the original contents which can result in long packets that need to be fragmented to not exceed size limits of the used protocol. If all fragmented packets get blocked by the firewall then the VPN will not be able to establish a connection, since its communication will be severely restricted. To avoid fragmented packets being used for attacks, while still allowing VPN traffic, a firewall that reassembles fragmented packets can be used. However this does not come without drawbacks. Reassembling of packets at the firewall uses a lot of resources and creates overhead, additionally it creates a bottleneck into the network that can easily be exploited and attacked by denial-of-service attacks [SH09].

To improve a basic packet filter and protect the network against more advanced attacks aiming to bypass it, a stateful filter can be implemented. Such a filter will, additionally to the functionality of the packet filter, also be able to keep track of state information regarding connections. Such a filter is transport layer aware, opposed to stateless filters that generally operate on the network layer. The rules from the underlying packet filter will still be applied first, already blocking messages that come from, or are sent to, blocked IP addresses. Packets that do pass the first ruleset will then be evaluated by the stateful filter. Such a filter uses a state table to keep track of established connections, this way packets that do not belong to an existing connection or violate other parts of the used protocol can be filtered out effectively. State tables usually keep information on:

- Source and destination IP
- Source and destination ports
- State information about the connection

While the exact information that is being tracked can vary, the aforementioned points are usually the minimum information that the state table tracks [SH09]. For connection-oriented protocols like TCP there are 3 states, initiating a connection, connection established and connection terminating. With this, the filter will be able to determine the expected state of future packets. A packet that does not already belong to an established connection needs to initiate a new one, if it does not it will be discarded. This protects against attacks where a packet is designed to look like it is already part of an established connection. If a new connection is established or an existing one is terminated, the state table is updated accordingly. The packet header is inspected to get information necessary to achieve this [SH09]. Since TCP and UDP are widely used protocols most stateful firewalls can analyze traffic that uses these protocols in greater detail. In addition to the information named above

it can also keep track of sequence numbers and reject messages that arrive out of order. When filtering stateless protocols, such as UDP, a stateful filter offers barely any improvements over a regular packet filter. As there is no state to be kept track of, the filter can not make decisions based on it. However, there are still some advantages, such as keeping track of sequence numbers for UDP and only allowing DNS responses after a DNS query was sent from the network initially [SH09].

To further improve upon stateful filtering Deep Packet Inspection (DPI) can be used. This type of filtering is used when working with an application firewall. This kind of firewall additionally comes with functionality to inspect protocols on the application layer. This is achieved via use of an inspection engine. This engine can be used to find deviations from regular protocol use by comparing analyzed packets to vendor-supplied profiles of regular protocol behavior. This makes it possible to filter incoming traffic depending on how a protocol is trying to run an application [SH09]

DPI allows a more in-depth approach to filtering. It makes it possible to filter, among other things:

- Traffic that uses the same port as another protocol while only one of the two protocols is allowed.
- Traffic that passes a certain file type over e.g. email or another similar protocol.
- Traffic that uses a specific command of an application like the put command of FTP.

Using it to filter specific commands can be helpful when trying to secure the network against specific active network content coming from web pages, such as ActiveX. It can also be used to deny traffic signed by a certain certificate authority. This might be useful if the CA has been compromised or is otherwise invalid. Additionally, it can be used to detect sequences of commands. Receiving the same command over and over can hint at a denial-of-service attack on the network. Receiving messages out of order can also be a sign for an attack or the exploitation of faults in protocol implementations such as HTTP. A regular packet filter and even a stateful filter would have no means to filter or even detect such sequences [SH09].

Additionally, DPI can also be used to validate that inputs passed via a protocol are valid. For example, length and data type of certain fields can be checked and filtered. Suspiciously long usernames or text fields containing binary data are generally suspicious and could be filtered out. However, an application firewall needs to be aware of a certain protocol to inspect it properly. These days common application firewalls are aware of the most used internet protocols, such as:

- HTTP and HTTPS.
- Different databases such as SQL.
- General email protocols such as the post office protocol POP and the simple mail transfer protocol SMTP.
- Voice communication protocols such as the voice over IP protocol VoIP.
- Markup languages like XML.

Furthermore, application firewalls can be used to enforce strict adherence to the protocol standard for supported protocols. This compliance checking can check messages for RFC compliance. RFCs are documents produced by the Internet Engineering Task Force (IETF) that describe and specify,

among other things, internet communication protocols [IET22]. However, it is not unusual for protocols to deviate from the RFC specification slightly in certain use cases. Enforcing RFC compliance will cause such non-malicious messages to be filtered out by the firewall [SH09].

Application firewalls are generally used alongside stateless and stateful filters. A complete Intrusion Detection and Prevention System (IDPS) still offers greater protection. It would additionally to the capabilities mentioned above also filter traffic by using signature and anomaly-based analysis. This can detect additional attacks and problems in traffic with remote locations [SH09].

3.3 Host Firewalls

The firewall technologies discussed so far have been focusing on protecting a local network from remote attackers. In such a case all ingress and at least some of the egress traffic is routed through the firewall. However, communication between hosts inside the network does not get checked in such a way. Traffic between hosts inside the same network does not pass through the firewall and is thus not checked. This can be a problem in certain cases. In networks where even a single local host gets compromised it can then send malicious unfiltered messages to all other hosts inside the network. Even if the network is protected by a proper firewall or IDPS single hosts might still get compromised due to unexpected exploits, user error or physical access to the host [SH09].

To prevent a compromised host from infecting other hosts a host firewall can be deployed. A host firewall, as the name implies, does not sit on the edge of the network but instead is deployed on an individual host. This can protect hosts from others inside the same network as every host can then filter all in- and outgoing traffic in such a scenario. Since different hosts usually have different communication needs, the use of personalized host firewalls allows for more precise filtering. E.g., if one host is never expected to be addressed using the SOME/IP protocol then the host firewall can block all incoming SOME/IP messages while other hosts in the same network will still be able to communicate with it. Generally, a host-based firewall can be used to log all incoming and outgoing traffic, or just traffic that violates the filtering rules specified. They can also be used for address- and application-based access control similarly to network firewalls. Such a firewall can be configured to take certain steps after a threat has been detected, such as switching to a more restrictive filtering ruleset. In this case the firewall acts like an IDPS [SH09].

A host firewall that is running on a PC used by an end-user is usually referred to as a personal firewall. While the general purpose of a personal firewall is the same as the one for a host firewall there is usually a user-interface that is easier to work with for the general end-user [SH09].

While the firewall type that is covered in this paper is host-based, there are still features that are traditionally only part of personal firewalls that would still be interesting for our use-case. One of these features is the possibility to have multiple firewall policy profiles that can be switched depending on the location of the computer. This has traditionally been useful for laptops that are being used in remote locations and untrusted networks. The same can apply for the firewall implementation in AUTOSAR Adaptive, where the vehicle is obviously not stationary and might connect to untrusted remote networks for maintenance or updates to local software. Additionally, personal firewall distribution and management is usually handled in a centralized manner where the firewall gets distributed to all local systems from a single point. This ensures that all systems can

be kept up to date and secured easily [SH09]. Rolling out the firewall policies in such a manner for in-vehicle systems might be an efficient way to handle updating certain rulesets and should be investigated in section 4.

The use of a firewall appliance, a designated piece of hardware running a personal firewall, can help increase the security for PCs. A firewall appliance is connected between the network and the PC and is able to perform advanced firewall techniques like maintaining VPNs and Deep Packet Inspection. Additionally, if the integrated personal firewall of the PC malfunctions, is turned off or just misconfigured the firewall appliance will still guard the PC. This essentially creates a 2-factor firewall system that is harder to circumvent than either system on its own [SH09]. As this would require additional hardware for each ECU it will most likely not be feasible for use in vehicles.

3.3.1 Related technologies

A firewall proxy server employed by a firewall will act as an intermediary between the source and destination servers. It will decouple communication between the servers and instead create two separate connections. One between the source and the proxy and one between the proxy and the destination. The source and destination servers will therefore never be in direct contact and will not be aware of this. Both servers will only see the other server as their communication partner, the proxy server itself is transparent. The internal servers will therefore not be accessible from the outside directly. This decouples the internal servers from the outside world, they will not be directly accessible and their internal IP addresses hidden [SH09].

The proxy server can then directly access the firewall ruleset and can decide whether or not to block incoming connections. Additionally, the proxy server can also authenticate users. This can be done via username and password or other common authentication methods such as ID tokens. The proxy server is generally used like an application firewall. It is application layer aware and able to inspect the protocols used to connect to it. The proxy also performs the required handshakes with the external source and is able to inspect the traffic thoroughly. If the traffic is deemed acceptable according to the ruleset it can then be forwarded to the designated internal server [SH09].

There are a few advantages that a proxy server has over an application firewall. These include, among others:

- No direct connections to the internal server.
- Can inspect all traffic for policy violations.
- Can decrypt packets to check them for malicious payloads and then re-encrypt them.

Some traffic might still not be decryptable by the firewall, traffic like that will be forwarded to the destination anyways [SH09].

Disadvantages that come with the use of application-proxy gateways should also be kept in mind. The most limiting disadvantages of such proxy servers are:

- Added overhead from deep inspection of packets. Limiting for high-bandwidth systems in general unless a specialized application proxy is used that can handle such traffic. Latency is generally also increased which can be troublesome in real-time systems.

- Slow adaptation to new standards and protocols. General traffic the proxy server does not understand will just get tunneled through and conventional proxy servers only provide support for general protocols and applications and are slow to get upgraded. This is a problem especially for a use-case where lesser used protocols need to be tunneled through the proxy.
- Inbound proxy servers must be able to provide an interface for all functions that the destination server provides.

In general, it is a trade-off between security and an increase in overhead. It is important to determine whether the use of an application-proxy is beneficial in a certain setting or if the use of a stateful or application firewall is enough [SH09].

Another widely used technology that needs to be considered in the context of firewalls are virtual private networks. They can be used to connect remote networks and establish a secure and trusted connection. VPNs need to be considered when designing a firewall since they play an integral role in VPN traffic. Generally, the encrypted traffic from a VPN connection gets decrypted by the firewall. This is necessary for the firewall to inspect the traffic properly. Firewalls also need to encrypt VPN traffic for outgoing connections. For this to work properly gateways need to be used that sit outside of the firewall that receive and send the VPN traffic [SH09].

Common implementations connect either separate networks from the same organization, e.g. multiple offices in different locations, or a single PC or server with the organization's network. The latter usually requires additional authentication via credentials from the remote user. For gateway-to-gateway VPNs where multiple remote sites are connected no additional authentication is needed by the users. Connecting to a machine from a remote location this way is transparent to the user and will seem like a connection to a local machine [SH09].

For user-to-gateway firewalls a VPN policy can be specified. With such a policy it is possible to limit network access to resources based on user authentication. Protocols used to authenticate users are usually either the Remote Authentication Dial In User Service protocol RADIUS or the Lightweight Directory Access Protocol LDAP [SH09].

As with proxy servers the use of VPNs adds a non-negligible amount of overhead, depending on the encryption method used and the amount of data needing to be transferred. This can be mitigated to a certain degree by implementing hardware acceleration for encryption in the firewall [SH09].

3.4 netfilter

netfilter is a linux kernel module that can hook into different stages of the linux network stack to allow packet filtering of all incoming and outgoing traffic, as well as allowing rerouting. Functionality of the netfilter software is grouped into different modules that are part of the kernel. They can be attached to the network stack through designated callback functions. The main functionality of netfilter lies within stateless and stateful filtering of IP packets, both IPv4 and IPv6, as well as network address and port translation. It is a community-driven project that is part of Linux kernels starting from the 2.4.x series. [web21].

3.4.1 nftables

nftables is a program that allows the creation of a firewall rules that get executed by the underlying netfilter software. It is the successor of the iptables tool, however it is not limited to IPv4 tables and can add rules for many different protocols, such as IPv6, Address Resolution Protocol or rules managing ethernet frames. Additionally, nftables gives more flexibility than iptables since there are no predefined tables and chains. They can instead be generated as needed. Rules are evaluated in descending order, where rules that are added first will also be evaluated first. Multiple statements within one rule get evaluated from left to right. If one rule matches the incoming or outgoing packet, depending on whether the hook for the chain is input or output, completely, the defined action will be taken. The most used actions are either drop or accept, allowing packets to either be dropped completely or to pass through the filter [web21].

