CHALMERS
UNIVERSITY OF TECHNOLOGY

Evaluation of Security Mechanisms for an Integrated Automotive System Architecture

Master's thesis in Computer Systems and Networks

JOAKIM KARLSSON

PONTUS KHOSRAVI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Master's thesis 2017

Evaluation of Security Mechanisms for an Integrated Automotive System Architecture

JOAKIM KARLSSON

PONTUS KHOSRAVI

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2017

Evaluation of Security Mechanisms for an Integrated Automotive System Architecture
Joakim Karlsson Pontus Khosravi

Supervisor: Tomas Olovsson, Department of Computer Science and Engineering
Examiner: Erland Jonsson, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2017

Abstract

Prior research on security mechanisms for the connected vehicle focuses on the current heterogeneous distributed *embedded electronic architecture* (EEA). However, due to increased demands on data-processing and bandwidth from future functionality, such as automated driving and sophisticated infotainment systems, vehicle manufacturers are now moving towards centralisation of functionality in one or more integrated electronic control units (ECUs). In this thesis, we identify security issues related to centralization of functionality in the automotive domain, and investigate mechanisms which mitigate these issues. Further, we investigate the support, both in hardware and software, for these mechanisms in the current and future automotive embedded systems.

To identify security issues related to the integrated EEA and to identify mechanisms which mitigate these issues, a literature review of other domains was performed. The *AUTomotive Open System ARchitecture* (AUTOSAR) development partnership has specified a software architecture with the aim of standardizing software development in the automotive industry. We have investigate whether the identified mechanisms are specified in the AUTOSAR framework by interviewing AUTOSAR specialists. Some form of hardware feature is typically needed to be able to use the mechanisms. Therefore, hardware requirements for each mechanism were identified and used to evaluate the support present in ECU hardware. We have found that there was lacking support for most of the identified mechanisms, in both the present ECU hardware as well as in the current AUTOSAR platform. However, the hardware targeted to be used in future ECUs, along with the next-generation software standard *Adaptive AUTOSAR*, show support for all the presented mechanisms. This shows that the automotive industry have realized that cyber-security will be a major challenge in the development of future connected autonomous vehicles and that is aims to adopt a strong security posture.

Keywords: automotive security, automotive cyber-security, AUTOSAR.

Acknowledgements

We want to extend our thanks to Atul Yadav & Christian Sandberg, our supervisors at Volvo Trucks, for the opportunity to work with them during this thesis. Further, we would like to thank Tomas Olovsson and Erland Jonsson, our academic supervisor and examiner at Chalmers University of Technology.

Joakim Karlsson & Pontus Khosravi, Gothenburg, December 2017

Contents

1

Introduction

Modern vehicles communicate with other devices through both wired and wireless interfaces and the trend towards increased connectivity is expected to continue with future deployment of car-to-car and car-to-infrastructure communication [1]. This communication exposes the in-vehicle network to the outside world and can thus be used as vectors for cyber attacks which can have financial, operational, privacy, and safety impacts [2]. Cybersecurity issues have largely been neglected by the automotive industry until recently, when a number of high profile incidents [3, 4] exposed security flaws of commercial vehicles which could prove fatal for passengers, and cause severe economic damages to the manufacturers. These incidents, along with governmental initiatives to introduce legislation within the automotive area regarding security and privacy have highlighted the importance of cybersecurity for manufacturers.

The automotive industry is currently experiencing a paradigm shift towards the connected autonomous vehicle. With this shift follows greater requirements for data-processing and bandwidth, due to sensor fusion and the machine learning algorithms needed to offer functionality associated with automated driving [5]. Increasingly sophisticated infotainment systems further exacerbates the data processing requirements. Current vehicles typically employ a heterogeneous distributed embedded electronic architecture (EEA), where dedicated hardware components known as *electronic control units* (ECUs) realise mostly independent or loosely interconnected functions [6]. However, the increased cost, weight, and complexity due to the proliferation of ECUs and communication buses means that this architecture is no longer scalable [6]. Additionally, the data-processing requirements of future functionality is pushing the limits of currently available microprocessor technology developed for the automotive domain [5]. Therefore, the trend is now moving towards centralisation of functionality in one or more high-performance nodes, which act as arbitrators between different subsystems.

Integrating functionality to the same hardware platform will give rise to new problems, not present in the current architecture, because of resource sharing between applications. It will be crucial to ensure that a security flaw in one application does not influence another. Therefore, it must be possible to isolate applications by restricting access to shared resources. In the current distributed architecture, resources are physically separated and separation of information flow can be achieved entirely

by network segmentation. With small degrees of centralisation these solutions may still be an option. But as the degree of centralisation grows, application isolation must be solved using software mechanisms, to contain damage to the system and user-data should an application be compromised. Further, different applications will have different security requirements and when integrated on the same platform will give rise to a mixed-criticality system. Finally, different security mechanisms may require varying complexity of hardware features, something that could increase the cost of the system. To minimize system complexity, the hardware requirements of identified security mechanisms need to be investigated.

1.1 Purpose

The aim of this thesis is to evaluate mechanisms which mitigate security issues associated with centralizing functionality to the same automotive hardware platform, with regard to the current and future automotive embedded systems.

To achieve this purpose we answer the following research questions:

- What are the security threats associated with centralizing functionality?
- Which mechanisms exist that could mitigate the identified security threats?
- What metrics can be used to evaluate such mechanisms?
- What support exists for these mechanisms in automotive embedded systems?

1.2 Definition of Support

This paper often refers to the 'support' for a mechanism in both hardware and software. Therefore, the following section defines the scope of the term 'support' as used throughout the paper. Some form of hardware acceleration features are typically needed to be able to efficiently use the mechanisms identified in this paper. Therefore, when discussing support for a mechanism in ECU hardware, we refer to whether or not a specific ECU includes the required hardware features needed to efficiently run that mechanism. When discussing support for a mechanism in the automotive software market, we refer to whether or not the mechanism is specified in the AUTOSAR framework. AUTOSAR support is desirable because AUTOSAR compliant products can easily be used across most of the commodity ECU hardware, independently of manufacturer.

1.3 Method

We have studied the state-of-the-art in automotive electronic embedded systems and the trend to centralize functionality in order to determine related security issues. We surveyed the present automotive threat analysis to identify high level automotive threats. These high level threats are applied to an integrated architecture in order to generate specific threats and to identify the assets and security attributes which these threats impact. To identify software mechanisms which mitigate these threats, we study mechanisms used in other domains. We consider how the mechanisms can protect each of the asset-attribute pairs previously obtained and we investigate the hardware features required for each mechanism.

To determine the support in automotive hardware we investigate the hardware features of common ECUs. To determine the support for the mechanism in the automotive software market, we interview AUTOSAR specialists at ARCCORE, a supplier of automotive software for Volvo Trucks.

1.4 Scope and limitations

We have investigated mechanisms to contain any breach from spreading from a compromised application to others. We only consider the security of applications on the same hardware platform and applications which can communicate with the vulnerable application via the internal network. However, we do not consider mechanisms aimed to secure the perimeter of the system. This means that we do not consider solutions for securing or authenticating any external communication. Future vehicles with automated driving capabilities must be able to securely communicate with other vehicles, but this is out of scope for this thesis. Instead we assume that secure external communication exist for applications which depend on this.

We assume that applications running on the same hardware is supplied from trusted third parties, but may contain vulnerabilities which can be used as attack vectors to the system. If we allow applications from untrusted sources, or from an "app store", we must consider issues such as verification of privilege requests and security evaluations of the source code, but this is outside our scope. Neither will we consider vulnerabilities of individual applications.

1.5 Related work

Most of the work related to security in the automotive domain has focused on the current distributed EEA. Because of this, most of the proposed security mechanisms are either aimed at perimeter security, or to prevent attack propagation on the internal network. The works of Miller & Vasalek can be credited with being the

main contributors to opening the eyes of the automotive domain to the security issues posed to them. In one paper [7], they examine the features, standards, and network architecture of over 20 vehicles to determine their attack surfaces. For each vehicle they list the layout of the internal vehicle network, detailing which ECUs pose remote attack surface and which has safety critical components. Further, they propose the use of layered security to protect against remote attacks, as the nature of the attacks are themselves layered. In a later work [3], they detail the remote exploitation of a Jeep Cherokee, the vehicle deemed to be most susceptible to attack in the previous work. The attack allowed them to gain remote control over the vehicles critical functions, and thus physical control of the vehicle. The attack required no physical interaction or modification of the vehicle, and led to a recall of 1.4 millions vehicles.

Studnia et al. [1] provides a detailed survey of security threats related to current automotive embedded systems, and proposed protection mechanisms. They analyze the currently used communication protocols in automotive embedded networks, to find threats targeting the in-vehicle network. As such, they consider only protection mechanisms which aim to stop an attack from propagating between different ECUs on the in-vehicle network. The HEAVENS project [8] have investigated different available security models and proposed a new security model aimed at performing threat analysis and risk assessment on automotive embedded systems. The model facilitates identification of security levels for threat-asset pairs, as well as deriving security requirements, by combining the threat-asset pair with security attributes and a security level. However, it does not suggest any security mechanisms which fulfill the derived security requirements according to the derived security level.

Most of the work related to integrating several functions on the same platform mostly considers the safety issues involved and gives little consideration to security. Obermaisser et al [9] considers fault-isolation, error containment, and state recovery when describing an integrated system architecture for the automotive domain. In a review covering most of the recent research in mixed-criticality systems, Burns & Davis identifies the fundamental issue in mixed-criticality systems as "how to reconcile the differing needs of separation (for safety) and sharing (for efficient resource usage)" [10]. They explain that much of the implementation work has focused on how to safely partition a system so that critical components can share different resources, while most of the theoretical research has focused on solving this issue with task scheduling. A recent work has shown the feasibility of integrating a real-time operating system (RTOS), running time-constrained tasks, with a general purpose operating system (GPOS) executing the resource intensive infotainment and connectivity tasks on the same hardware platform using hypervisors [11]. However, they focus only on safety and reliability aspects of centralizing functionality, and do not consider security implications.

1.6 Outline

This thesis is structured as follows:

- Chapter 1 - In this section the purpose of the thesis is presented and motivated and a method of how to achieve this purpose is explained. Further, some limitations are made and some important related work is presented.
- Chapter 2 - This section provides the necessary background information. In the first part the automotive embedded systems are explained in terms of hardware and software architectures. In the second part automotive cybersecurity is explained including a general threat model and some history on cyber-attacks.
- Chapter 3 - In this section issues related to centralization of functionality are investigated. Based on these issues and some of the content in Section 2, a threat analysis is performed to identify what kind of protection that is required.
- Chapter 4 - Based on the threat analysis in section 3, the chosen security mechanisms are presented. Each mechanism is explained and evaluated with regard to known strengths and weaknesses. Further, the hardware requirements for each mechanism is investigated.
- Chapter 5 - The first part of this chapter evaluates the breadth of protection o□ered by the chosen security mechanisms. The second part investigates the support for these mechanisms in automotive hardware. The final part investigates support in standardized software frameworks.
- Chapter 6 In this final chapter of this thesis, a conclusion is presented and some of the results from the evaluation in Chapter 5 are discussed. Further, some recommendations for future work is made.

2

Background

2.1 Automotive embedded systems

The use of embedded electrical and electronic (E/E) systems have revolutionized the automotive industry in the latest decades. A majority of automotive innovations would not be possible without the use of E/E systems and software, which are now essential to much of the functionality of modern vehicles. Embedded systems help control the movement of the vehicle, the chemical and mechanical processes taking place in it, and perhaps most importantly, to help ensure the safety of its passengers [12]. More recently, embedded systems are also used to entertain passengers and establish connectivity with the rest of the world, through Internet connectivity and other wired and wireless interfaces [12]. Perhaps the most important function of automotive embedded systems is to help achieve the stringent safety requirements posed on the vehicle. *Passive safety* systems refer to those systems which aim to reduce the a□ect of potential accidents, such as airbags and seat belts, while *active safety* systems work preemptively with the aim to avoid accidents. Examples of active safety systems are anti-lock braking, lane-keeping, and electronic stability control.

2.1.1 Current distributed architecture

Current vehicles comprise of a complex distributed system of sensors, actuators, dedicated *electronic control units* (ECUs), and several internal communication networks. This architecture is typically referred to as a *federated* architecture, where an ECU is responsible for one specific task or functionality and communicates with sensors, actuators, and other ECUs over one of several communication networks to perform their task. This paradigm arose because automotive manufacturers typically purchase subsystems designed for specific functionality from suppliers, and integrate them into their vehicle, in order to reduce costs [13]. By connecting ECUs over communication buses, functions distributed over several ECUs could be realized. These functions support tasks of di□ering constraints and requirements, as specified by their functional domains, which led to the introduction of several types of communication networks, to suit the needs of each domain. This has resulted

in a distributed architecture consisting of as many as 100 ECUs [14] in a plethora of ECU-subsystems connected by many different in-vehicle networks such as CAN, MOST, etc. An example of a federated architecture is shown in Figure 2.1.

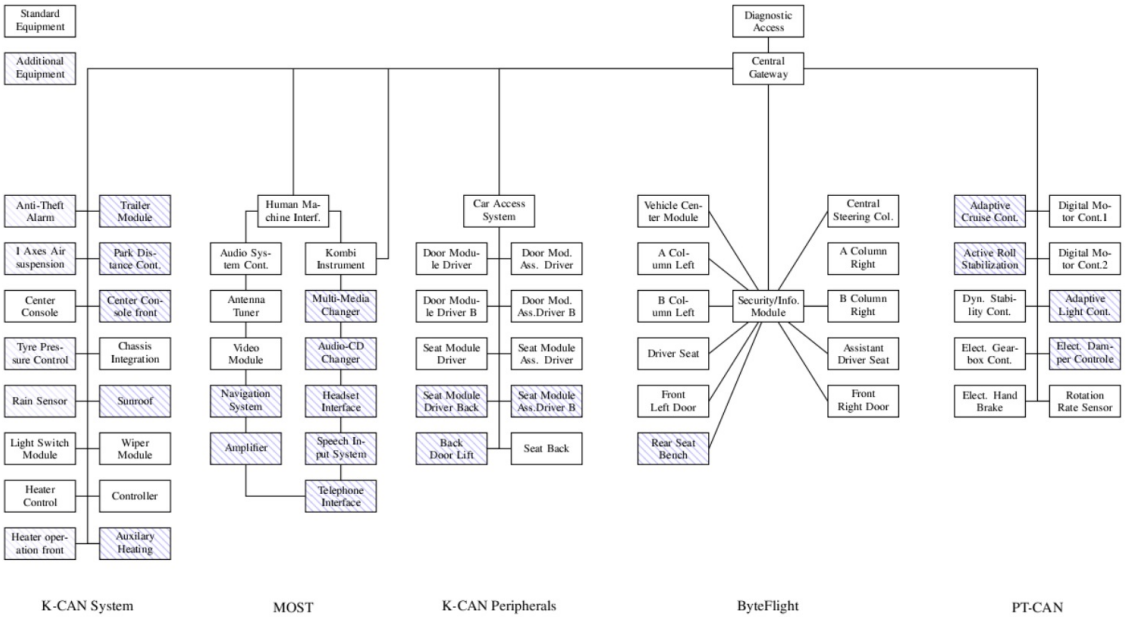

Figure 2.1: Example of a federated architecture [15].

The severe hardware limitations of ECUs in the current heterogenous EEA make many traditional security mechanisms unsuitable or impossible to employ. They would either take too long to compute, violating real-time constraints, or be so weak that an attacker could easily break them [2]. Centralisation of functionality may have the added benefit of allowing traditional security mechanisms to be used to protect the system.

