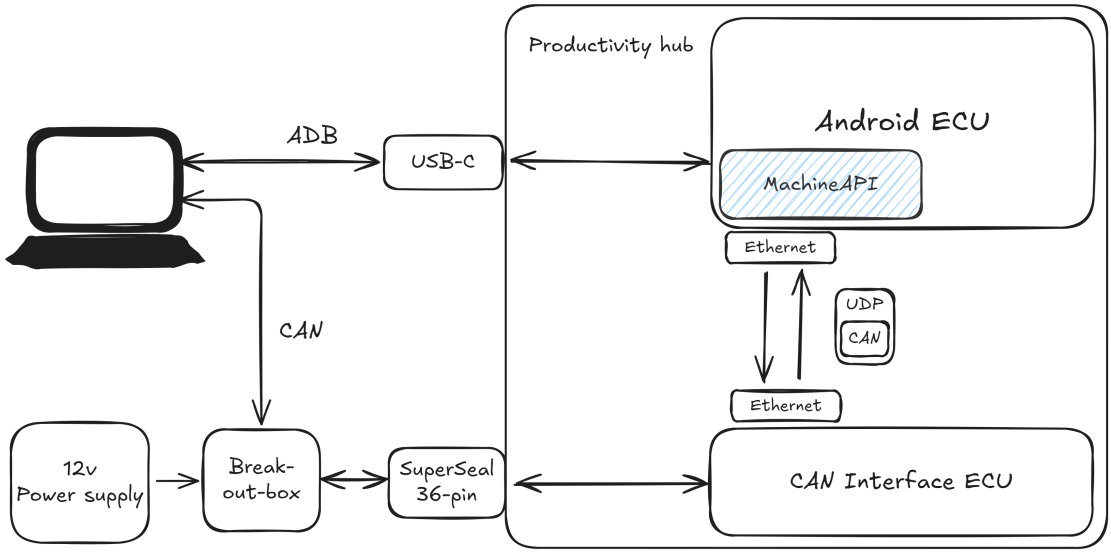
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Enhancing the Security of Android-Based Infotainment Systems

Implementing a CAN Access Control System using Android Permissions, guided by the Principle of Least Privilege

Master's Thesis in Computer Science and Engineering

Emil Lindblad
Amanda Papacosta

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Enhancing the Security of Android-Based Infotainment Systems

Implementing a CAN Access Control System using Android Permissions, guided by the Principle of Least Privilege

Emil Lindblad
Amanda Papacosta

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Enhancing the Security of Android-Based Infotainment Systems
Implementing a CAN Access Control System using Android Permissions, guided by the Principle of Least Privilege
EMIL LINDBLAD
AMANDA PAPACOSTA

Advisor: Mathias Andreasson, CPAC Systems
Advisor: Lennart Dahlström, CPAC Systems
Supervisor: Tomas Olovsson, Department of Computer Science and Engineering
Examiner: Tomas Olovsson, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: System setup, described in more detail in Section 4.1

Typeset in LaTeX
Gothenburg, Sweden 2025

Enhancing the Security of Android-Based Infotainment Systems
Implementing a CAN Access Control System using Android Permissions, guided by the Principle of Least Privilege
EMIL LINDBLAD
AMANDA PAPACOSTA
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern infotainment systems are highly integrated with both the vehicle's Controller Area Network (CAN) bus and external internet services. Such connectivity capabilities make the infotainment system a viable entry point for adversaries targeting the vehicle's internal components and subsystems. Therefore, this thesis investigates how to implement a granular access control mechanism for the CAN bus, guided by the Principle of Least Privilege (PoLP). The objective is to create a more secure infotainment system that incorporates multiple layers of protection. In addition to adhering to the PoLP, the proposed solution aims to comply with two important cybersecurity standards and regulations: EN 18031-1:2024, part of the Radio Equipment Directive (RED), and ISO/SAE 21434. The latter providing a framework for Threat Modeling, the process of analyzing a system, identifying threats, and deciding on appropriate mitigation.

A key component of the access control system is the use of Android permissions, which provide enhanced granularity in managing application access. In addition to Android permissions, the configuration scheme of the infotainment system was extended with application level whitelisting for CAN data.

Benchmarking tests were conducted to evaluate the performance impact of the proposed solution, and the results did not show any significant additional overhead. The final implementation provides effective and scalable protection of the CAN bus, with granular access control, improving security in the infotainment system.

Keywords: Cybersecurity, Android Permissions, Android, Principle of Least Privilege, CAN Bus, In-Vehicle Network.

Acknowledgements

We would like to extend our deepest gratitude to CPAC Systems for providing us with the opportunity to work at the company and for granting us access to valuable resources and insightful discussions with its employees. We are especially grateful to our advisors Mathias Andreasson and Lennart Dahlström for their support and excellent supervision throughout the entire project. Finally, we would like to express our sincere thanks to our supervisor and examiner Tomas Olovsson for his role in shaping the project proposal during its initial stages, as well as for constructive feedback throughout the course of the project.

Emil Lindblad, Amanda Papacosta
Gothenburg, June 2025

Contents

List of Acronyms

The following is a list of acronyms used in the thesis, displayed in alphabetical order.

ADB Android Debug Bridge

API Application Programming Interface

ART Android Runtime

CAN Controller Area Network

CIE CAN Interface Electronic Control Unit (ECU)

DFD Data Flow Diagram

DoS Denial of Service

ECU Electronic Control Unit

IDS Intrusion Detection System

IVN In-vehicle Network

MITM Man-In-The-Middle

OEM Original Equipment Manufacturer

OTA Over-the-Air

PoLP Principle of Least Privilege

RED Radio Equipment Directive

SDK Software Development Kit

TARA Threat Analysis and Risk Assessment

UDP User Datagram Protocol

List of Figures

List of Tables

List of Listings

1

Introduction

Modern trucks and construction equipment can be fitted with smart infotainment systems, offering productivity apps and various forms of connectivity such as 5G and Wi-Fi. Infotainment systems with such features essentially make the vehicle a smartphone on wheels, with the same connectivity capabilities as the device in our pockets. As a result of the connectivity capabilities, these infotainment systems can serve as an exposed entry point to the vehicle's internal systems. Therefore, designing and implementing infotainment systems securely has become paramount. Acquiring unauthorized access to a vehicle can result in many different consequences, both for the vehicle itself and the environment around it. These security breaches can vary in severity and impact ranging from obtaining access to sensitive data to controlling the vehicle's movement. Cybersecurity concepts such as Principle of Least Privilege (PoLP) and multi-layered security can be used to create a secure implementation that gives developers control over how the infotainment system can access the rest of the vehicle.

This thesis was done in collaboration with CPAC Systems, a technology company that produce a productivity and infotainment device for construction vehicles, boats, and trucks. The report further references this device as the *productivity hub*. The research and development conducted in this thesis was carried out using a concept model of the productivity hub intended for use in heavy-duty trucks.

1.1 Automotive Electronics

The purpose of this section is to serve the reader with a basic understanding of automotive electronics, to better grasp the purpose and motivation of this thesis.

Trucks, construction equipment, or any other type of vehicle with electronics consist of Electronic Control Units (ECUs). These devices are responsible for controlling different vehicle functions [1]. An ECU can have a specific purpose, e.g., controlling the brakes, or be multifunctional such as CPAC's productivity hub, combining multiple sub-ECUs in a single device. Modern vehicles contain hundreds of interconnected ECUs [2], which together create the In-vehicle Network (IVN). Communication between ECUs is done using Controller Area Network (CAN), a multicast and multimaster network protocol [3]. These properties entail that messages do not have a specific recipient, thus any ECU on the network can transmit data or read any message present on the bus.

CAN was created in the 1980s and was not designed with cybersecurity in mind, resulting in later efforts to create new versions of the protocol with integrated security measures [4]. Despite these efforts, adoption of more secure versions of CAN is a complex task and more suited for complete-vehicle manufacturers. For a single ECU manufacturer, whose product is meant to be installed in different types of vehicles from different manufacturers, a better approach is instead to implement internal controls and limit communication between an infotainment ECU and the rest of the IVN.

1.2 Purpose and Motivation

The purpose of this thesis is to investigate how the security of an infotainment system can be improved by adopting a more granular approach to CAN bus access, based on the PoLP. The motivation stems from the current access control mechanism, consisting of two Android permissions that grant full read and write capabilities to the CAN bus. As a consequence, any app with these permissions can access the entire IVN.

This "all-or-nothing" approach is problematic from a security standpoint, as it may grant certain parts of the infotainment system unintended read and write access to other systems in the vehicle. Therefore, it would be beneficial to take a granular approach, implementing more precise access control based on the PoLP. Such an approach entails that apps only get access to what they need, for example to read and write only certain CAN messages. This would ultimately lower the risk of unauthorized access to the IVN.

Previous research on the topic of Android security, specifically Android permissions, has mainly focused on smartphones and tablets. In those scenarios, the user can freely install apps and decide how permissions are granted, e.g., giving an app access to the microphone. However, the conditions for the system in this thesis differ from the previous research, since all applications on the productivity hub are pre-installed, and it is not intended for users to install third-party applications.

Additionally, new regulations and standards that Original Equipment Manufacturer (OEM) must follow are continually introduced. Consequently, the regulations outlined in Section 2.6 will influence both the project's implementation decisions and the approach taken to perform the threat analysis. Adhering to such regulations ensures that potential future integration of the projects findings can be achieved with minimal effort.

1.3 Research Questions

With the purpose and motivation defined, this thesis aims to answer the following three questions:

Q1: Which historical and existing security-related vulnerabilities exist in Android-based systems?

Q2: What is the approach to designing a granular access control system in Android Automotive for CAN bus access based on the PoLP?

Q3: What is the method for developing security-focused software while ensuring compliance with regulations and industry standards?

1.4 Limitations

The project limits the scope to two different ECUs present on the productivity hub. These are responsible for running the Android Automotive operating system and providing a CAN interface to the IVN respectively. In terms of Android development, the goal is to not create any modifications on the operating system level, commonly referred to as platform changes. As a result, the implementations will be done on the app level. This is because updating individual apps is generally a more straightforward and efficient process compared to updating the entire operating system, thus making development and deployment easier.

Furthermore, the project focuses on securing the CAN bus from unauthorized access via the productivity hub. Thus, no aspect of physical security or tampering will be investigated. An adversary with physical access to a vehicle and targeting the CAN bus is able to bypass the productivity hub entirely. In such a scenario, security mechanisms implemented at other system levels will not be beneficial in protecting the CAN bus.

1.5 Report Structure

Chapter 2 describes the relevant background theory needed to understand the project. This is then followed by a literature review on Android permissions and historical vulnerabilities related to the topic in Chapter 3. Chapter 4 presents the methodology used in all stages of the project, from research to final solution. Threat modeling is also included in the chapter. The results of tests, as well as the threat analysis, can be found in Chapter 5. In Chapter 6, the results are discussed, and Chapter 7 concludes the whole project.

2

Background

The following chapter begins with an overview of the Android operating system and its underlying security model. It then proceeds to present key in-vehicle components and communication protocols relevant to the context of this thesis. Subsequently, foundational cybersecurity concepts are introduced, along with a step-by-step guide on how to conduct a threat analysis and modeling process. Finally, the chapter concludes with an overview of the regulations and standards adhered to by CPAC, which are essential to the thesis.

2.1 The Android Operating System

Android is an open source, touchscreen-based operating system for mobile devices and infotainment systems, with the latter using an extension of the base operating system called Android Automotive [5]. It provides additional features and requirements to integrate into automotive systems and interface with the IVN. Android is built on a modified version of the Linux kernel, allowing it to use several security features such as user permissions and process isolation [6]. Android security is further described in Section 2.1.1.

Apart from the operating system itself, Android offers an Application Programming Interface (API) for developers to build Android apps [7]. There are four key components of an app, each serving as a potential entry point:

- **Activities** - Represents different interfaces that the user can interact with, e.g., write an email or show a list of emails. Each activity is an independent view, working together to create a seamless user experience.
- **Services** - Services keep apps running in the background without directly interacting with the user through a user interface. Their primary purpose is handling operations that run for a longer period of time.
- **Broadcast Receivers** - Apps can use broadcast receivers to listen for system-wide or app-specific broadcast messages, such as when the user turns off the screen or when the device's battery is low.
- **Content Providers** - Content providers are used to manage an app's data. If an app wants to share its data with other apps, it can permit the usage of its content provider.