The nft command line tool can be used to set up additional tables, chains and rules directly in the command line. Tables are used as top level data structures that can contain chains, sets or other stateful object. The type of a table consists of a family of protocols. The families that are currently available are:

- ip: This family can filter IPv4 packets [net21b].
- ip6: This family can filter IPv6 packets [net21b].
- inet: This family can filter both IPv4 and IPv6 packets. Rules addressing a network layer protocol still have to address a single protocol, either IPv4 or IPv6. IPv4 packets ignore IPv6 rules during rule matching and vice versa [nft21].
- arp: The arp family can filter low level traffic, such as datalink layer traffic. This family hooks into the network stack before the kernel handles any network layer related tasks [nft21].
- bridge: The bridge family is aware of traffic that passes through network bridges.
- netdev: The netdev family allows tables to be attached to a single network interface, such as an ethernet or wifi interface. All network traffic that passes through the attached interface will be filtered this way. This family can be used in conjunction with an ingress hook. This allows filtering of packets early on which makes it an efficient way to drop packets related to DDoS attacks [nft21].

Only packets of the tables family will get passed along to the chains. Therefore it is important to choose the appropriate family for your use-case [net21b].

Chains group a set of rules together while providing a default policy. If no matching rules were found for a chain, the action specified by the default policy is taken instead. A priority can also be attached to the chain to define the order in which chains are getting evaluated as well as a hook to specify what kind of packets the chain evaluates. The default command to generate a new chain is: *nft add chain [<family>] <table_name> <chain_name> <parameters>* where the family is optional since it is only needed if the table does not have the ip family and the parameters can contain the hook, priority, default policy and a comment. All of these parameters are optional so a chain without any hooks could be generated, however no packets would be evaluated when checking the rules that are part of the chain. The default priority is 0 while the default policy is to accept all packets. Both of these parameters can be changed [net21a].

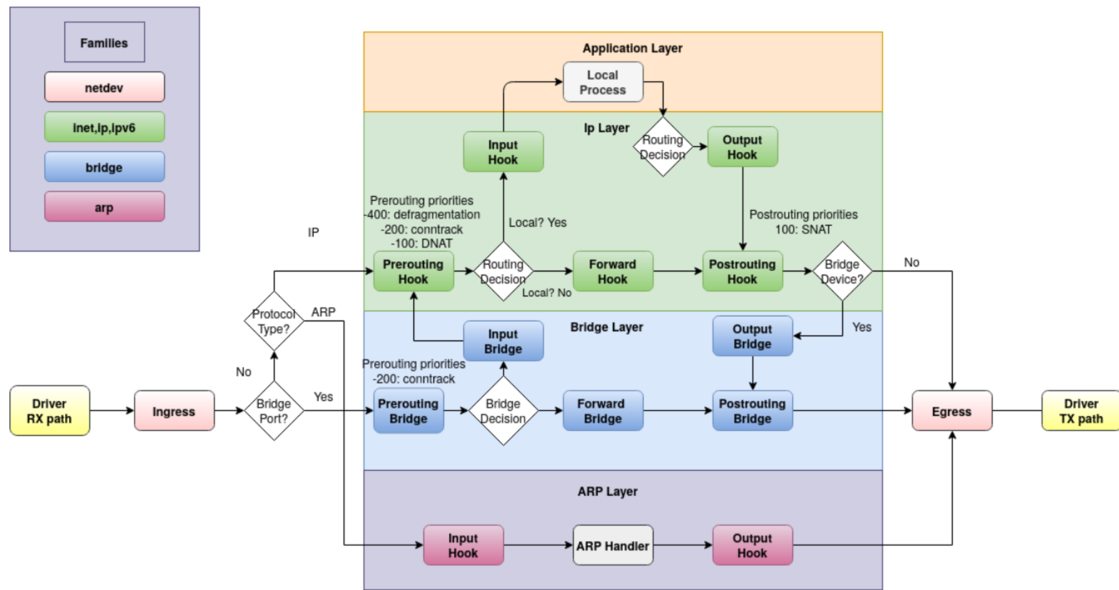

Figure 3.1: The netfilter hook structure along with the packet flow for incoming and outgoing packets [net21d].

The most important chain type for a firewall is the filter type as it allows to filter packets against the given rules. There are two more chain types, route and nat, which allow to reroute matching packets or perform Network Address Translation respectively. For a firewall the most crucial ones are the input and output hook, as they allow basic capturing of incoming and outgoing traffic. Other base chain hooks are:

- Ingress: As mentioned before, the ingress hook allows filtering of packets before any routing is done, even before the prerouting hook.
- Prerouting: This hook gets to detect packets before routing decisions are made.
- Forward: The forward hook detects packets that are not directly addressed to the system but are forwarded to a remote system instead.
- Postrouting: The postrouting hook detects packets after all routing decisions have been processed, just before the packets arrive at the local system.

Depending on what rules are added to the chain different hooks and priorities are needed to filter packets before e.g. reassembly of packets is executed. Figure 3.1 shows at what point of the traffic flow the different netfilter hooks register [net21a].

Priority of a chain dictates when it is evaluated. Chains with a lower priority get evaluated before chains with a higher priority. Having a lower priority can also cause the chain to be checked before internal operations take place. Using the right priority can, for example, cause your chain to be placed before certain routing decisions are applied. If more than one chain uses the same hook they are evaluated from lowest to highest priority. If the default policy of a chain is accept and no rules match, or if a rule matches which results in the packet being accepted other chains of the same hook

with higher priority will still be checked for this packet. Only if all chains accept the packet it will be allowed to pass through the filter. However, if any chain decides to drop the packet it will be dropped immediately, even if chains with higher priorities have not been checked yet [net21a].

This table, taken from the nftable manual page [net20], shows the standard priority names that are provided by netfilter. However any numeric value can be used as a priority:

Name	Value	Families	Hooks
raw	-300	ip, ip6, inet	all
mangle	-150	ip, ip6, inet	all
dstnat	-100	ip, ip6, inet	prerouting
filter	0	ip, ip6, inet, arp, netdev	all
security	50	ip, ip6, inet	all
srcnat	100	ip, ip6, inet	postrouting

Some of the priorities are only available on certain hooks while arp and netdev families only support the filter priority by default [net20].

Adding new rules to a chain can be done by executing the command:

nft add rule [<family>] <table_name> <chain_name> <expressions> <statements>

The family is again needed only if the table is not of type ip, both the table name and the chain name are required. Zero or more expressions can be attached to a single rule, if no expressions are given then the statements will be executed immediately. If at least one expression is provided then all expressions will be checked from left to right to see if they match an incoming packet. Only if a packet matches every single expression will the statements be executed. Expressions usually start with the protocol they are meant for and then filter against header files of that protocol, such as tcp destination ports, udp sender ports or sender IP addresses for IPv4 and IPv6. Statements can either be used to log, count, redirect or drop and accept packets. Additionally statements can refer to other chains that can evaluate the packet further [net21e]. With this, a basic stateless firewall can be configured by adding the desired rules to the right tables, bearing in mind what priority and hook is needed for the rules to be evaluated at the right moment.

3.4.2 Connection tracking

To set up a proper stateful firewall with netfilter and nftables, connection tracking is needed. This is possible via the conntrack system provided by netfilter. This system, also referred to as ct, can track existing connections and associate incoming packets by assigning them a state. This state depends on the state of the current connection, as tracked by conntrack, along with the headers of the incoming packet [net21c]. By default this system only tracks connections, no filtering or routing is performed unless specified in designated rules. They will again have to be properly registered in the appropriate tables and chains. There are four different states that conntrack can attach to an incoming packet. These are:

- NEW: This is the first state that packets of a new connection can be assigned with. It implies that the packet is part of proper initialization messages for a protocol. It also shows that, so far, the communication has only taken place one-way and no response packets have been received or sent yet.
- ESTABLISHED: The established state is assigned to packets that have been part of a valid communication, packets have been both sent and received for this connection and initialization of the protocol, e.g. the TCP handshake, has been completed properly.
- RELATED: This state is assigned to messages from a protocol that is somehow expected by the system. For example, this state would be assigned to ICMP messages belonging to an established IPv4 connection.
- INVALID: This state is reserved for packets that violate the expected traffic flow of a protocol. For example messages that try to close an existing connection before the connection was initiated.

As stated before, even messages in the INVALID state do not get filtered by default, however it is easy to add a rule that drops all packets with the INVALID state. Additionally, state and connection information will be attached to not just TCP connections but to other protocols as well, even though they might be stateless protocols. Currently supported are TCP, UDP and ICMP [h06].

The first connection tracking (conntrack) callback is registered early in the package flow in the prerouting hook. This callback checks for the proper formation of the incoming packet, such as checking the packet header for any malformations, and creates a new conntrack connection for this packet if none was found. If the packet belongs to an already established connection, a specific flag will be set in the related conntrack to signal this. For new connections, if the packet is allowed to pass through the firewall, meaning that it was not dropped at any point, it will be added to the hash table containing all conntracks. This allows efficient lookup of already established connections. Is the packet invalid or belongs to an invalid connection, the related conntrack is set to null. If the connection is explicitly untracked, a dummy conntrack will be used instead [h06].

For some higher-level protocols, such as FTP, additional information is required to properly relate messages. FTP uses the port 21 to send messages but does not have a predefined port for responses. To figure out what port the response is going to arrive at, a connection tracking helper can be used. Helpers define expectations that tell conntrack to expect a certain connection in a specified amount of time. Expectations usually contain information that is hard to track for conntrack, such as the port number for the FTP protocol. When an expectation has been defined, incoming messages will be matched against this expectation. If there is a match, the message is marked as related to the message that initially created the expectation [h06].

4 The proof-of-concept

The goal of the proof-of-concept (POC) is to have software that is able to parse a configuration file and set up a working firewall based on that. The proof-of-concept should be able to run on an operating system that is able to implement the AUTOSAR Adaptive platform like QNX or Linux. As this is a POC and not meant to be an actual implementation that is usable in a real life environment, there are no resource constraints for latency or memory usage. However, the program should still be able to run on a target ECU that implements AUTOSAR Adaptive without issue.

For testing, the WSL2 kernel from Microsoft was used as a basis, however the kernel was recompiled to include more netfilter modules that were missing in the official release. The module *CONFIG_NF_CT_NETLINK_TIMEOUT* was added to the kernel to allow adding of custom timeouts such as the default TCP session timeout. Most other netfilter modules that were needed are already part of the official WSL2 kernel.

One of the first decisions for the POC that was needed to be made, was where and how to implement it. The firewall itself can run either as a daemon on the actual system or in a piece of hardware that intercepts all messages meant for the target machine. We chose the daemon approach for two main reasons:

- Implementing in hardware would require additional hardware to be installed into automobile systems which requires more space and money.
- Having additional hardware that intercepts incoming communication causes additional overhead.

How to go about capturing incoming traffic also posed a problem. Initially, the idea was to use a similar approach as the IDPS concept [AUT20] from AUTOSAR. However, while this concept is able to detect all incoming traffic, it only works on copies of the incoming messages. The actual messages can not be blocked this way. Instead, the netfilter system was used to capture incoming traffic and set up firewall rules.

While the prototype itself does not have any resource constraints the actual implementation will. Therefore it makes sense to build the prototype in an environment that is suitable for the actual firewall if that is possible.

4.1 Specification

The specification for the configuration file is described here. An XML schema based on this specification has also been created as part of this thesis and will be provided in the appendix 7. The specification detailed here is an adaptation of the official AUTOSAR specification for the firewall

concept. Things that were deemed out of scope for this thesis, such as protocol support for additional application level protocols were left out here. However, some functionality has also been simplified, such as being able to configure the modes for the firewall through a designated field.

The configuration file should be an arxml file, just like the AUTOSAR Manifests, so that it is compliant with the AUTOSAR configuration specification. The general idea about what rules should be configurable was provided by the concept owner. It has been adapted to fit the scope of this prototype. All firewall rules should be defined in this single document.

The firewall should be mode dependent. This means that depending on the current vehicle state or mode there can be different firewall rulesets. Changing from one ruleset to another should be possible during runtime and should not allow the system to accept all incoming messages while switching modes. This could happen if the default action for the new mode is to accept all incoming traffic and adding the new ruleset would not be atomic such that some communication can happen before the rules have been added.