2.1.2 Centralization of functionality

In recent years the current distributed automotive architecture model has been increasingly challenged due to the increased system complexity, costs due to the proliferation of electronic components, under-utilization of ECUs, and increased functional complexity [6]. As car manufacturers continue to add functionality to their vehicles by purchasing self-contained hardware platforms, in the form of ECUs, the total cost of embedded electronics rises to unwanted levels. Most of the commonly used communication buses can only accommodate a certain number of ECUs, which means that the number of buses grows with the number of ECUs. The increased number of electrical components add weight to the vehicle, which leads to reduced fuel efficiency. The use of dedicated ECUs for infrequently used functions, such as seat or mirror adjustment, means unnecessary added weight and occupied space but also wasted computational power as these ECUs are idle for a majority of the

time. Finally, future functionalities associated with autonomous driving will fuse data from multiple sensors and require huge amount of data-processing, which will push the limits of the currently available microprocessor technologies [5].

To solve these problems, several automotive manufacturers have started to rethink their architectures, moving to an *integrated* architecture. The idea is to use a central processor to serve as an arbitrator and decision maker between different subsystems, designed to perform the same work as multiple ECUs in order to reduce system complexity, cost and mass [5]. Because the heavy data processing would be done centrally, future generations of sensors can be made smaller and more affordable, as there is no longer a need to include high-end microprocessors. This design is more future proof as there is enough memory and processing headroom to add more functionality.

2.1.3 AUTOSAR

AUTOSAR (Automotive Open System Architecture) was founded in 2003 and is a set of specifications detailing software architecture components and specifying their interfaces. The goal of AUTOSAR is to introduce an open industry standard for the automotive software architecture between manufacturers and suppliers in order to decrease the growing complexity of software [16, 17]. There are nine core companies developing AUTOSAR, including Ford, BMW and General Motors [18]. AUTOSAR creates an abstract layer of the underlying hardware, thus software running on top of AUTOSAR is hardware independent. AUTOSAR also enables several software modules to run on the same ECU independently of the suppliers [17].

2.1.3.1 AUTOSAR Classic

On the highest abstractions level AUTOSAR Classic consists of three software layers: Basic Software, Runtime Environment, and Application. The Basic Software layer is further divided into four layers as shown in Figure 2.2.

Microcontroller Abstraction Layer - Contains internal drivers which has direct access to the microcontroller and makes higher software layers independent of the microcontroller [19].

ECU Abstraction Layer - Contains drivers for external devices and also interfaces the drivers of the Microcontroller Abstraction Layer to offer an API for access to any peripheral or device. Makes higher software layers independent of the ECU hardware layout [19].

Complex Drivers - Enables the possibility of implementing special purpose functionality which might not be specified within AUTOSAR Classic [19].

Service Layer - Provides basic services for applications, for example operating

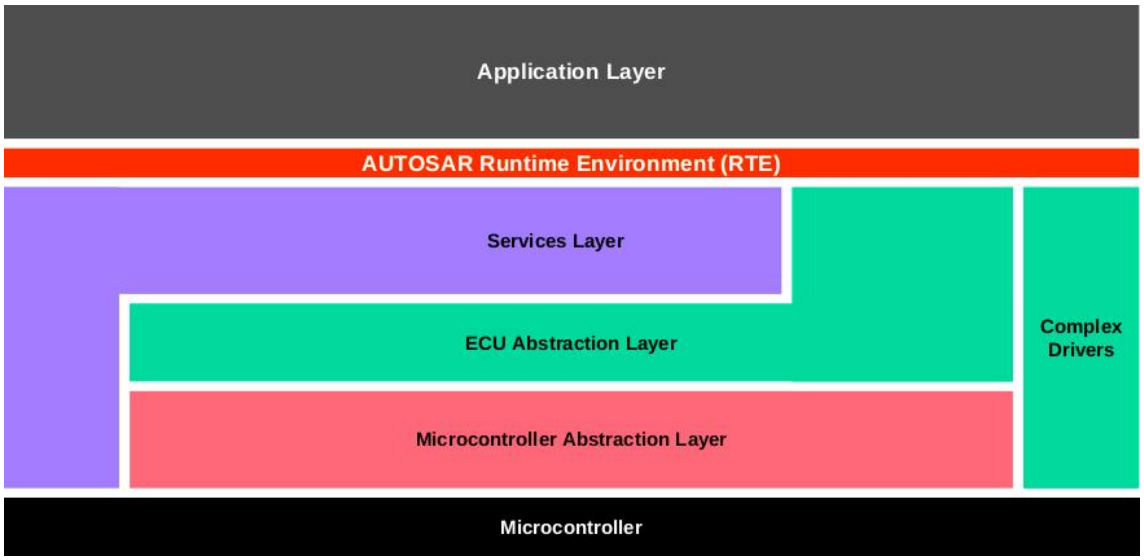

Figure 2.2: The layered software architecture of AUTOSAR Classic [19].

system functionality, vehicle network communication, and cryptographic services [19].

Runtime Environment - Enables AUTOSAR software components to communicate with other type of components, within and/or outside the ECU, and services. Makes the software independent from the mapping to a specific ECU [19].

AUTOSAR Classic has three main security mechanisms; Crypto Service Manager (CSM), Crypto Abstraction Library (CAL), and Secure On-Board Communication (SecOC). The CSM is located in the Service layer and provides cryptographic services to applications on higher levels while CAL is a static library, very similar to CSM in functionality, but also o□ers cryptographic services to software modules in the Basic Software layer [20]. CSM has the advantage of utilizing cryptographic hardware if present, while CAL can not. There are no cryptographic algorithms defined for CSM or CAL by AUTOSAR, instead the implementer chooses which algorithms to include [20]. SecOC o□ers an authentication mechanism for critical data which can be utilized by any ECU requiring secure communication. This module can provide protection against injection, alteration and replay attacks on the in-vehicle network [16].

2.1.3.2 Adaptive AUTOSAR

The *AUTOSAR Adaptive Platform for Connected and Autonomous Vehicles* is designed to meet the new use cases emerging with the connected vehicle. The main drivers for the initiative are use cases such as highly automated driving, Car-2-X applications, vehicle-in-the-cloud and increased connectivity. Because of this, future cars are expected to have a heterogeneous electronic architecture, mixing deeply em-

bedded systems and more dynamic systems, creating a need for a complementing architecture. The Adaptive platform is meant to close the gap between the Classic AUTOSAR platform, which focuses on real-time requirements and safety, and dynamic applications needing high computing power and security. It will support adaptive deployment, complex microcontrollers, and interaction with non-AUTOSAR systems [21]. In contrast to the Classic Platform, the Adaptive Platform Runtime Environment dynamically links services and clients during runtime [22]. The architecture of Adaptive AUTOSAR is shown in Figure 2.3

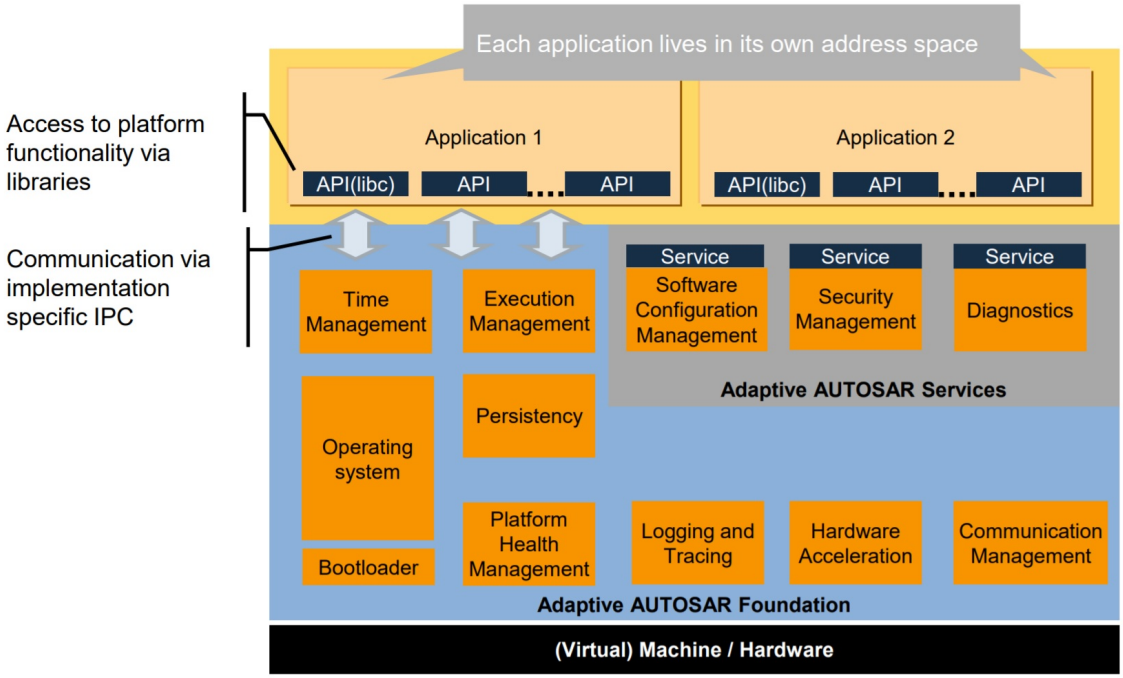

Figure 2.3: The software architecture of Adaptive AUTOSAR [21].

2.2 Automotive cybersecurity

2.2.1 History of automotive cyber-attacks

Previous to the concept of the connected vehicle, automotive OEMs did not consider cyber-security that much since an attack was only possible if the attacker had physical access to the vehicle. The modern vehicle however, has multiple wireless connections to outside networks and devices, for example the Internet and the connection of a smart device over bluetooth. Additionally, vehicles will communicate with each other and also the infrastructure with the common goal of increasing safety, e□ciency, and driving experience. As the complexity of the connected vehicle increases, so will the security threats by the increase of attack surface. The attack surface comprises of all accessible services and connection points of a vehicle

and is used to gain unauthorized access to the vehicle system. Without an attack surface, vulnerabilities can not be exploited to launch attacks. However, the attack surface of the connected vehicle will only continue to grow and securing the entire attack surface is not feasible, thus perimeter security is no longer enough. As briefly presented below, history has shown that systems will be breached and there needs to exist additional levels of security closer to individual resources.

In 2016, researchers from the Chinese security firm Tencent discovered a flaw in the WiFi connectivity of all Tesla car models that would allow them them to gain remote access to the driving and braking system [16]. The attack was made possible due to four independent vulnerabilites [23]. First, the researchers set up a fake Tesla WiFi hot-spot by simply using the same name and password used in Tesla dealership hot-spots. The car would then auto-connect to the fake hot-spot when it was in close proximity and the hot-spot was configured so that the Tesla webbrowser would load a webpage designed by the researchers when it connected. Second, a vulnerability in the webbrowser allowed the researchers to run malicious code, fetched from the malicious website, in the browser. Then, a vulnerability in Tesla's Linux operating system gave the researchers full privileges on the car's head unit. Finally, a vulnerability allowed the researchers to overwrite the firmware of a gateway that separated the head unit from the internal CAN network, thus providing them with access to the driving and braking system.

This attack is a good example of how one small vulnerability can jeopardize the entire system. With additional security mechanisms the attack could have been stopped at four di□erent stages. The concept of defence-in-depth should be applied in automotive security to prevent one breached system to cause further damage.

Automotive security involves more than just the functional safety of the vehicle, it should also consider other issues, like privacy. The Nissan LEAF is one of the most sold electric cars in the world [4] and comes with an companion-app which the owner can use to observe the status of the car. Researcher Troy Hunt discovered a vulnerability in the app that allows anyone to collect sensitive information, like the GPS history of the car, with a simple GET message [4]. No authentication is needed, besides providing the VIN (Vehicle Identificaiton Number) of the car, which is physically displayed on the windshield of the car. Although this vulnerability doesn't pose any direct danger to the passengers, it can still be used for malicious activity. The GPS history can be used to map the driving patterns of the owner and can be used by burglars to estimate which hours of the day the owner is not at home.

2.2.2 Systems cybersecurity engineering

To reduce the likelihood of successful cyber-attacks on vehicles principles of system cybersecurity engineering should be applied to the design and development of cybersecurity-critical vehicle systems. A cybersecurity-critical system is a system which may incur financial, operational, privacy, or safety losses if compromised due

to a vulnerability in the system [24]. Vehicles are safety critical systems that may cause harm to life, property, or the environment should it not operate as intended. Therefore they also inherently constitute a cybersecurity-critical system since an attack on the system may incur safety losses. Vehicle safety has long been a focus for the automotive industry, while cybersecurity has been neglected. However, due to increased connectivity it is no longer acceptable to assume that safety-critical systems are immune to security risks. This has led several industry alliances to release guidelines for securing cyber-physical vehicle systems [24, 25, 26]. The consensus advocates a holistic approach to securing the connected vehicle by building security into the design, rather than being added on to an existing design .

Clearly there is some overlap between system safety engineering and system cybersecurity engineering, since all safety critical systems are also cybersecurity-critical. However, the focus of cybersecurity engineering is much broader, as it also includes non safety related systems by considering privacy, operational, and financial losses. The connected vehicle is susceptible to each of these aspects; privacy related information may be stored in infotainment systems or diagnostics; a high profile attack may incur financial losses due to recalls or loss of OEM reputation; finally, features or functionality might be altered to increase the operational performance of the vehicle. Further, potential security threats involve intentional malicious actions which are more difficult to address than safety hazards which occur due to random hardware failure. An acceptable risk level of a safety hazard can be claimed by applying statistical analysis while assessing the risk of a cybersecurity threat must consider factors such as the experience and knowledge of an attacker. Further, the malicious intent of an attacker means that a state considered safe from a safety engineering perspective must be assessed as to whether it can be exploited by an attacker to compromise the system from a cybersecurity perspective. Finally, cybersecurity risks will evolve over time as motivations and capabilities of attackers change. This makes cybersecurity engineering especially difficult, because the system must be defended against vulnerabilities which did not even exist at the time of implementation.

2.2.3 Designing secure automotive systems

To allow for a system to protect itself and the information it processes against attacks it must be architected, designed, and implemented with security in mind [27]. The aim when designing for security is to reduce the attack surface of the system, and limit damage in case of a compromise. This is achieved by key design principles, such as employing layered defenses, restricting access to system services, and applying the principle of least privilege.

No complex system can achieve perfect security, instead designers must assume that the systems contains security flaws and design to minimize harm should these flaws be abused. This is especially true for the automotive domain, where OEMs often purchase ready-made software from suppliers without access to the source code. A layered approach to security hardens the vehicles electronic architecture against potential attacks and mitigates the ramifications should an attack be successful [26].

Every system component which impacts security should be secured by a security mechanism appropriate for the associated level of risk. At the application layer, appropriate perimeter defenses should be applied to secure specific applications against external threats. To further strengthen the security of the system, additional security mechanism should be applied at lower layers to contain damage if a vulnerability at the application layer is exploited. Security mechanisms at these layers will be crucial for securing safety-relevant applications. These applications will typically not be directly exposed to external threats, but an attacker may try to use vulnerabilities in other, externally facing, applications to gain control of cyber-physical features.

Software vulnerabilities may allow an attacker to gain control of legitimate processes and execute malicious code assuming the identity of an authorised user, thus fully utilising all of their privileges. To allow for secure simultaneous execution of applications and protection of data at rest, the default state of the system should promote security by minimizing its attack surface. This means running software with only the privileges, services, and resources it needs to perform its function.

2.2.4 Threat model

2.2.4.1 Potential assets

An asset can be anything that has some kind of value, and the value itself can be highly abstract. Parts of a system can be assets, where the system as a whole is also an asset with equal or greater value than its parts. Thus, the assets form a hierarchy which can be observed from different points of view. For a fleet owner, the fleet is an asset as well as each vehicle in the fleet. From this view, each vehicle is like a black box and assets within the vehicle are not easily identified. However, from the view of the OEM's, each vehicle is like a white box and can be divided further into lower level assets. Beside the physical view of assets, there is also a functional view. A vehicle feature is a typical functional asset; a feature can be disabled by default and later enabled when paid for, thus having a value and considered an asset. The HEAVENS project [28] has presented assets of the automotive domain in terms of activity, physical view, and functional view in table 2.1.