If needed, an app can also launch other apps' components using intents. This allows for reuse of existing functionality across the system instead of re-implementing the same functionality in every app. An app sends an intent whenever it wants to perform an action. For example, when an app needs to open a webpage, it sends an intent to the operating system specifying the URL it wants to display. The operating system then forwards it to the responsible unit, which handles the intent (a web browser app in this example).

All components of an app must be declared in a special file called `AndroidManifest.xml` [7]. This file is used as a manifest by the Android operating system to know which components belong to which app. In addition to declaring components, the manifest file also contains other relevant information for the application, such as the minimum required API or any permissions the app needs. The latter being a fundamental part of Android's security model.

2.1.1 Android's Security Model

The Android operating system offers several security features, which together constitute the Android security model. One such feature is the Application Sandbox, which is a way to isolate an app's resources from other apps in the system [7]. Each app is assigned a unique UID (User ID), and as a result runs in its own process [8]. Furthermore, every app also has its own instance of the Android Runtime (ART) [9].

By default, all components of a given app execute within the same process and share a single thread, known as the main thread [10]. However, it is possible to change so that different components of the same application run in separate processes and threads if desired. The Application Sandbox runs at the kernel level and can therefore utilize Linux facilities to ensure the separation of processes [8]. Thus, all software running above the kernel is encapsulated by the Application Sandbox. The Linux file system permissions are also utilized to separate each apps files and storage [6].

Another key feature in Android's security model is permissions. As a result of application sandboxing, apps must request permission before accessing resources from other apps or the operating system [8], [11]. Permissions are declared and defined in the manifest file, with each permission having its own unique label. Listing 1 shows an example of declaring and defining permissions.

```
<manifest package="com.example.app" >
    <!-- declare to use a permission -->
    <uses-permission
        android:name="com.example.permission.OPEN_CAMERA"
    />

    <!-- define a custom permission -->
    <permission
        android:name=="com.example.permission.READ_SIGNAL"
        android:permissionGroup="MACHINE_SIGNALS"
        android:protectionLevel="normal"
    />
</manifest>
```

Listing 1: Snippet of manifest file declaring use of a permission and defining a new one.

There are three different permission levels in Android: *normal*, *signature*, and *dangerous* [12]. Each level corresponds to an increase in the sensitivity of data and actions available to the app.

Normal and signature permissions are granted automatically by the operating system at install time, and are thus called install-time permissions. Normal permissions give an app access to external resources and introduce a low risk to user privacy. They also do not have a significant impact on other apps. Signature permissions are granted to an app only if signed with the same certificate as the application or operating system that defined the permission. Similarly to normal permissions, the user does not have to explicitly approve signature permissions [13].

On the contrary, dangerous permissions (also called runtime permissions) can only be accepted or denied by a user at runtime [12]. Such permissions allow an app to access sensitive data (e.g., location) or perform actions that have a greater impact on other apps or the system (e.g., modify a file). Additionally, when an app requests a dangerous permission, a notification is presented on the screen for the user to allow or disallow the request. Dangerous permissions can also be revoked by the user at any time, whereas normal and signature permissions cannot be revoked once granted.

It is also possible to group similar permissions together into so-called permission groups [12]. For example, permissions to send and receive text messages could belong to the same permission group. When a user is prompted to allow or deny a set of permissions, those belonging to the same group are displayed together, and the user's choice affects all permissions within that group simultaneously.

Apart from system permissions, which are permissions defined by the operating system and built-in system apps, custom permissions also exist. Third-party apps have the ability to define custom permissions to share their resources and services with other apps [14]. Additionally, system apps are installed before the installation

of third-party apps, and thus, no custom permissions will directly affect system resources.

In summary, Android adheres to the concept Principle of Least Privilege, meaning that each app is only able to access the components it needs. As a result, apps cannot access system locations for which they lack the required permissions [7].

2.2 Principle of Least Privilege

There are several security-focused design principles for computer systems, with one of the most widely recognized being the Principle of Least Privilege (PoLP) [15]. This principle was introduced in 1975 by Saltzer and Schroeder as part of their list of eight fundamental design principles for computer security. The principle states that every program and user should not be granted any set of privileges, except for those necessary to complete the work. This limits the risk of unauthorized programs or users being able to access resources and services. Additionally, by adhering to the principle, the extent of the damage caused by such a situation is reduced due to the small number of privileged programs.

2.3 CIA triad

The CIA triad is a fundamental concept within computer security. As can be seen in in Figure 2.1 it consists of three elements: Confidentiality, Integrity, and Availability. The model can be used both as a way to find vulnerabilities within a system, as well as methods to remove them [16].

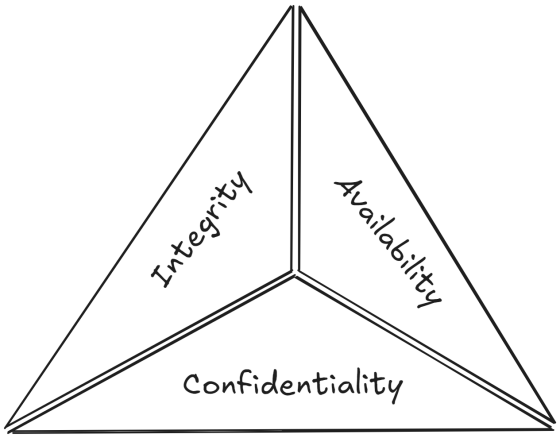


Figure 2.1: The CIA Triad.

Confidentiality refers to the requirement of merely granting access to sensitive data to authorized individuals [17]. It is ensured by implementing access control mechanisms, encryption, and classification of the data based on its level of sensitivity. Integrity implies that data must remain unaltered and untainted. It guarantees the author's authenticity and preserves the original version of the data. Integrity

can be upheld by implementing the following techniques: hash functions to create uniqueness of versions, digital signatures, and maintaining version control. Finally, availability entails that individuals who need the data must have access to it at any time. Availability is maintained through several mechanisms, including uptime management, service level agreements that define expected performance standards, and redundancy and backup strategies to mitigate unexpected downtime.

2.4 In-Vehicle Network

The following section provides an overview of key vehicle components and communication protocols commonly found within the In-vehicle Network (IVN). It also outlines their respective functions and roles in facilitating the exchange of information between different Electronic Control Units (ECUs).

2.4.1 Electronic Control Unit

An Electronic Control Unit (ECU) is an electronic component dedicated for managing specific vehicle functionalities and ensuring optimal performance. Common examples of functions controlled by ECUs include self-parking, braking, headlight control, and cruise control [1]. As modern vehicles continue to increase in complexity, the number of ECUs integrated into a single vehicle has also risen. As of 2020, vehicles typically contain between 80 and 150 ECUs, depending on the level of advanced functionality provided [18]. Communication between ECUs is facilitated through various IVN protocols, with CAN being one of the most widely used [1]. Further details regarding the CAN protocol are provided in the following section.

2.4.2 The CAN protocol

The Controller Area Network (CAN) protocol was created in the 1980s at Robert Bosch GmbH as a way to simplify inter process communication between components in automotive vehicles [19]. It utilizes the concept of broadcasting for sending messages over the network, and it is therefore not possible to send a message (called a frame) to a specific node directly. Instead, it is up to each receiver node to decide whether the data is relevant to that device or not.

The structure of a CAN frame can be found in Figure 2.2, and it contains the following segments [20]:

- **SOF (Start of Frame)** - Specifies the start of a new transmission of a frame
- **Arbitration ID** - Declares when the frame gets passed onto the bus (lower ID equals higher priority)
- **Control** - Control bits
- **DLC (Data Length Code)** - Specifies how long the data segment is
- **Data** - The data intended for transmission

- **CRC (Cyclic Redundancy Checksum)** - Utilized by recipient ECUs for verifying the integrity of the data field
- **ACK (Acknowledge)** - The recipient ECU can confirm the successful transmission of the CAN frame
- **EOF (End of Frame)** - Specifies the end of the frame

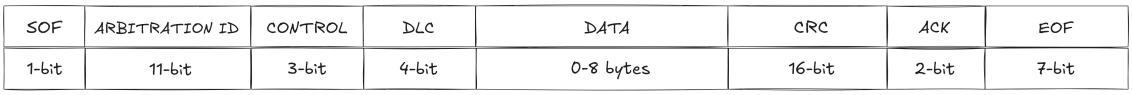


Figure 2.2: CAN frame structure [20].

The CAN protocol has many advantages for use in IVNs compared to other communication protocols, hence its vast global usage [21]. Firstly, CAN is a lightweight protocol that enables fast data transmission between ECUs. Its simple wiring design also makes it a cost-effective solution. Secondly, the protocol is known for its robustness and reliability since all nodes on the network are notified when a fault is detected. Moreover, the CAN bus is resistant to interference from electrical noise. Lastly, CAN offers flexibility, as nodes are not restricted to predefined actions or specific data. This allows for easy addition or removal of nodes within the network as required.

A major disadvantage of the CAN protocol is its lack of security. It does not adhere to the CIA principle since it lacks confidentiality, integrity, and authentication [20]. All traffic sent over the CAN bus is unencrypted, and every node on the network can listen to any message, thus violating the aspect of confidentiality. Furthermore, the protocol lacks authentication features, which prevents it from ensuring data integrity. Additionally, due to the use of arbitrary IDs, it is possible for an ECU to continue transmitting frames uninterrupted as long as its ID is lower than the frames from other ECUs. As a result, the availability of the CAN bus could be jeopardized if an attacker floods the bus with high-priority messages.

The lack of security in the CAN protocol originates from its initial design, during which security was not considered to be a priority [22]. However, as the automotive industry has advanced and vehicles have become more complex, the demand for robust security has significantly increased. Consequently, various solutions have been proposed to enhance the security of the CAN bus, either by embedding security mechanisms directly within the protocol or by implementing safety measures at higher system levels to restrict access to the CAN bus. Integrating security within the protocol can impact performance and introduce additional overhead, creating a need for balance between security and efficiency.

An example of such a security mechanism is incorporating encryption of the data transmitted over the bus. However, encrypting CAN data has its limitations. Firstly, due to the broadcast nature of the CAN protocol, all nodes on the network need to change to be able to encrypt and decrypt messages [23]. Secondly, the additional computations made, due to the implementation of cryptographic algorithms,

increase message latency. This can be dangerous in systems that are dependent on real-time constraints. Furthermore, some algorithms, such as MACs and digital signatures, also require a change in the CAN protocol to be able to handle the verification tags. Lastly, as the CAN bus enables communication among all ECUs in a vehicle, the encryption strength depends on a shared key, which introduces a potential vulnerability to the system. As a result, given the scope and focus of this thesis, security mechanisms were implemented at higher system levels rather than modifying the protocol itself, as this approach is more straightforward and practical within the project's constraints.

2.5 User Datagram Protocol

UDP is defined in RFC768 [24] and is a transport protocol that provides fast but unreliable transport of data between apps that operate over IP [25]. Its unreliable nature is due to the fact that it does not establish an end-to-end connection between systems, and datagrams can therefore get lost in transmission. However, due to the lack of established connections, the protocol is able to offer fast and efficient transmission of datagrams (i.e. low latency).

UDP packets contain the fields listed below, as shown in Figure 2.3. All fields mentioned, except the data field, together construct the UDP header.

- **Source Port** - Indicates the port from which the datagram is sent, and a reply is expected to. This field is optional.
- **Destination Port** - Indicates the port to which the datagram should be sent.
- **Length** - The length of the whole UDP datagram.
- **Checksum** - Used to check if any errors have occurred to the datagram during transmission [26]. This field is optional.
- **Data** - Application data to be sent.

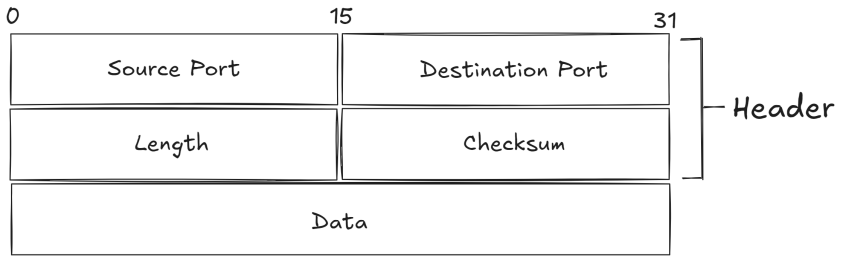


Figure 2.3: UDP packet format [24].