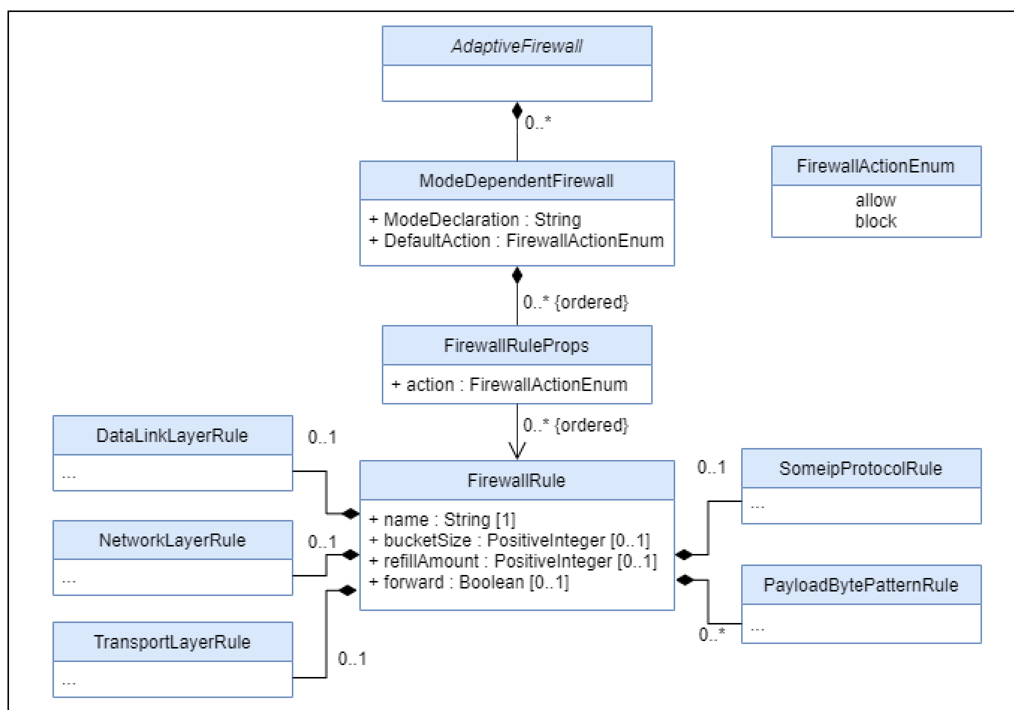

Figure 4.1: The UML diagram shows the architecture of the firewall configuration file. The protocol and layer subrules of the FirewallRule each contain header fields of the related protocols.

Figure 4.1 shows the structure of the firewall configuration. There is one top level element that contains all sub elements, the AdaptiveFirewall element. It can contain any number of ModeDependentFirewall elements. They are the main firewall elements that contain the underlying ruleset. They contain, additionally to the FirewallRuleProps subelement, information about the current mode and the default action to be taken if no rules match. This means that the firewall can load a different ruleset depending on the current mode the vehicle is in. If it is in maintenance mode for example, the related mode dependent firewall rules for the maintenance mode should be loaded. This allows adapting rulesets to different situations where the communication requirements might change.

The FirewallRuleProps elements are a collection of ordered elements that contain an action to be taken in case of a rule match and an ordered list of FirewallRule elements. Each FirewallRule contains each of its sub elements either one or zero times. Both the FirewallRuleProps and the FirewallRule elements are ordered and should be parsed in the given order. This allows control over the order rules are added to the firewall in. This is important in certain cases where rules that affect many packets are added as they should be checked first for performance reasons. If both, rules that accept and rules that drop messages, are part of the firewall the order they are processed in is important as well, especially if rules from both categories can affect the same packet.

Information about the bucketSize, refillAmount and the name of the rule can be provided. Both, the refillAmount and bucketSize, refer to variables used when implementing a leaky bucket algorithm. They are meant to configure congestion control for incoming packets. The elements named after OSI layers can be used to configure rule parts related to their respective network layer. The SomeipProtocol element is meant to configure rules regarding the SomeIP protocol and is an application level rule. The PayloadBytePatternRule should allow filtering of arbitrary bytes.

To have a matching FirewallRule all provided sub rules must match. For example, if we have a rule with a datalink layer rule that specifies the ethernet type and a network layer rule that specifies a certain Ipv4 address then the rule will only match incoming packets if both the ethernet type and the ip is the same as in the rule. In case of a match the action set in the FirewallRuleProps element will be taken.

One attribute of a FirewallRule element is called forward. It is meant to determine whether this rule addresses messages meant for the local host or if they are meant for messages that will be forwarded further. This attribute could have also been part of the parent element, the FirewallRuleProps to further distinguish rules that would affect locally targeted traffic from forwarded traffic. It was added to the FirewallRule element as to keep the FirewallRuleProps element as simple as possible. It was also decided to have the forward chain mimic the default decision of the ModeDependentFirewall. Alternatively it could have been set to always forward any traffic meant to be forwarded, as it was the case before the forward functionality was added to the POC. This would keep firewall configurations simpler in environments where this is the case by default, however, it was deemed unintuitive to have a firewall that is configured as drop by default but allows any incoming messages to be forwarded.

The content of the individual sub-rules that has been omitted in figure 4.1 is going to be explained briefly in the next subsections.

Datalink Layer Rules

The datalink layer rules deal with stateless filtering of datalink level headers:

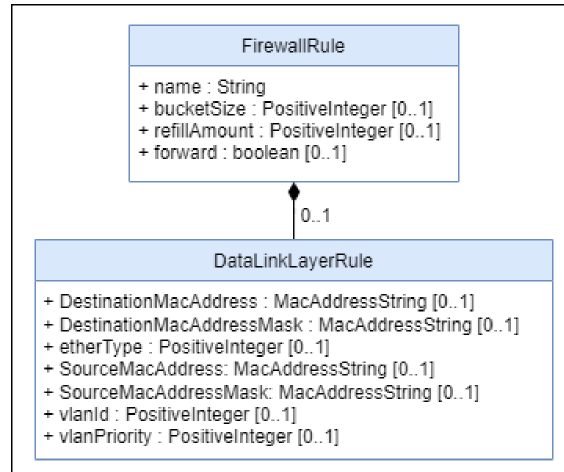

Figure 4.2: Contents of datalink layer rules.

Most of these options are supported by default by nftables. However, the mac address masks are also provided in the default mac address string format of MM:MM:MM:MM:MM:MM where each MM represents one byte. A mask of this format would contain full bytes (FF) for the fields of the address that should be considered. A mask of the format FF:FF:00:00:00:00 would cause only the first two octets of the address to be filtered against.

To properly add this rule to the nftables configuration a suffix of the form "/number" can be used where the number defines the amount of bits to mask from left to right. Therefore the internal call needs to be adapted accordingly

Network Layer Rules

A network layer rule can either be an IPv4 or an IPv6 rule. Depending on which protocol the rule should filter either of the two are chosen. In both cases the rules will have an ICMP subrule to configure internet control message protocol related rules. This protocol deals with error and notification messages for both IPv4 and IPv6.

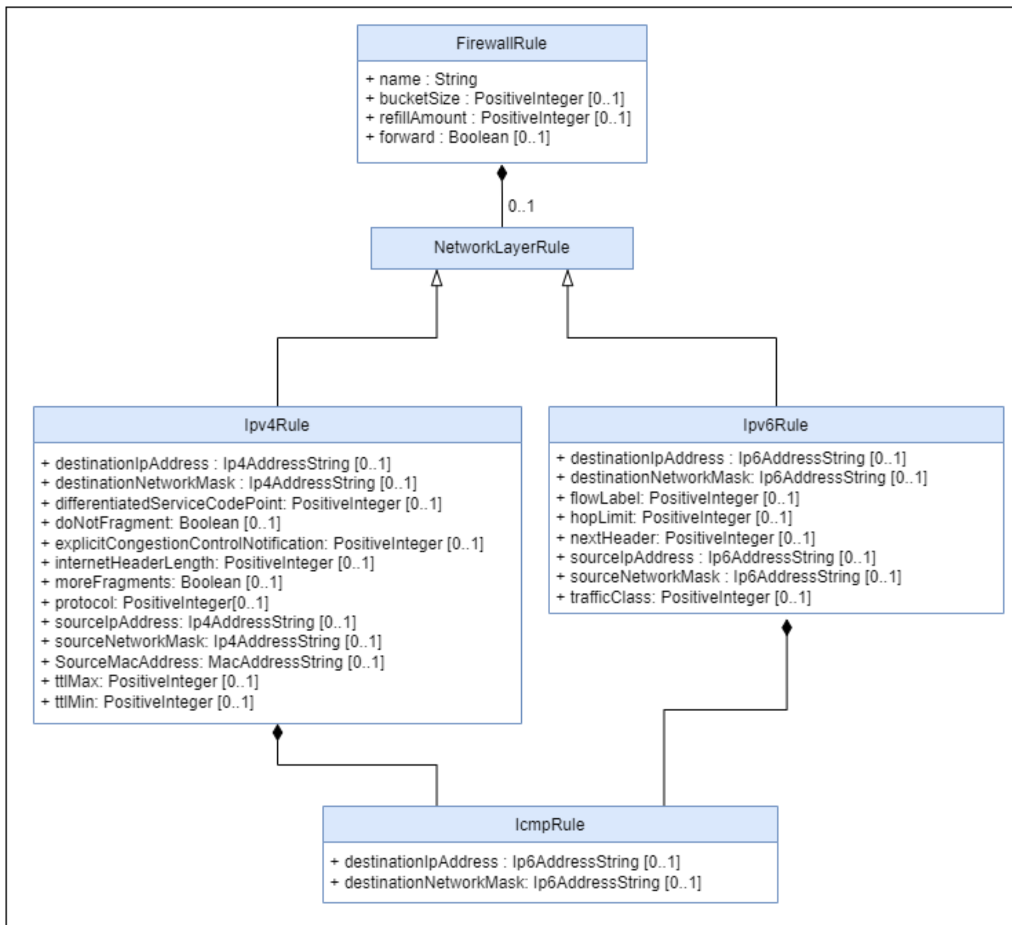

Figure 4.3: Contents of network layer rules.

As shown in figure 4.3 the headers that can be filtered depend on what protocol is being used for the rule. The masks are, as with the data link layer rules, in the same format as the addresses. For IPv4 the address format consists of four bytes concatenated with a dot between each byte. The mask is filled from left to right to show what parts of the original address should be considered and which should be ignored. As an example the address 127.128.168.65 masked with 255.255.0.0 will result in filtering any address that starts with 127.128.

IPv6 addresses and masks work similarly. The Ip6AddressString type consists of 128 bits. Every 16 bits are divided by a colon. Trailing zeroes can be omitted by adding two colons. The address mask has to have 1-bits on parts of the address that should be static. Masking the address 3244:4535:1233:FE12:2345:1234:64AB:ABBA with the mask FFFF:FFFF:: results in the masked address 3244:4535::/32. The “/32” shows that only the first 32 bits of the address should be considered when filtering.

Transport Layer Rules

The transport layer rules are split into two parts, much like the network layer rules. However there is a base set of headers that can be filtered. Then, depending on whether the transport layer rule is a UDP or TCP rule, additional fields will be added. The common headers that can be filtered for both protocols, the source and destination ports, are part of stateless filtering. If the transport layer rule is defined as a TCP rule however, additional fields for stateful filtering become available.

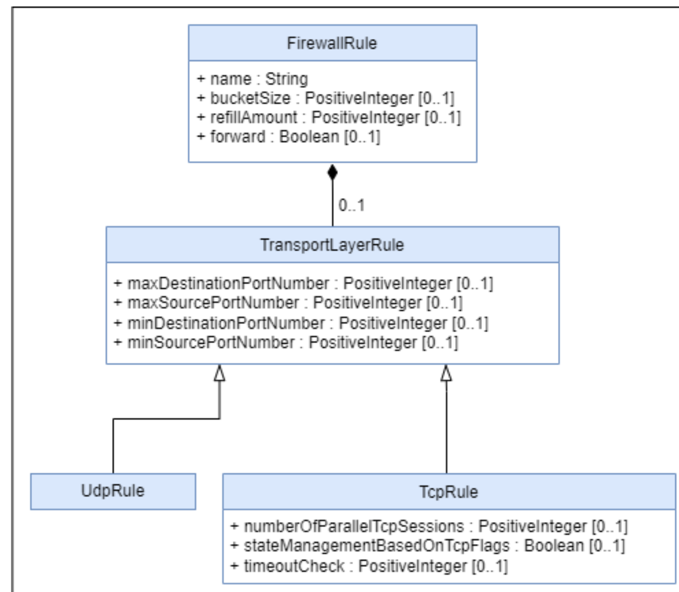

Figure 4.4: Contents of transport layer rules.

The first field in the TcpRule, numberOfParallelTcpSessions, sets the amount of TCP sessions that can be established at the same time. This means that once the set amount of connections has been established any messages trying to establish a new TCP connection should be dropped. The timeoutCheck field should specify the default TCP session timeout in seconds. If the number of parallel TCP sessions is limited then it makes sense to also decrease the default session timeout such that idle connections will not block new connections from being established. The state management option determines whether the connection tracking system considers TCP flags when classifying connections.