Activity	Physical view	Functional view
Computing	CPU or dedicated hardware accelerator	Processing task
Communication	Wired bus or network, wireless link	Send/receive messages on logical channel
Storage	Memory (RAM, ROM, flash)	Read/write data from/to address spaces
Acquisition	Sensors (wired, wireless)	Get measurements from the environment
Command	Actuators	Do actions on the environment
Implementation	Software and data	Implemented feature, functionality and service
Product	Vehicles produced by OEM	Provided services, features and OEM reputation
Back o□ce	OEM Back o□ce containing databases and application store	Diagnostics services, Software update, etc.

Table 2.1: Assets in terms of activity, physical view and functional view [28].

2.2.4.2 Security attributes

Security attributes are a set of attributes of on asset and/or an item that are used to define and enforce security requirements [28]. The CIA triad (Confidentiality, Integrity, Availability) has been a core principle of cyber-security for decades and these attributes are often referred to as the primary security attributes. However, cyber-security has evolved drastically in recent years and is nowadays one of the central concerns in almost every industrial domain. This has motivated the extension of the CIA triad to include additional security attributes. The HEAVENS project [28] consider eight security attributes, listed below, in their model.

- Confidentiality - The property of information not made available or disclosed to an unauthorized party.
- Integrity - The property of protecting the completeness and accuracy of data or an entity.
- Availability - The property of being accessible and usable when needed by an authorized party.
- Authenticity - The property that an entity can be verified and trusted as what/who it claims to be.
- Authorization - The property of access control, to access privileges granted to an entity or the process of granting an entity privileges.

- Non-repudiation - Also known as auditability, is the property of two parties not being able to deny processing of information sent/received.
- Privacy - A special form of the confidentiality attribute. Privacy applies to a set of information and an entity, and the property is guaranteed if this relation is confidential.
- Freshness - The property of uniquely identifying a message. This basically means adding a unique time stamp and can be used to prevent replay attacks.

2.2.4.3 Generic automotive threats

The EVITA project [29] has considered the following four high level security objectives:

- Operational – to maintain the intended operational performance of all vehicle and ITS functions.
- Safety – to ensure the functional safety of the vehicle occupants and other road users.
- Privacy – to protect the privacy of vehicle drivers, and the intellectual property of vehicle manufacturers and their suppliers.
- Financial – to prevent fraudulent commercial transactions and theft of vehicles.

Based on these security objectives, the EVITA project [29] has performed threat identification by using 'dark side' scenarios and attack trees to identify the following seven generic security threats for the automotive industry:

- Interference with safety functions of a specific vehicle
- Interfere with safety functions of many vehicles or tra□c management functions.
- Theft of vehicle information or driver identity, vehicle theft, fraudulent commercial transactions.
- Interference with operation of vehicle functions.
- Interference with operation of tra□c management functions or tolling systems.
- Avoiding liability for accidents, vehicle or driver tracking.
- Interference with operation of vehicle functions, acquiring vehicle design information.

3

Security analysis of the integrated architecture

This chapter details security issues related to centralisation of functionality in one or more integrated ECUs in an automotive system.

3.1 Issues related to centralization

Integrating functionality to the same physical hardware means that applications will be sharing resources, such as the memory, network interfaces, etc. This can have great implications for the security of the vehicle if access cannot be properly restricted. Some applications may store privacy sensitive data, such as locational information or driving patterns which must not be accessed by other applications. Thus, the system must implement a security policy that specifies who or what may access a specific resource and the type of access permitted. Further, it must be possible to ensure that one application cannot starve others of access to shared resources.

The degree of centralization and cooperation between applications will determine to what extent access to shared resources must be restricted. Simple ECUs, executing only a few applications with similar resource demands, may be su□ciently protected using only static resource allocation. However, a powerful ECU executing a diverse set of applications with varying resource usage will need more complex mechanisms, to ensure that system breaches can be contained. These mechanisms must support both fine and coarse grained specifications to allow access to be regulated at the level of individual entities and files, as well as classes of entities and resources [30].

3.1.1 Memory corruption

Separating access to physical memory is crucial to ensure spatial isolation of applications and limiting damage from potential security breaches. A benefit of the federated architecture is that only one application has access to the physical memory of the ECU. In an integrated architecture several applications must share the

same physical address space which means that mechanisms to manage memory accesses are needed. However, securing an ECU against malicious actors requires more sophisticated mechanisms than what safety standards require. Ensuring fault-containment simply means that applications must not be allowed to access regions of memory belonging to other applications [31]. Securing against intentionally malicious actors means that extra measures must be taken to also ensure that an application cannot corrupt its own assigned memory space. The reason for this is that an attacker may be able to abuse a vulnerable application such that only memory belonging to the application itself is corrupted to gain access to the system.

An attacker can gain remote code execution by abusing programming errors which allow writing more data to a buffer than what is actually allocated to it, causing subsequent memory addresses to be corrupted. *Stack smashing* attacks abuse such buffer overruns to overwrite memory addresses which belong to the applications call stack, allowing the attacker to modify the return address of a functional call to point to attacker-controlled data [32]. A variation of this attack is *return oriented programming* [33], where the return address is modified to point to subroutines already present in the executable memory of the process, such as linked library functions or system calls. Buffer overflow vulnerabilities can be mitigated by simply validating the size of the input data, but OEMs typically do not have access to source code of commodity software. Thus, protection against such vulnerable applications must exist at system level.

3.1.2 Data storage

In a federated architecture where each ECU only implements one application, information required by the application could be stored in non-volatile memory. The integrated architecture, along with the introduction of new functionality, will bring a greater need to store larger amounts of data, and to efficiently share data between applications. Filesystems are typically used to control how data is stored and retrieved. Some applications may store privacy sensitive data, such as locational information or driving patterns, in files which must not be accessed by other applications. Thus, the system must implement a security policy that specifies who or what may access a filesystem and the type of access permitted

3.1.3 Denial-of-Service

Protection against denial-of-service (DoS) attacks is crucial to uphold the temporal isolation of safety critical partitions and to ensure that applications cannot starve each other of system resources. Temporal isolation of partitions is typically achieved by running tasks on real-time operating systems with deterministic execution scheduling to guarantee each task sufficient computing time and resources to meet their timing constraints [10]. When tasks with hard real-time constraints are integrated on the same ECU as applications assuming only best-effort deliv-

ery, a compromised application cannot be allowed to interfere with the execution of another application by starving it of resources.

Resource exhaustion attacks are a form of DoS attack where a vulnerable application is abused to starve the rest of the system of some resource, such as memory, CPU-time, disk, or operating system descriptor tables. A *Fork bomb* is an attack which targets both CPU-time and the operating systems process-table by repeatedly creating new processes with the fork system call [34]. *Regular Expression Denial of Service* attacks are a form of algorithmic complexity attack which uses the exponential worst case execution times of *regular expressions* to exhaust CPU-time [35]. Attacks such as *SYN flooding* and *ping flooding* use network packets to exhaust the targets descriptor tables and bandwidth respectively [36]. These attacks are more effective when the attackers computational power and bandwidth is larger than that of its target. The integrated architecture will have one or more nodes with much greater computational power than the rest of the in-vehicle network which makes such attacks very powerful. A compromised process on a more powerful node may be used to flood the network such that other nodes will be denied communication.

3.1.4 Message injection

Trust between applications is crucial to ensure correct operation of any distributed system where the computational outcome depends on data passed in messages from other applications [37]. This is especially true for automotive systems where the safety of the vehicle and passengers depend on the authenticity and integrity of communicated data. If an application blindly trusts received messages an attacker who is able to tamper with messages or inject messages of their own choosing may be able to fool other applications to perform unsafe operations or gain elevated privileges on the system. Further, even if the senders authenticity can be verified, an attacker with access to old legitimate messages may use these messages to perform replay-attacks to fool the receiver.

3.2 Threat analysis

To derive threats for the integrated automotive EEA we assume an architecture with at least one centralized ECU integrating several applications. This integrated ECU is assumed to be connected to an in-vehicle network consisting of several other nodes. One example of such an integrated architecture is the domain-controller architecture, shown in Figure 3.1.

To identify specific threats for the integrated architecture, we assume that one application in the ECU has been compromised and consider how the issues identified in Section 3.1 could be used to spread an attack to realize the generic threats detailed in Section 2.2.4.3. For each identified specific threat, we determine which asset and security attribute it affects. To simplify the analysis, the only considered attributes

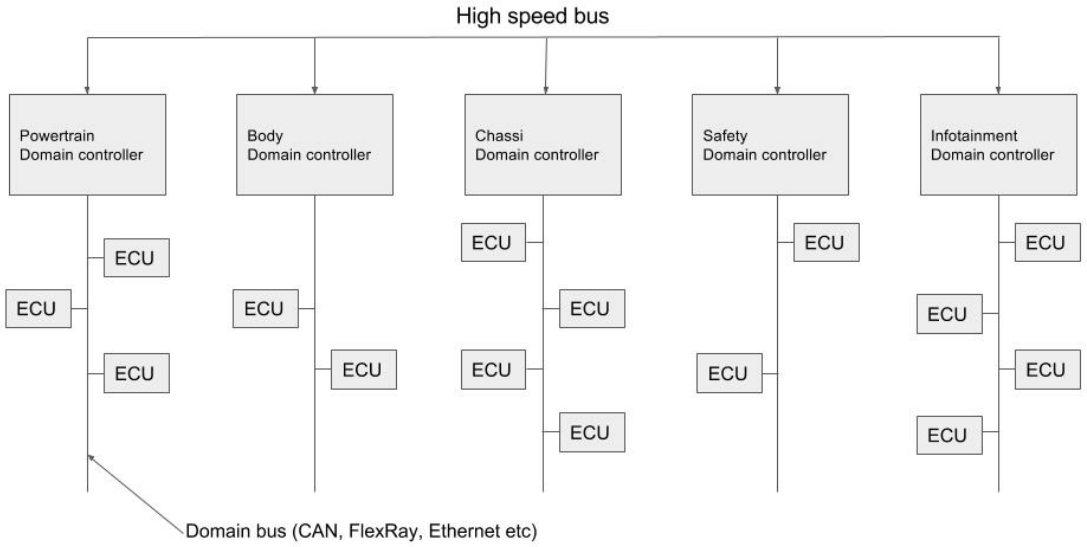

Figure 3.1: Domain controller architecture.

are Confidentiality, Integrity and Availability, which is a subset of the attributes described in Section 2.2.4.2. Because of the limited time available for this project, the only considered assets are computing, memory, storage, and communication. Further, we only consider the generic threats which a□ect one specific vehicle. The result is a set of specific threats, each with an associated asset-attribute pair. The aim of this threat analysis is not to identify all possible threats, but rather to identify a set of threats to fit within the scope of this thesis. The derived specific threats, grouped per generic threat, are presented below:

- Interference with safety functions of a specific vehicle.
  - T1: A compromised application may tamper with the memory space of a safety-related application.
    - Assets: Memory
    - Attribute: Integrity
  - T2: A compromised application performs a denial of service attack on a safety-related application.
    - Assets: Memory, Storage, Computing, Communication
    - Attribute: Availability
  - T3: A compromised application injects false sensor data on the in-vehicle network to a□ect a safety-related application.

- Assets: Communication
- Attribute: Integrity

- Theft of vehicle information, driver identity, or intellectual property.
  - T4: An attacker uses a compromised application to read stored privacy related information or intellectual property.
    - Assets: Storage
    - Attribute: Confidentiality
  - T5: An attacker uses a compromised application to eavesdrop on privacy or vehicle related information sent on the in-vehicle network.
    - Assets: Communication
    - Attribute: Confidentiality
- Interference with operation of vehicle.
  - T6: An attacker may use a compromised application to modify software or data related to the operation of the vehicle.
    - Assets: Storage, Memory
    - Attribute: Integrity
  - T7: An attacker uses a compromised application to inject false sensor data on the in-vehicle network to a□ect another application related to the operation of the vehicle.
    - Assets: Communication
    - Attributes: Integrity
  - T8: A compromised application performs a denial of service attack on another application related to the operation of the vehicle.
    - Assets: Memory, Storage, Computing, Communication
    - Attribute: Availability
- Avoiding liability for accidents or hiding vehicle tracking information.
  - T9: An attacker uses a compromised application to tamper with stored logging data.
    - Assets: Storage
    - Attribute: Integrity

– T10: An attacker uses a compromised application to inject false data to the OEM.

  - Asset: Communication
  - Attribute: Integrity

4

Security Mechanisms

In this chapter, we study mechanisms used in other domains to mitigate the threats identified in section 3.2. We investigate how these mechanisms can be used to protect assets with regard to asset-attribute pairs derived from the identified threats. Further, problems and security vulnerabilities related to each mechanism are investigated and advantages are detailed. Finally, the required hardware features of each mechanism is identified in order for us to later be able to determine the support for each mechanism in automotive embedded systems.

4.1 Access Control

4.1.1 Discretionary Access Control

The Trusted Computer System Evaluation Criteria (TCSEC), also known as the orange book, defines Discretionary Access Control (DAC) as “a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control)” [38].

The UNIX access control system, which has been a part of UNIX based systems for decades, is a widely used implementation of DAC [39]. The UNIX access control system has three types of permissions on an object; read, write and execute, and the owner of the object decides which permissions other subjects have.

DAC can be used to protect the confidentiality and integrity of storage by setting appropriate subject permissions on file objects. For example, if a subject does not have read or write permissions on a file, the confidentiality and integrity of the file is protected from that subject. The DAC permissions are typically stored in either Access Control Lists (ACLs) or Capabilities Lists. When using ACLs, each object has a list with subjects and their individual permissions, and when using Capabilities lists, each subject has a list with its permissions for every object. These lists need to be securely stored in memory so that no unauthorized process can overwrite them. To further enforce the access control policy, processes must execute in user mode

and access requests must be handled by the kernel.

The primary benefit of DAC is that it offers fine-grained control over system objects, which can be used to implement least-privilege access. The implementation of DAC is intuitive and can for the most part be invisible to users [40]. DAC is also a very dynamic policy since subjects can pass object-permissions to eachother without the involvement of a central authority.

In a typical UNIX system, a user is a passive entity for whom authorization can be specified and a process executes on behalf of a user. However, DAC does not make this distinction. When a process submits an access request, DAC only considers the privileges of the user who owns the process. This makes DAC vulnerable to processes executing malicious code exploiting the permissions of the user. In Linux systems, a process runs with the privileges of the effective user id, which can differ from the real user id when running setuid programs. In a setuid program, privileges can be escalated by temporary setting the user id to a user id with higher privileges, like root, to perform some privileged operations. If such a program can be exploited the attacker can gain total control of the system by acting as root [41].

Further, DAC does not enforce any control of information flow once a process acquires information. Because of this, processes can leak information to users who does not have the permission to read the information, breaking confidentiality [42]. Additionaly, verification of security principles for a DAC system is very difficult since users are allowed to share permissions of owned objects [40].

Hardware requirements:

- Dual-mode
- MPU/MMU

4.1.2 Mandatory Access Control

Mandatory access control (MAC) can be defined as any access control mechanism where the security policies are enforced by a central authority, and can not be affected by user actions [42, 40]. Unlike DAC, MAC makes the distinction between user and subject. Users are humans accessing the system while subjects are processes executing on behalf of users.