Another consequence of UDP's unreliability is its lack of implemented security mechanisms [25]. As a result, the protocol does not offer any prevention from tampering, eavesdropping, or message forgery, and instead, security has to be enforced at higher system levels.

2.6 Relevant Cybersecurity Regulations and Standards

The field of cybersecurity is constantly evolving, particularly in the context of connected vehicles. With that comes new regulations and requirements that the OEM has to comply with. Several of these regulations and standards apply to the majority of CPAC's product portfolio and thus are of importance to this thesis.

2.6.1 The Radio Equipment Directive

One example of a regulation that is relevant for this thesis is the Radio Equipment Directive (RED), established by the European Union. RED is a framework of rules that targets any form of radio equipment placed on the European market [27]. The directive was introduced in 2014, but was expanded in 2021 to increase the level of cybersecurity for radio equipment. In the context of this thesis, Article 3(3), point (d) of the legislation is particularly relevant as it applies to all radio equipment with network communication capabilities [28]. The European standard EN 18031-1:2024 [27] specifically addresses the requirements outlined in point (d). RED is also relevant to the thesis since the hardware used in the project has network capabilities and thus falls under the targeted devices.

2.6.2 ISO/SAE 21434 Road vehicles - Cybersecurity Engineering

ISO/SAE 21434 [29] is a standard developed by the International Organization for Standardization (ISO) in collaboration with the Society of Automotive Engineers (SAE). The standard can be summarized as a set of requirements and guidelines for cybersecurity engineering for vehicles. Organizations that adhere to the standard demonstrate a robust and systematic implementation of cybersecurity measures in all steps of the development process and the life cycle of the vehicle.

In addition to the guidelines for product development, the standard also provides methods for establishing a proactive cybersecurity culture within an organization. It ensures that cybersecurity is being considered in management, company policies, and daily operations, not just in technical development.

The standard also includes a comprehensive breakdown of all components and so-called work products required to create an effective and well-structured Threat Analysis and Risk Assessment (TARA).

2.7 Threat Analysis and Risk Assessment

A Threat Analysis and Risk Assessment (TARA) is defined as a collection of methods and work products used to identify threats to a system or component, assess the likelihood of those threats occurring, and evaluate their potential impacts. These impacts are typically considered from multiple perspectives, such as financial cost, harm to users, or damage to a vehicle or device.

In the context of automotive engineering, ISO/SAE 21434 [29] provides information on how a TARA should be structured for evaluating road vehicles. The standard describes different items that should be in place and what needs to be done to produce those items. Examples are definitions of the device or process being evaluated, cybersecurity requirements, and damage scenarios.

However, ISO/SAE 21434 only aims to define what should be included, not to give a definitive guide on exactly how the TARA should be performed. For this, there are different frameworks and methods on how to perform a TARA that can be used. These can be internal company documents, or public like HEAVENS 2.0 [30].

Not all elements and items described in ISO/SAE 21434 will be applicable or feasible to address in the thesis. As a result, OWASP's Threat Modeling Process [31] and accompanying cheat sheet [32] were used for the threat modeling methodology. These guides provide a more lightweight and software-focused approach compared to the more comprehensive guide from ISO/SAE 21434. Both ISO/SAE 21434 and OWASP cover the core items of a TARA, but OWASP's structured step-by-step approach is better suited to the software of the productivity hub.

2.8 Threat modeling

The OWASP cheat sheet [32] is based on the Threat Modeling Manifesto, which defines threat modeling as "analyzing representations of a system to highlight concerns about security and privacy characteristics" [33]. The manifesto defines the following questions as the foundation of a threat model:

- What are we working on?
- What can go wrong?
- What are we going to do about it?
- Did we do a good enough job?

In addition to the questions, the manifesto also defines values and principles for integrating the threat modeling process in development teams. Being high-level questions and more abstract principles, performers of the threat model can choose different strategies and approaches that best fit the project.

The purpose of doing a threat model is to understand how a system works and how data flows through it, with an emphasis on the security of the system. By having a detailed understanding of the system, one can identify implementation

issues that may result in security vulnerabilities and mitigate them. For the most optimal results, the threat modeling process should be integrated with the system's development process.

According to the Threat Modeling Cheat Sheet published by OWASP, the following four parts define a complete threat modeling process, and will be followed in this project [32]: **(1) System Modeling**, **(2) Threat Identification**, **(3) Response and Mitigations** and **(4) Review and Validation**.

2.8.1 Step 1 - System Modeling

The first step in a threat modeling process is to model the system. The key question to address in this stage is "What are we working on?" [32]. This step is crucial as it lays the foundation for the whole threat model. Also, without a comprehensive understanding of the system, it is impossible to identify potential threats effectively. One technique for modeling a system is using data flow diagrams (DFDs). These diagrams should clearly illustrate the following elements [31]:

- Trust boundaries: Illustrates the trust level changes throughout the system.
- Data flows: Represents the data movement within the system.
- Data storage: Where the data is stored.
- Processes: The tasks responsible for managing the data.
- Any external entities present: An entity that interacts with the system through an entry point.

2.8.2 Step 2 - Threat Identification

When the system has been modeled, the second question from the manifesto should be answered, which is "What can go wrong?" [32]. Finding possible threats should be done with the system model in mind. One way of identifying and categorizing threats is by using a framework called STRIDE. This technique is an acronym for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege (see Table 2.1). Categorizing threats in this manner gives a structured picture of potential weaknesses and helps make decisions for the remaining steps of the threat modeling process.

The CIA triad is the foundation for the STRIDE framework, as it builds upon the same core principles: confidentiality, integrity, and availability. Additionally, other desired properties are non-repudiation, authenticity, and authorization.

Table 2.1: The STRIDE framework.

Threat	**Targeted Property**	**Description**
Spoofing	Authenticity	Pretending to be someone or something else to access information or perform actions.
Tampering	Integrity	Maliciously modifying data.
Repudiation	Non-repudiability	Claiming that some action did not occur or was not performed by someone.
Information Disclosure	Confidentiality	Obtaining information they are not authorized to access.
Denial of Service	Availability	Exhausting a system, making data and services unavailable to valid users.
Elevation of Privilege	Authorization	Gaining privileged access to perform actions they are not authorized to do.

Risk

After applying STRIDE, it is common to rank the discovered threat scenarios based on a risk level. The risk level of a specific threat scenario can be calculated by combining threat level and impact level [34], using the risk matrix in Table 2.2. The threat level refers to the likelihood that a threat scenario will occur, while the impact level refers to the severity of the consequences. The resulting risk level is used to prioritize threat scenarios and determine appropriate mitigation strategies mentioned in Section 2.8.3

Table 2.2: Matrix for determining risk level based on threat and impact levels [34].

Risk Level		**Threat Level**				
		Non	**Low**	**Medium**	**High**	**Critical**
Impact Level	**Critical**	Low	Medium	High	High	Critical
	High	QM	Medium	High	High	High
	Medium	QM	Low	Medium	High	High
	Low	QM	Low	Low	Medium	Medium
	No Impact	QM	QM	QM	QM	Low

Impact Level

Impact level reference the extent of the exploit, i.e., the damage potential and the number of affected components [32]. The damage potential increases significantly if the adversary is able to gain administrative privileges, assume full control over the system, cause a system crash, or access sensitive information. Additionally, the feasibility of any of these outcomes contributes to a higher impact level.

Furthermore, the number of affected components can be divided into two subsections: the number of linked data sources and the number of layers of the infrastructure that the attacker must traverse. If any of the numbers are high, it will affect the overall impact level of the exploit.

Threat Level

Threat level refers to the feasibility of the exploit happening. It is based on the following three criteria: Discoverability, Exploitability, and Reproducibility, as seen in Figure 2.4.

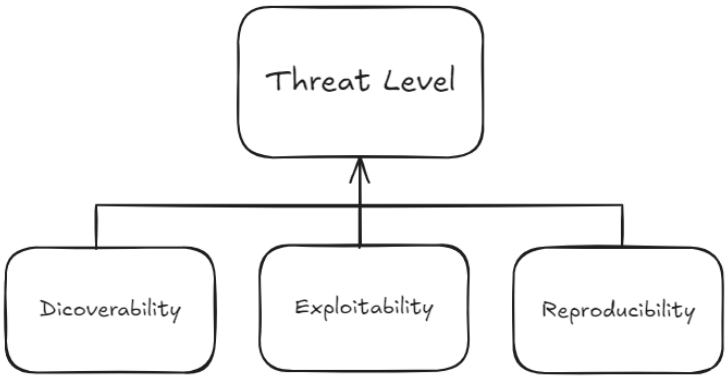


Figure 2.4: Threat Level Categories.

Discoverability refers to how easily an attacker can identify entry points to exploit a system. This can be determined by factors such as whether or not authentication is required to perform the exploit. If authentication is necessary, the difficulty of discovery increases, thereby reducing overall attack feasibility [32]. However, discoverability is not considered in the context of this thesis. The threat modeling process assumes that the adversary has already succeeded in installing a malicious app on the system. As a result, no entry points can be discovered, and discoverability will therefore be excluded from the threat level assessment for each exploit discussed in Section 5.1.

Exploitability, on the other hand, concerns how easily an attacker can exploit a discovered vulnerability. This thesis adopts an attack vector-based approach to evaluate exploitability, as outlined in the ISO/SAE 21434 standard [29]. This entails that the greater the level of remoteness required for an attacker to carry out an exploit, the higher the resulting attack feasibility rating. This assessment is based on the assumption that remote attack vectors, such as those accessible via the internet,

are exposed to a broader range of potential adversaries compared to vectors that require physical access, thereby increasing the likelihood of exploitation. Based on the identified attack vector, each case is assigned an attack feasibility rating of *very low*, *low*, *medium*, or *high*, as illustrated in Table 2.3.

Table 2.3: Attack vector-based approach [29].

Attack feasibility rating	**Criteria**
High	Network: The attack path originates within the network stack and is not constrained by any predefined access controls or security policies.
Medium	Adjacent: The attack path also originates within the network stack, but is constrained by a limited physical or logical connection.
Low	Local: The attack path is not originated from the network stack, thus the adversaries need direct access to the item.
Very low	Physical: The attack path requires physical access to be realized.

Reproducibility refers to how easily an adversary can replicate an attack either at a later time or across multiple components of the same system, or even on similar systems [32]. A key factor affecting reproducibility is the extent to which the attack can be automated. If the attack can be automated, the threat level increases since it allows the attacker to repeat the exploit easily or target many systems with little effort. However, in the case of this thesis, since there are multiple types of privileges that can be escalated or because multiple vehicles of the same type exists, each found exploit can be replicated easily across multiple vehicles or additional times within the same system. Therefore, the reproducibility criterion is assumed to always be high when discussing the threat level of found exploits in Section 5.1.

2.8.3 Step 3 - Response and Mitigations

The next step in the threat modeling process is to address the third key question, "What are we going to do about it?". Each identified threat must have one of the following responses:

- **Mitigate**: Implement a measure that will reduce the risk of the threat happening.
- **Eliminate**: Eliminate the system component responsible for the threat.
- **Transfer**: Delegate the responsibility to another entity.

- **Accept**: Do not perform any of the above-mentioned actions.

Furthermore, it is essential that the mitigation strategies are actionable and not solely hypothetical. In other words, the mitigations must be possible to integrate into the system. However, one of the responses is not favorable over another. Instead, the best response differs for each scenario and system.

2.8.4 Step 4 - Review and Validation

Lastly, the final question in the manifesto, "Did we do a good enough job?", is answered by step four. This is done via a review and evaluation from the stakeholders, not only developers or security experts. The review process focuses on at least one of the following areas:

- Do the Data Flow Diagrams (DFDs) or similar models provide an accurate representation of the system?
- Is there a response strategy in place for each identified threat?
- For the identified threats that should be mitigated, do the mitigation strategies reduce the risk level?
- Is the threat model documented and available to those in need?
- Is it possible to test and assess whether or not the mitigations, requirements, and recommendations have been successfully implemented?

3

Historical Android Permission Vulnerabilities

Several studies have been conducted over the years to find security issues in the Android permission model. This chapter reviews key findings from the literature and provides examples of how Android permissions have been used in potential exploits.