Payload Byte Pattern Rules

The goal of the payload byte pattern rules is to be able to filter on arbitrary byte values.

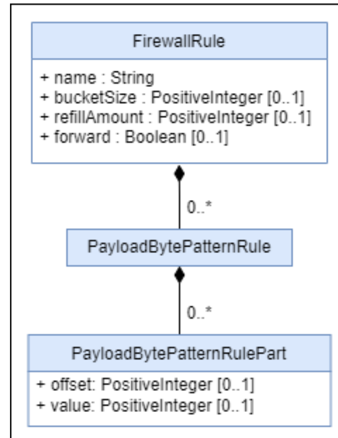

Figure 4.5: Contents of the payload byte pattern rules.

A single payload byte pattern rule consists of a set of payload byte pattern rule parts. Each of those parts has the value that we want to filter and the offset where that value is expected. This should allow filtering of arbitrary fields if the protocol and offset is known.

SOME/IP Rules

The SOME/IP protocol rules allow filtering of headers of the standard SOME/IP protocol.

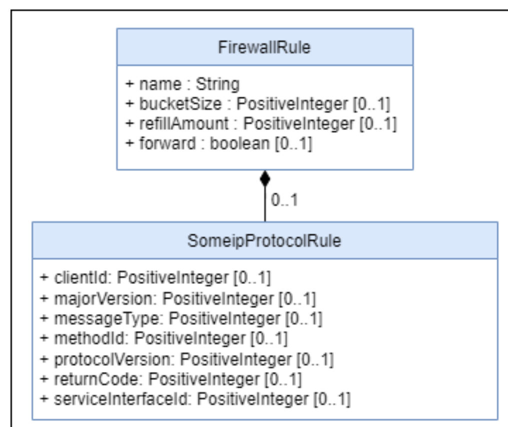

Figure 4.6: Contents of the SOME/IP protocol rules.

This is able to filter both the regular SOME/IP protocol header as well as the header of a SOME/IP protocol running in end-to-end encryption (E2E) mode. In this mode the original header layout is preserved, additional information for the E2E mode is appended to the original header before the payload. As the encryption only affects the payload the header fields can still be accessed as usual.

4.2 Implementation

This part of the thesis presents the actual implementation and inner workings of the prototype. Its general task is to parse the configuration file specified in section 4.1. The general idea of the firewall concept would be to have it running as a daemon. The POC, however, runs as a standalone and execution blocking program as of now. As it is just a proof of concept this should suffice to check whether it can parse the configuration file and set up the specified firewall rules properly. Subsequently transferring this functionality to a standalone daemon should be an easy task.

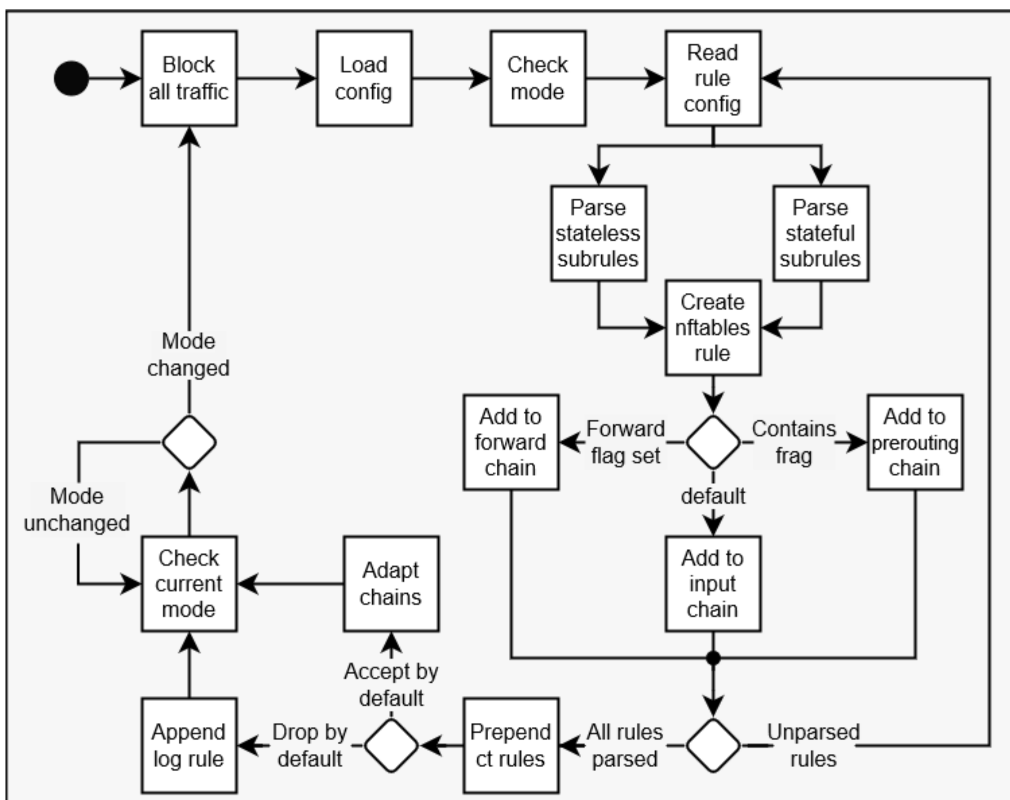

Figure 4.7: Activity diagram of the POC that shows the general program flow.

Figure 4.7 shows the program flow of the proof of concept. The ideas behind the individual stages of the prototype as well as more information regarding them will be presented in this section as well.

The very first action the POC takes after being started is that all incoming traffic is blocked. The reason behind this is to block all potentially malicious or unwanted messages as soon as there is a need for a firewall. Since it takes time to properly parse the configuration file and to subsequently set up the related firewall rules there is a brief window where the need for the firewall is known while no rules have been added to the firewall yet. Also, if legitimate messages get dropped during firewall setup they should be able to be resent once the firewall is operational.

Along with blocking all incoming packets the firewall sets up a set of nftable chains. Those are meant to be populated by the actual firewall rules once the config file is getting parsed. It does this by loading a predefined base configuration that sets up all chains as needed.

baseConf.conf

```
flush ruleset

table inet filter {
  chain input {
    type filter hook input priority 0; policy drop;
  }
  chain output {
    type filter hook output priority 0; policy accept;
  }
  chain prerouting {
    type filter hook prerouting priority 0;
  }
  chain forward {
    type filter hook forward priority -450; policy drop;
  }
}
```

This ruleset can be loaded with the nftables utility, loading external configurations is supported by the command-line interface (CLI) with the *-f* flag. It loads the provided ruleset as a whole. The POC does this internally via a system call and uses the nftables utility. The related command to load the config is: *sudo nft -f baseConf.conf*.

Next, the POC loads the configuration file. As it is an *.arxml* file it can be read by standard XML parsers. For the prototype RapidXml was chosen to parse the config. It was chosen because it is a fast and stable xml parser [Kal09]. It is also compatible with C++, which makes it easy to include into the POC.

After the configuration file has been loaded the mode of the vehicle is checked. This currently happens to a dummy call that always returns the default mode. For the actual Firewall concept this functionality has to be provided over an API. A manufacturer implementing the AUTOSAR Adaptive platform would need to provide the current vehicle mode via this API. As there is currently no standardized set of vehicle modes nor the option to globally set a vehicle mode this would have to be implemented by the end user.

All ModeDependentFirewall rules are then checked. The first one to be associated with the mode element matching the current mode is then parsed further to get the current firewall configuration. The first rule of the first FirewallRuleProps element is then loaded via RapidXml. Both its stateless and stateful subrules get loaded to create a single nftables rule along with the action defined in the FirewallRuleProps element.

What chain of the base configuration the rule gets added to, depends on two things. First, if the forward flag is set as part of the FirewallRule element the parsed rule gets added to the forward chain. Subsequently this rule only affects messages passing through the forward hook. If any fragmentation information is part of the rule then it will be added to the prerouting chain. This chain will affect both local and forwarded messages as the prerouting hook is traversed by both. If neither are the case the rule gets added to the default chain.

Once all rules that are part of this ModeDependentFirewall have been parsed and added to their respective chains, basic connection tracking rules will be prepended to the input chain. There are two connection tracking rules that will get added in this step. The first one is *ct state invalid drop*.

This rule drops any incoming messages that are deemed to be invalid. The rule *ct state established, related accept* accepts any incoming messages that are deemed to belong to an existing connection or are related to an existing one. The actions taken for these rules will stay the same, independently of the actual firewall configuration. They will be inserted before any other rules are checked as this allows immediate filtering of messages that belong to an existing connection which will increase performance, if a lot of messages that are exchanged belong to existing connections.

Next, the chains are adapted in case accept is set as the default action. Since the base configuration has the chains set to drop by default, this has to be changed. In cases where drop remains the default action, a single log statement is added to the input table to log all messages that did not match any rules and will subsequently be dropped.

This concludes setup of the firewall ruleset. Now the POC polls for state changes of the vehicle. If one is detected the whole program flow is repeated and the new firewall configuration is loaded. This could also be implemented by a callback or interrupt instead of having the program poll actively.