There are many implementations of MAC for different operating systems. For example, Android uses SELinux to enforce MAC on all applications [43]. SELinux is a security enhancement to Linux originally developed by the national security agency (NSA) [44], and based on its popularity and origin it will be used to exemplify the principles of MAC. In SELinux systems, access to resources must be explicitly granted. This means that no process can access any resource by default, thus the confidentiality and integrity of storage is protected by default. To enforce the access control policy, processes must execute in user mode and access requests must be

handled by the kernel.

Access is explicitly granted, with access rules, by the system administrator. The primary access control model in SELinux is domain type enforcement, where each subject belongs to a domain and each object has a class and a type [44]. Consider the following access rule:

```
allow user_t bin_t :  file {read write}
```

This rule allows any process who belongs to the user_t domain to read and write from/to any file object with a type of bin_t. Thus, configuring SELinux is mainly about applying this type enforcement across labels, and properly labeling subjects and objects [45].

MAC can be used to achieve complex functions, for example to create a sandbox. In SELinux systems, running the command sandbox will start the cmd application within a strictly confined SELinux domain [46]. The sandbox domain only allows read and write to the standard input and output descriptors by default. If specified, an alternative home directory with copies of chosen files can be created for the sandbox. The default SELinux policy does not allow the sandbox access to any other files, capabilities or networks access.

Another example is to use MAC to enforce a strict network policy. SELinux has labeled several network objects including interfaces, internet nodes, and ports, and access control can be applied to all of these objects to enforce a network policy. Consider the following SELinux rule:

```
allow user_t http_port_t :  tcp_socket {send_msg recv_msg name_bind}
```

This rule allows any process under the user_t domain the ability to bind to port 80 and also send and receive tcp messages on port 80. Note that if no network rules exist, the default SELinux policy would deny all processes any type of network capabilities. Thus, the integrity of communication can be protected using MAC.

MAC systems o□er strong added security assurances along with defence in depth, and the design allows for fine grained decision making [45]. Further, the permissions of a process do not depend on the user who owns the process, which make MAC systems less vulnerable to malicious processes compared to DAC systems. MAC can also be used to control the flow of information using multilevel security (MLS). The Bell-LaPadula model uses MLS to control the flow of information by assigning security levels to subjects and objects [40]. A subject may not read from an object with a higher security level and a subject may not write to an object with a lower security level.

In a MAC system with many subjects and objects, access control lookups can be expensive. To somewhat ease this problem, a software cache can be used to store the latest lookups and results [44]. Access requests would then first be checked against the cache before a lookup is made in case of a miss. Vulnerabilities in MAC systems are typically found in the policy itself or the applied rule-set. The rule-set can be

very complex and hard to implement and maintenance of the rule-set can also be very challenging. A single weakness in the policy or rule-set can compromise the entire system by opening up a significant attack surface [45]. A MAC system relies on the fact that the operating system kernel can be trusted [40]. This fundamental limitation is a problem when considering the large kernel attack surface, proving that MAC cannot be the sole defence of the system.

Hardware requirements:

- Dual-mode
- MPU/MMU

4.2 Static memory allocation with memory protection

Many simpler embedded microcontrollers in the automotive domain often comprises of a simple task-switching real-time operating system and its tasks, which together have been statically linked as a single monolithic entity. Thus, the memory layout is fixed at load time and represents the entire system. The memory architecture of these microcontrollers integrates some amount of RAM and ROM on the chip. The ROM is also known as *program memory* and is a non-volatile memory that primarily stores instructions, along with constants and static data values. The program memory typically comes in the form of FLASH memory (other available kinds are EPROM, EEPROM etc) which is read-only during runtime and requires a separate *reprogramming* procedure to write to. Upon boot, code is executed directly from the program memory without being copied to RAM before execution. The RAM is used to store mutable data to record changes in state as the program executes. In this memory architecture, the RAM and ROM are often uniformly mapped in different regions to the same address space along with other special purpose regions which allow programs to directly address the registers of on-chip peripherals and any external RAM or devices (a method known as memory-mapped I/O).

All the required memory is allocated at compile time, and the compiler divides the memory into different segments for code, read-only data, and read-write data. The code and read-only data segments are flashed directly to the microcontroller by loading it to the ROM, while the RAM is initialized during system boot. Because all the memory is statically allocated during compile time, the amount of system memory and its layout can be fully determined at load time. Thus, there is little risk of any runtime errors due to memory management, as the compiler ensures that the allocation cannot exceed the maximum available memory. Further, without the need for dynamic memory management it is possible to store code or data belonging to different applications/tasks in different contiguous memory segments.

A *memory protection unit* (MPU) can be used to divide the addressable memory map in a number of regions and to specify access permissions for memory addresses in each region. Most MPUs allows defining read and write permissions on data memory accesses, as well as whether instruction fetches are allowed within a region. Disabling instruction fetches from certain memory addresses is known as executable space protection and is one way to protect against buffer overflow attacks. Any attempt to access memory locations which are not permitted by the region settings will raise a memory management exception. The MPU can be used to [47]:

- Prevent user applications from corrupting data used by the operating system
- Separate data between processing tasks by blocking tasks from accessing others' data
- Define memory regions as read-only so that vital data can be protected
- Detect unexpected memory accesses (such as stack corruption)

An MPU is typically controlled by a number of registers which are used to configure regions. At minimum there will be registers which specify the region to be defined, the starting address of that region, the size of that region, and the access permissions for that region. The MPU must be reprogrammed by the operating system on each context switch to allow applications to access their code and data, as well as any other resources they need. However, this is done simply by writing to the MPU control registers which have low latency and a minimal impact on context switching.

The number of regions which can be defined, and thus the granularity of memory protection, is dependent on the MPU used. The ARM MPU also allows defining subregions and a default background memory map for privileged access only. This can be used to define several privileged regions by means of one background region, to protect privileged code (such as the OS kernel and exception handlers) and data. Overlapping regions are then defined which take precedent and gives user applications permission only to the regions they explicitly need access to. Thus, static memory allocation together with a MPU can protect the integrity and confidentiality of an applications memory since all memory accesses are checked against permissions set in the MPU. Since the memory is statically assigned at compile time it also offers improved Availability, as tasks/processes are not able to dynamically request more memory in order to starve other tasks of memory resources. Further, if external storage is accessed by memory mapped I/O, then the integrity and confidentiality of storage is protected.

Hardware requirements:

- Dual mode
- MPU

4.3 Virtual memory

Virtual memory is a memory management technique that has become an essential component of contemporary operating systems [48]. This technique allows separating virtual memory, as perceived by a process, from physical memory. Its three basic elements are the virtual address space, the main (physical) memory, and the auxiliary memory [49]. A process's virtual address space is its own virtual view of how it is stored in main memory [50]. This view is typically a contiguous range of virtual memory addresses, starting at address zero and differs from how it is actually stored in physical memory.

The physical memory is organized in chunks of linear memory addresses called memory *pages* and the pages which make up a process need not be stored contiguously in physical memory. Memory pages which have not yet been loaded in main memory, or that have been replaced by another page, are stored on disk (auxiliary memory). The operating system, together with the address translation hardware, is responsible for fetching a programs memory pages on demand, allowing the execution of a program that is not completely loaded in memory.

The main benefits of virtual memory are increased system utilization, due to being able to keep more processes in main memory, and to conceptually use more memory than is physically available. However, it also has the added benefit of increased security due to process memory isolation [48]. All modern operating systems rely on a model of execution where an instance of a program corresponds to the existence of one or more processes [48]. Information about each process is stored in kernel data structures known as *process control blocks* (PCB), which consists of the program code and associated data that describe the state of the process. Virtual memory provides process memory isolation by giving each process a unique non-overlapping virtualized view of memory. Each process is under the illusion that is has exclusive access to the full range of memory, limited only by the addressing scheme used by the system.

Processes reference virtual memory locations only, and the translation between virtual addresses and physical addresses is transparent to the process. This translation requires the support of hardware, in the form of a *memory management unit* (MMU) interposed between the processor and memory, through which each memory reference passes. The mappings between virtual and physical memory addresses are stored in *page tables*, which map virtual memory pages to equally sized physical page frames. The PCB of an individual process typically contains a pointer to the page table (typically stored in a cache or memory) storing the mappings between that process's virtual and physical address space. Upon a context switch to another process, the kernel informs the MMU of the location of the new process's page table, before execution is resumed. Unfortunately, this scheme has the effect of doubling the memory access time, because every virtual memory reference requires one memory access to fetch the correct page table entry, plus one memory access to fetch the desired data. This problem can be overcome by using a high-speed cache, called a

translation lookaside buffer (TLB), which caches recently used page table entries.

The operating system ensures that virtual addresses in different page tables never map to the same physical address (unless memory sharing is used). Whenever a new page is brought into main memory, it is either assigned a free page or it will replace an existing one. If it is assigned a free page this means that no page table entry maps to this page in memory. Otherwise, the page table entry pointing to the page to be replaced is updated to indicate that this page is no longer resident in memory (it now resides on disk), this ensures that only one page table entry points to this memory page.

Virtual memory protects the integrity and confidentiality of a process's memory because the operating system, together with the MMU, ensures that a process cannot reference memory locations outside its virtual address space. Thus, it is impossible to access memory belonging to other processes. Whenever a process wants to access a memory location, the kernel will interpret this location as a virtual address and perform address translation via the MMU. Because only the page tables entries for the currently executing process is used for the translation, the process cannot access pages that have not been explicitly allocated to it, since each page table entry will either point to a page allocated to the process, or be empty. Further, only the kernel, running in privileged mode, is allowed to modify the page table entries. This ensures that a user process cannot maliciously modify its page table entries to point to physical memory locations allocated to other processes.

Hardware requirements:

- Dual mode
- MMU

4.4 Packet filter firewall

Ethernet will play a crucial role in automotive communications and with TCP/IP already being an essential part of AUTOSAR [51, 52], in-vehicle networks can now be inspected in a more traditional way with the use of firewalls. Any host-based firewall inspects incoming and outgoing network traffic on a host and decides if the traffic should be allowed or blocked according to a policy. Most major general purpose operating systems, like Windows and Linux, implement a host-based firewall, based on packet filtering [53].

In a network consisting of single application ECUs, traffic could be filtered based on OSI layer 2 (data link) addressing, like for example MAC filtering, since only one application on each node can send or receive traffic. If several applications are integrated on the same ECU however, the node needs a way to direct traffic to specific applications. The most common solution is to make use of ports with the TCP/IP protocol.

Packet filter firewalls can be used in any network that depend on OSI layer 3 addressing [54], like the IP protocol. Tra□c to and from a host is filtered according to a rule-set which resides in kernel space and can only be modified in kernel mode. When considering TCP/IP tra□c the packet filter mainly operates on the network and transport layer of the TCP/IP stack and packets are filtered according to the following parameters [55]:

- Source and destination address
- Source and destination port (TCP or UDP)
- Protocol (identifying the type of data)

The firewall inspects every incoming or outgoing packet and compares the parameters mentioned above to the rule-set in order to decide if the packet should be allowed or denied. For example, one ingress (incoming packet) rule might be:

Source address	Source port	Destination address	Destination port	action
192.168.0.5	22	Any	Any	Deny

This rule would deny all communication from 192.168.0.5:22 with the node.

To protect the integrity of the network, no application on the node should be able to send packets claiming to originate from another node, known as spoofing. The firewall can prevent this by not allowing packets to be transmitted if the source address does not match the address of the node. Further, the firewall can control if an application is allowed to communicate with an application on another node by only allowing certain source and destination port pairs. Further, a firewall can also limit the maximum rate at which certain packets can be sent, preventing flooding of the network to maintain availability of communication.

The main strengths of a packet filter firewall is speed and flexibility [54]. The network can be partitioned to control the flow of tra□c. Packet filters have several weaknesses however; it is not trivial to configure the firewall to deny all unwanted tra□c due to the complexity of the rule-set and protocols. Further, a packet filter can not protect against attacks on layers above the transport layer nor can it provide any advanced user authentication schemes [54].

Hardware requirements

- Dual mode
- MPU/MMU

4.5 Message authentication

Message authentication is the process of confirming that the message comes from the stated sender, providing authenticity, and that the message has not been al-

tered, providing integrity. In addition, the message can be encrypted to provide confidentiality. These properties are crucial for secure communication in any type of network.

4.5.1 Message Authentication Code (MAC)

A Message Authentication Code (MAC) is a small piece of data which authenticates a message. The IPSec, TLS, and SSH protocol all make use of MAC to provide the necessary communication security. To generate a MAC, the sender uses a symmetric key and the message as input to a cryptographic function which generates a fixed size block of data (the MAC), which is then appended to the message. The receiver performs the same action, using the received message and their copy of the key, to generate a new MAC, which is then compared to the received MAC. If they match, the receiver can be sure that the message has not been altered, thus the integrity of communication is protected. A MAC is typically based on either a block cipher or a hash function.

Cipher-based MAC (CMAC)

A message authentication code based on a cryptographic block cipher is known as a CMAC. The National Institute of Standards and Technology (NIST) has issued a CMAC standard with two approved block ciphers, 3DES and AES, and a minimum key size of 128 bits is recommended [56, 57]. The length of the CMAC can be implementation specific and is a trade-o between security and performance, however NIST recommends a CMAC length of at least 64 bits.

Keyed hash MAC (HMAC)

A message authentication code based on a cryptographic hash function is known as a HMAC. NIST has issued a HMAC standard with seven approved Secure Hash Functions (SHA) which produce a HMAC of 160-512 bits, depending on which hash function is used [58]. A minimum key size of 128 bits is recommended [57].

The size of the message has a big impact on the performance of cryptographic hash functions as padding adds a constant cost. This leads to a large overhead for small messages when using HMAC [59]. However, hash functions generally execute faster in software compared to block ciphers. The computation of cryptographic functions is known to be expensive and support should be added through hardware acceleration.

MAC also provides authenticity of the message by proving that the sender knows the secret key. However, if more than two parties share the key the receiver can not be sure who sent the message, only that the sender is trusted. Anyone who possesses the symmetric key can both generate and validate a MAC, thus non-repudiation is not achieved since it can not be proven who generated the MAC, not even if only two parties share the key. An entity could, through hardware support, be limited to only validating MACs [60], thus improving security and in the special case where all parties but one are limited to validation non-repudiation is achieved. This is

also known as a symmetric system with asymmetric properties. To prevent replay attacks, timestamps and/or sequence numbers should be added to the message before authentication is made.

Key management is an important factor whenever cryptographic functions are used. A symmetric key should have an appropriate lifespan, and the system should be able to securely generate new symmetric keys. Whenever a new key is generated, it needs to be distributed in a secure way. One common solution is to use a public key infrastructure to distribute symmetric session keys. However, in a public key infrastructure one needs to be able to trust that the public key belongs to the entity claiming ownership. This can be achieved by using a trusted third party, like a public key authority, who can verify the ownership of the public key. Finally, symmetric keys need to be securely stored in the system. Storing them in memory is less secure compared to using a cryptographic hardware module.

Hardware requirements:

- Dual-mode
- MPU/MMU
- Cryptographic ISA extension or a cryptographic hardware module.

4.5.2 Authenticated Encryption

A MAC only provides authentication of the message and anyone who can eavesdrop on the communication will be able to read the plaintext message. To also protect the confidentiality of communication, the message has to be encrypted, also known as authenticated encryption. There are three approaches to authenticated encryption [61]:

- *MtE (MAC-then-Encrypt)* - MtE doesn't provide integrity of the ciphertext, only the plaintext, since there is now way of knowing if the message is valid until after decryption. The MAC doesn't provide any information about the plaintext. The SSL/TSL protocol uses this approach.
- *EtM (Encrypt-then MAC)* - EtM provides integrity of both plaintext and ciphertext and the MAC doesn't provide any information about the plaintext. The IPsec protocol uses this approach.
- *E & M (Encrypt-and-MAC)* - E&M does not provide integrity of the ciphertext, only the plaintext, since the MAC is performed on the plaintext. The MAC might reveal information about the plaintext. The SSH protocol uses this approach.