3.1 Bypassing Required Permissions

One example of an exploit, found by G. Tuncay [35], is the possibility of executing a side channel attack to access a user's phone number in the mobile app Viber. Ideally, Android apps should only access phone numbers through the `READ_PHONE_STATE` or `READ_PHONE_NUMBERS` permissions. However, Tuncay bypassed this boundary by setting the Android permissions `READ_CONTACTS` and `GET_ACCOUNTS` instead. The attack is possible since Viber sets the account name to the user's phone number. The root of the problem therefore lies in the fact that the data associated with a resource protected by permissions is not itself secured by corresponding permissions.

Another example, also discovered by G. Tuncay, involves unauthorized access to all files on an Android device without the required permissions [35]. Beginning with Android 10, apps with a target Software Development Kit (SDK) of level 29 or higher are restricted by a feature known as scoped storage, limiting access to only photos and media files. This contrasts with legacy storage, which grants access to any file on the device. The vulnerability lies in the fact that an app can escalate its privileges from scoped storage to legacy storage without the user's awareness. After acquiring the related permission for scoped storage, this privilege escalation can be achieved in two ways:

- Setting the target SDK level for the app to 28 or lower
- Setting `requestLegacyExternalStorage` to true in the app's manifest file

The adversary then updates the app after either of these actions have been performed. Now, when requesting the storage permission an additional time, the user will not be prompted to accept or deny the action through a permission dialog. Instead, it will be automatically granted. The vulnerability results from improper handling of app downgrades within the Android system.

3.2 Elevated Privileges

In the first exploit mentioned by G. Tuncay *et al.* in [36], the adversary creates an app that requests a normal or signature custom permission and assigns it a system permission group. The user then installs the app on their device. Next, the adversary changes the permission level to `dangerous`, and the user installs an updated version of the app. Since dangerous permissions are granted on a group basis, the user is now entirely unaware that the app acquired a `dangerous` permission without their consent, while also gaining access to other permissions within the same system group. This stems from how the Android permission system is structured. Once a user approves a single dangerous permission from a permission group, the app is allowed access to all other dangerous permissions of that same group if such a request is made. It is therefore possible for an attacker to exploit this behavior to elevate its privileges by quietly acquiring multiple sensitive permissions through a single approval within any permission group.

Another attack mentioned by the same authors [36] is based on reusing the name of a custom permission to gain access to resources protected by a `signature` permission of the same name. For the exploit, the adversary creates two apps. First app A, which reuses the exact name of the custom permission from the victim app to spoof the permission, but with the protection level set to `dangerous`. Then app B, which solely requests said permission. Two different apps are needed to perform the exploit because Android does not allow two apps defining permissions with the same name to exist simultaneously on the same device.

The attack works as follows (see Figure 3.1 for visualization of execution order), the user first installs app A followed by app B. After app A has granted permission to app B at runtime, it can be uninstalled to remove the custom permission definition from the device. This is then followed by an installation of the non-malicious victim app on the device, which defines a custom permission with the same name as app A did. After installing the victim app, app B can launch an attack on the victim app to access its components protected by a `signature` custom permission, even if app B itself is not signed with the same certificate. This is due to the permission of the same name being previously granted by app A.

Figure 3.1: Order of execution for attack found by Tuncay *et al.* [36].

Furthermore, a study conducted by R. Li *et al.* [37] highlights another issue regarding custom permissions. In the exploit, the focus is on dangling custom permissions. In the exploit, the adversary creates two apps, app-d and app-r. App-d defines a custom permission `cp`, with protection level set to `normal`. App-r then requests that permission. In addition, two different versions of app-d exist, with the first being the one to remove the definition for `cp`, and the second redefines its definition

to `dangerous`. By following the order of execution visualized in Figure 3.2, app-r obtains the custom permission `cp` and thus gains elevated privileges.

Figure 3.2: Order of execution for the exploit found by Li *et.al.* [37].

3.3 Summary of Findings

This chapter presented some examples of insecurities related to the Android permission model. Although these vulnerabilities have all been patched, they remain relevant since they highlight that historically the model has not been fully secure, and suggest that new vulnerabilities will likely be discovered in the future. Furthermore, the vulnerability found by G. Tuncay [35] regarding accessing a user's phone number highlights the need for more fine-grained control of resources through permissions, which aligns with the underlying purpose of the thesis. However, this project does not aim to assess the security of Android's permission model. **As such, Android permissions will be treated as secure system within the scope of this work and will therefore not be investigated further.**

4

Methods

The following chapter presents the overall approach taken for completing the project, divided into smaller steps that are addressed in order of execution.

4.1 System Setup

Testing and development during the project was carried out using the setup visualized in Figure 4.1. Out of all the sub-components present on the productivity hub, two of them are the focus of this thesis: the Android ECU and the CAN Interface ECU (CIE). As the name suggests, the Android Automotive operating system is located in the Android ECU. External communication with the Android ECU is done through a USB-C connection using Android Debug Bridge (ADB).

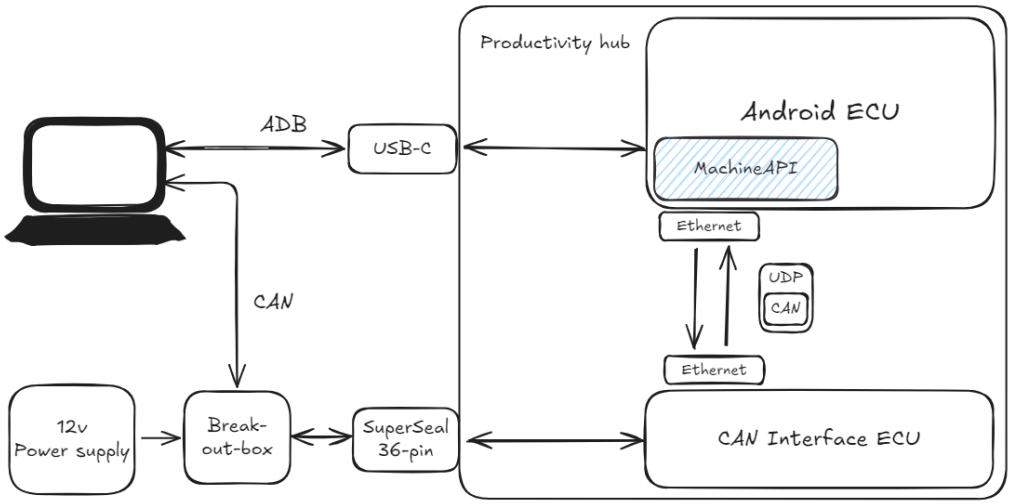


Figure 4.1: Overview of system setup.

When the productivity hub is installed in a vehicle, communication with the IVN is done through a SuperSeal connector, which provides a single connector for several data interfaces such as multiple CAN buses and power to the productivity hub. During development, a so-called break-out box is used to separate all the different interfaces from the SuperSeal connector into individual connectors. This enables the connection of a CAN-to-USB adapter to send CAN messages from a computer to a

specific CAN bus, while also powering the productivity hub using a 12-volt power supply.

As shown in Figure 4.1, the Android ECU and the CIE are physically separate devices. This means that the Android Automotive operating system does not have direct contact with the IVN and can only access it via the CAN Interface ECU. The two ECUs are connected on an internal network with Ethernet interfaces. The CAN data is encapsulated in UDP packets when transmitted between the devices. This separation enhances access control, provides multiple layers of security, and ensures proper verification of the data exchanged between the productivity hub and the IVN.

4.1.1 Handling CAN Data

There are two different ways of handling CAN data in the productivity hub. First, through precise definitions of all bits and their respective positions in the frame, and the frame ID. This is known as a machine signal, and the APIs that handle reading such signals only return the parsed value of the requested signal. The other method is to request a specific frame ID, without being concerned about which data the frame contains, and these APIs return the entire CAN frame to the caller. Machine signals ensure detailed control over which data each app needs and requests, but they also prevent flexibility. By having an app register frame IDs to read, the data can be parsed more dynamically, allowing for sending data across many frames. An example of such a use case is sending data using the CAN-TP protocol, which is a transport protocol over a CAN bus that allows transmission of data larger than 8 bytes [38].

MachineAPI is the Android service responsible for providing communication between apps running on the Android ECU and the CIE. It provides all necessary APIs for apps that need to interact with the IVN. Every read or write method call is forwarded to a permission validator to ensure that the calling app has the correct permission. To acquire vehicle configuration data, MachineAPI communicates with the ConfigHandler service. This service parses vehicle configuration parameters and returns various types of information, such as which machine signals are able to be read.

4.1.2 Installing Apps on the Infotainment System

The aforementioned model of the productivity hub does not have Google Play Services installed, resulting in no Google Play Store. Thus, the user cannot install any third-party apps onto the infotainment system. Furthermore, all apps on the system must go through a signing process with OEM certificates to be installed. Essentially, for a successful app installation, a unique token specified in its manifest file has to correspond with the OEM certificate verified during the installation process. This gives the OEM complete control over which apps are present and their intended functionalities. This means that once an app is installed, it is implicitly trusted by the system.

This thesis states that apps should be explicitly trusted instead. In the scenario that an adversary manages to compromise existing apps or install their own, additional verification should be in place. Apps should not be implicitly trusted solely because they are present on the productivity hub. Therefore, it is necessary to explore and implement additional protections.

4.2 Threat Analysis and Risk Assessment

The threat analysis conducted in this thesis was based on the information presented in the ISO/SAE 21434 standard [29] regarding required items and how these were to be obtained. Since the standard does not include a definite guide on how to perform a TARA, internal documentation was used as a guide instead. Furthermore, since not all aspects of the standard apply to the thesis, the threat modeling process was instead based on the OWASP guide [32]. This process is further described in the next section.

4.3 Approach for Threat Modeling

The OWASP guide [32] used for the threat modeling process is divided into different steps, as mentioned in Section 2.8. The following subsections present a more in-depth description of the approach taken for each step in the process, from system modeling to review and evaluation.

4.3.1 Step 1 - System Modeling

The first step was to get a better understanding of potential entry points, assets, and the general flow of data within the system. This can be visualized by the Data Flow Diagram (DFD) in Figure 4.2, which illustrates how CAN frames are sent between the CIE, MachineAPI, and the app. It also highlights which Android permissions are needed to traverse the privilege boundaries. There are two points of entry to the system: the app and the CIE. The CIE is responsible for sending and receiving CAN frames to the IVN, whereas the app requests specific frames or signals and receives the data if it has the correct permissions.

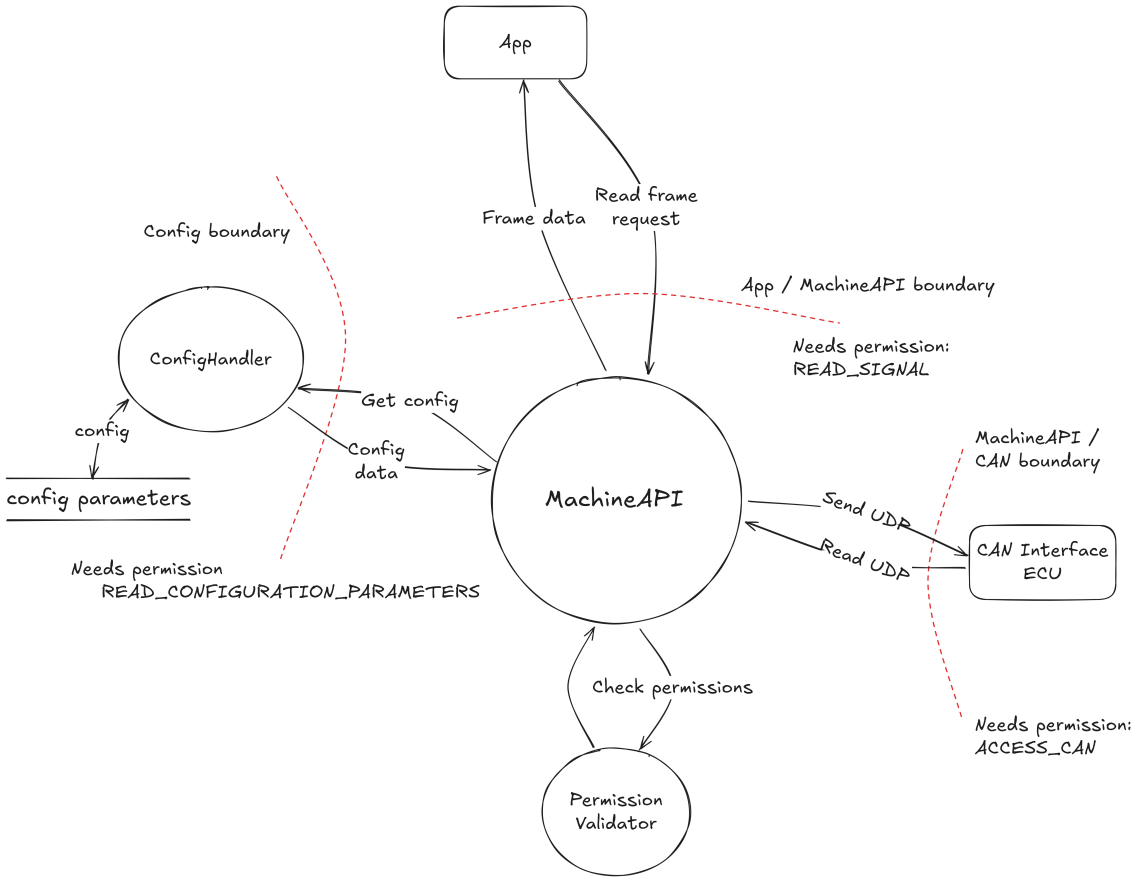


Figure 4.2: Data flow diagram for the process of an app reading CAN frames.