4.3 Parsing rules

Parsing the firewall configuration and creating a single nftables rule from one FirewallRule element can be done by using variables provided by nftables for supported protocols. The nftables utility supports all low level protocols the POC was designed to filter. Therefore it is easy to add the stateless subrules to the firewall configuration in most cases. The rule of the configuration file gets traversed and each subrule added to a single string. This string is then passed to the nftables CLI by using a system call.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<AdaptiveFirewall>
    <ModeDependentFirewall defaultAction="allow">
        <ModeDeclaration>default</ModeDeclaration>
        <FirewallRuleProps action="block">
            <FirewallRule name="example-rule">
                <NetworkLayerRule>
                    <Ipv4Rule>
                        <sourceIpAddress>127.25.126.168</sourceIpAddress>
                        <sourceNetworkMask>255.255.0.0</sourceNetworkMask>
                        <protocol>1</protocol>
                    </Ipv4Rule>
                </NetworkLayerRule>
            </FirewallRule>
        </FirewallRuleProps>
    </ModeDependentFirewall>
</AdaptiveFirewall>
```

Figure 4.8: Example ruleset for one IPv4 rule.

The related system string the POC would generate for the example ruleset 4.8 would be:
ip saddr 127.25.126.168/16 ip protocol 1 comment example-rule
The generated string will then be added to the nftables configuration with the following system call:
system(sudo nft add rule inet filter input ip saddr 127.25.126.168/16 ip protocol 1 log drop comment example-rule)

This way all subrules specified in 4.8 will have to match an incoming packet for the specified action to be taken.

Even though nftables supports all low level protocols the POC needs, creating the system string needs to transform the input of the configuration file into an nftables compatible format. For example, as seen in the configuration file 4.8, the network mask is provided as an IPv4 string. The nftables utility uses another notation when adding mac, IPv4 and IPv6 address masks. Therefore the internal representation of the address mask needs to be transformed into the one used by nftables.

Adding fragmentation rules

While nftables provides functionality to filter low level headers, it does not provide any direct access to the fragmentation control flags of the IPv4 protocol. Both the “do not fragment” and “more fragments” flags do not come with their own handle.

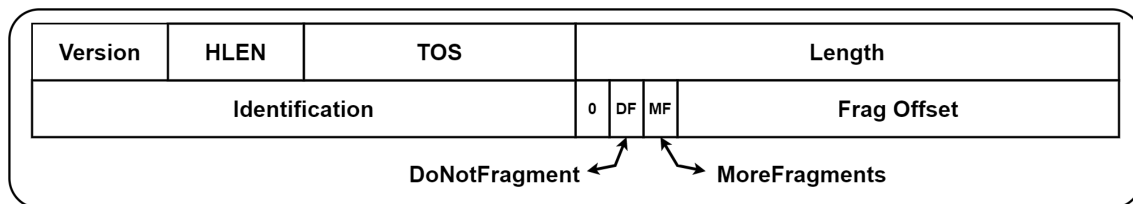

Figure 4.9: Abbreviated IPv4 header.

As seen in figure 4.9, the fragmentation control flags are set before the actual fragmentation offset value. The bit before the two flags is a zero-bit and is currently unused.

The problem, when trying to filter for these fragmentation control flags, is that nftables does not provide any direct access to them. Instead, they are grouped together with the fragmentation offset value. Using the designated “frag-offset” handle, a 16-bit string would be returned, starting with the 0 bit, then both flags and lastly the actual fragmentation offset value.

The following rule would fetch this whole 16-bit string and compare it to a value:

ip frag-off <value>

Bitwise operations can then be added to the frag-off notation to access parts of the 16-bit string. Accessing the fragmentation control flags can then be done in the following way:

do not fragment flag: ip frag-off & 0x4000 != <value>
more fragments flag: ip frag-off & 0x2000 != <value>

In this case, if one of the fragment flags is set, the bitwise operation would return the value 0x4000 or 0x2000 respectively. If we want to check if either flag is set the value specified in the <*value*> field should be 0.

To check for the actual fragmentation offset the frag-off value needs to be transformed like this:

fragment offset: ip frag-off & 0x1FFF <value>

With this command the <*value*> field should contain the actual offset that should be matched with the rule.

Any rules that contain fragmentation related information, such as the fragmentation control headers or the actual fragmentation offset, need to be added to the prerouting chain of the base configuration. This is needed since the “nf_defrag_ipv4” netfilter module reassembles packets at the prerouting level if it is enabled. Packet reassembly is generally needed for proper connection tracking and should be enabled for the prototype.

As this way of accessing the fragmentation control flags is not directly supported by nftables this way of accessing them was tested thoroughly. The hping3 [Lim22] tool was used to send fragmented IPv4 packets. The rules were then added to the nftables configuration with counters to check the amount of packets that match each of the rules.

Adding SOME/IP rules

Two things netfilter does not support is filtering of both high-level protocols, protocols that are transported over the application layer, and proprietary protocols. As SOME/IP fits both of those categories, being an application layer protocol and having been specifically developed for AUTOSAR, it is not supported by nftables by default. Additionally, the netfilter subsystem, and subsequently nftables, does not support actual Deep Packet Inspection. There is no way to check for sequences of the same command, for example.

Instead we aim to perform shallow packet inspection for SOME/IP. With this, the header values of the SOME/IP protocol can be used to filter messages.

Since there is no SOME/IP protocol support by nftables the header values need to be accessed in a different manner. We assume for this that SOME/IP communication takes place over UDP exclusively and that the SOME/IP header is not encrypted. While the first assumption might not be strictly true, the following method can be adapted for the TCP protocol as well. Furthermore SOME/IP communication generally takes place over UDP, and all SOME/IP communication can be sent over UDP if required.

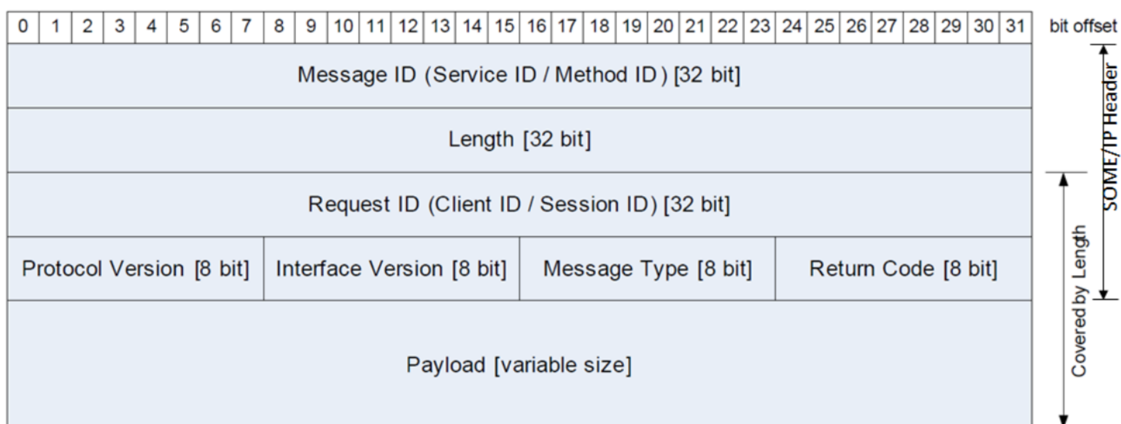

Figure 4.10: Standard SOME/IP header format including all required fields [AUT21e].

To actually filter the SOME/IP header fields the “@th” token, provided by nftables, can be used. This token allows pointing to the beginning of the transport header, defining an offset and the length we want to read. Since the SOME/IP header starts right after the UDP header ends we can use this

token to point to the SOME/IP header field we want to filter. The offset used needs to be the offset of the header field that should be filtered added to the length of the UDP header. The UDP header has a standardized length of 64 bit [sam21].

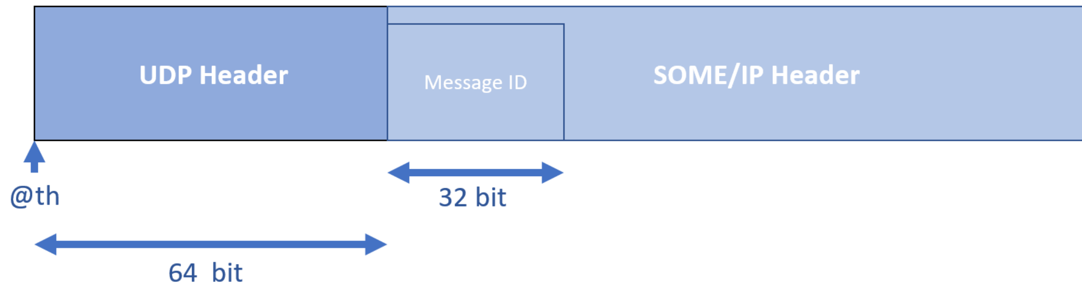

Figure 4.11: Offset and length of the message ID field in the SOME/IP header.

Figure 4.11 shows how the message ID of a SOME/IP message can be accessed. An example rule that would filter for this message ID would look like this:

ip protocol udp @th,64,32 <value>

Additionally, it is checked whether UDP is used as an underlying protocol. As the SOME/IP protocol is not supported by default there is no in-built functionality to check if the SOME/IP protocol is actually in use. To check if an incoming message does contain a SOME/IP part, it could be checked if a valid Message Type was provided.

The first attempt for checking for SOME/IP messages was to verify that the supposed length field actually corresponds to the length of the SOME/IP message. However, such functionality is not provided by nftables. The length value could be read using an @th token with the related offset and length, but we cannot check this value against another.

When filtering other header fields both the offset and the length of the field need to be adapted accordingly:

Header Field	Offset	Length	Rule
Service ID	64	16	@th,64,16
Method ID	80	16	@th,80,16
Length	96	32	@th,96,32
Client ID	128	16	@th,128,16
Protocol Version	160	8	@th,160,8
Interface Version	168	8	@th,168,8
Message Type	176	8	@th,176,8
Return Code	184	8	@th,184,8

Accessing the individual SOME/IP header fields like this had to be tested. As the POC was developed and deployed on a Linux virtual machine, actually sending valid SOME/IP messages posed a problem. Since there is no command line utility that sends proper SOME/IP messages, a custom script was used. It is shown in appendix 7. This tool allowed sending arbitrary messages

over UDP. SOME/IP messages were constructed as defined in the official documentation [AUT21e]. Firewall rules were then set up to only allow messages with certain SOME/IP header field values to pass through the host firewall. Additionally the netcat utility [ubn18] was used to listen to incoming connections. As a result, only messages with the defined header values were able to be received by netcat, all other incoming UDP messages were dropped.

Adding payload byte pattern rules

The payload byte pattern rules are supposed to allow filtering of arbitrary byte values. The approach to implementing this is similar to the SOME/IP approach. Instead of using the “@th” token the “@ll” token is used. It points to the beginning of the datalink layer header.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<AdaptiveFirewall>
    <ModeDependentFirewall defaultAction="allow">
        <ModeDeclaration>default</ModeDeclaration>
        <FirewallRuleProps action="block">
            <FirewallRule name="example-rule">
                <PayloadBytePatternRule>
                    <PayloadBytePatternRulePart>
                        <offset>0</offset>
                        <value>255</value>
                    </PayloadBytePatternRulePart>
                    <PayloadBytePatternRulePart>
                        <offset>8</offset>
                        <value>123</value>
                    </PayloadBytePatternRulePart>
                </PayloadBytePatternRule>
            </FirewallRule>
        </FirewallRuleProps>
    </ModeDependentFirewall>
</AdaptiveFirewall>
```

Figure 4.12: An example payload byte pattern rule.

The example ruleset 4.12 consists of one payload byte pattern rule with two rule parts. The internal call of the POC would transform this configuration into the following call:

sudo nft add rule inet filter input @ll,0,8 255 @ll,8,8 123 log drop comment example-rule

The actual nftables rule will look a little different since it merges multiple rule parts into one if the “@ll” or “@th” tokens are used and the fields to be accessed are next to each other. Instead of comparing the two payload byte pattern rule parts separately a single comparison over 16 bits is added to the firewall:

@ll,0,16 65403 log drop comment ”example-rule”

The same thing happens for the SOME/IP header fields if they are next to each other.

4.4 Ruleset example

To create a proper firewall implementation that not only secures the system against adversaries, but also allows all desired communication to still take place, the exact communication needs of the host need to be known. For machines running the AUTOSAR Adaptive platform this information can

generally be extracted from the related machine manifest. The machine manifest would specify all information regarding needed ports, broadcast addresses and the address of the machine itself. However, a general approach was taken for the example ruleset instead, since actual machine manifests can contain sensitive information.

In general, communication between AUTOSAR Adaptive machines is done over SOME/IP. This protocol, as presented in section 2.3 almost exclusively uses UDP for communication. It is also possible that long messages are sent over TCP instead, but this is generally not the case. Therefore a set of UDP ports needs to be opened to allow proper SOME/IP communication while also opening at least one TCP port in case a TCP connection needs to be established. Additionally, a port for SOME/IP service discovery needs to be provided for the SOME/IP protocol to work properly. Available services and functions on the network need to be known for machines to use them properly. Another UDP port for general notifications should also accept incoming messages.

The general policy of the firewall should be, as stated previously, to drop all packets that do not match any rules.

The exact port numbers and IP addresses for this example are meant to illustrate the setup of firewall rules and are not meant to actually be used in a production environment.

For ports that tend to expect less traffic, such as the TCP port and the UDP port meant for notifications the bandwidth is capped at 10 packets per second with a 10 packet overflow. This is meant to limit the amount of incoming data on ports where little data is expected such that suspiciously high traffic will get blocked.

For the TCP port the amount of parallel connections will also be limited. In this example no more than three TCP connections can exist at the same time. The session timeout for the TCP protocol has been reduced as well. A timeout of two minutes should be enough in an automotive context.

Traffic that is meant for the same subnet the machine is in should also be forwarded. The forward chain gets configured accordingly, assuming the machine is part of the *127.168.123.0/24* subnet.

These rules are part of the *default* mode which is meant for normal operation. Another mode *maintenance* has been provided where only the TCP port used for SSH is open. This allows access to the machine over SSH for maintenance and debugging. Depending on what kind of maintenance is needed it makes sense to also add the default rules to this mode.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<AdaptiveFirewall>
    <ModeDependentFirewall defaultAction="block">
        <ModeDeclaration>default</ModeDeclaration>
        <FirewallRuleProps action="allow">
            <FirewallRule name="Notif-Port-25255">
                <refillAmount>10</refillAmount>
                <bucketSize>10</bucketSize>
                <TransportLayerRule>
                    <UdpRule>
                        <minDestinationPortNumber>25255</minDestinationPortNumber>
                        <maxDestinationPortNumber>25255</maxDestinationPortNumber>
                    </UdpRule>
                </TransportLayerRule>
            </FirewallRule>
            <FirewallRule name="UDP-Port-5050-5052">
                <TransportLayerRule>
                    <UdpRule>
                        <minDestinationPortNumber>5050</minDestinationPortNumber>
                            <maxDestinationPortNumber>5052</maxDestinationPortNumber>
                    </UdpRule>
                </TransportLayerRule>
            </FireWallRule>
            <FirewallRule name="SD-Port-30405">
                <TransportLayerRule>
                    <UdpRule>
                        <minDestinationPortNumber>30405</minDestinationPortNumber>
                            <maxDestinationPortNumber>30405</maxDestinationPortNumber>
                    </UdpRule>
                </TransportLayerRule>
            </FireWallRule>
            <FirewallRule name="TCP-Port-5050">
                        <refillAmount>10</refillAmount>
                <bucketSize>10</bucketSize>
                <TransportLayerRule>
                    <TcpRule>
                        <minDestinationPortNumber>5050</minDestinationPortNumber>
                        <maxDestinationPortNumber>5050</maxDestinationPortNumber>
                        <numberOfParallelTcpSessions>3</numberOfParallelTcpSessions>
                        <timeoutCheck>120</timeoutCheck>
                    </TcpRule>
                </TransportLayerRule>
            </FireWallRule>
            <FirewallRule name="Local-Forward" forward="true">
                <NetworkLayerRule>
                    <Ipv4Rule>
                        <destinationIpAddress>127.168.123.0</destinationIpAddress>
                        <destinationNetworkMask>255.255.255.0</destinationNetworkMask>
                    </Ipv4Rule>
                </NetworkLayerRule>
            </FirewallRule>
        </FirewallRuleProps>
    </ModeDependentFirewall>
    <ModeDependentFirewall defaultAction="block">
        <ModeDeclaration>maintenance</ModeDeclaration>
        <FirewallRuleProps action="allow">
            <FirewallRule name="SSH-Port">
                <TransportLayerRule>
                    <TcpRule>
                        <minDestinationPortNumber>22</minDestinationPortNumber>
                            <maxDestinationPortNumber>22</maxDestinationPortNumber>
                    </TcpRule>
                </TransportLayerRule>
            </FireWallRule>
        </FirewallRuleProps>
    </ModeDependentFirewall>
</AdaptiveFirewall>
```

Figure 4.13: Example ruleset for an AUTOSAR Adaptive machine.