EtM has been shown to be the only generically secure method of the three and also has the advantage of verifying the message before decryption, saving resources in case the MAC is invalid [61]. Authenticated encryption increases the demand on

cryptographic hardware support.

4.6 Linux Containers

Linux Containers offer a form of virtualization known as *Operating System-Level virtualization*, which differ from traditional virtualization technologies in that the virtualized artifacts are global kernel resources, as opposed to hardware, and as a result they incur less CPU, memory, and networking overhead [62]. As such, OS-level virtualization is used to achieve increased availability and security by isolation in *High-Performance Computing Clusters* and distributed hosting data centers (such as Akami, Google App Engine, Heroku and Amazon EC2) [63]. It is also used for resource contrained environments, such as mobile and embedded devices [62]. Additionaly, a subset of the features used to build containers are used in ChromeOS and the Chrome Web Browser for sandboxing.

The linux kernel offers several components which together can be leveraged to isolate processes inside *containers*, appearing to run on isolated kernel instances while in fact sharing the same kernel [64]. These components can be combined differently to allow for a flexible range of isolation features, from imposing simple resource restrictions, to virtualisation of entire linux distributions, or full application sandboxing. Linux container solutions typically comprise of three major components:

- Capabilities allows fine-grained control of permissions and thus avoiding the use of `root`, which bypasses all permission checks, to perform privileged actions [65]. This allows practicing the principle of least privilege more easily.
- Kernel Namespaces provides process isolation by abstracting global resources to make them appear as a separate instance to processes within each namespace. They are the fundamental building block of linux containers and allow for isolation of resources such as network, processes, users, and the filesystems.
- Control Groups provides a hierarchical interface for managing hardware resource limits and device access, including CPU allocation and usage, block device I/O, and more.

4.6.1 Capabilities

Traditional UNIX systems distinguish between privileged and unprivileged processes when performing permission checks. Privileged processes run with effective user ID 0 (superuser `root`) and bypass all kernel permission checks, granting it full control over the system. On the other hand, unprivileged processes run with a non-zero user ID and are subject to full credential based permission checking. This approach has long been a source of security issues because of vulnerable setuid programs executing

as root [41]. A setuid program allows unprivileged users to run an executable file with the permissions of the owner of the file. When an unprivileged user needs to perform some privileged operations, it can run a setuid program owned by root to perform the task. Ideally, the program should only offer a temporary elevation of privilege by performing the privileged instructions as soon as possible and then drop all elevated privileges. However, if the program has a vulnerability while running with root privileges, an exploit may grant root access to an unprivileged user. The shortcomings of this model is that programs which need only some privileges is afforded full privileges on the system [66]. This means that the attack surface for root privilege escalation extends across all processes running as root, all setuid-root binaries, and any libraries they interact with [45].

Capabilities divide the privileges traditionally associated with the superuser root into distinct units, allowing finer control of privileges [65]. The key insight to this model is that programs, not users, exercise privilege. Therefore, capabilities are a per-thread trait, so can be used to execute programs with only the privileges which they need to execute. This is in contrast to setuid programs which grant processes all the privileges afforded to the owner of an executable file, even though the process may not be acting on behalf of the user. The capabilities available to a process during runtime is determined from its *thread capability sets* and the *file capability sets* associated with the executed binary.

The three main thread capability sets are:

- Effective: The set of capabilities used by the kernel to perform permission checks for the process. When capabilites are enabled and the process performs a privileged operation, the kernel will no longer check that the user ID of the process equals zero. Instead it checks the process' effective capability set for the appropriate capability.
- Permitted: A limiting superset for the effective and inheritable capability sets, indicating which capabilities a process is allowed to use. If a capability is a member of the permitted set but not the effective set, it means that the process has temporarily dropped that capability. Processes are allowed to add members of the permitted set to the effective set, but can not regain capabilities dropped from the permitted set.
- Inheritable: The capabilities inherited by a new process, when executed by the current process. (i.e. the set is preserved across a call to execve(2)).

Executable files have the same conceptual capability sets. Together with the thread capability sets of the process executing the binary, they determine the process' new thread capability set upon execution. Note that the thread capability sets are simply copied to the child upon a call to fork(2), the new set is only computed when executing a binary. The file capability sets are as follows:

- Permitted: These capabilities are added to the threads effective capability set automatically, regardless of all other capability sets.

- Inheritable: The intersection of this set and the threads inheritable set is added to the threads permitted set, along with the the file permitted set (i.e. only the inheritable capabilies which are present in both inheritable sets are allowed to the thread).
- E□ective: Actually just a single bit, indicating whether the threads computed permitted set should be added to the e□ective thread set upon execution (i.e. if set, the thread will acquire all its permitted capabilities).

File capabilities reduces the risk associated with running `setuid` programs, instead assigning only the minimal set of capabilities needed to an executable binary. Only processes with the `CAP_SETFCAP` capability can add file capabilities. However, once a process has started executing, its permitted capability set can only be reduced, not increased. Further, it is possible to restrict the use of executable binaries to only certain users, by granting inheritable capabilities to individual users. Now, instead of adding the required capabilities directly to the permitted set of the executable file, the capabilities are added to the inheritable set only, and the e□ective bit is set. This ensures that the required capabilities are added to the threads permitted (and thus e□ective) set only if the process executing the file has the required capabilities in its inheritable set (per the rules for computing the thread permitted set). Thus, when a user without the inheritable capabilites executes the file, the process's e□ective set will be empty and the privileged operation denied.

4.6.2 Namespaces

Linux kernel namespaces are logical constructs dealing with scope and segmentation of operating system resources, which allow creating di□erent userland views. Namespaces wrap global resources in an abstraction, such that processes within a namespace will appear to have access to their own isolated instance of the resource. [67]. Currently, there are six di□erent namespaces which all running processes share initially. Calling the `clone(2)` system call with accompanied `CLONE_NEW` flag splits the global resource identifier tables, creating partitions which provide processes a unique view of the resource. Processes can join namespaces using `setns(2)`[68], while `unshare(2)` [69] allows creating new namespaces without creating a new process.

4.6.2.1 UTS Namespace

Provides domain and hostname isolation, allowing processes in di□erent namespaces to appear to have di□erent domain and hostnames. While not being very security relevant, it o□ers functionality crucial to many web services.

4.6.2.2 Mount Namespace

Provides per-process isolated views of the filesystem hierarchy. UNIX's "everything is a file (descriptor)" approach means I/O resources, inter-process and network communications are exposed as byte streams through filesystem mount points. A newly created mount namespace is initialized with a copy of the parent filesystem, but any subsequent mounts or unmounts do not propagate to the parent. This means that processes in different mount namespaces can mount or unmount hardware devices, virtual filesystems, etc without affecting the filesystem view of processes in other mount namespaces.

4.6.2.3 IPC Namespace

Isolates inter-process communication resources identified by means other than filesystem pathnames. Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue filesystem. Objects created in an IPC namespace is visible to all members of the same namespace, but not to processes in other IPC namespaces. Thus, inter-process communication is restricted to process within the same namespace. Note that this namespace does not isolate any shared memory regions, as Linux creates and mounts shared memory segments as objects in the virtual filesystem.

4.6.2.4 Network Namespace

Provides isolation of system networking resources such as network devices, IP protocol stacks and routing tables, firewall rules, and more [67]. Each physical network device can only exist in one namespace, but virtual network device pairs can be used to connect processes in different namespaces or to create a bridge to the physical network device in another namespace.

4.6.2.5 PID Namespace

Protects against cross-application attacks and information leaks between processes in different PID namespaces [45] by allowing processes in different PID namespaces to use the same process identifier [70]. PID namespaces can be nested to create fully isolated process trees where the first created process in each namespace will be the "init" process with PID 1. Processes in a PID namespace is only visible to other processes in the same namespace and to processes in the direct ancestor. Therefore, system calls which target other processes by specifying a process ID (such as `kill(2)`, `setpriority(2)` or `ptrace(2)`) can only be used on processes in the same namespace or on processes in descendant namespaces. Further, any call to `reboot(2)` will now signal to the "init" process of the targeted namespace (instead of rebooting the system) and will only affect its descendant processes.

4.6.2.6 User Namespace

This is the most recently introduced namespace, and perhaps the most important one with regards to container security. User namespaces isolates security-related attributes such as user and group IDs, root directories, and capabilities between namespace instances [71]. This allows running processes in *unprivileged containers*, in which processes can have an unprivileged user ID outside its associated namespace while having user ID 0 (root), and thus full privileges, inside the namespace. This means that if an attacker manages to escape the container they will find themselves running only with the capabilities a□orded to the "nobody" user outside the container. User namespaces can be nested, such that each newly created user namespace creates a parental relationship where the parent of the new namespace is the user namespace where it was created.

User namespaces are implemented as mappings from user and group IDs insde a user namespace to a corresponding set of IDs in its parent user namespace. The mappings define a range of user and group IDs internal to the namespace to be mapped to an ID range of the same length in the parent user namespace. Essentially, each user namespace is assigned a one-to-one mapping of some range of IDs in the initial user and group ID range which are shifted inside the user namespaces, such that each user namespace can have an internal ID 0 which map to di□erent non-zero IDs in the initial range. Whenever a process performs an operation requiring permissions based on user or group ID the kernel maps the internal ID to the ID in the initial range to perform the permission check. For instance, when a process creates a file, the kernel assigns an ID in the initial ID range as the file owner. Further, when a process executes a set-user-ID program, its e□ective user ID is changed to the mapped value in the initial ID range. However, if no such ID is mapped for that particular user namespace, the e□ective ID is left unchanged. This ensures that a process cannot execute a set-user-ID program to gain the privilege of a user ID outside its ID range in the initial user namespace.

An individual process is a member of exactly one user namespace and can reassociate to another user namespace only if it has the `CAP_SYS_ADMIN` capability in the target namespace [68]. Processes created in a new user namespace start out with a complete set of capabilities in that namespace. The same is true for existing processes joining newly created or existing user namespaces. Further, any capability that a process is a□orded in a user namespace, it is a□orded in all descendant user namespaces as well. However, all capabilities in the current namespace are dropped whenever a process joins, or is created in, another user namespace. Thus, even if the new namespace is created or joined by a process with `root` user ID in the initial namespace, it will no longer have any capabilities in the previous or parent namespace. Therefore, a process can never have any capabilities in an ancestor user namespace. This also protects against processes trying to gain elevated privileges in the current namespace by leaving it and later rejoining. Because the `CAP_SYS_ADM` capability is dropped when leaving the namespace, any attempt to rejoin the namespace will be denied. Further, processes are not allowed to join their already associated user namespace, as this could be used to regain any dropped privileges.

Creating new non-user namespaces require the CAP_SYS_ADMIN capability. However, this is not the case for user namespaces, which can be created by any unprivileged process. As processes start out with a complete capability set in their new user namespace, they are free to create non-user namespaces inside their user namespace. A newly created non-user namespace is owned by the user namespace in which the process that created it resides at the time of creation. The interpretation of effective capabilities is changed when applied inside user namespaces. A process has a capability in a particular user namespace only if it is a member of said namespace, or one of its ancestors, and its effective capability set includes the capability. Further, the capabilities afforded to a process within a certain user namespace only permit it to perform privileged operations on resources governed by the non-user namespaces associated with that user namespace. This is because the kernel will now perform all permission checks for global resources governed by a non-user namespace against the capabilities in the associated user namespace. Therefore, the isolated view of global resources which non-user namespaces provide can only be modified by privileged processes in the owner user-namespace (or its ancestors) of each non-user namespace.

Consider creating a new user namespace from the initial user namespace without creating any new associated non-user namespaces. A process running in this user namespace will not be able to perform any privileged operation on any global resource, even though it has a full set of capabilities in its user namespace. This is because it shares all non-user namespaces with the initial user namespace, in which it has no capabilities. If it wants to mount a filesystem it must create a new mount namespace, or if it wants to change its hostname it must create a new UTS namespace, and so on. Furthermore, there are some resources which are not yet namespace aware, on which privilege operations require privilige in the initial user namespace. For example, loading kernel modules, setting process priorities, and creating device nodes.

In conclusion, user namespaces allow unprivileged users to use functionality otherwise limited to the root user, while limiting the scope of that privilege to a user namespace. Thus isolating the effects of said functionality from the runtime environment of the wider system.

4.6.3 Control groups

Control groups [72] are a kernel feature which allow organizing processes into hierarchical groups (known as cgroups), and applying hardware resource limiting and monitoring, as well as access control, to these resources on a per-group basis. Thus, cgroups isolate and limit resource usage over a group of processes, to control system performance or security [45]. Limiting and monitoring of resources is implemented in a set of per-resource-type subsystems, known as resource controllers, that modify the behaviour of processes in a cgroup attached to a given controller. When attached to a cgroup, controllers allow for control of how processes in the cgroup can use the associated resource. For instance, it is possible to control things such as

memory usage and allotted CPU time, or to restrict the processes to execute only on a specified set of processor cores.

Cgroups form a hierarchy, where the limits defined for a cgroup at one level will generally affect the descendant cgroups at lower levels. This ensures that limits placed on a cgroup at a higher level cannot be exceeded by any descendant. Therefore, limits for cgroups at lower levels can only be constrained further, never extended. Further, the rules for how cgroups and controllers are allowed to be associated ensures that there is only one way that a process can be limited or affected by a given controller. The Linux kernel provides twelve controller subsystem, each representing a single resource. Different cgroups can be associated to different instances of each controller subsystem, to assign different limits per cgroup. The following are the controller subsystems relevant for this thesis:

- cpu: Controls scheduling of CPU access to cgroups by assigning either relative CPU shares, or ceiling enforcements [73]. Assigning relative shares means that each cgroup will receieve a share of CPU time equal to its share of the sum of shares for all cgroups on the system. Thus, the actual amount of CPU time available to a cgroup will depend on the number of cgroups on the system and their relative shares. However, these limits only apply when the system is busy. When CPU shares are used, a cgroup may receive additional CPU time whenever there are idle CPU cycles. If hard limits are needed for the amount of CPU time a cgroup can utilize, ceiling enforment can be used. This allows specifying an upper limit of CPU time allocated to a cgroup during each scheduling time period. Therefore, when ceiling enforcement is used, a cgroup cannot use more than a fixed amount of CPU time even if there are idle CPU cycles available.

- cpuset: Facilitates assigning individual CPUs and memory nodes to cgroups (this includes all logical processing units on which a process can execute). This can be used to restrict processes to execute only on a subset of available CPUs and memory nodes. Further, it is possible to give a cgroup hierarchy exclusive access to its assigned CPUs and memory nodes. This means that no other cpusets other than direct ancestors and descendants can share the CPUs or memory nodes specified for the cpuset.

- memory: Supports limiting of memory allocation and swap space usage to processes in a cgroup. By default, if processes tries to consume more memory than they are allowed, they are immediately terminated. Optionally, this feature can be disabled such that processes are paused until memory has been freed.

- devices: Can allow or deny access to devices by processes in a cgroup. Is typically used to implement a whitelist of devices from the *Linux Device List* which members of a cgroup is allowed to access. Further, it is possible to specify whether the access type should read, write, or both, and whether a cgroup is allowed to create new devices.

- blkio: Controls I/O devices to limit bandwidth or access on block devices (for example disks) by processes in cgroups. Two policies for I/O limiting is available. *Proportional weight division* allows setting weights to specific cgroups, thus allocating a percentage of all available I/O operations per cgroup. *I/O throttling* sets an upper limit on I/O rates, either as the number of operations per second or as the number of bytes per second.
- net_cls: Allows tagging network packets with a class identifier which can be used to identify from which cgroup a network packet originates from. This in turn can be used in firewall rules to filter or limit packets sent from different cgroups.
- net_prio: Provides a way to dynamically set the priority of network traffic originating from various cgroups, for each network interface. The priority is used internally by the system and network devices to decide which packets are sent, queued, and dropped.
- pids: Can be used to limit the maximum number of processes that may be created in a cgroup hierarchy. Helps to protect against fork bombs [45].