Both the CIE and the app communicate with MachineAPI, which in turn checks the validity of permissions from the app through the PermissionValidator. This is necessary since different parts of the system should only be accessed if the proper permissions are held. To use the APIs exposed by MachineAPI the app must have the `READ_SIGNAL` and `WRITE_SIGNAL` permissions. In order for MachineAPI to communicate with CIE, it must have the `ACCESS_CAN` permission.

The valuable asset in the system is the CAN bus, since it gives access to the whole IVN. The project, therefore, aims to protect this entity by implementing additional protection layers in the system.

4.3.2 Step 2 - Threat Identification

Once a model of the system had been created, threats could be identified using the DFD and examining the source code. As mentioned in Section 2.8.2, STRIDE was used during the Threat Identification to sort the identified threats into different categories. Each threat was also subject to several iterations of the ranking process for Risk, Impact and Threat level. This was done both as more threats were identified and could be weighted against each other, but also as more knowledge about the system resulted in different perspectives and reasoning about the threats.

4.3.3 Step 3 - Response and Mitigations

The third step involved conducting a risk mitigation analysis. This was carried out by again examining the DFD and reviewing the codebase, along with the information about the identified threats to create potential mitigation strategies. The process of developing the mitigations are further described in Section 4.5 and its subsections. Furthermore, section 5.2 details how each mitigation correspond to the identified threats.

4.3.4 Step 4 - Review and Validation

The fourth and final step in the threat modeling process involved reviewing and validating the entire procedure. This was done mostly in regards to if the implemented mitigation strategies actively secures the system from the identified threats or not. By returning to the original DFD and comparing it to the new source code with implemented mitigations, an updated version was created and could be examined the same way as in Step 4.3.1. The outcomes of this step are presented in Section 6.2.

4.4 Creating a Testing App

A testing app was created to aid in the development of the access control mechanisms and automate different actions for benchmarking. The app uses MachineAPI in the same ways as regular apps on the productivity hub, thereby enabling the simulation of how a malicious app could behave and function. The user interface is very straightforward, consisting of buttons and forms to perform different actions, such as sending or reading predefined signals or sending a custom frame using the input data from a form.

During development, the workflow consisted of using methods exposed by MachineAPI, investigating the data flow using logging and debuggers, and implementing protections in MachineAPI where needed. Once a mitigation strategy had been implemented, the functionality could be tested by attempting to perform both permitted and forbidden actions with the testing app.

4.5 Implementing the Access Control

Part of our proposed access control system is built using Android permissions in a layered fashion. The lower layers control access to the most sensitive system components, and higher layers are more granular, such as listing allowed CAN frame IDs for specific apps. Figure 4.3 shows an overview of the different layers. In addition to Android permissions, system configuration parameters are utilized for granular control over app-specific privileges.

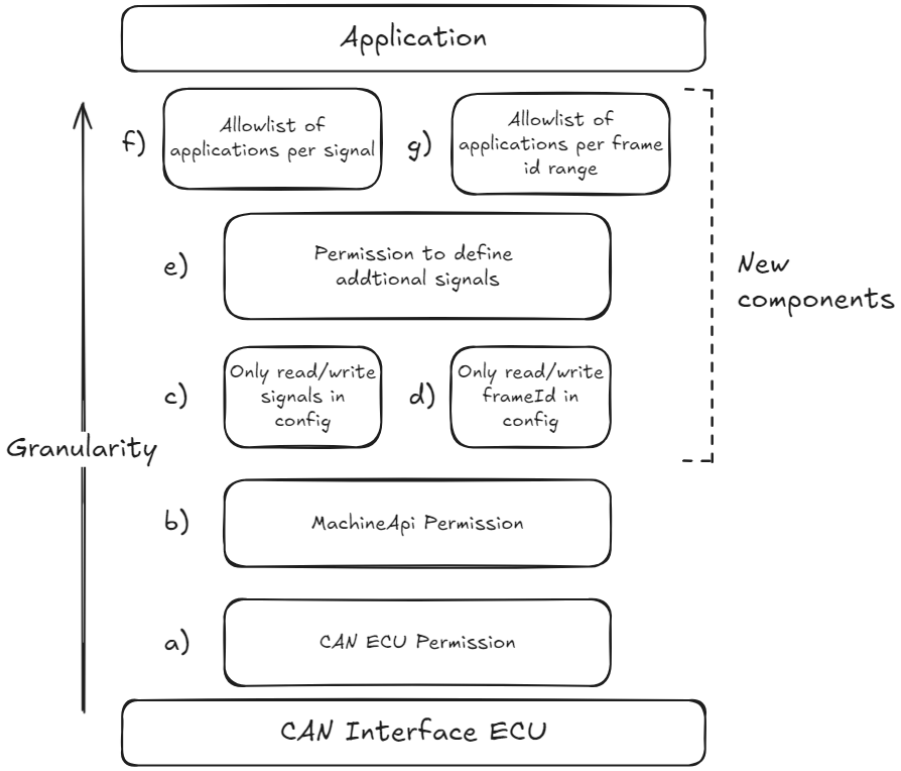


Figure 4.3: Different components of the access control organized in order of granularity.

The proposed system contains the following components:

(a) **Android permission for sending UDP packets to the CIE network interface**
This is the most essential protection, contained at the operating system level of the Android platform. This permission is only given to system apps and verified by special platform signatures. In the case of this thesis, only MachineAPI has this permission. This means that any app wishing to communicate with the CIE is required to use MachineAPI and its exposed library methods.

(b) **Android permission for interacting with MachineAPI**
Essential for protecting the APIs within MachineAPI, which in turn talk to the CIE. The permission can be defined by internal app developers, and proper usage can be controlled through internal code reviews. An addtional permission has been added to this layer, specific to the ability to read CAN frames.

(c) **Defining and enforcing a list of allowed CAN signals**
These signals are defined in the system configuration file. This ensures that apps can read and write only the signals specified in the OEM provided configuration file.

(d) **Defining a range of allowed CAN frame IDs**
For CAN frame IDs, the system configuration file defines specific hexadecimal

ranges of IDs that can be read and written to the CAN bus.

(e) **Android permission to define new signals to read and write in addition to the configuration file**
A special permission that can be granted to apps that need to define additional signals at runtime, which are not already specified in the vehicle's configuration file.

(f) **Listing which apps are allowed to read and write a certain signal**
Each configured signal includes a list of app package names that are allowed to read and write it.

(g) **Listing which apps are allowed to read and write a frameID within a certain range**
Each configured frameID range includes a list of package names that are allowed to read and write it.

The following subsections present how these components were implemented and the thought process behind them.

4.5.1 System Configuration

The system configuration is the underlying component of layers C,D,F and G. The current solution for configuring the productivity hub to a specific vehicle type is a JSON file containing the necessary configuration parameters. This configuration scheme was extended to include permitted read and write signals and a filtering mechanism specifying which CAN frame IDs can be read and sent. Each rule for permitted signals or frameIDs is tied to a specific app, ensuring fine-grained control over which application has access to what resources. The configuration file also ensures that only relevant frames and signals for a particular type of vehicle are used. The configuration file serves as the authoritative source for IVN access control, and the file's integrity must therefore be protected. To ensure this, preventative measures have been implemented to prevent overwrites or modifications of existing signal and frame definitions, using the MachineAPI.

Specific ranges of hexadecimal values are defined in the configuration file in the following way. The configuration in Listing 2 shows a filter being applied to CAN bus 4. The `filter` and `mask` entries are used to calculate if a frameID is within the valid range through equation 4.1:

$$frameID \ \& \ mask == filter \tag{4.1}$$

From the calculation, IDs ranging from `0x18F00000` to `0x18FFFFFF` are valid in the example in Listing 2. Finally, the package `com.example.mypackage` is the only package allowed to read frames with IDs that adhere to the calculated filter.

```
"com.example.machineapi.allowedCanIdsRx": {
    "type": "array",
    "value": [{
        "canBus": 4,
        "filter": "0x18F00000",
        "mask": "0x1FF00000",
        "allowedPackages": ["com.example.mypackage"]
    }]}
```

Listing 2: An entry in the configuration file declaring a range of allowed CAN frameIDs with a single allowed package.

As outlined in Section 2.1, content providers are used to facilitate data sharing between apps in Android. The configuration file is made accessible to apps via the content provider implemented in the ConfigHandler. Access to the ConfigHandler is protected by a specific permission, as illustrated in Figure 4.2. As a result, only apps that have been granted the appropriate permission are authorized to access configuration parameters.

4.5.2 Permission Control for Registering New CAN Frames

Section 4.1.1 mentions how CAN is handled in two different ways on the productivity hub. The original system allowed for new CAN frames to be registered by an app as long as it had permission to read and write machine signals. However, this does not adhere to the PoLP since it gives the application more access than needed. Therefore, a new permission has been added to layer B, regarding reading frames. This ensures that any application that needs to register new frames must explicitly declare the permission in its manifest file.

4.5.3 Define Permitted CAN Signals and FrameID Ranges

By having defined lists of signals and ranges of allowed CAN frame IDs that an app can register or write, it adheres to the PoLP where only the minimum access necessary is granted. Layers C and D extends the configuration file with a list of allowed signals that can be defined. For frameIDs, a developer specifies a filter and a mask for each allowed range of transceiver and receiver frames. By performing a bit-wise AND operation between the frameID that the app intends to register or write and the mask, and then comparing the result to the filter, it can be determined whether the frameID falls within the allowed range. If it returns true, and thus is within range, the app is allowed to register or write the frame. However, if it returns false, MachineAPI throws a `SecurityException` to the calling app.

4.5.4 Permission Control for Defining New Machine Signals

Previously, two permissions (to read and write signals respectively) were used to limit how the developers could define new signals not present in the configuration

file. However, this is not optimal since solely being able to, e.g., read machine signals should not automatically give full permission also to define new ones. Therefore, layer E adds two new permissions for defining new read and write signals. Preventative measures were also implemented to prevent overwrites of signals already existing in the configuration file.

4.5.5 Enforce App-Level CAN Access

The final layers F and G consist of app level verification using the package names defined in the system configuration. This allows the rules for signals and CAN frameIDs (layers C and D) to be enforced on an app basis. Checks are performed to verify that the app requesting a certain signal or frame is indeed allowed to do so. If the app is not allowed to read or write a specific signal or frame, MachineAPI will throw a `SecurityException` to be handled by the calling app.

4.6 Benchmarking

To ensure that the implementation of the access control did not negatively impact the performance of the productivity hub, two benchmark tests were conducted to measure the difference in execution time compared to the original system.

4.6.1 Writing Frames from Android to the CIE

This test was performed to verify if the additional checks and computations from the access control implementation resulted in a significant increase in the time between when an app does a method call to write a frame, to when the frame is actually sent. The method `SystemClock.elapsedRealtimeNanos()`, which gives the time since system boot in nanoseconds, was used to measure the elapsed time between the events. By recording this value at two points in time, the elapsed time between those points could be calculated by taking the difference between the recorded values. Values were recorded after the initial method call from the app, and immediately after the method sending the UDP packet had finished. This measured the time taken from the top of the software stack (starting at the app layer), through the system, down to the bottom layer where a UDP packet is transmitted.