```
table inet filter {
        ct timeout input-timeout {
                protocol tcp
                l3proto inet
                policy = { established : 120 }
        }

        chain input {
                type filter hook input priority filter; policy drop;
                ct state invalid log drop
                ct state established,related accept
                limit rate 10/second burst 10 packets udp dport 25255 accept comment "Notif-Port-25255"
                udp dport 5050-5052 accept comment "UDP-Port-5050-5052"
                udp dport 30405 accept comment "SD-Port-30405"
                ip protocol tcp ct count over 3  log drop
                limit rate 10/second burst 10 packets tcp dport 5050 accept comment "TCP-Port-5050"
        }

        chain output {
                type filter hook output priority filter; policy accept;
                ct timeout set "input-timeout"
        }

        chain prerouting {
                type filter hook prerouting priority raw; policy accept;
        }

        chain forward {
                type filter hook forward priority filter; policy drop;
                ip daddr 127.168.123.0/24 accept comment "Local-Forward"
        }
}
```

Figure 4.14: Resulting nftables configuration when the example ruleset is parsed by the program.

As no fragmentation control was defined, the prerouting chain remains empty. The forward chain contains the rule allowing forwarding of messages to machines in the same subnet. The custom TCP session timeout is set in the output chain. This sets the TCP session timeout for all TCP connections.

Network separation example

The prototype can also be used for network separation. In this case a special flag needs to be set for the FirewallRule element that specifies that this rule is meant for the forward chain of nftables. This was already done in the last example to allow forwarding to local machines, however a setup for proper network separation looks differently.

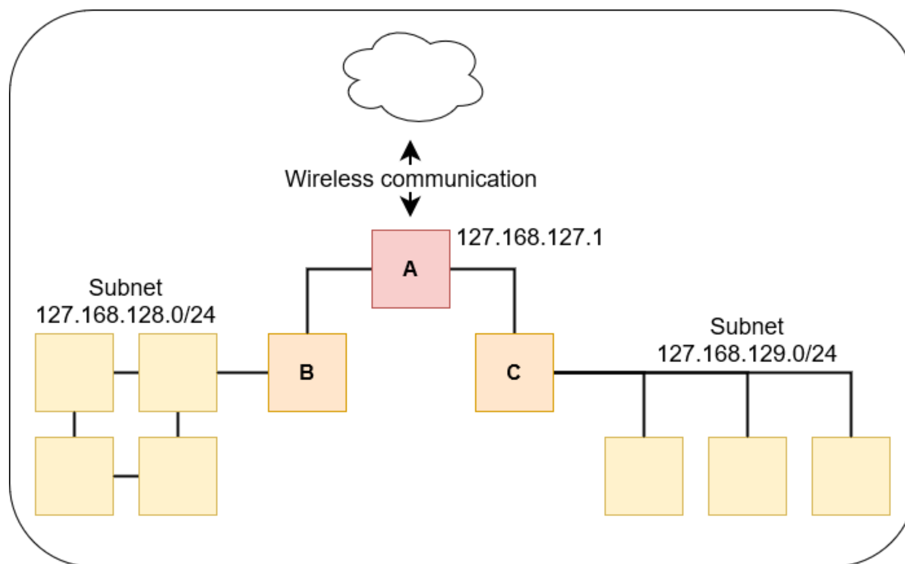

Figure 4.15: Each rectangle represents a single ECU. The ones in orange are domain gateways while the one in red is the central gateway that is also able to communicate with an outside server over the internet.

For this part the example architecture given in figure 4.15 is considered. Two subnets and three gateways are part of the architecture. Additionally, one of the gateways is able to communicate with the outside world to receive e.g. updates or additional information. In this setup we assume that the two subnets will not have to communicate with each other at all. Instead, most communication happens within one subnet with occasional updates to and from the central gateway *A*. How this can be set up was already shown in the previous example. The goal of network separation in this case is to prevent direct access from one subnet to the other. To achieve this, the following configuration is used on gateway *B*:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<AdaptiveFirewall>
    <ModeDependentFirewall defaultAction="block">
        <ModeDeclaration>default</ModeDeclaration>
        <FirewallRuleProps action="allow">
            <FirewallRule name="Subnet-127.168.128.0/24-Ingress" forward="true">
                <NetworkLayerRule>
                    <Ipv4Rule>
                        <destinationIpAddress>127.168.128.0</destinationIpAddress>
                        <destinationNetworkMask>255.255.255.0</destinationNetworkMask>
                        <sourceIpAddress>127.168.127.1</sourceIpAddress>
                    </Ipv4Rule>
                </NetworkLayerRule>
            </FirewallRule>
            <FirewallRule name="Subnet-127.168.128.0/24-Egress" forward="true">
                <NetworkLayerRule>
                    <Ipv4Rule>
                        <sourceIpAddress>127.168.128.0</sourceIpAddress>
                        <sourceNetworkMask>255.255.255.0</sourceNetworkMask>
                        <destinationIpAddress>127.168.127.1</destinationIpAddress>
                    </Ipv4Rule>
                </NetworkLayerRule>
            </FirewallRule>
        </FirewallRuleProps>
    </ModeDependentFirewall>
</AdaptiveFirewall>
```

Figure 4.16: Example configuration for network separation.

This configuration has one rule for traffic from gateway *A* to reach the *127.168.128.0/24* subnet and one for traffic from said subnet back to gateway *A*. A similar configuration, with updated subnet IP, needs to be deployed on gateway *C* as well. This prevents any direct communication between both subnets. They will only be able to communicate within the subnet and with gateway *A*.

```
table inet filter {
        chain input {
                type filter hook input priority filter; policy drop;
                ct state invalid log drop
                ct state established,related accept
        }

        chain output {
                type filter hook output priority filter; policy accept;
        }

        chain prerouting {
                type filter hook prerouting priority raw; policy accept;
        }

        chain forward {
                type filter hook forward priority filter; policy drop;
                ip saddr 127.168.127.1 ip daddr 127.168.128.0/24 accept comment "Subnet-127.168.128.0/24-Ingress"
                ip saddr 127.168.128.0/24 ip daddr 127.168.127.1 accept comment "Subnet-127.168.128.0/24-Egress"
        }
}
```

Figure 4.17: Resulting nftables configuration for the network separation example ruleset.

All rules are part of the forward chain. No traffic will be accepted on the gateways as they are only meant to relay messages.

5 IAM

Identity and Access Management (IAM) allows to manage both, authentication of users and authorizing access to sensitive data, among other things. This kind of service has become increasingly important with the ubiquitous use of cloud systems nowadays. As companies outsource more and more services, users need to remember an ever increasing amount of credentials to securely access the companies cloud resources. IAM aims to change this by providing single-sign on (SSO) [al15]. SSO allows logging into all of your accounts through a single master password [H R05]. Additionally more information can be factored into what kind of data you are allowed to access, like geographic location, source network, history or other factors. This allows a fine grained configuration of access rights for individual users even though one password is in use only. However multi-factor authentication is also required to increase security by IAM. Just like with firewalls, policies can be created that determine what privileges a user has and how user accounts get provisioned [al15].

The Identity Management part of IAM allows users to prove their identity, while Access Management is focused on providing authentication and authorization. The access management part also deals with policy changes and management. There are five main services that IAM provides. They are:

- Identity Management: The main focus of this service is to create a digital identity for every new account a user should have access to. Using this digital ID the user can then access data he has permission for. In line with this, the digital identity should also be removed if the user loses the right to access the resources for any reason, e.g. quitting their job at the company.
- Authorization Management: This service uses a policy to decide whether a certain user has the necessary rights to access a resource. It assigns the necessary privileges to a user depending on the roles related to their account.
- Authentication: The authentication services main objective is, as the name implies, the authentication of users. With IAM this requires multi-factor authentication along with a conventional password. However, the additional token that needs to be provided via a multi-factor authentication method is something the user does not have to memorize. Generally the token is generated and sent to one of the users devices or something the user possesses like an ID card or their fingerprint. Additionally, if risk based authentication is used, more information about the connection is used to determine the risk of allowing the client to access the requested resource. The transaction can then be approved or dropped depending on the calculated risk level.
- Identity Federation: This allows the delegation of authentication by a trusted identity provider to other services, instead of signing in via username and password. When trying to log into a new service the identity provider authenticates the user with a token by using an identity protocol. Common protocols for this use case are OAuth or OpenID.
- Compliance Management: This service deals with auditing, logging and monitoring of any records to ensure compliance with existing security standards.

Along with these services the user management is also part of IAM. This includes creation and deletion of user credentials as well as maintaining and updating them if necessary [al15].

While traditional IAM is meant to deal with user access and credentials the concept can be adapted to other use cases as well. Services like remote IAM [AUT21c] or SCREIAM are adaptations of the traditional IAM concept. They aim to provide an IAM system for individual ECUs to prevent them from accessing resources beyond their intended functionality by using identifiers of the source ECU. SCREIAM was developed for the AUTOSAR Systems for this use case [AUT21a].

5.1 Proof-of-concept comparison

To compare the firewall concept with an IAM concept the focus will be on the host firewall proof-of-concept presented in section 4 and a variation of IAM that is designed to work with ECUs as described in the AUTOSAR specification. The general idea of this section is to compare both concepts in relation to their similarities, advantages, disadvantages and differences to see which of the two concepts is better suited to secure ECUs.

Both concepts aim to do the same thing fundamentally. Both try to limit or prevent unauthorized or malicious access to certain resources. The difference is the way both concepts achieve this and on what level.

Generally, as presented in section 3, firewalls work on the lower end of the OSI model. They check up to the transport layer for simple firewalls while sophisticated ones manage to also filter application level protocols via Deep Packet Inspection. However, the proof-of-concept presented in section 4 only does shallow packet inspection. It does not look at the payload of the application layer protocols and instead just checks the headers. While this does allow some control over the contents of high level protocols the payload of those protocols can not be filtered by the firewall. Additionally, a firewall has full access to the original packet and is able to filter on many things that are no longer available as the packet passes through the network stack, such as low level information like virtual lan ID or ether type.

The IAM framework from AUTOSAR works with individual Adaptive Applications instead of authenticating full ECUs. Here, each individual application does have certain intents that are specified in the Application Manifest. With these intents an application shows which resources it should be allowed to access. Only if an application possesses all required intents for a certain AUTOSAR resource will access be granted. Additionally, general access to resources is forbidden by default. Only access to designated interfaces and well-defined resources is allowed, and only if the required intents are present for the requestor [AUT21c].

One of the main differences between the concepts is that for IAM the individual applications need to be aware that IAM is being used. They need to be configured accordingly in their respective manifest to provide them with the needed intents. If they are not configured in such a way every request to a resource implementing IAM will be denied [AUT21c]. For the firewall concept the sender does not have to be aware that a firewall is being used. With an ideal firewall implementation all legit communication is allowed to pass through the firewall. Only malicious, corrupted or invalid messages should be blocked.

An advantage of IAM is that it also authenticates the Adaptive Application that is requesting the resources. A requesting AA gets cryptographically signed to allow this [AUT21c]. The firewall concept does not authenticate other ECUs at all.

One of the key advantages of the firewall concept is that a firewall can validate incoming traffic and only allow traffic to reach the ECU that actually adheres to a certain protocol standard, e.g. only allowing SOME/IP messages to pass into the system. This helps to prevent attacks that use malformed protocol messages from ever reaching the system. As IAM only deals with messages that have already reached the system it will not be able to prevent such attacks. On the other hand IAM can allow certain messages from a single ECU while blocking others from the same source if the targeted resources need a different level of access [AUT21c]. This is not possible with the presented firewall prototype as Deep Packet Inspection is not supported to this extent.