4.6.4 Security offered

The advantages, in terms of security, offered by Linux containers is the ability to greatly reduce attack surfaces and to isolate applications, and thus attacks, to only the required components, interfaces, libraries, and network connections [45]. This can be achieved with negligible impact on performce, when compared to running Linux without containers.

Integrity and Confidentiality of the filesystem (Storage) is protected (beyond simple Discretionary Access Control) through the use of separate namespaces, along with a least privilege capabilities usage. The mount namespace gives processes inside a container an isolated view of the filesystem, thus allowing to isolate access to sensitive files between containers. The use of different network namespaces allow to completely deny processes in a namespace access to any network devices, or to specify a more fine grained policy through the use of per namespace firewall rules, thus protecting the Integrity of communication. This means that a process can be completely restricted from the network, or to only be allowed to communicate with certain other nodes on the network. Thus, network namespaces allow specifying a per-process least privilege model on communication, ensuring that an attack can only spread to any nodes which the process is allowed to communicate with.

Further, user namespaces and unprivileged containers offer an added layer of protection against privilege escalation attacks and container escapes. A container escape occurs if an application interacts with the host or another container in a manner not intended. When using user namespaces, in the event of a container escape, the compromised process will have no privilege outside the container, and as such the

harm it can inflict is limited to that of the nobody user. Therefore, the compromise of a privileged process running inside a container is isolated to that container.

Integrity and Confidentiality of memory is protected due to the Linux kernel's use of virtual memory and the process abstraction, protecting memory access to the memory space of another process. Again, user namespaces add another layer of protection of the memory, as processes running with privilege to access the memory space of other processes (f.e. `root`) will not have those privileges outside of its container. A process can use the `ptrace(2)` syscall to read or manipulate the memory space of another process only if it is executing with `root` privileges in the initial user namespace or if it has the `CAP_SYS_PTRACE` capability in the user namespace of the target process. Because a process only has any capabilities in its own user namespace, such privileged processes are only allowed to access memory of other processes inside the same container.

Additionally, Linux Containers can protect the Availability of computing, memory, storage, and communication through resource management with cgroups. Setting limits on the amount of disk I/O, memory, and CPU time that processes are allowed to use mitigates denial-of-service attacks that aim to exhaust these resources [74]. However, the *blkio* cgroup only allows limiting of disk I/O bandwidth, it does not allow setting any limits on the total amount of disk usage. Instead, using separate disk partitions mounted in different mount namespaces protect against denial-of-service attacks where a compromised application tries to exhaust the entire disk space, further protecting the availability of storage.

Denial-of-service on the network can be mitigated by tagging network traffic with class identifiers and priorities. The class identifiers can then be used to filter or limit egress traffic in the Linux kernel firewall, which can mitigate attacks where a compromised process is used to exhaust network bandwidth to disrupt other nodes on the network. Setting priorities on the network traffic can protect from denial-of-service attacks where an application tries to exhaust network bandwidth from other processes on the same system. Traffic with higher priority will take precedence in the send queue, and if a network device is exhausted the low priority traffic will be dropped from the send queue.

4.6.5 Container threats

The Linux kernel features which comprise Linux containers are relatively new, and still incomplete, when compared to hardware virtualization. Grattafiori [45] has performed an extensive examination of container attack surfaces, threats and hardening techniques which highlight the fundamental risks associated with the shared kernel and the incomplete implementation of kernel namespaces. Further, Hertz [75] discuss several security pitfalls intrinsic to the design of the kernel container features and analyze historic container attacks to the highlight the complexity and attack surface of the container system.

Clearly, the shared kernel is the fundamental risk of using linux containers to isolate applications, as sharing the kernel also means that any kernel vulnerabilities are also shared. Linux is a general-purpose operating system exposing roughly 400 system calls to user space applications, this is a large amount of code which many applications might not need to fulfill their functionality. The attack surface of the kernel is huge, and several of these system calls have been vulnerable to privilege escalation attacks. Additionally, loading of outdated or obscure networking kernel modules have resulted in many trivial privilege escalation attacks. Kernel vulnerabilities have been discovered in several other features, such as filesystem implementation, the crypto API, and device drivers (see [45] for a list of CVE links of such vulnerabilities). To reduce this attack surface, container platforms (such as LXC and Docker) allow the use of system call filtering to restrict which system calls are allowed to be called by processes inside a container. `seccomp-bpf` is a system call filtering method which allow implementing either a whitelist or blacklist of system call IDsr. This can reduce the kernel attack surface for application to only the needed system calls.

Capabilities are meant to split the role of `root` into smaller pieces, to make it easier to practice a least-privilege model. Unfortunately, capabilities are still under development and the capabilities model is not entirely intuitive. Kernel developers are encouraged to associate any new kernel features with existing capabilities, instead of creating new ones, to keep the number of capabilities manageable [65]. This has led to a capabilities model where the role of `root` is split into quite large pieces, which can be confusing and hard to use correctly. This is highlighted by Spengler [76], who details how 19 of the available capabilities can be used to gain full `root` privileges. For example, the `CAP_SYS_MODULE` allows to modify the kernel at will, by loading arbitrary kernel modules, e□ectively subverting all system security. The `CAP_SYS_ADM` capability is the most egregious avenue for privilege escalation, as it has become a catch-all capability for kernel developers [65]. These issues highlight the need to drop all unnecessary capabilities, as they are complicated and may o□er more privilege than perceived. An overview of all capabilities and potential abuses are given in [45].

Another issue is that the namespace implementation is also incomplete, meaning several kernel features are not yet namespace aware and can be leveraged to allow information exposure or trivial container escapes [75]. Examples of such features are the `devfs`, `sysfs`, and `procfs` virtual filesystems, which act as interfaces to internal kernel data structures. These filesystems allow reading information about kernel subsystems, processes, hardware devices, etc., and to configure certain kernel parameters. This greatly increases the attack surface to compromise the host or other containers. One example of an attack using these features involve changing the location of a utility that the kernel will call when certain events occur [75]. By changing this location to point to a file within the container, an attacker can gain remote code execution outside the container. Various other files within these filesystems can expose information used for information gathering, or to perform other attacks.

This incompleteness of capabilities and namespaces highlight why the user names-

pace is a great security advancement for containers. Prior to the introduction of user namespace, all containers were privileged containers where `root` inside a container equals `root` on the host. When using privileged containers, protecting the host relies on dropping of privileges, Mandatory Access Control of non-namespace aware resources, and system call filtering. These containers are not considered root-safe by the developers [64]. When using unprivileged containers (facilitated by user namespace), Capabilities, Mandatory Access Control, and system call filtering are used as defense-in-depth in the case of kernel security issues, but the security model no longer relies on them. This is because capabilites now only has meaning within the user namespace of the process, and modifying non-namespace aware resources such as `procfs` requires capabilities in the initial user namespace. However, user namespaces are relatively immature and due to their sensitive nature and large code base, since their addition to the kernel several serious vulnerabilities have been discovered. Further, although a privileged process within a container is unprivileged on the wider system, that process now has access to potential exploits in kernel code previously only accessible by `root`.

4.6.6 Hardware requirements:

- Dual mode
- MMU

4.7 Hypervisors

Hypervisors are relatively small control programs placed between one or more *hosted* operating systems and the physical hardware which enable the abstraction, or *virtualization*, of hardware computing resources into several different execution environments. These operating systems are referred to as *guests* and are hosted on the Hypervisor in a sandboxed environment called a *Virtual Machine* (VM) which appear to the guests to be native hardware. The VM behave towards the guest as native hardware, giving the illusion that the guest is executing on an isolated instance on its private hardware but can, transparent to the guest, limit, manage, and protect it and its resources while multiplexing the physical resources between several VMs. Access to the shared physical resources are arbitrated by the Hypervisor similarly to how a traditional OS manages concurrent execution of processes.

Virtualization is a proven technology which have been instrumental to the server and data center markets for decades by enabling server consolidation through the ability to concurrently execute multiple OSs [77]. One of the key features of server virtualization technologies is the ability to provide securely isolated (sandboxed) environments for the execution of untrusted software. The resources are abstracted such that each VM is almost entirely isolated from every other VM running on the Hypervisor. This abstraction is achieved with a code base which size is only a frac-

tion of that of a traditional operating system [78, 79]. This ability to isolate guests allows addressing the strictest legal, security, and safety design concerns when implementing a system. Due to this and other beneficial features, major mobile and embedded technology providers have started using virtualization in their devices. Additionally, there are now several embedded hypervisors available from major embedded software providers, such as Wind River Systems, Green Hills Software, and others.

The level of abstraction needed to present guests with an exact duplicate of the underlying system while managing the software complexity and performance overhead is specific to the architecture, hardware, and guest OSs. There are two main approaches used to achieve this abstraction:

- Full Virtualization: The guest assumes native control of handware and is unaware that it's being controlled by an underlying Hypervisor. This requires no modification of guest OSs and is typically implemented with the *trap-and-emulate* approach. All instructions with potential to impact the Hypervisors control of resources are trapped and handled by the Hypervisor. This approach o□ers the best isolation and security for VMs.

- Paravirtualization: The guest OS is modified to work with the Hypervisor through *hyper calls* which the Hypervisor exposes in an interface to the guests. The intent of this approach is to work around the overheads associated with Full Virtualization, and have thus been traditionally used for embedded Hypervisors. However, it lacks the portability o□ered by full virtualization, as guest operating systems must be modified to work with the paravirtualized VMM. This also implies a lower degree of isolation o□ered, compared to full virtualization because guest operating systems are aware of the VMM. Additionally, many architectures now include hardware assistance to overcome these overheads. Although no extensions to the ISA are technically needed to support virtualization with this approach„ many of the now available hardware assists can be used with paravirtualization to further increase performance.

Hypervisors, also known as Virtual Machine Monitors (VMM), can be classified in two general forms; type 1 and type 2 hypervisors. Type 1 hypervisors are operating-system-like software built specifically to support virtualization of other operating systems and are installed as the primary boot system with native control of hardware. Type 2 hypervisors operate within the context of a host operating system and provide virtualization features to run guest operating systems. Because of the added overhead of type 2 hypervisors, in terms of performance and security, these are not considered in this thesis.

To cost e□ectively implement e□cient virtualized systems require support and acceleration in the hardware running the Hypervisor. Memory management in particular is troublesome to virtualize without hardware assists and can have severe impact on system performance and reliability [77]. I/O management is another obstacle which hardware assistance can speed up while o□ering increased protection for VMs.

Popek & Goldberg [80] formalized the requirements and characteristics of an *e cient* Virtual Machine Monitor by describing a set of architecture features of the physical machine su cient to support e cient system virtualization. These requirements define three properties of interest for an arbitrary application running under the control of a VMM, and still guide virtualization e orts to this day:

- Equivalence (Fidelity): Software on the VMM executes identically to its execution on hardware, barring timing e ects..
- E ciency (Performance): An overwhelming majority of guest instructions are executed by the hardware without the intervention of the VMM.
- Resource control (Safety): It must be impossible for that arbitrary program to a ect the system resources, i.e. memory, available to it; the allocator of the control program is to be invoked upon any attempt.

Thus, in order to build a VMM supporting system virtualization, all instructions which may a ect the VMM itself must be able to be trapped and handled (emulated) by the VMM, in order to fullfil the *Resource control* property. An architecture which can be virtualized purely with the trap-and-emulate approach is said to be *classically* virtualizable [81]. For an instruction set architecture (ISA) to be classically virtualizable it must support a clean separation of privileged and unprivileged instructions. This means that an instruction must either behave equally in user mode and supervisor mode, or an exception must be thrown when used in user mode to allow trapping to the VMM. *Privileged* instructions are those which trap when the processor is in user mode and that do not trap when the processor is in supervisor mode. *Sensitive* instructions are those that attempt to change the configuration of system resources or that illustrate di erent behavior or result depending on the configuration of the system. Architectures such as PowerPC, Motorola 68010, and IBM S/370 do not include any unprivileged sensitive instructions, and can therefore be classically virtualized without any architecture extensions.

However, some architectures do not have a clean separation of instruction privilege, as they were designed before the concept of virtualization was proposed. The x86 and ARM architecture, for instances, were historically not virtualizable because of sensitive instructions which do not generate traps. They contain instructions which return di erent values depending on the mode in which it was executed, which could allow the guest to observe that it was de-privileged. An example of this is the x86 `popf` instruction which replaces all the flags in the flag register with the contents of the stack when executed in privileged mode, while only replacing some of the flags when executed in user mode. A de-privileged guest's attempt to replace the remaining flags are simply suppressed without generating a trap, rendering the trap-and-emulate approach useless. These issues have been solved by extending the architectures with hardware features which typically add one or more modes of operation to the processors previous dual modes of operation. This allows to fully virtualize the CPU and fullfil the properties of a classically virtualizable system. Sensitive operations, such as reading the flag register, can then be emulated in a virtual CPU and will therefore not a ect the host system.

Because privileged instructions must be trapped and handled in the VMM they create extra overhead and cause the guest to run slower than it would natively. The e□ciency property says that all innocuous (unprivileged) instructions must be handled natively by the hardware, without intervention by the VMM. This property is what di□erentiates system virtualization from full emulation, where both privileged and unprivileged instructions are intercepted by the emulator. The performance issues of using trap-and-emulate is exacerbated by privileged data residing in memory, such as guest *page table entries* which store mappings and permissions for memory pages. If the architecture supports virtual memory the guest OS cannot be allowed direct access to the MMU, since the VMM would no longer be in control of all system resources. The guests cannot be allocated true physical memory, which only the VMM is allowed to manage, instead they are allocated intermediate addresses to reference. Therefore, one extra stage of address translation is needed to map guest virtual addresses to host physical addresses. If the hardware provides only one stage of address translation the VMM must manage the relationship between these in software. This is typically implemented by maintaining *shadow page tables* which store information about the physical location of guest memory. The VMM must ensure that changes to the guest page tables are reflected in the corresponding shadow page tables. To maintain coherency between guest page tables and shadow page tables, accesses to privileged in-memory data must trap to the VMM. The perfomance overhead associated with maintaining coherency of the shadow page tables were big enough that the first generation of x86 hardware extensions, which enabled full virtualization, performed worse than pure software virtualization [81]. Second Level Address Translation (SLAT) is a hardware feature providing MMU virtualization which eliminates the need for maintaining shadow page tables in software and the performance overhead associated with keeping them up-to-date [82]. It extends the CPU page-table walking function to include an extra intermediate address layer and when a virtual address is accessed the hardware walks both levels to perform the translation from guest virtual to host physical address. This composite translation eliminates the need for software shadow tables, but add a performance penalty to TLB misses. However, this penalty is low compared to the gains from eliminating the added traps to the VMM [82].

Another obstacle which must be overcome to meet the resource control property is that of I/O management, especially if the system allows interaction with direct-memory-access (DMA) capable devices. The DMA controller allows I/O devices to transfer data to or from its own bu□er storage directly to memory. This happens without intervention by the CPU, which is instead notified via an interrupt once the transfer of a block has completed. The issue, again, is that the DMA capable devices are not aware of the mapping between guest-physical and host-physical addreses. If a guest initiates a DMA transfer using its perceived physical address this might corrupt the memory of the VMM or other guests. Therefore, address translation is required when initiating DMA transfers. This translation is possible to manage in software but (just like with guest virtual address translation) adds significant overhead and also requires device driver porting [77]. Further, a software only solution is still susceptible to physical memory corruption by erroneous or malicious

DMA capable devices. A DMA attack is a physical side-channel attack which uses an accessory with maliciously constructed device driver to gain access to physical memory through DMA. An IOMMU is a hardware feature which provides hardware assisted DMA and adds a level of indirection to DMA by applying the concepts of virtual memory to it. It performs the address translation without VMM intervention and also ensures that interrupts are delivered only to the correct guest, allowing DMA transfers to be passed through between guest and device transparently to minimize performance overhead. It also protects against faulty or malicious devices by explicitly assigning memory regions which devices are allowed to read or write from.