Each test run was composed of a total of 1000 transmitted frames, sent with an interval of 100 frames per second. Such an interval puts a higher performance load on the system compared to normal use case, which was measured to be approximately four frames per second. The time between the initial method call and the frame being sent was recorded for each frame. Using the timings for all frames, an average delay was calculated for that particular test run. A total of 20 test runs were concluded, ten with the original code and ten with the new implementations. Based on this the final average delay was calculated for both variants, using the average of each individual run.

4.6.2 Reading Incoming Frames from the CIE

Similarly to the previous test, two timestamps were recorded to measure the elapsed time. In this test, the first timestamp is set once a UDP packet has been parsed to a CAN frame, which is then accessed as an attribute of a CAN frame object. The second timestamp is taken right after a valid message from the `readFrameMsg` method has returned. As in the previous test, 1000 frames were read in each run and the send interval was 100 frames per second.

5

Results

This chapter presents the project's results based on the threat model, including the detected threats and their respective mitigation strategies. This is followed by an evaluation of how well the proposed solution aligns with the EN 18031-1:2024 standard [27], and concludes with results from the benchmarking tests.

5.1 Identified Threats

This section presents the results of the second step in the threat modeling process. The threats were identified by examining the source code, performing system testing, and by consulting the system's DFD, illustrated in Figure 4.2, which was developed in Section 4.3.1. All threats are presented in Table 5.1.

Table 5.1: Threats found in the system, with their corresponding STRIDE category, possibility, and impact level.

Id	**Threat Type**	**Short Description**	**Threat level**	**Impact Level**	**Risk Level**
T1	Spoofing	Send CAN frames or signals that appear to originate from other ECUs	Low	Critial	Medium
T2	Repudiation	Capture a legitimate CAN message to perform a replay attack	Low	High	Medium
T3	Information Disclosure	Opening socket to read CAN data	Low	High	Medium
T4	Denial of Service	Overload the CIE with CAN frame requests	Low	High	Medium
T5	Elevation of Privilege	Gaining privileged access to the CAN bus	Low	High	Medium

T1: This threat serves as the foundation for the motivation of this thesis. It exists in the original system because apps that hold the `WRITE_SIGNAL` and `READ_SIGNAL` or `ACCESS_CAN` permissions have no additional restrictions regarding allowed CAN

frames and signals. Should these apps be compromised or the permissions be obtained, an adversary could send any type of CAN frames or signals from the productivity hub.

Impact
CAN frames can be created so that they appear to originate from another ECU inside the IVN. Depending on the contents of the frame, it is possible for an adversary to perform malicious actions with the vehicle, such as controlling its movement.

This scenario allows for a broad level of control over the vehicle and can affect multiple components across various sub-systems, and is thus rated as critical. Moreover, this scenario of unrestricted CAN access can be seen as the foundation of the remaining found threat scenarios. In which each one eventually lead to unrestricted CAN access.

Threat Level
There would be two ways to exploit this scenario, through compromising existing apps or by gaining permissions with a custom app. Developers may unintentionally create public methods or expose services, resulting in other applications being able to send CAN data via these existing applications. However the possibility of such a scenario is deemed low.

The possibility of gaining the `WRITE_SIGNAL` or `READ_SIGNAL` permissions on a custom app without the manufacturers intent or through exploits are deemed low as well. Resulting in a low overall threat level.

Risk Level
Despite the critical impact level, due to the actual low threat of the scenario the overall risk results in a medium level.

T2: Due to the lack of encryption in the CAN protocol, all messages can be copied in plain text and sent again by an adversary at a later time, a so called replay attack. The same permissions listed in T1 are required for CAN access.

Impact Level
The damage potential level is determined based on which CAN messages the adversary manages to read. Still, there is a possibility that the adversary, e.g., can access vehicle movement messages, in which case they can later control the vehicle themselves. Since such a scenario is possible, the impact level is high.

Threat Level
The adversary needs direct access to an app in order to perform the exploit, thus making exploitability low. The attack is also possible to automate, thus adding to the high reproducibility. However, accessing the required permissions (or bypassing them completely) is a difficult task, and thus reproducibility can be neglected. As a result, the threat level is considered low.

Risk Level
Due to the low threat level, but the high impact level, the overall risk level is considered medium.

T3: As can be seen in Figure 4.1 the Android ECU and CIE communicates via UDP, and data is being sent on 6 ports (one for each CAN bus). At initialization, MachineAPI binds UDP sockets to these ports. If an app manages to bind its own sockets to one of those ports first, it prevents MachineAPI from binding to the same port. This results in the app being able to communicate with the CIE via UDP, bypassing MachineAPI entirely.

This scenario requires an adversary to unbind the present sockets and create new ones for an app that they control, either through crashing MachineAPI or by creating sockets before MachineAPI can.

Impact Level
The impact level is high due to the high damage potential where the adversary would be able to read and send any CAN data, potentially leading to complete control over the IVN.

Threat Level
The likelihood of binding sockets before MachineAPI is rather low due to the startup process of the productivity hub, in which MachineAPI is launched well in advanced of other apps. Another approach is overloading the memory of the MachineAPI to the extent at which Android itself terminates the service [39]. This second scenario also has a low possibility of happening, and as a result of the exploit being difficult to perform the threat level is considered low.

Risk Level
The associated risk level is set to medium due to the high impact level and low threat level.

T4: The CIE is vulnerable to a Denial of Service (DoS) attack if an app can send a large number of CAN frames to the CIE and maximize the capacity of the bus. This would prevent valid data from going through. The spoofing can, for example, be done by setting the arbitration ID of the CAN frame to the lowest value present on the bus. This gives the node the highest priority, making other nodes on the bus (which might have valid CAN frame requests) pause their transmission.

Impact Level
The damage potential is high since it is possible for the adversary to fully control the CAN bus and determine which frames and signals are transmitted.

Threat Level
However, to perform such an exploit, one would have to acquire the `ACCESS_CAN` and `WRITE_SIGNAL` permissions, which is relatively difficult. The adversary also needs a direct access port to MachineAPI to perform the exploit, thus making exploitability low. As a result, the overall threat level is considered low as well.

Risk Level
The risk level associated with the vulnerability is considered medium due to the high impact level and low threat level.

T5: Elevation of privilege in this scenario would entail that the app is able to increase its privilege level and bypass the application sandbox entirely, or obtain necessary permissions it is otherwise not allowed to. This would in turn increase the level to which it can access the CAN bus, i.e., which frames and signals it is allowed to read and write.

Impact Level
The impact level depends on the extent of the privilege escalation, but could ultimately lead to bypassing protection layers and result in complete control over the IVN. Thus, this threat is given a high impact level.

Threat Level
Elevating privileges is a complex task, requiring exploiting the security of the Linux kernel. In normal operations, an app's permissions are decided at install time based on its manifest file and cannot be changed afterwards. Bypassing this behavior would require advanced manipulation of Androids permission system, thus making the exploitability rating low.

Risk Level
Even though the impact level is high, the difficulty in performing the attack results in an overall medium risk level.

5.2 Threat Scenario Mitigations

The following mitigations are found to be effective for each threat stated in Table 5.1.

T1: In the proposed solution, the CAN signals and frames that an app can read and write have been limited to specific IDs or ranges. Thus, it is no longer possible for an app to write arbitrary signals via MachineAPI. Furthermore, only apps specified in the configuration file can communicate with the CAN bus. As a result, the threat has been mitigated.

T2: This threat has been mitigated in the proposed solution, since only apps explicitly stated in the configuration file can read from and write to the CAN bus. Thus, a malicious app cannot gain access to the CAN bus to perform the replay attack.

T3: To perform the exploit, the adversary has to either manage to bind a socket to a port before MachineAPI does, or make Android terminate the MachineAPI service completely. In order to send UDP data to the CIE the `ACCESS_CAN` permission is needed. It is, however, possible to read UDP data without the permission. Both taking over the sockets and bypassing permissions are deemed very challenging to achieve. Therefore, no action will be taken to mitigate the threat scenario, treating it as accepted.

T4: The threat scenario is mitigated as a result of the new layers implemented in the access control. Only CAN data specified in the configuration is allowed, making it harder to flood the CAN bus with malformed data.

T5: This threat has been accepted. However, depending on the level of escalation achieved, the remaining security layers are able to protect the CIE. Bypassing the security mechanisms of Android and the Linux kernel are deemed very complex and would most likely require finding zero-day exploits. Protection against such exploits can only be done after the attack has been discovered and patched.

5.3 Updated Data Flow Diagram

As a result of Step 4 in the threat modeling process, revisiting and re-modeling the system leads to updated DFDs. The updated DFD in Figure 5.1l, showcasing the scenario of writing a frame, illustrates how two additional privilege boundaries have been added inside MachineAPI. After passing the first boundary corresponding to layer B, the updated flow includes checks for frameID (layer D) as well as the package name for the calling application (layer G).

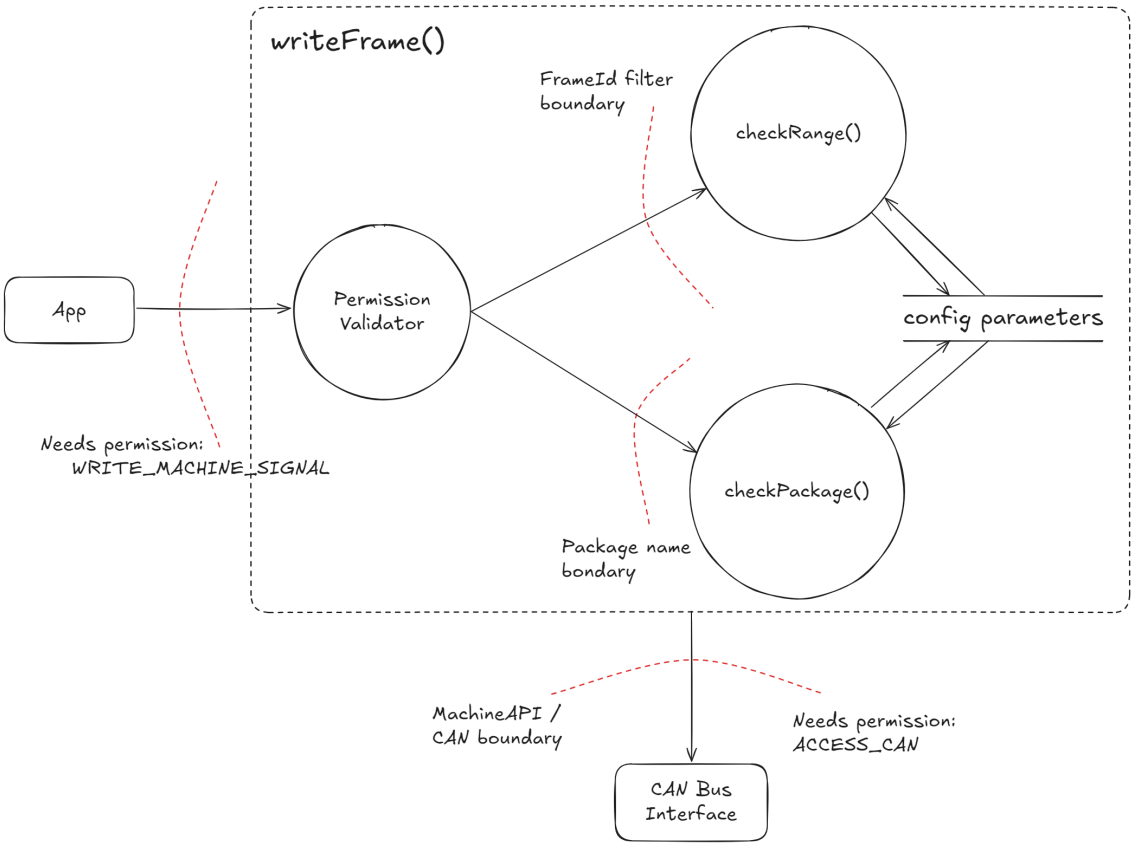


Figure 5.1: Updated DFD for `writeFrame()`.