In case of a denied request the incoming packet gets logged for the firewall concept while the IAM concept raises a notification. Both concepts can also remember certain access decisions. For the firewall concept this is the case for established connections. Once a connection has successfully been established it is allowed to pass through the firewall without having to re-check all the firewall rules. With IAM an Adaptive Application gets authenticated once when it is started initially. It is assumed that escalating privileges is not possible so any request from an already authenticated AA will be accepted without checking again. This should be ensured by the used runtime environment [AUT21c].

It is a challenge to compare the performance of both concepts as they are both implemented on different layers and in different ways. However, the IAM concept should have additional overhead every time an application's intents are checked, depending on how many intents the requested resource requires. The main overhead comes from getting the requestors app ID from the Execution Management FC and the related Application Manifest from the database to check the requestors intents [al15]. Since applications that have been checked once do not need to be re-checked, unless they are restarted, the overhead should remain small. With the firewall concept however every single incoming message will have to pass through the filter. This means that every message gets checked against every single rule until a match is found. With a recommended firewall setup that drops all messages, that did not match any rule, a message has to get checked against all firewall rules before it can be dropped. The resulting overhead applies to all incoming messages. The use of proper ordering of rules can mitigate the overhead to some extent. Rules that are most likely to match incoming messages, such as connection tracking rules for established connections, should be checked first. However in such a case blocked messages cause the biggest overhead meaning that a flood of messages that won't match any rule might be able to create a denial-of-service attack on the machine. This could be avoided with further denial-of-service detection functionality, but the prototype cannot determine or prevent denial-of-service attacks at the moment. The firewall rules also need to be set up when the daemon is launched initially, however the time this takes should be negligible as it only needs to do this once when the machine starts.

	Firewall	IAM
Overhead	Variable overhead, scales with checked rules.	Low overhead, intents for a new requestor need to be checked once.
Setup	Easy basic ruleset based on Machine Manifest.	Easy setup via intents, need to know which resources the application needs.
Reusabilty	General ruleset for every machine, minor adjustments as needed.	Intents configured once per application, high reusability.
Working level	Mostly low level network stack.	Application level.
Security	Restricts incoming messages by header information.	Restricts access to resources via intents.

This table summarizes the results of this section. Generally, the IAM concept has a lower overhead if the same applications keep communicating with each other, an easier setup and better reusability than the firewall concept. However, a firewall can protect against more attack vectors than a simple IAM system and is generally protocol independent as even higher level protocols like SOME/IP rely on low level protocols like TCP/UDP and IP for transport.

In conclusion, even though both concepts try to provide the similar things, they achieve it in a very different way. However, the concepts themselves do not interfere with each other as they target different things. The firewall concept monitors the network stack and all incoming traffic, while the IAM functionality is implemented in the applications themselves [AUT21c]. Therefore, IAM as well as a firewall can be used to create a multi-level security structure. Even if some malicious request manages to circumvent the firewall it might get caught by IAM and vice versa. As with many security features it makes sense to have more than one line of defense like this. Something similar was discussed in chapter 3 where the use of both network and host firewalls simultaneously was recommended. It needs to be considered that both the firewall concept and IAM cause additional overhead. If only one of the two systems can be used for whatever reason, the firewall concept should be used over the IAM concept. As the firewall can be configured with highly detailed rules, it can be adapted to the exact communication needs of a platform. Any communication that might not have been intended gets blocked this way. This helps limit access to the ECU to trusted sources with trusted protocols, preventing attack vectors and securing the system. Furthermore, the concept can also be used to achieve network separation as shown before which can be a useful tool in security critical environments.

6 Related Work

As the topic of this thesis covers a very specific use case, a prototype for host firewall implementations for embedded environments running AUTOSAR Adaptive, the amount of related work remains low. The main focus of the related work will be on firewall and IAM implementations in embedded environments in general.

Other work based around embedded firewalls mainly focuses on resource constraints and performance of the firewall implementation [PS17; SZD16], as well as improving the firewall based on the underlying hardware architecture [FHSM17]. Schmidt et al [SZD16] propose the use of a holistic security concept for any external communication. They provide patterns for general firewall systems in automotive systems and what kind of underlying hardware aspects are required for a firewall. As they focus on securing ECUs that are able to communicate with external networks they propose a network firewall system, as opposed to the host firewall concept presented in this thesis.

Another paper by Pesé and Schmidt [PS17] proposes a multi-layer approach to securing modern vehicles. A part of this approach is the implementation of an embedded firewall. It focuses on how the main tasks of a firewall can be distributed on a hardware and software implementation and discusses the resource consumption and hard resource constraints found in embedded vehicle environments. Fiessler et al [FHSM17] propose an embedded packet filter that is being run in a specifically developed massively parallel specialized circuitry. This circuitry is able to run simple firewall rules with high efficiency but is not able to run complex rules. More complex rules will instead be part of a separate software firewall. Both of these approaches split the actual firewall implementation into both a software and hardware part. Compared to that the prototype presented in this thesis sets up a software firewall only and favors high configurability and access to complex rules.

Payne and Markham [PM04] investigate a host firewall on embedded devices. Their approach is based on two main requirements. First, they acknowledge that even ECUs protected by the firewall might not be trustworthy. Second, to combat the first requirement, their firewall implementation needs to be tamper proof. Even if the host is compromised it will not be able to tamper with the firewall implementation. They achieve this by separating the actual firewall from the hosts operating system. The approach in this paper also assumes that firewall protected ECUs might get compromised, however, instead of trying to prevent disabling of the firewall the focus is instead shifted on only policing incoming traffic. Outgoing traffic is not policed at all as a compromised ECU would be able to turn off the firewall in our case, so only incoming traffic will be filtered to block malicious messages.

Most IAM implementations that could be used in a vehicle context seem to be meant for Internet of Things (IoT) devices. Here, Azhar [Azh21] and Cremonezi et al [CVNN20] both research use cases for Identity and Access Management in an embedded context for highly interconnected devices.

This approach checks devices privileges before any access is granted. However, no comparisons between firewall implementations are drawn. Furthermore, this approach checks device privileges while the IAM concept by AUTOSAR is based around privileges of individual applications.

7 Conclusion and Outlook

In conclusion, the prototype presented in this thesis showed how a highly configurable host firewall can be set up by using netfilter and nftables on a Linux virtual machine. A custom kernel was built to support the needed netfilter modules. Additionally, while allowing a high grade of flexibility for the host firewall configuration, the configuration file was well defined by both a UML diagram as well as an XML schema. This allows a clearer structure for the configuration file and the subsequent firewall configuration. This is important for maintenance of firewall rules, especially for more complex firewall setups. It was shown that the POC can be used to configure a working firewall setup for an AUTOSAR Adaptive ECU. It was also shown how network separation can be achieved with the POC.

Using the firewall concept along the AUTOSAR IAM concept shows a promising approach to a multi-layer security system for individual ECUs as well. Having more than one way to control access of certain resources is beneficial for ECUs as it prevents a single point of failure for the access control systems. Since the integration points of these protocols do not overlap, using them both at the same time should not be a problem.

Outlook

While the POC builds a good foundation for the general firewall concept there are still a lot of things that could be implemented or improved upon in future work.

First of all, having the software that parses the configuration file run as an actual daemon is desired. Currently, the POC runs as a normal application, with a blocking loop that checks for the current vehicle state. Having this functionality move to a daemon would make sense, as there is no need for direct user interaction with the firewall configuration software.

Adding proper support for the SOME/IP protocol to netfilter is possible through custom netfilter modules. Writing such a module would help make SOME/IP rules more readable as the *@th* notation is hard to understand. Since netfilter is extensible via custom modules there is designated functionality for extending the basic netfilter functionality. It would also allow verifying incoming messages being actual SOME/IP messages.

Similarly, support for other commonly used high level protocols could be added to the firewall concept. Notably the SOME/IP Service Discovery protocol (SOME/IP-SD), the Diagnostics over IP protocol (DoIP) and the Data Distribution Service protocol (DDS) should be supported in the final version. As with SOME/IP, the creation of custom netfilter modules to support them should be considered.

Since the software presented in this thesis was a proof of concept, no actual resource constraints were considered. Doing proper resource testing needs to be done to ensure the firewall concept does not cause too much overhead for embedded devices in an automotive context. Both initial setup time for a firewall configuration and switching between configurations of different modes should be considered. Additionally, the time it takes to check different rules should also be measured to see what the median overhead is for incoming packets. Worst case overhead for packets that do not match any rules should also be tested.

The packet inspection part of the prototype that was presented was dealing with shallow packet inspection as netfilter does not support any Deep Packet Inspection functionality. Looking into custom modules that would allow such functionality would be beneficial. However, it's not clear if DPI can be added to netfilter via a custom module in the first place or if it is even feasible in an automotive and embedded environment. Deep Packet Inspection can require more resources than both stateless and stateful packet inspection, depending on the implementation and how many features should be supported. Therefore, testing the resource requirements for proper DPI should be considered as well.

Bibliography

[al15] M. A. S. et al. “Identity and Access Management- A comprehensive Study”. Study. Jagannath University, 2015 (cit. on pp. 57–59).

[atl22] atlassin. *Was bedeutet Agile?* 2022. URL: https://www.atlassian.com/de/agile (cit. on p. 15).

[AUT17] AUTOSAR. *Specification of Manifest*. Mar. 31, 2017. URL: https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_TPS_ManifestSpecification.pdf (cit. on p. 19).

[Aut17] Autosar. *SOME/IP Service Discovery Protocol Specification*. Oct. 27, 2017. URL: https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-2/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf (cit. on p. 22).

[AUT20] AUTOSAR. *Specification of Intrusion Detection System Protocol*. 2020. URL: https://www.autosar.org/fileadmin/user_upload/standards/foundation/20-11/AUTOSAR_PRS_IntrusionDetectionSystem.pdf (cit. on p. 37).

[AUT21a] AUTOSAR. *AUTOSAR Release R20-11 veröffentlicht*. 2021. URL: https://www.autosar.org/fileadmin/user_upload/AUTOSAR_Release_R20-11_Press_DE.pdf (cit. on p. 58).

[AUT21b] AUTOSAR. *Explanation of Adaptive Platform Design*. Nov. 2021. URL: https://www.autosar.org/fileadmin/user_upload/standards/adaptive/21-11/AUTOSAR_EXP_PlatformDesign.pdf (cit. on pp. 13–19).

[AUT21c] AUTOSAR. *Explanation of Adaptive Platform Design*. 2021. URL: https://www.autosar.org/fileadmin/user_upload/standards/adaptive/21-11/AUTOSAR_EXP_PlatformDesign.pdf (cit. on pp. 58–60).

[AUT21d] AUTOSAR. *Explanation of Adaptive Platform Software Architecture*. Nov. 2021. URL: https://www.autosar.org/fileadmin/user_upload/standards/adaptive/21-11/AUTOSAR_EXP_SWArchitecture.pdf (cit. on p. 13).

[AUT21e] AUTOSAR. *SOME/IP Protocol Specification*. Nov. 25, 2021. URL: https://www.autosar.org/fileadmin/user_upload/standards/foundation/21-11/AUTOSAR_PRS_SOMEIPProtocol.pdf (cit. on pp. 19–22, 48, 50, 71).

[AUT22] AUTOSAR. *HISTORY*. 2022. URL: https://www.autosar.org/about/history/ (cit. on p. 11).

[AVM22] AVM. *Security functions (firewall) of the FRITZ!Box*. 2022. URL: https://en.avm.de/service/knowledge-base/dok/FRITZ-Box-7590/57_Security-functions-firewall-of-the-FRITZ-Box/ (cit. on p. 23).

[Azh21] I. Azhar. *Identity and Access Management for the Internet of Things*. Sept. 14, 2021 (cit. on p. 61).

[CVNN20] B. Cremonezi, A. Vieira, J. A. Nacif, M. Nogueira. *Survey on Identity and Access Management for Internet of Things*. Sept. 8, 2020 (cit. on p. 61).

[Dal20] A. Dalton. *Sending multiple packets of hex data with UDP*. Oct. 4, 2020. URL: https://unix.stackexchange.com/questions/612667/sending-multiple-packets-of-hex-data-with-udp (cit. on p. 71).

[ele22] elektronik-kompendium. *ESP - Encapsulating Security Payload*. 2022. URL: https://www.elektronik-kompendium.de/sites/net/1410261.htm (cit. on p. 24).