The properties of system virtualization and the use of a Hypervisor o□ers improved system protection in terms of Confidentiality, Integrity, and Availability through the high degree of isolation achieved. Software running in di□erent virtual machines, on the same hardware, are considered more isolated than when running on a shared OS because of the properties of a VMM and because VMMs typically have a much smaller trusted code base. The smaller code base, and thus reduced attack surface, means that the VMM is less complex to prove secure and o□ers a higher level of containment which must be breached to break the isolation. The resource control property ensures that the VMM can observe or intervene in everything the guests does to ensure Integrity of virtual resources, including Memory, Storage, and Communication. Confidentiality of memory and Storage is ensured as the resources are isolated between VM instances. Further, because the VMM has full control of all resources, it can manage the resource usage for individual VMs to protect against cross-VM resource exhaustion attacks and o□er improved Availability of Memory, Storage, Computing, and Communication. The VMM can ensure that software running in one VM does not exhaust software in other VMs of their resources, either by partitioning the resources beforehand, or by setting limits on the amount of resources or the frequency of operations on a per-VM basis.

Hardware requirements:

- Dual-mode
- Virtualizable ISA / CPU Virtualization support
- MPU / MMU
- SLAT (nested page tables)
- IOMMU (optional. Protects against DMA-attacks)

5

Evaluation

In this Chapter, the mechanisms presented in Chapter 4 are evaluated with regards to the support in automotive embedded systems by investigating support in hardware and the AUTOSAR platforms. Also, the breadth of protection o□ered by the mechanisms is investigated.

5.1 Protection o□ered

Table 5.1 shows that the presented security mechanisms in Chapter 4 o□ers protection for all the identified asset-attributes pairs in Chapter 3.2. It also shows that for some assets all attributes can be protected by implementing either one complex mechanism or by implementing several less complex mechanisms. The abbreviations used in the tables throughout this chapter is explained below:

- SM - Static Memory
- VM - Virtual Memory
- DAC - Discretionary Access Control
- MAC - Mandatory Access Control
- CTN - Containers
- HYP - Hypervisor
- HMAC - Message authentication
- AE - Authenticated Encryption
- FW - Packet filter firewall

Asset	SM	VM	DAC	MAC	CTN	HYP	HMAC	AE	FW
Computing					A	A			
Memory	C I A	C I			C I A	C I A			
Storage			C I	C I	C I A	C I A			
Communication				I	I A	I A	I	C I	I A

Table 5.1: The breadth of protection offered by the presented security mechanisms in terms of security attributes (CIA) per asset.

5.2 Hardware Support

The support for the presented security mechanisms in ECU hardware is presented in this section, in form of tables. The hardware requirements for all mechanisms are summarized below:

- SM - Requires Dual-mode and an MPU/MMU.
- VM - Requires Dual-mode and an MMU.
- DAC - Requires Dual-mode and an MPU/MMU.
- MAC - Requires Dual-mode and an MPU/MMU.
- CTN - Requires Dual-mode and an MMU.
- HYP - Requires Dual-mode, Virtualizable ISA, an MPU/MMU, and SLAT.
- HYP+ - Same requirements as Hypervisor but also requires an IOMMU.
- HMAC - Requires Dual-mode, an MPU/MMU and either cryptographic ISA extension or a cryptographic hardware module.
- AE - Requires Dual-mode, an MP/MMU, and either cryptographic ISA extension or a cryptographic hardware module.
- FW - Requires Dual-mode, an MPU/MMU.

5.2.1 Microcontrollers

The list of microcontrollers (MCUs) in this section are provided and supported by ARCCORE AB. These MCUs are commonly used in the automotive domain and are often used for AUTOSAR Classic supported software. For this thesis, they represent the current hardware used in automotive embedded systems. For each MCU, the hardware requirements for each security mechanisms found in Section 4 are checked to determine the current support. Table 5.2, 5.3, and 5.4 show the support for each mechanism in Renesas, ARM, and PowerPC MCUs.

MCU	SM	VM	DAC	MAC	CTN	HYP	HYP+	HMAC	AE	FW
RH850F1L	X		X	X						
RH850F1H	X		X	X				X	X	X

Table 5.2: Renesas RH850/F1x MCUs

MCU	SM	VM	DAC	MAC	CTN	HYP	HYP+	HMAC	AE	FW
STM32F107										
Zynq 7000	X	X	X	X	X					
TMS570LS12	X		X	X						
TMS570LS1114	X		X	X						
TMS570LC43	X		X	X						
Jacinto 6	X	X	X	X	X	X	X	X	X	X

Table 5.3: MCUs with ARM architecture.

MCU	SM	VM	DAC	MAC	CTN	HYP	HYP+	HMAC	AE	FW
MPC5567	X	X	X	X	X					X
MPC5604B	X		X	X						
MPC5606B	X		X	X						
MPC5606S	X		X	X						
MPC5607B	X		X	X						
MPC5634M	X	X	X	X	X					
MPC5643L	X	X	X	X	X					
MPC5644A	X	X	X	X	X					
MPC5645S	X	X	X	X	X					
MPC5668G	X	X	X	X	X					X
MPC5744P	X		X	X						X
MPC5746C	X		X	X				X	X	X
MPC5748G	X		X	X				X	X	X
MPC5777M	X		X	X				X	X	X
SPC56EL70	X	X	X	X	X					
SPC560B54	X		X	X						

Table 5.4: MCUs with PowerPC architecture.

5.2.2 Cores

In this section, a number of cores currently found in automotive ECUs are presented, along with a number of more advanced processor cores planned to be used in future ECUs. The list of cores was given by ARCCORE AB and Volvo Trucks AB but is not an exhaustive list of cores used in the automotive domain but aim to show support in the most common ECUs. The hardware requirements for each security

mechanisms found in Section 4 are checked to determine the current and future support.

5.2.2.1 Current

The Arm Cortex-R series include real-time processors aimed at systems where reliability, high availability, fault tolerance and/or deterministic real-time responses are needed [83]. Further, the Cortex-R series is used in systems which require functional safety to avoid hazardous situations, like medical and autonomous systems. Table 5.5 shows the support for each mechanism in Cortex-R cores.

CPU	SM	VM	DAC	MAC	CTN	HYP	HYP+	HMAC	AE	FW
R4	X		X	X						X
R5	X		X	X						X
R52	X		X	X		X	X			X
R7	X		X	X						X
R8	X		X	X						X

Table 5.5: ARM Cortex-R cores

The Arm Cortex-M series include a range of scalable and energy e□cient processors optimized for cost and power-sensitive MCU and mixed-signal SoCs for several domains including automotive, industrial, and energy grid [84]. Table 5.6 shows the support for each mechanism in Cortex-M cores.

CPU	SM	VM	DAC	MAC	CTN	HYP	HYP+	HMAC	AE	FW
M0										
M0+	X		X	X						X
M3	X		X	X						X
M4	X		X	X						X
M7	X		X	X						X
M23	X		X	X						X
M33	X		X	X						X

Table 5.6: ARM Cortex-M cores

Arm Cortex-A is a series of more powerful processors used in mobile devices, networking infrastructure, home and consumer devices, automotive in-vehicle infotainment and driver automation systems, and embedded designs [85]. The Cortex-A series supports a wide range of full operating systems like Linux, Android, and Chrome. Table 5.7 and 5.8 show the support for each mechanism in Cortex-A cores.

CPU	SM	VM	DAC	MAC	CTN	HYP	HYP+	HMAC	AE	FW
A5	X	X	X	X	X					X
A7	X	X	X	X	X	X	X			X
A8	X	X	X	X	X					X
A9	X	X	X	X	X					X
A15	X	X	X	X	X	X	X			X

Table 5.7: Currently used ARM Cortex-A cores.

5.2.2.2 Future

CPU	SM	VM	DAC	MAC	CTN	HYP	HYP+	HMAC	AE	FW
A17	X	X	X	X	X	X	X			X
A32	X	X	X	X	X	X	X	X	X	X
A35	X	X	X	X	X	X	X	X	X	X
A53	X	X	X	X	X	X	X	X	X	X
A55	X	X	X	X	X	X	X	X	X	X
A57	X	X	X	X	X	X	X	X	X	X
A72	X	X	X	X	X	X	X	X	X	X
A73	X	X	X	X	X	X	X	X	X	X
A75	X	X	X	X	X	X	X	X	X	X

Table 5.8: Future ARM Cortex-A cores.

Apollo Lake is a series of cores developed by Intel intended for domains including new in-vehicle experiences and advancements in industrial and o□ce automation [86]. Table 5.9 show the support for each mechanism in Apollo Lake cores.

CPU	SM	VM	DAC	MAC	CTN	HYP	HYP+	HMAC	AE	FW
Pentium J4205	X	X	X	X	X	X	X	X	X	X
Pentium N4200	X	X	X	X	X	X	X	X	X	X
Celeron J3455	X	X	X	X	X	X	X	X	X	X
Celeron J3355	X	X	X	X	X	X	X	X	X	X
Celeron N3350	X	X	X	X	X	X	X	X	X	X
Celeron N3450	X	X	X	X	X	X	X	X	X	X
Atom x7-E3950	X	X	X	X	X	X	X	X	X	X
Atom x5-E3940	X	X	X	X	X	X	X	X	X	X
Atom x5-E3930	X	X	X	X	X	X	X	X	X	X

Table 5.9: Intel Apollo lake series of cores.

5.3 Recommended usage

When performing threat modeling of a system/product, the components are often assigned a criticality level ranging from 1 (low) to 5 (high). To simplify the process of choosing mechanisms to protect components, it is useful if also the mechanisms are assigned criticiality levels for which they are suited for.

Table 5.10 summarizes our recommendations for the suitable use of each mechanism, with regards to the determined criticality level of an application. The recommendations are based on the breadth of protection o□ered, in combination with the complexity of each mechanism. The complexity is measured both in terms of hardware features needed for e□cient use, as well as any issues identified in Section 4, which might circumvent the protection o□ered. Examples of issues which might impact negatively on our recommendation include known vulnerabilities or complex management and configuration. To simplify the evaluation, the summary only includes the mechanisms suited to separate components running on the same hardware platform.

The base criteria used to determine the highest possible criticality level suited for a mechanism is the degree of separation it o□ers, in terms of the amount of assets it protects. The following scoring criteria was used to assign a highest suited criticality level:

- Critical: All four assets must be protected.
- High: Any three assets must be protected.
- Medium: Any two assets must be protected.
- Low: At least one asset must be protected.
- QA: Protection not strictly required, but adds quality assurance.

If two or more mechanisms are assigned the same maximum criticality level, issues identified in Section 4 were used to determine the relative strength and maturity of these mechanisms and potentially decrease assigned levels.

To ensure that unnecessarily complex or demanding mechanisms are not used to protect applications of lower criticality levels, a lower bound for the suitable criticality levels were assigned. The baseline lower bound was assigned based on the required hardware features identified in Section 4, and were determined as follows:

- High: Dual Mode, MMU, and Virtualization Extension (SLAT, IOMMU)
- Low: Dual Mode & MMU
- QA: Dual Mode & MPU

Additionally, issues detailed in Section 4 were used to determine whether any known

complexities makes a mechanism unsuited for a lower level of criticality. If so, the lowest suited level was raised accordingly.

Level	HYP	CTN	MAC	VM	SM	DAC
Critical	X					
High	X	X				
Medium		X	X			
Low		X	X	X	X	X
QA					X	X

Table 5.10: This table states, for each isolation mechanism, the levels of criticality they are suited for.

According to the critera mentioned above, hypervisor and container were both assigned Critical as the maximum level. However, considering the issues related to containers detailed in Section 4.6.5, the maximum level of containers was lowered to High. Similarly, MAC and DAC were both assigned QA as the minimum level, but due to the complex management of MAC (see Section 4.1.2) the minimum level of MAC was raised to low.

5.4 AUTOSAR Support

5.4.1 AUTOSAR Classic

The list below shows whether the identified security mechanisms are specified in the AUTOSAR Classic specifications. If vendors offer known implementations of not specified mechanisms, this is also listed. An interview was conducted with ARCCORE AB to provide the basis for the list.

- Static memory allocation with memory protection - *Specified*

  AUTOSAR OS provides protective functions (memory, timing etc.) at runtime. The specification states that the OS module shall prevent write accesses, and may prevent read accesses to an applications private data sections by other non-trusted applications. Further, it specifies that the OS may provide an application the ability to protect its code sections from being executed by other non-trusted applications [87]. This was also confirmed by ARCCORE.

- Virtual Memory - *Not specified*

  AUTOSAR OS requires that all applications must use the same address space, i.e. virtual address spaces are not allowed [87]. ARCCORE also stated that Virtual memory is not supported since memory must be configured statically.

- Discretionary Access Control - *Not specified*

There is no concept of users specified in AUTOSAR OS. ARCCORE stated that there has been no demand for this type of access control from its customer base.

- Mandatory Access Control - *Not specified*

  ARCCORE stated that there has been no demand for this type of access control from its customer base.

- Packet filtering firewalls -*Not specified*

  Although the AUTOSAR Classic specification contains modules specifying both Ethernet and TCP/IP interfaces, it does not specify anything about TCP/IP or Ethernet packet filtering. However, some suppliers of AUTOSAR services have implemented such functionality in their Service Layer o□erings [88].

- Message Authentication - *Specified*

  The Secure On-board Communication (SecOC) module provides secure communication services, including message authentication and encrypted communication [89].

- Containers - *Not specified*

  AUTOSAR classic does not support virtual memory, which is a requirement for containers.

- Hypervisors - *Not specified*

  The Classic specification does not specify anything about support for virtualization of the platform. However, several vendors have implemented embedded hypervisors which support running AUTOSAR Classic systems in parallel with other systems [90] [91].

5.4.2 AUTOSAR Adaptive

The AUTOSAR Adaptive Platform implements the AUTOSAR Runtime Environment for Adaptive Applications (ARA) which exposes an operating system API conforming to POSIX specification PSE51, and an interface to services in the layer beneath. The PSE51 profile is targeted to small embedded devices with no support for filesystems or MMU. However, even though the API presented to the applications is only allowed to be PSE51, the operating system underneath may be full POSIX and the Foundation and Service layers can make use of that API to implement the functionality they provide to the applications. The PSE51 restriction is for the applications only, to allow for portability between di□erent operating systems.

In contrast to the Classic platform, Adaptive AUTOSAR does not define its own

operating systems. Instead, any POSIX compliant OS can be used [92] and AUTOSAR has indicated that they are willing to cooperate with suppliers to define standardized interfaces to operating systems such as Linux, Android, Windows, and more [21]. This means that all of the mechanisms evaluated in this thesis can be supported, since they are supported in most modern operating systems.

6

Conclusion

The current distributed automotive electronic architecture is no longer scalable, due to increased cost, weight, and system complexity. Therefore, automotive manufacturers are moving towards centralizing functionality of several ECUs to one or more high performance nodes. The aim of this thesis was to evaluate mechanisms which mitigate security issues associated with centralizing functionality to the same automotive hardware platform. To achieve this, we answer the following research questions:

- What are the security threats associated with centralizing functionality?

  The centralization of functionality means that applications must now share resources which were previously physically separated, leading to an environment similar to that of modern general-purpose systems. Thus, the associated security issues are related to the sharing of critical resources such as memory, filesystem, processor time, and communication. The issues include illegal use of memory and filesystem, Denial-of-Service attacks, and impersonation of other applications on the same node. Based on the identified issues, specific threats are identified. These issues show that separation of resources is critical to protect the safety and privacy of individuals, as well as operational and financial objectives of organizations.

  Denial-of-Service attacks, such as resource starvation attacks, are a bigger issue in an automotive embedded systems than in most other domains. This is because the loss of service might impact the safety of individuals, whereas in other domains loss of service might simply be an inconvenience. To mitigate such attacks require the use of more complex security mechanisms, such as Hypervisors or Containers, which o□er granular resource management capabilities.

- Which mechanisms exist that could mitigate the identified security threats?

  To protect against the presented issues, it is crucial to be able to separate access to resources between the integrated software applications. A set of security mechanisms which o□er varying breadth of protection against the identified

security issues were presented. It was shown that these mechanisms can be combined to mitigate all or a subset of the identified issues. It was found that the achievement of the Availability attribute required relatively more complex mechanisms, as this typically involves resource management in some form and thus a rather complex operating system.

- What metrics can be used to evaluate such mechanisms? This thesis considered the following metrics when evaluating mechanisms:
  - Breadth of protection o□ered
  - Hardware requirements
  - Strengths and known weaknesses

  These metrics were combined in order to recommended a suitable criticality level for each of the mechanisms. A scoring criteria which considers each of the metrics were derived for this use.

  The Confidentiality, Integrity, and Availability attributes were found to be good parameters to determine the breadth of protection o□ered. These attributes are at the heart of information security and all other attributes, which were considered for use, were found to be special cases of the CIA triad. The breadth of protection o□ered by each mechanism were mapped in a table so that correct mechanisms can be selected based on security needs. Further, a mapping between the mechanisms and their identified hardware requirements was produced, which can be used to determine the support for the mechanisms in other hardware.

- What support exists for these mechanisms in automotive embedded systems?

  The support for security was shown to be lacking, in both the available ECU hardware as well as in the AUTOSAR Classic platform, the de-facto open industry standard for automotive software. However, the hardware targeted to be used in future ECUs, along with the next-generation software standard Adaptive AUTOSAR, show support for all the presented security mechanisms. This shows that the automotive industry have realized that cyber-security will be a major challenge in the development of future connected autonomous vehicles and aim to adopt a strong security posture.

7

Discussion and recommendations for future work

We only used the Confidentiality, Integrity, and Availability security attributes as a metric when evaluating the protection o□ered, because of the limited time available even though we considered using an extended set of security attributes. When reflecting back on this decision, we feel that this was the correct decision as extending the set of attributes would only make the evaluation method more complex. Many of the other attributes do not apply to the assets considered in this thesis. Therefore, presenting them for some of the assets could create a misconception that some of the mechanisms do not o□er wide enough protection.

This thesis work only considered the breadth of protection o□ered by each of the identified mechanisms. The result o□ers a guide for selecting the appropriate mechanisms to protect the desired assets, in terms of the CIA security attributes. However, we do not evaluate the relative strength of the protection o□ered between the individual mechanisms, i.e. the depth of protection of the security attributes which they o□er. For example, two mechanisms which protect the same security attributes of the same asset may be considered to o□er varying strength of protection depending on, for instance, the complexity or maturity of the mechanism. A recommendation for future work might be to develop a scoring system which can be used to rank the security mechanisms with regards to the strength of protection o□ered. This scoring might include metrics such as the complexity of implementation (in terms of the Trusted Computing Base), the maturity of the solution, management overhead, performance overhead, hardware requirements, etc. When presenting the mechanisms we detailed strengths and weaknesses of each mechanism which can be factored in the scoring system.

Bibliography

[1] I. Studnia, V. Nicomette, E. Alata, Y. Deswarte, M. Kaâniche, and Y. Laarouchi. Survey on security threats and protection mechanisms in embedded automotive networks. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–12, June 2013.

[2] Lotfi Ben Othmane, Harold We□ers, Mohd Murtadha Mohamad, and Marko Wolf. *A Survey of Security and Privacy in Connected Vehicles*, pages 217–247. Springer New York, New York, NY, 2015.

[3] Chris Valasek and Charlie Miller. Remote exploitation of an unaltered passenger vehicle. `http://www.ioactive.com/pdfs/IOActive_Remote_Car_Hacking.pdf`, 2015. Accessed: 2017-02-26.

[4] Troy Hunt. Controlling vehicle features of nissan leaf. `https://www.troyhunt.com/controlling-vehicle-features-of-nissan/`, 2016. Accessed: 2017-03-16.

[5] New EEAs for the connected, autonomous future. *SAE Automotive Engineering*, pages 14–17, January 2017.

[6] M. Di Natale and A. L. Sangiovanni-Vincentelli. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. *Proceedings of the IEEE*, 98(4):603–620, April 2010.

[7] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. *black hat USA*, 2014.

[8] HEAVENS, Deliverable D2, “Security Models", HEAVENS project consortium. March 2016.

[9] Roman Obermaisser and Donatus Weber. Architectures for mixed-criticality systems based on networked multi-core chips. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–10. IEEE, 2014.

[10] Alan Burns and Robert Davis. Mixed criticality systems- a review*. *Department of Computer Science, University of York, Tech. Rep*, 2016.

[11] Karthikeyan Ravindran and Xingge Xu. Autosar and linux - single chip solution

implementation of automotive multipurpose ecu prototype system using hypervisor solution. Master's thesis, Institutionen för data- och informationsteknik (Chalmers), Chalmers tekniska högskola, 2015.

[12] Alberto Sangiovanni-Vincentelli. Automotive electronics: Trends and challenges. In *Convergence 2000 International Congress on Transportation Electronics*, 2000.

[13] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, Feb 2007.

[14] AMPG Body Electronics Systems Engineering Team. Future advances in body electronics. http://www.nxp.com/assets/documents/data/en/white-papers/BODYDELECTRWP.pdf. Accessed: 2017-05-07.

[15] Dominik Reinhardt and Markus Kucera. Domain controlled architecture - a new approach for large scale software integrated automotive systems. 2013.

[16] Adi Karahasanovic. Automotive cyber security. Master's thesis, Department of Computer Science and Engineering, Chalmers university of technology, 2016.

[17] Simon Furst. Autosar – a worldwide standard is on the road. 2009.

[18] AUTOSAR. Autosar webpage. https://www.autosar.org. Accessed: 2017-08-20.

[19] AUTOSAR. Autosar layered software architecture. Document ID: 053.

[20] AUTOSAR. Utilization of crypto services 4.3.0. 2016.

[21] Simon Fürst. The autosar adaptive platform for connected and autonomous vehicles.

[22] AUTOSAR. Adaptive platform : Autosar. https://www.autosar.org/standards/adaptive-platform/. (Accessed on 09/22/2017).

[23] Andy Greenberg. Tesla responds to chinese hack with a major security upgrade. https://www.wired.com/2016/09/tesla-responds-chinese-hack-major-security-upgrade/, 2016. Accessed: 2017-03-15.

[24] Society of Automotive Engineers. Cybersecurity guidebook for cyber-physical vehicle systems. https://saemobilus.sae.org/content/J3061_201601, 2016. Accessed: 2017-05-07.

[25] Automotive Information Sharing and Analysis Center. Automotive cybersecurity best practices. https://www.automotiveisac.com/best-practices/, 2016. Accessed: 2017-05-07.

[26] National Highway Tra□c Safety Administration. Cybersecurity best practices for modern vehicles. (report no. dot hs 812 333). https://www.nhtsa.gov/

staticfiles/nvs/pdf/812333_CybersecurityForModernVehicles.pdf, 2016. Accessed: 2017-05-07.

[27] Steve Lipner. The trustworthy computing security development lifecycle. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 2–13. IEEE, 2004.

[28] HEAVENS, Deliverable D1.1, “Needs and requirements", HEAVENS project consortium. March 2016.

[29] E-safety vehicle intrusion protected applications (EVITA). Deliverable D2.3: Security requirements for automotive on-board networks based on dark-side scenarios. Dec 2009.

[30] William Stallings. *Cryptography and network security: principles and practices*. Pearson Education India, 2006.

[31] ISO 26262-1:2011(E) Road vehicles — Functional safety. Standard, International Organization for Standardization, Geneva, Switzerland, 2011.

[32] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Bu□er overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE, 2000.

[33] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[34] N. Ye. *Secure Computer and Network Systems: Modeling, Analysis and Design*. Wiley, 2008.

[35] OWASP. Regular expression Denial of Service - ReDoS. https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS. Accessed: 2017-05-07.

[36] Felix Lau, Stuart H Rubin, Michael H Smith, and Ljiljana Trajkovic. Distributed denial of service attacks. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 3, pages 2275–2280. IEEE, 2000.

[37] PREparing SEcuRe VEhicle to X Communication Systems (PRESERVE). Deliverable 1.1: Security Requirements of Vehicle Security Architecture. 2011.

[38] DOD. Trusted computer system evaluation criteria. *Department of Defense 5200.28-STD*, 1985.

[39] Ninghui Li. How to make discretionary access control secure against trojan horses. *Paralell and Distributed Processing*, 2008.

[40] Ryan Ausanka-Crues. Methods for access control: Advances and limitations. 2001.

[41] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *USENIX Security Symposium*, pages 171–190, 2002.

[42] Capitani de Vimercat Pierangela SamaratiSabrina. Access control: Policies, models, and mechanisms. *Foundations of Security Analysis and Design*, 2001.

[43] Android. Security-enhanced linux in android. `https://source.android.com/security/selinux/`. Accessed: 2017-05-20.

[44] SELinux Wiki contributors. Selinux wiki. `https://selinuxproject.org/`. Accessed: 2017-05-15.

[45] Aaron Grattafiori. Understanding and hardening linux containers. `https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers/`, 2016. Accessed: 2017-05-07.

[46] Linux man page. sandbox(8). `https://linux.die.net/man/8/sandbox`. Accessed: 2017-05-16.

[47] Joseph Yiu. *The Definitive Guide to the ARM Cortex-M3, Second Edition*. Newnes, Newton, MA, USA, 2nd edition, 2009.

[48] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008.

[49] Richard William Carr. Virtual memory management. In *UMI Research Press, Ann Arbor, Mich*. Citeseer, 1984.

[50] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[51] Sangrok Han and Hyogon Kim. On autosar tcp/ip performance in in-vehicle network environments. 2016.

[52] Lucia Lo Bello. Novel trends in automotive networks: A perspective on ethernet and the ieee audio video bridging. 2014.

[53] Guidelines on firewalls and firewall policy. Special publication 800-41, National Institute of Standards and Technology, 2009.

[54] Secure on-board architecture specification. Deliverable d3.2, EVITA, 2011.

[55] André Perez. *Network Security*. Wiley, 2014.

[56] Recommendation for block cipher modes of operation: The cmac mode for authentication. Special publication 800-38b, National Institute of Standards and Technology, 2005.

[57] Recommendation for key management. Special publication 800-57, part 1, revision 4, National Institute of Standards and Technology, 2016.

[58] Recommendation for applications using approved hash algorithms. Special publication 800-107, revision 1, National Institute of Standards and Technology, 2012.

[59] Howard M. Heys Janaka Deepakumara and R. Venkatesan. Performance comparison of message authentication code (mac) algorithms for the internet protocol security (ipsec). 2003.

[60] Marko Wolf Thomas Wollinger André Groll, Jan Holle. Next generation of automotive security: Secure hardware and secure open platforms. 2010.

[61] Hugo Krawczyk. The order of encryption and authentication for protecting communications) algorithms for the internet protocol security (ipsec). 2001.

[62] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. *Security of OS-Level Virtualization Technologies*, pages 77–93. Springer International Publishing, Cham, 2014.

[63] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[64] linuxcontainers.org. Linux containers. `https://linuxcontainers.org/`. Accessed: 2017-05-07.

[65] Linux Programmer's Manual. capabilities - overview of linux capabilities. `http://man7.org/linux/man-pages/man7/capabilities.7.html`. Accessed: 2017-05-07.

[66] Serge E Hallyn and Andrew G Morgan. Linux capabilities: Making them work. In *Linux Symposium*, volume 8, 2008.

[67] Linux Programmer's Manual. namespaces - overview of linux namespaces. `http://man7.org/linux/man-pages/man7/namespaces.7.html`. Accessed: 2017-05-07.

[68] Linux Programmer's Manual. setns - reassociate thread with a namespace. `http://man7.org/linux/man-pages/man2/setns.2.html`. Accessed: 2017-05-07.

[69] Linux Programmer's Manual. unshare - disassociate parts of the process execution context. `http://man7.org/linux/man-pages/man2/unshare.2.html`. Accessed: 2017-05-07.

[70] Linux Programmer's Manual. pid_namespaces - overview of linux pid namespaces. `http://man7.org/linux/man-pages/man7/pid_namespaces.7.html`. Accessed: 2017-05-07.

[71] Linux Programmer's Manual. user_namespaces - overview of linux user

namespaces. http://man7.org/linux/man-pages/man7/user_namespaces.7.htmll. Accessed: 2017-05-07.

[72] Linux Programmer's Manual. cgroups - linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed: 2017-05-08.

[73] Redhat Enterprise Linux. Resource management guide: 3.2. cpu. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-cpu.html. Accessed: 2017-05-08.

[74] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar. Securing docker containers from denial of service (dos) attacks. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 856–859, June 2016.

[75] Jesse Hertz. Abusing privileged and unprivileged linux containers. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/june/abusing-privileged-and-unprivileged-linux-containers.pdf, 2016. Accessed: 2017-05-14.

[76] Brad Spengler. grsecurity forums: False boundaries and arbitrary code execution. https://forums.grsecurity.net/viewtopic.php?f=7&t=2522#p10271. Accessed: 2017-05-14.

[77] Roberto Mijat and Andy Nightingale. Virtualization is coming to a platform near you: The arm® architecture virtualization extensions and the importance of system mmu for virtualized solutions and beyond. 2011.

[78] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[79] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[80] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[81] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, October 2006.

[82] VMWare. Performance Evaluation of Intel EPT Hardware Assist. *VMware ESX builds 140815 & 136362 (internal builds)*.

[83] ARM. Arm cortex-r webpage. https://www.arm.com/products/processors/cortex-r. Accessed: 2017-09-29.

[84] ARM. Arm cortex-m webpage. https://www.arm.com/products/

processors/cortex-m. Accessed: 2017-09-29.

[85] ARM. Arm cortex-a webpage. https://www.arm.com/products/processors/cortex-a. Accessed: 2017-09-29.

[86] Intel. Intel apollo lake webpage. https://www.intel.com/content/www/us/en/embedded/products/apollo-lake/overview.html. Accessed: 2017-09-29.

[87] AUTOSAR. Specification of operating system (version 4.3.0). https://www.autosar.org/fileadmin/files/standards/classic/4-3/software-architecture/system-services/standard/AUTOSAR_SWS_OS.pdf, 2016.

[88] Vector. Microsar: AUTOSAR basic software product information. https://vector.com/portal/medien/cmc/info/MICROSAR_ProductInformation_EN.pdf. Accessed on: 09/22/2017.

[89] AUTOSAR. Specification of module secure onboard communication 4.3.0. 2016.

[90] Mentor Graphics. Mentor graphics hypervisor delivers high performance and security for multicore processors and enables multi-os consolidation - mentor graphics. https://www.mentor.com/embedded-software/news/mentor-embedded-hypervisor. (Accessed on 10/22/2017).

[91] ARM Limited. Automotive safety hypervisor announced for arm cortex-r52 - arm. https://www.arm.com/about/newsroom/automotive-safety-hypervisor-announced-for-arm-cortex-r52.php. (Accessed on 10/22/2017).

[92] Patrick Markl. Vector congress 2016: The adaptive platform for future use cases. (Accessed on 09/15/2017).