5.4 Compliance to the EN 18031-1:2024 Standard

Based on the EN 18031-1:2024 [27] standard mentioned in Section 2.6.1, the following three requirements are deemed relevant to the thesis. Each requirement is assessed as either *pass* or *fail* based on the assessment criteria specified in the documentation for the standard [27].

Table 5.2: Requirements from the EN 18031-1:2024 standard and their corresponding STRIDE threat categories [27].

Requirement	**Description**	S	T	R	I	D	E
ACM-1	Access control mechanisms should be implemented to secure the assets from unauthorized parties		x		x	x	x
ACM-2	The implemented access control mechanisms in ACM-1 ensure the assets are protected		x		x	x	x
GEC-6	All input which could negatively impact the assets should be validated by the equipment		x		x		

ACM-1: PASS
The requirement is considered relevant for the project since public accessibility of the CAN bus is not intended, no physical or logical measures are already in place to limit access to authorized entities, and there are no legal implications that prevent the implementation of access control mechanisms for the CAN bus. Furthermore, access to the CAN bus has been made more fine-grained and narrow through the implemented mechanisms described in Section 4.5. As a result, the requirement passes the assessment criteria.

ACM-2: PASS
This requirement is closely linked to ACM-1, as it essentially ensures that the access control mechanisms have been properly implemented. Since all access control measures outlined in ACM-1 are designed to protect the CAN bus from unauthorized access, this requirement also passes the assessment criteria.

GEC-6: PASS
Since the productivity hub does not handle any user input, "input" in this context refers to any data sent by an app. The following mechanisms have been introduced or enhanced to ensure proper input validation for any action involving access to the CAN bus, as a means to prevent unauthorized access to the productivity hub.

Firstly, all variables regarding the CAN bus have explicit types. For instance, this ensures that a CAN frame cannot be of any other type than a `String`. Secondly, checks have been implemented to ensure only CAN frames within a specific ID range, specified in the configuration file, can be handled by the system. This prevents

apps from, for example, attempting to register frames they are not authorized to access. Lastly, it is not possible for an app to define a signal already present in the configuration file. This prevents the configuration file from being overwritten by input from a malicious app. As a result, the requirement meets the assessment criteria.

5.5 Benchmark Tests

This section presents the results of the benchmark tests, in which the average execution time was compared between the system with the access control mechanisms enabled and its original, unmodified state. Two tests were conducted to measure the time required for a CAN frame to travel between the Android application and the CIE, in both directions.

5.5.1 Writing Frames from Android to the CIE

The following measurements were taken in the case of Android sending a CAN frame down to the CIE. Measurements for a system with active access control mechanisms are found in Table 5.3, whereas Table 5.4 shows a system in its original state without any modifications.

Table 5.3: Execution time in milliseconds with access filters enabled.

Test run	**Average time difference (ms)**
1	0.506
2	0.531
3	0.532
4	0.528
5	0.539
6	0.539
7	0.525
8	0.525
9	0.519
10	0.529
Average time difference across all runs	**0.527**

Table 5.4: Execution time in milliseconds with access filters disabled.

Test run	**Average time difference (ms)**
1	0.386
2	0.375
3	0.388
4	0.330
5	0.347
6	0.330
7	0.372
8	0.338
9	0.356
10	0.342
Average time difference across all runs	**0.356**

To determine the average increase in execution time between the original system and the system with the newly implemented access control mechanisms, the difference in the *Average time difference across all runs* values was calculated between the two tests. This gave an average time increase of 0.171 ms. Thus, introducing the access control mechanisms increases the execution time by approximately 48%.

5.5.2 Reading Incoming Frames from the CIE

The following measurements were obtained when reading CAN frames sent from the CIE to the Android app. Table 5.5 presents the execution time for each run with active access control mechanisms, whereas Table 5.6 showcases the system in its original state with no filters enabled.

Table 5.5: Execution time in milliseconds with access filters enabled.

Test run	**Average time difference (ms)**
1	0.139
2	0.141
3	0.140
4	0.136
5	0.140
6	0.136
7	0.146
8	0.139
9	0.150
10	0.145
Average time difference across all runs	**0.141**

Table 5.6: Execution time in milliseconds with access filters disabled.

Test run	**Average time difference (ms)**
1	0.154
2	0.137
3	0.129
4	0.138
5	0.145
6	0.145
7	0.130
8	0.129
9	0.132
10	0.136
Average time difference across all runs	**0.137**

Comparing the average difference between the two scenarios shows that using access filters adds 0.004 ms in processing time, equivalent to an approximate increase of 1.59%.

6

Discussion

The following chapter outlines the thought process behind trusting Android's security system, explores abandoned ideas for the project, and suggests potential directions for expanding the project further in the future.

6.1 Trust in Android Security

Although Android has been the target of numerous attacks throughout the years, as discussed in Chapter 3, we have chosen to trust Android's built-in security features (such as the permissions system and content providers) to help protect the IVN from unauthorized access. This decision is based on the fact that Android is a widely used operating system with a large team at Google continuously working to improve its security. Additionally there is also the organizations and individuals providing development and building the security for the Linux kernel. Given the scale of resources and expertise available for these two projects, we believe it is more practical and secure to trust the Android platform rather than attempting to develop our own security infrastructure from the ground up.

6.2 Threat Modeling Evaluation

The TARA conducted in this project initially followed the OWASP Threat Modeling Guide [32], which provided a structured set of steps for the analysis. However, since the project focuses on autonomous vehicles, the ISO/SAE 21434 [29] standard was incorporated as a complementary reference for steps 1 and 2, as it offers more details specifically tailored to the automotive domain. Since the standard does not prescribe a specific methodology, an internal Volvo TARA guide [34], which is based on ISO/SAE 21434, was also followed as a way to align the process more closely with industry practices. As a result, the TARA used in this thesis is a hybrid approach, drawing from multiple sources and customized to suit the scope and constraints of the project.

Since the ISO/SAE 21434 standard does not present a strict guide on how to perform the TARA, it is up to those conducting the TARA to decide on the best approach. Therefore, the hybrid approach taken in this project is the most optimal one based on the project focus.

6.3 Revisiting the Research Questions

As mentioned in Section 1.3, the project aimed to answer three questions related to the Android operating system, CAN bus access, and industry standards and regulations. The following section aims to evaluate the project based on the three questions and present an answer to each one.

Q1: Which historical and existing security-related vulnerabilities exist in Android-based systems?
Chapter 3 presents some previous vulnerabilities in the Android operating system. These vulnerabilities highlight that historically, Android has been susceptible to different attacks regarding its permission model and unauthorized privilege escalation. However, since the objective of the thesis was not to analyze the security of Android's permission model in depth, the research question was intended only to highlight its limitations.

Q2: What is the approach to designing a granular access control system in Android Automotive for CAN bus access based on the PoLP?
The methodology for designing and implementing an improved access control system for the productivity hub is presented in Chapter 4. The proposed solution incorporates multiple layers of protection, including additional validation of CAN frameIDs and signals, as well as precise control over which apps are permitted to interact with the CAN bus. The added protection layers together act as a way to restrict an app's access to the CAN bus to the least possible degree, thus adhering to the PoLP.

Q3: What is the method for developing security-focused software while ensuring compliance with regulations and industry standards?
Both a TARA and a threat model were performed to comply with the standards outlined in Section 2.6. The project aligns with the ISO/SAE 21434 standard [29], as a TARA was carried out following the guidelines specified in the standard's documentation. Additionally, the EN 18031-1:2024 standard [27] from RED is adhered to through the approaches detailed in Section 5.4, with STRIDE selected as the threat modeling framework.

6.4 Quality of benchmark tests

When attempting to measure execution times in the manner described in Section 4.6, there are numerous uncertainties that may affect the accuracy and reliability of the end results. Such uncertainties include OS-level thread scheduling, caching of data and dynamic frequency scaling on the CPU.

As shown in the tables from Section 5.5, the execution time increases in both directions with access control filters enabled. This outcome is expected, as the introduction of additional processing often leads to an increase in execution time. However, the measured increase is minimal in both tests. Given that the system is not subject to strict real-time constraints, such a small variation in execution time does not significantly impact overall system performance or reliability. Therefore, the observed

time differences can be considered negligible, and no corrective action is considered necessary.

The tests for reading from CIE showed a significantly less time increase compared to the writing frames to CIE tests. Potential reasons for this difference comes down to the testing methodology as well as numerous uncertainties associated with measuring execution times. Some system modifications were required to retrieve the timestamps in a sensible matter, resulting in measurements that may not accurately reflect how a real world system would behave and perform.

Despite the potential uncertainties and difference in measured performance impact, the measuring approach was deemed to be appropriate for the goal of the benchmarking, to answer if there is a negative performance impact using the new access control mechanism. Using `SystemClock.elapsedRealtimeNanos()` is a better approach compared to using external clocks, so called wall-clock timing. External clocks can be susceptible to changes or be de-synchronized during measurement. While the internal time since boot is unaffected by external changes such as timezone and offer nanosecond resolution for the timestamps.

6.5 Future Work

This section presents different concepts and ideas that serve as potential paths for further development and expansion of the project in the future.

6.5.1 Extend Scope to Include Over-the-Air Updates

Exploring how to securely perform Over-the-Air (OTA) updates for the system is highly relevant to the project since such updates offer many advantages to the infotainment system, while simultaneously introducing many risks. For example, it simplifies the process for the OEM to fix bugs and patch identified vulnerabilities, as the vehicle no longer needs to be taken to a repair facility. Instead, the owner can simply download the latest software update, where those bugs have been removed.

However, as mentioned previously, having the possibility of OTA updates also simultaneously introduces more vulnerabilities and attack vectors to the system. For instance, adversaries might inject malware through Man-In-The-Middle (MITM) attacks during software download, or execute supply-chain attacks on the cloud infrastructure. This requires the need for end-to-end encryption and integrity checks using, e.g., hashes and certificate-based authentication, which could be explored further in the future.

6.5.2 Implement an Intrusion Detection System

Implementing an Intrusion Detection System (IDS) may be beneficial in further enhancing the security of the CAN bus. Such systems monitor data transmitted to and from the bus to identify potentially malicious activity. Two common types of IDS

approaches are signature-based and anomaly-based detection, each with different advantages and disadvantages.

A signature-based IDS is effective at identifying known attacks, as it relies on predefined patterns. However, it is unable to detect novel or unknown threats that fall outside its programmed signatures [40]. In contrast, an anomaly-based IDS flags any behavior that deviates from predefined patterns. While this allows it to potentially detect previously unseen attacks, it also makes it more susceptible to false positives. Additionally, anomaly-based systems typically require significant computational resources to analyze and monitor traffic patterns, resources that are often limited in many ECUs. Therefore, it is essential to carefully evaluate all relevant factors and prioritize them based on the specific requirements and constraints of the vehicle system when selecting and implementing an IDS.

In the context of this thesis, incorporating a more dynamic approach to attack detection would be beneficial. Current protection mechanisms are static and predefined, making identifying previously unknown or unforeseen attack vectors challenging. An anomaly-based IDS could be an effective solution to address this limitation, as it offers a more adaptive and proactive alternative than signature-based methods. With respect to system resources, the productivity hub utilized in this project likely possesses enough computational capacity to support the monitoring and analysis of network traffic between the ECUs.

6.5.3 Alternatives to the Current Permission Management

Currently, the system fully trusts company developers to identify when an app requests more permissions than necessary. This is managed through code reviews, where the reviewing developer assesses whether or not the requested permissions are appropriate for the application. The decision-making process regarding which permissions to grant could be explored further in the future. For instance, automating or integrating this permission-granting process directly into the system may be possible.

6.5.4 Validate Configuration File to JSON Schema

At present moment, no checks have been implemented to validate the configuration file against its JSON schema, except for those already integrated into the Integrated Development Environment. Thus, it is technically possible to download an incorrect configuration file onto the productivity hub, which could potentially contain malicious content. An effective way to prevent this would be to implement a way for the system to check if the configuration file follows that schema as it is downloaded onto the productivity hub. If not, the installation cannot be completed. This ensures that incorrect values are not written to the configuration file, and thus reduces the risk of it containing incorrect content.

6.6 Abandoned Ideas

Throughout the project, various ideas were considered for securing the system. However, upon further discussion both internally and with our supervisors, the following ideas were deemed not applicable to the project within its current context and limitations.

6.6.1 Datagram Transport Layer Security

Since the CAN frames are transmitted over a UDP network inside the productivity hub, one idea was to implement Datagram Transport Layer security (DTLS). DTLS is similar to TLS in many ways, with the main difference being the fact that it operates over UDP instead of TCP [41]. Furthermore, it offers secure transport of UDP packages, which UDP in itself does not. However, upon further research and finalizing the project scope, DTLS was decided to not be entirely relevant to the project. Instead, the project focuses on higher system levels, more specifically, the application layer of the OSI model.

6.6.2 Zero Trust

When research on PoLP was conducted, it simultaneously sparked the idea of implementing the closely related concept of Zero Trust into the system. The core idea in a Zero Trust Architecture is to never trust and always verify all traffic (both internal and external), and thus restricting access to resources to the least amount possible [42]. Additionally, the trust level of a program or user should go through continuous checks to ensure no changes have occurred since the last access was granted. However, Zero Trust is not fully applicable to this project because the focus is on a small vehicle subsystem, whereas Zero Trust is more relevant when considering the entire vehicle system.

7

Conclusion

This project explored how to improve the security of an Android Automotive based infotainment system, by implementing a granular access control mechanism for the CAN bus, based on the Principle of Least Privilege (PoLP). This is necessary since the original system used in the project adopts an "all-or-nothing" approach to CAN bus access, which presents a security risk. Malicious or compromised apps could therefore potentially access the entire In-vehicle Network (IVN) if it obtains only two permissions.

Apart from the PoLP, the implemented solution is also supported by resources from two key cybersecurity standards and regulations: EN 18031-1:2024 [27], under the Radio Equipment Directive (RED), and ISO/SAE 21434 [29]. For RED, the project meets the relevant criteria ACM-1, ACM-2, and GEC-6. To systematically identify threat scenarios and ensure that the implementations mitigate the threats, a threat modeling process was conducted using the STRIDE framework and incorporating selected elements of the TARA methodology outlined in ISO/SAE 21434.

Android permissions serve as the foundation of the implemented access control and were used to achieve higher granularity within the system. Investigating the security of the Android permission model was deemed out of scope and it was instead fully trusted and regarded as a solid foundation to build the access control on.

The following protection mechanisms were implemented as part of the access control system. First, permission restrictions were added to limit read and write access to CAN frames and signals for apps. Additional permission boundaries have also been added for interacting with the CAN bus and MachineAPI (the service handling app-to-CAN bus interactions). Apart from these permission restrictions, changes related to the configuration file for each vehicle have been implemented. These measures include rules for restricting the allowed range of CAN signals, allowed CAN frames through bitmasks and filters, preventing overwriting of the rules in the configuration file, and requiring the explicit declaration of package names for allowed apps to ensure that only authorized apps can be installed on the productivity hub.

To test whether or not the implemented protection mechanisms in the proposed solution resulted in significant overhead to the system, benchmarking tests were conducted. This was concluded to be false. Receiving CAN frames from the bus showed an increase of 0.004 milliseconds, equivalent to an increase of approximately 1.59% compared to the original system. Regarding writing CAN frames to the bus,

the proposed solution resulted in a time increase of 0.171 milliseconds, which equals approximately 48%. Since the system is not dependent on any real-time constraints, such an increase in time will not significantly affect the system's performance and reliability and is therefore deemed negligible.

The result is a simple but effective system for managing access control to the CAN bus and IVN.

Bibliography

[1] N. Sugunaraj and P. Ranganathan, "Electronic Control Unit (ECU) Identification for Controller Area Networks (CAN) using Machine Learning," in *2022 IEEE International Conference on Electro Information Technology (eIT)*, IEEE, May 2022, pp. 1–7, ISBN: 978-1-6654-8009-3. DOI: 10.1109/eIT53891.2022.9813928.

[2] S. Jiang, "Vehicle E/E Architecture and Its Adaptation to New Technical Trends," SAE International, Apr. 2019. DOI: 10.4271/2019-01-0862.

[3] Robert Bosch GmbH, *Can specification 2.0*, 1991. [Online]. Available: https://web.archive.org/web/20250118165130/http://esd.cs.ucr.edu/webres/can20.pdf (visited on 05/09/2025).

[4] CAST and Fraunhofer IPMS, "White Paper - CANsec: Security for the Third Generation of the CAN Bus," Tech. Rep., Oct. 2024. [Online]. Available: https://www.cast-inc.com/blog/white-paper-cansec-security-third-generation-can-bus.

[5] Android Developer Guide. "What is Android Automotive." (Feb. 2025), [Online]. Available: https://source.android.com/docs/automotive/start/what_automotive (visited on 03/10/2025).

[6] Android Developer Guide. "System and kernel security." (2025), [Online]. Available: https://source.android.com/docs/security/overview/kernel-security (visited on 05/08/2025).

[7] Android Developer Guide. "Application fundamentals." (Feb. 2025), [Online]. Available: https://developer.android.com/guide/components/fundamentals (visited on 03/10/2025).

[8] Android Developer Guide. "Application Sandbox." (Feb. 2025), [Online]. Available: https://source.android.com/docs/security/app-sandbox (visited on 03/10/2025).

[9] Android Developer Guide. "Platform architecture." (Feb. 2025), [Online]. Available: https://developer.android.com/guide/platform (visited on 05/18/2025).

[10] Android Development Guide, *Processes and threads overview*, Jan. 2024. [Online]. Available: https://developer.android.com/guide/components/processes-and-threads.html.

[11] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2021, pp. 70–86, ISBN: 978-1-7281-8934-5. DOI: 10.1109/SP40001.2021.00070.

[12] Android Developer Guide. “Permissions on Android.” (2025), [Online]. Available: https://developer.android.com/guide/topics/permissions/overview (visited on 01/29/2025).

[13] Android Developer Guide. “<permission>.” (Feb. 2025), [Online]. Available: https://developer.android.com/guide/topics/manifest/permission-element.

[14] Android Developer Guide. “Define a custom app permission.” (2025), [Online]. Available: https://developer.android.com/guide/topics/permissions/defining (visited on 05/20/2025).

[15] J. Saltzer and M. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975, ISSN: 0018-9219. DOI: 10.1109/PROC.1975.9939.

[16] Spyridon Samonas and David Coss, “The cia strikes back: redefining confidentiality, integrity and availability in security,” *Journal of Information System Security*, vol. 10, no. 3, 2014, ISSN: 1551-0808.

[17] K. A. Cochran, “The CIA Triad: Safeguarding Data in the Digital Realm,” in *Cybersecurity Essentials*, Berkeley, CA: Apress, 2024, pp. 17–25. DOI: 10.1007/979-8-8688-0432-8{_}2.

[18] Syrma SGS, *Automotive ECU: Core Component for Connected Cars*, Sep. 2020. [Online]. Available: https://syrmasgs.com/ecu/ (visited on 05/22/2025).

[19] M. Farsi, K. Ratcliff, and M. Barbosa, “An overview of controller area network,” *Computing and Control Engineering*, vol. 10, no. 3, pp. 113–120, Jun. 1999. [Online]. Available: https://digital-library.theiet.org/doi/epdf/10.1049/cce%3A19990304 (visited on 05/20/2025).

[20] Brooke Lampe and Weizhi Meng, “Intrusion Detection in the Automotive Domain: A Comprehensive Review,” *IEEE Communications Surveys & Tutorials*, vol. 25, no. 4, pp. 2356–2426, Aug. 2023. DOI: 10.1109/COMST.2023.3309864.

[21] F. Fakhfakh, M. Tounsi, and M. Mosbah, “Cybersecurity attacks on CAN bus based vehicles: a review and open challenges,” *Library Hi Tech*, vol. 40, no. 5, pp. 1179–1203, Nov. 2022, ISSN: 0737-8831. DOI: 10.1108/LHT-01-2021-0013.

[22] M. Bozdal, M. Samie, and I. Jennions, “A Survey on CAN Bus Protocol: Attacks, Challenges, and Potential Solutions,” in *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*, IEEE, Aug. 2018, pp. 201–205, ISBN: 978-1-5386-4904-6. DOI: 10.1109/iCCECOME.2018.8658720.

[23] S. Adly, A. Moro, S. Hammad, and S. A. Maged, “Prevention of Controller Area Network (CAN) Attacks on Electric Autonomous Vehicles,” *Applied Sciences*, vol. 13, no. 16, p. 9374, Aug. 2023, ISSN: 2076-3417. DOI: 10.3390/app13169374.

[24] J. Postel, “User Datagram Protocol,” Tech. Rep., Aug. 1980. DOI: 10.17487/rfc0768.

[25] L. Eggert, G. Fairhurst, and G. Shepherd, “UDP Usage Guidelines,” Tech. Rep., Mar. 2017. DOI: 10.17487/RFC8085.

[26] R. Braden, D. Borman, and C. Partridge, “Computing the Internet checksum,” Tech. Rep., Sep. 1988. DOI: 10.17487/rfc1071.

[27] European Union, *Common security requirements for radio equipment Part 1: Internet connected radio equipment EN 18031-1:2024*, 2024.

[28] European Commission, Directorate-General for Internal Market, Industry, Entrepreneurship and SMEs, *COMMISSION DELEGATED REGULATION (EU) 2022/30*, Jan. 2022. [Online]. Available: https://eur-lex.europa.eu/eli/reg_del/2022/30/oj/eng (visited on 02/14/2025).

[29] ISO/TC 22/SC 32, *ISO/SAE 21434:2021 Road vehicles Cybersecurity engineering*, 2021. [Online]. Available: https://www.iso.org/standard/70918.html.

[30] A. Lautenbach, M. Almgren, and T. Olovsson, "Proposing HEAVENS 2.0 an automotive risk assessment model," in *Computer Science in Cars Symposium*, New York, NY, USA: ACM, Nov. 2021, pp. 1–12, ISBN: 9781450391399. DOI: 10.1145/3488904.3493378.

[31] Larry Conklin. "Threat Modeling Process." (2025), [Online]. Available: https://owasp.org/www-community/Threat_Modeling_Process (visited on 03/04/2025).

[32] OWASP Cheat Sheets Series Team. "Threat Modeling Cheat Sheet." (2025), [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Threat_Modeling_Cheat_Sheet (visited on 03/04/2025).

[33] Braiterman Zoe *et al.* "Threat Modeling Manifesto." (2020), [Online]. Available: https://www.threatmodelingmanifesto.org/. (visited on 03/04/2025).

[34] Volvo Group, "Threat Analysis and Risk Assesment Instruction," 2022, Proprietary.

[35] G. S. Tuncay, "Android Permissions: Evolution, Attacks, and Best Practices," *IEEE Security & Privacy*, vol. 22, no. 6, pp. 40–49, Nov. 2024, ISSN: 1540-7993. DOI: 10.1109/MSEC.2024.3461629.

[36] G. S. Tuncay, S. Demetriou, K. Ganju, and C. A. Gunter, "Resolving the Predicament of Android Custom Permissions," in *Proceedings 2018 Network and Distributed System Security Symposium*, Reston, VA: Internet Society, 2018, ISBN: 1-891562-49-5. DOI: 10.14722/ndss.2018.23210.

[37] R. Li, W. Diao, Z. Li, S. Yang, S. Li, and S. Guo, "Android Custom Permissions Demystified: A Comprehensive Security Evaluation," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4465–4484, Nov. 2022, ISSN: 0098-5589. DOI: 10.1109/TSE.2021.3119980.

[38] Microchip Technology Inc, *1.1 CAN-TP Overview*. [Online]. Available: https://onlinedocs.microchip.com/oxy/GUID-9C356E20-C5BD-430F-8C0B-CCA1B85ECC7C-en-US-3/GUID-F040354D-0842-4EFC-99F2-F1B8A649D106.html (visited on 05/19/2025).

[39] Android Developer Guide, *Services overview*, Jan. 2025. [Online]. Available: https://developer.android.com/develop/background-work/services (visited on 06/11/2025).

[40] S.-F. Lokman, A. T. Othman, and M.-H. Abu-Bakar, "Intrusion detection system for automotive Controller Area Network (CAN) bus system: a review," *EURASIP Journal on Wireless Communications and Networking*, vol. 2019, no. 1, p. 184, Dec. 2019, ISSN: 1687-1499. DOI: 10.1186/s13638-019-1484-3.

[41] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," Tech. Rep., Jan. 2012. DOI: 10.17487/rfc6347.

[42] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., Aug. 2020. DOI: 10.6028/NIST.SP.800-207.