[f522] f5. *What is an Application Firewall?* 2022. URL: https://www.f5.com/services/resources/glossary/application-firewall (cit. on p. 26).

[FHSM17] A. Fiessler, S. Hager, B. Scheuermann, A. W. Moore. *HyPaFilter — A versatile hybrid FPGA packet filter*. Report. IEEE, Feb. 16, 2017 (cit. on p. 61).

[H R05] J. Z. H. Roßnagel. “Single Sign On mit Signaturen”. Essay. DuD, 2005 (cit. on p. 57).

[h06] P. N. A. h. *Netfilter’s connection tracking system*. paper. institution, June 2006 (cit. on p. 36).

[IET22] IETF. *RFCs*. 2022. URL: https://www.ietf.org/standards/rfcs/ (cit. on p. 30).

[IF02] K. INGHAM, S. FORREST. “A History and Survey of Network Firewalls”. In: (2002). URL: https://www.cs.unm.edu/~treport/tr/02-12/firewall.pdf (cit. on p. 23).

[Kal09] M. Kalicinski. *RAPIDXML*. 2009. URL: https://rapidxml.sourceforge.net/ (cit. on p. 45).

[Lim22] O. S. Limited. *hping3 Usage Example*. 2022. URL: https://www.kali.org/tools/hping3/ (cit. on p. 48).

[Mic22] Microsoft. *Turn Microsoft Defender Firewall on or off*. 2022. URL: https://support.microsoft.com/en-us/windows/turn-microsoft-defender-firewall-on-or-off-ec0844f7-aebd-0583-67fe-601ecf5d774f (cit. on p. 23).

[net20] netfilter. *NFT*. 2020. URL: https://www.netfilter.org/projects/nftables/manpage.html (cit. on p. 35).

[net21a] netfilter. *Configuring chains*. 2021. URL: https://wiki.nftables.org/wiki-nftables/index.php/Configuring_chains (cit. on pp. 33–35).

[net21b] netfilter. *Configuring tables*. Apr. 17, 2021. URL: https://wiki.nftables.org/wiki-nftables/index.php/Configuring_tables (cit. on p. 33).

[net21c] netfilter. *Connection Tracking System*. 2021. URL: https://wiki.nftables.org/wiki-nftables/index.php/Connection_Tracking_System (cit. on p. 35).

[net21d] netfilter. *Netfilter hooks*. Apr. 19, 2021. URL: https://wiki.nftables.org/wiki-nftables/index.php/Netfilter_hooks (cit. on p. 34).

[net21e] netfilter. *Simple rule management*. 2021. URL: https://wiki.nftables.org/wiki-nftables/index.php/Simple_rule_management (cit. on p. 35).

[nft21] nftables. *Nftables families*. 2021. URL: https://wiki.nftables.org/wiki-nftables/index.php/Nftables_families (cit. on p. 33).

[PM04] C. Payne, T. Markham. *Architecture and Applications for a Distributed Embedded Firewall*. 2004 (cit. on p. 61).

[PS17] M. D. Pesé, K. Schmidt. *Hardware/Software Co-Design of an Automotive Embedded Firewall*. Mar. 28, 2017 (cit. on p. 61).

[Rod12] R. Rodriguez. *IPv4 Scarcity*. 2012. URL: https://www.internetsociety.org/blog/2012/11/ipv4-scarcity/ (cit. on p. 25).

[sam21] sam_2200. *Examples on UDP Header*. 2021. URL: https://www.geeksforgeeks.org/examples-on-udp-header/ (cit. on p. 49).

[Sch19] E. Schneider. "Sichere Kommunikation unter Verwendung von Adaptive Autosar". Bachelors Thesis. Universitaet Stuttgart, Oct. 9, 2019 (cit. on pp. 11, 13, 14, 16, 17).

[se22] ionos se. *Was ist ICMP? – Das steckt hinter dem Nachrichtenprotokoll*. 2022. URL: https://www.ionos.de/digitalguide/server/knowhow/was-ist-das-icmp-protokoll-und-wie-funktioniert-es/ (cit. on p. 24).

[Sec22] N. Security. *TCP vs. UDP – die beiden Protokolle im Vergleich*. 2022. URL: https://nordvpn.com/de/blog/tcp-vs-udp/ (cit. on p. 24).

[SH09] K. Scarfone, P. Hoffman. "Guidelines on Firewalls and Firewall Policy". In: (Sept. 2009). URL: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-41r1.pdf (cit. on pp. 23–32).

[Sri22] A. Srivastava1. *Internet Protocol Authentication Header*. 2022. URL: https://www.geeksforgeeks.org/internet-protocol-authentication-header/ (cit. on p. 24).

[SZD16] K. Schmidt, H. Zweck, U. Dannebaum. *Hardware and Software Constraints for Automotive Firewall Systems?* SAE Technical Paper. Society of Automotive Engineers (SAE), Apr. 5, 2016 (cit. on p. 61).

[ubn18] ubntuusers. *netcat*. 2018. URL: https://wiki.ubuntuusers.de/netcat/ (cit. on p. 50).

[web21] T. N. webmasters. *The netfilter.org project*. 2021. URL: https://www.nftables.org/ (cit. on pp. 32, 33).

[wol22] wolfSSL. *TLS 1.3 PROTOCOL SUPPORT*. 2022. URL: https://www.wolfssl.com/docs/tls13/ (cit. on p. 11).

All links were last followed on November 7, 2022.

XML Schema Specification

An XML schema was derived from the UML diagrams presented in section 4.1. It was appended here for completeness and will not be explained in detail:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="AdaptiveFirewall">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ModeDependentFirewall" type="ModeDependentFirewallType" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="ModeDependentFirewallType">
        <xs:sequence>
            <xs:element name="ModeDeclaration" type="xs:string"/>
            <xs:element name="FirewallRuleProps" type="FirewallRulePropsType"
                minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="defaultAction" type="FirewallActionEnum"/>
    </xs:complexType>


    <xs:complexType name="FirewallRulePropsType">
        <xs:sequence>
            <xs:element name="FirewallRule" type="FirewallRuleType" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="action" type="FirewallActionEnum" use="required"/>
    </xs:complexType>


    <xs:simpleType name="FirewallActionEnum">
        <xs:restriction base="xs:string">
          <xs:enumeration value="allow"/>
          <xs:enumeration value="block"/>
        </xs:restriction>
    </xs:simpleType>


    <xs:complexType name="FirewallRuleType">
        <xs:sequence>
            <xs:sequence>
                <xs:element name="bucketSize" type="xs:positiveInteger" minOccurs="0"/>
                <xs:element name="refillAmount" type="xs:positiveInteger" minOccurs="0"/>
                <xs:element name="PayloadBytePatternRule" type="PBPType" minOccurs="0"/>
                <xs:element name="DataLinkLayerRule" type="DLLType" minOccurs="0"/>
                <xs:element name="NetworkLayerRule" type="NLType" minOccurs="0"/>
                <xs:element name="TransportLayerRule" type="TLType" minOccurs="0"/>
            </xs:sequence>
            <xs:choice>
                <xs:element name="SomeipProtocolRule" type="SomeipPType" minOccurs="0"/>
            </xs:choice>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="forward" type="xs:boolean"/>
    </xs:complexType>

    <xs:complexType name="PBPType">
      <xs:sequence>
        <xs:element name="PayloadBytePatternRulePart" type="PBPPartType"
            minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="PBPPartType">
      <xs:sequence>
        <xs:element name="offset" type="xs:string"/>
        <xs:element name="value" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
```

Figure 1: First part of the XML schema. Contains the top level elements and some rule types.

```
<xs:complexType name="DLLType">
  <xs:sequence>
    <xs:element name="destinationMacAddress" type="xs:string" minOccurs="0"/>
    <xs:element name="destinationMacAddressMask" type="xs:string" minOccurs="0"/>
    <xs:element name="etherType" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="sourceMacAddress" type="xs:string" minOccurs="0"/>
    <xs:element name="sourceMacAddressMask" type="xs:string" minOccurs="0"/>
    <xs:element name="vlanID" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="vlanPriority" type="xs:positiveInteger" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="NLType">
  <xs:choice>
    <xs:element name="Ipv4Rule" type="Ipv4Type" minOccurs="0"/>
    <xs:element name="Ipv6Rule" type="Ipv6Type" minOccurs="0"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="Ipv4Type">
  <xs:sequence>
    <xs:element name="destinationIpAddress" type="xs:string" minOccurs="0"/>
    <xs:element name="destinationNetworkMask" type="xs:string" minOccurs="0"/>
    <xs:element name="differentiatedServiceCodePoint" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="doNotFragment" type="xs:boolean" minOccurs="0"/>
    <xs:element name="explicitCongestionNotification" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="internetHeaderLength" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="moreFragments" type="xs:boolean" minOccurs="0"/>
    <xs:element name="protocol" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="sourceIpAddress" type="xs:string" minOccurs="0"/>
    <xs:element name="sourceNetworkMask" type="xs:string" minOccurs="0"/>
    <xs:element name="ttlMax" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="ttlMin" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="IcmpRule" type="IcmpType" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="IcmpType">
  <xs:sequence>
    <xs:element name="code" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="type" type="xs:positiveInteger" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Ipv6Type">
  <xs:sequence>
    <xs:element name="destinationIpAddress" type="xs:string" minOccurs="0"/>
    <xs:element name="destinationNetworkMask" type="xs:string" minOccurs="0"/>
    <xs:element name="flowLabel" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="hopLimit" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="nextHeader" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="sourceIpAddress" type="xs:string" minOccurs="0"/>
    <xs:element name="sourceNetworkMask" type="xs:string" minOccurs="0"/>
    <xs:element name="trafficClass" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="IcmpRule" type="IcmpType" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="TLType">
  <xs:choice>
    <xs:element name="UdpRule" type="UdpType" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="TcpRule" type="TcpType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:choice>
</xs:complexType>
```

Figure 2: Second part of the XML schema. Contains most of the OSI layer rules and type definitions.

```
    <xs:complexType name="TLType">
      <xs:choice>
        <xs:element name="UdpRule" type="UdpType" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="TcpRule" type="TcpType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:choice>
    </xs:complexType>

    <xs:complexType name="UdpType">
        <xs:sequence>
            <xs:element name="maxDestinationPortNumber" type="xs:positiveInteger" minOccurs="0"/>
            <xs:element name="maxSourcePortNumber" type="xs:positiveInteger" minOccurs="0"/>
            <xs:element name="minDestinationPortNumber" type="xs:positiveInteger" minOccurs="0"/>
            <xs:element name="minSourcePortNumber" type="xs:positiveInteger" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="TcpType">
      <xs:sequence>
        <xs:element name="maxDestinationPortNumber" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="maxSourcePortNumber" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="minDestinationPortNumber" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="minSourcePortNumber" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="numberOfParallelTcpSessions" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="stateManagementBasedOnTcpFlags" type="xs:boolean" minOccurs="0"/>
        <xs:element name="timeoutCheck" type="xs:positiveInteger" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="SomeipPType">
      <xs:sequence>
        <xs:element name="clientId" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="lengthVerification" type="xs:boolean" minOccurs="0"/>
        <xs:element name="majorVersion" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="messageType" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="methodId" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="protocolVersion" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="returnCode" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="serviceInterfaceId" type="xs:positiveInteger" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Figure 3: Third part of the XML schema. Contains the transport layer and SOME/IP rules and type definitions.

SOME/IP test script

The script used to test SOME/IP rules is presented in this part of the appendix. It was adapted from a general script for sending multiple UDP packets of hex data.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>


#define DESTINATION_ADDRESS "127.0.0.1"
#define DESTINATION_PORT    8080
#define MAX_MESSAGE_LENGTH 16

struct message {
    int length;
    char bytes[MAX_MESSAGE_LENGTH];
};

int main(void)
{
    const int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    const struct sockaddr_in servaddr = {
        .sin_family      = AF_INET,
        .sin_port        = htons(DESTINATION_PORT),
        .sin_addr.s_addr = inet_addr(DESTINATION_ADDRESS),
    };

    const struct message m = {.length=16, .bytes =
        {0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0xDE, 0xAD, 0xBE, 0xEF, 0x01, 0x01, 0x01, 0x00}};


    if (sendto(sockfd,
                m.bytes,
                m.length,
                0,
                (const struct sockaddr*) &servaddr,
                sizeof(servaddr)) < 0) {
        perror("sendto");
    }


    close(sockfd);
    return EXIT_SUCCESS;
}
```

Figure 4: C script to send custom UDP packets. Adapted from Dalton's example [Dal20].

The example message m that is being sent in figure 4 is the magic cookie message detailed in the SOME/IP specification [AUT21e].

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature