CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Design and implementation of an intrusion detection system (IDS) for in-vehicle networks

Master's thesis in Computer Systems and Networks

NORÄS SALMAN
MARCO BRESCH

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

Master's thesis 2017

Design and implementation of an intrusion detection system (IDS) for in-vehicle networks

NORÄS SALMAN
MARCO BRESCH

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

Design and implementation of an intrusion detection system (IDS) for in-vehicle networks
NORÄS SALMAN
MARCO BRESCH

Supervisor: Tomas Olovsson, Computer Science and Engineering.
Advisor: Nasser Nowdehi, Volvo Car Corporation
Examiner: Erland Jonsson, Computer Science and Engineering.

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

Typeset in LaTeX
Gothenburg, Sweden 2017

Design and implementation of an intrusion detection system (IDS) for in-vehicle networks

NORÄS SALMAN & MARCO BRESCH

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Controller Area Network (CAN) was specified with no regards to security mechanisms at all. This fact in combination with the widespread adoption of the CAN standard for connecting more than a hundred Electrical Control Units (ECUs), which control almost every aspect of modern cars, makes the CAN bus a valuable target for adversaries. As vehicles are safety-critical systems and the physical integrity of the driver has the highest priority, it is necessary to invent suitable countermeasures to limit CAN's security risks. As a matter of fact, the close resemblances of in-vehicle networks to traditional computer networks, enables the use of conventional countermeasures, e.g. Intrusion Detection Systems (IDS).

We propose a software-based light-weight IDS relying on properties extracted from the signal database of a CAN domain. Further, we suggest two anomaly-based algorithms based on message cycle time analysis and plausibility analysis of messages (e.g. speed messages). We evaluate our IDS on a simulated setup, as well as a real in-vehicle network, by performing attacks on different parts of the network. Our evaluation shows that the proposed IDS successfully detects malicious events such as injection of malformed CAN frames, unauthorized CAN frames, speedometer plausibility detection and Denial of Service (DoS) attacks.

Based on our experience of implementing an in-vehicle IDS, we discuss potential challenges and constraints that engineers might face during the process of implementing an IDS system for in-vehicle networks. We believe that the results of this work can contribute to more advanced research in the field of intrusion detection systems for in-vehicle networks and thereby add to a safer driving experience.

Keywords: Controller area network, in-vehicle network, embedded security, intrusion detection system, engineering, project, thesis.

Acknowledgements

We want to extend our thanks to our supervisors: Nasser Nowdehi at Volvo Car Corporation and Tomas Olovsson at Chalmers University of Technology. Last but not least we want to thank Erland Jonsson, at Chalmers University of Technology, for being our examiner.

Noräs Salman & Marco Bresch, Gothenburg, June 2017

Contents

List of Figures

List of Tables

1

Introduction

In the last decades, technology has affected many different aspects of our lives with the aim to bring more comfort and ease of performing everyday tasks. Automotive vehicles have been a fundamental tool for our everyday life for the last century. The car evolution went through several dramatic changes. For instance, cars went from having steam powered engines to gasoline engines and now it is more common to see electrical powered engines. Not long ago, cars were completely mechanical systems that consisted of only mechanical components and some electrical units connected by wires. One of the most important changes was the introduction of Electrical Control Units (ECU), which are small embedded systems.

The in-vehicle network of a modern car consists of more than a hundred ECUs that control almost every functionality of the car, including safety critical functions, such as acceleration, braking and steering. These ECUs communicate over multiple sub-networks and gateways, using different communication technologies such as CAN, LIN, MOST, and FlexRay.

The Controller Area Network (CAN) which was proposed by Bosch in 1983 [11], enables the ECUs to communicate over a serial-bus. The CAN specification [2], includes different message formats, error handling mechanisms and how to do bus arbitration. However, the CAN bus was not designed with security in mind and cannot guarantee any security properties, such as confidentiality, authenticity, availability, integrity and non-repudiation [25]. For example, the CAN bus is prone to spoofing attacks because messages on the bus are broadcasted and ECUs have no mechanisms to verify the authenticity of the messages. Other attack vectors exist, for instance, it is possible for an attacker to perform a Denial of Service attack (DoS) on a CAN bus by misusing the error detection mechanism of CAN.

Cars are safety-critical systems, where malfunction of in-vehicle ECUs can lead to serious injuries or death of passengers or near-by pedestrians. Therefore, it is of highest importance to develop mechanisms to detect such attacks and protect the in-vehicle network against them. In traditional IT security, Intrusion Detection Systems (IDS) are well-known measures for securing communications. An IDS is a software for monitoring the network or the host for malicious activities, which are defined by rules or patterns. However, when working with ECUs the engineers have to deal with several resource constraints, such as limited processing power and limited memory, as well as compatibility and backward compatibility requirements.

In this thesis, we explore the state of the art approach for IDS development

for attacks against the Controller Area Network. Furthermore, we discuss the engineering challenges to build such systems and deploy them as a feasible security solutions.

1.1 Scope

Modern cars have several built-in communication technologies used for entertainment, diagnosis, customer service and internal control. All these technologies can be used as an attack vector by malicious individuals. However, this thesis is only concerned with the security of the internal communication and the CAN bus in particular.

Multiple types of CAN frames exist, such as flexible data-rate CAN (CAN-FD) and ISO 11898-4 time triggered CAN (TTCAN), in addition to the standard CAN frames. This thesis focuses on the investigation of standard CAN frames (ISO 11898-1). Figure 1.1 gives an overview over the different branches the CAN standard is divided into.

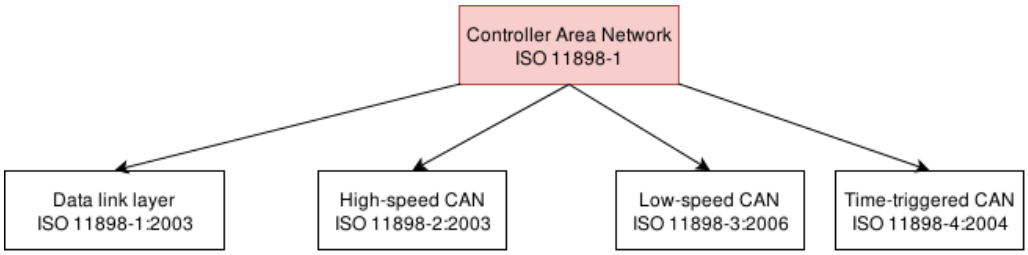

Figure 1.1: Control Area Network standards

1.2 Objectives

This thesis aims to identify the constraints and requirements of developing IDSs for modern vehicles by implementing a prototype of a lightweight IDS on an experimental in-vehicle network. The findings in this thesis can contribute to a deeper understanding of the practical challenges of implementing IDS for in-vehicle networks, which can result in a further step towards a more secure driving experience. Hence, the specific questions that drive this research are:

- Which aspects of the architecture must be considered in the design of an IDS?
- How should monitoring sensors be distributed over different domains?
- What aspects of the communication must be inspected for attack detection?
- What type of attacks can be locally detected?

The practical goal of this project is to build a lightweight prototype IDS that is able to successfully detect simulated attacks.

1.3 Limitations

It is beyond the scope of this thesis to investigate the process of how an attacker can get access to the network. We assume that an adversary already found an entry point to the CAN bus and is using a compromised ECU to conduct the attacks. This assumption allows the attacker to alter the ECU behaviour partially or completely. Additionally, the main focus lies in implementing the core functionality of an IDS, namely detection of attacks, the work is not concerned with any prevention mechanisms of attacks.

1.4 Methodology

Our work is divided into several steps. The first step consists of a literature review, which is necessary to understand the architecture of the in-vehicle network, in particular CAN and identifying its limitations and the state of the art of designing an in-vehicle IDS. Further, we investigate the signal database, which defines the properties of the CAN network, the ECUs connected to the bus and the transmitted CAN messages. This helps to identify the types of messages that are important to consider when creating the rules for the IDS. The following step is to analyze CAN frames collected from the communication inside an experimental in-vehicle network. We then gather further information about the structure and behaviour of CAN frames, identify patterns and extract signatures that can be used for the implementation of the IDS.

Based on the gained knowledge, we implement a lightweight IDS for the CAN bus. We use a software called CANoe with a special hardware component that helps us in simulating a CAN network. A major advantage of CANoe is that it provides a full framework for developing, simulating and testing in-vehicle networks. Using CANoe, we are able to simulate different attacks to evaluate the effectiveness of our IDS.

The detection rate and the effectiveness of an IDS are evaluated by measuring the occurrences of true-positive, false-positive, true-negative and false-negative results. The same evaluation criteria is used when presenting the results and testing our prototype IDS.

1.5 Structure of the report

This report has been organized in the following way. The first Section gives a short introduction of the subject together with the methodology that we have used. Section 2, begins by laying out the theoretical background for the research, and looks at different vehicular communication types and their architecture. Next in Section 3, we give a comprehensive review of the CAN standard, its protocols, followed by a list of security considerations in Section 4. In Section 5, we give a brief introduction to

IDS systems, their types and architecture and in Section 6, we summarize previous in-vehicle network IDS research.

The remaining part of the report proceeds as follows: in Section 7, we include a detailed overview of the experimental setup we have used. In Section 8, we continue with describing our implementation of the lightweight IDS for in-vehicle networks and discuss the results in Section 9.

In Section 10, we discuss the implication of our findings in regards to the constraints of the in-vehicle network and we identify areas that require further research. Section 11, concludes the entire thesis connecting various theoretical and practical strands in order to evaluate our discoveries and results.

2

Background

2.1 Vehicular communication technologies

Different components inside the vehicle require cooperation from other devices and sensors in order to perform their assigned tasks. This requires a one-directional or sometimes multi-directional communication line between these devices. All the communications between the internal components of the vehicle are referred to as internal vehicular communication. Other communication technologies exist, that provide a communication interface for outside devices to perform tasks such as diagnostics or firmware updates. Moreover, functionalities that give the passengers the ability to stay connected to the internet are getting more popular. These kind of communications and all communications that include an outside party are referred to as external communication.

2.2 External communications

Cars are getting more computerized and more communication technologies are used to remotely control several features of the car, and connectivity in modern cars has become a necessity.

Manufacturers are trying to give the consumer more ways to remotely control several aspects of the car [23] using more than the traditional radio-controlled door unlocking functionality. For example, WiFi (IEEE 802.11) and Cellular communication such as GSM, 3G, 4G are becoming a more standard option. These communication technologies are also used to control some aspects of the vehicle like turning on air conditioning and even starting the engine. GPS for navigation, and Bluetooth for hands free usage of smart phones, have been used in the past decade. All these technologies and more [14] are becoming available as stock options for the consumers.

Additionally, from the manufacturers point-of-view, diagnostics messages are required for pushing remote Firmware update Over The Air (FOTA) [22] to make sure that the customer gets the best aftermarket experience. Manufacturer may also request to receive periodical report about the vehicles to provide remote support. These communications are usually done through VPNs provided by the manufacture or third-party services.

New vehicle communication technologies such as Vehicle-to-Vehicle (V2V), Vehicle-to-Infrastructures (V2I) and Vehicle Ad-hoc Networks (VANETs) [29] are said to be auspicious, and the usage of such technologies promises a safer driving experience.

All of the previous concepts show how much the technology is affecting the design and driving experience of modern cars. Figure 2.1 presents a breakdown of the several communication technologies that are influencing the car.

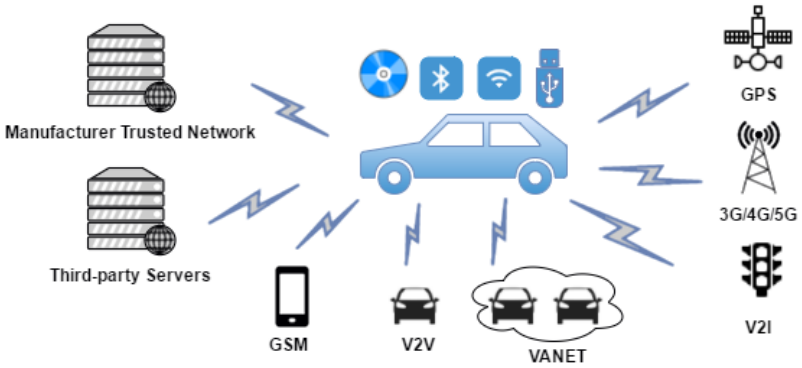

Figure 2.1: External vehicle communications

2.3 Internal communications

Almost all functions in the modern car are controlled by one or more Electronic Control Units (ECU). An ECU is a small size embedded computer system that has real time computing, time constraints, and low power consumption. The ECU's main responsibility is to collaborate to share sensor information using messages. The ECUs can be categorized [18] into five different groups:

- Power-train
- Comfort-train
- Safety
- Infotainment
- Telematics

The power train is responsible for handling car control and navigation functionalities such as brakes, gearshifts and engine acceleration. The safety group includes the control of passenger safety functionalities, for instance, airbags, tire pressure and collision avoidance. The comfort train covers components such as thermal control, windows control, parking assistance. Infotainment includes multimedia, audio and video streaming, traffic and weather information. Finally, telematics covers the external network applications and mobile communication such as WiFi, internet and mobile application controlled services.

An in-vehicle network consists of several buses and ECU's that form a network. The ECUs in the car are connected through multiple communication technologies

such as CAN, LIN, FlexRay, MOST and Ethernet. They transfer and read messages transmitted within multiple in-vehicle local area network (LAN) domains, in order to coordinate and collaborate on controlling a range of operations. Special ECUs known as gateways, forward the messages from one domain to another in case the sender and receiver do not share the same LAN. Figure 2.2 shows how an in-vehicle network can be constructed.

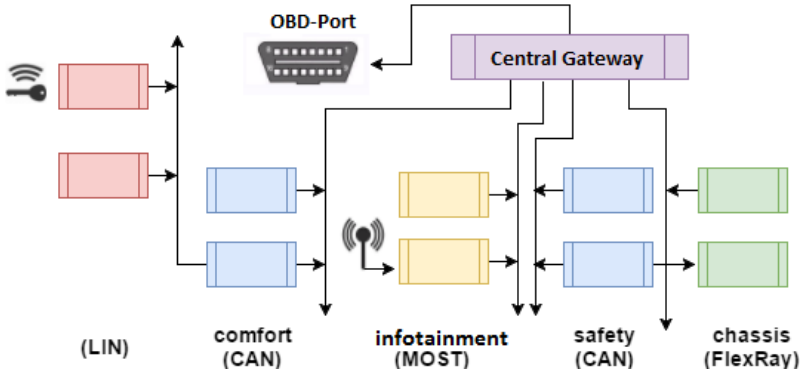

Figure 2.2: Internal communications

The most common bus technology in modern cars is the Control Area Network (CAN), which provides an efficient, fast, reliable and economical link between the ECUs. It allows the connected ECUs to broadcast messages bi-directionally over the bus using only a single channel.

Local Interconnect Network (LIN) is another serial bus intercommunication technology that works like CAN. It has a lower cost, and it is often used for applications that are not relying on strict real-time deadlines to function correctly.

FlexRay is a superior protocol to the CAN that provides high speed communication for time sensitive functionalities including backbone systems, drive-by-wire and brake-by-wire. It was designed with a much faster and more reliable communication than CAN. Unlike CAN, FlexRay offers two-channel communication and multiple topologies such as star, ring and more. In terms of cost, FlexRay is more expensive.

Media Oriented Systems Transport (MOST) is another high speed bus communication standard for multimedia networks in vehicles that use a ring topology. It is used to transmit video and audio data, and has a high bandwidth.

Audio Video Bridging (Ethernet AVB) and its superior Time-Sensitive Networking (Ethernet TSN IEEE 802.1) are getting more used in modern cars' in-vehicle multimedia fields to transfer audio and video data. It provides a high speed communication with synchronization, and resolves problems such as buffering, lag and jitter.

2.4 Accessing the network

The On-Board Diagnostics port (OBD) provides uncomplicated access to the in-vehicle network for troubleshooting problems by performing sensor readings, firmware modification detections and error codes readings.

Different countries have different legislations and standards for this port. For instance, in the United States, the OBD port should be able to perform readings from specific domains of the CAN bus, such as the power-train.

Its successor, the Unified Diagnostic Service (UDS) standard gives more capabilities and more access to the CAN network, such as flashing ECU firmware, read and write ECU memory locations and override ECU I/O [9]. However, being able to perform such access usually requires going through an authentication-like scheme, e.g. challenge-response.

3

Controller Area Networks

3.1 Background

The controller area network was created in the beginning of 1980s by Robert Bosch Inc, an automotive supplier from Germany, in an effort to reduce the convoluted wiring inside the vehicle and substitute it with a two-wire bus [19]. It is standardized as ISO 11898 and ISO 11898-2 (which includes among others, a faster transmission rate ISO 211898-2 is not part of this thesis).

The standard was rapidly accepted by a large number of automotive manufacturers and it has been adapted to other areas as well, e.g the European Organization for Nuclear Research (CERN) relies on CAN for acquiring and controlling parameters in their Proton Synchrotron Booster [1].

The original specification paper from BOSCH [2] points out the following properties of CAN, which led to the widespread adoption of the standard:

- Prioritization of messages.
- Guarantee of latency times.
- Configuration flexibility.
- Multicast reception with time synchronization.
- System wide data consistency.
- Multimaster.
- Error detection and signalling.
- Automatic re-transmission of corrupted messages as soon as the bus is idle again and distinction between temporary errors and permanent failures of nodes and autonomous switching off of defect nodes.

3.2 CAN Architecture OSI

This section explains the architecture and different components of CAN and CAN nodes. Furthermore, it outlines how CAN is classified in the ISO/OSI reference model for network protocols.

The ISO/OSI model consists of seven different layers. CAN mainly operates in two of them. The physical layer is used for for signaling on the bit level, e.g. bit time synchronization or bit encoding and decoding.

The main features of CAN are located in the data link layer, which is further segmented into Logical Link Control layer (LLC) and Medium Access Control layer (MAC). The capacity of the LLC include, among other things, overload notifications and assistance for data transfer. MAC contains the central features that make CAN reliable, these features include: methods for confining faults and signaling of different errors; regulating the frames on the bus; and most importantly carrying out arbitration on the frames in case two nodes want to send frames at the same time. These concepts are explained in greater detail in the following sections. Figure 3.1, displays the distinct layers of the OSI model and highlights the operational layer for the CAN bus.

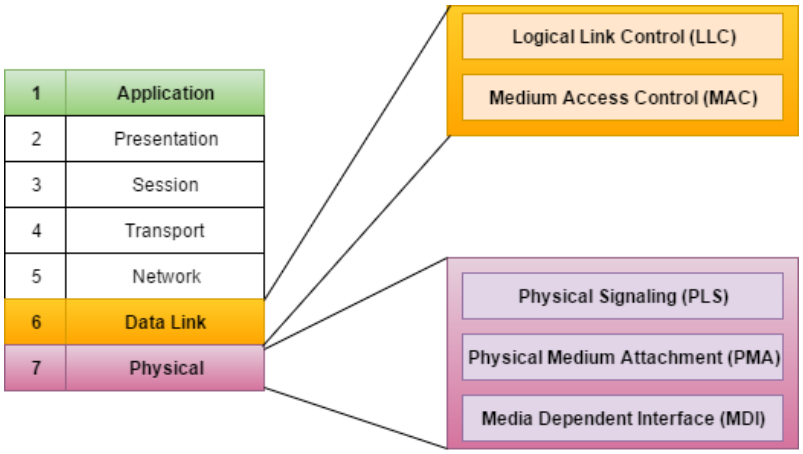

Figure 3.1: The operational layers of CAN in the OSI model

3.3 Physical layer

The physical layer of CAN is concerned with defining how to send signals and the correct encoding of bits. In order to accomplish this task, several concepts are specified in the original CAN specification. This section explains these concepts.

3.3.1 Bit representation

CAN uses Non-Return-to-Zero (NRZ) bit encoding [7]. This encoding is rather trivial and describes that after a value of 1 is detected in the bit stream, the following bit does not have to be changed to a 0 immediately and the voltage can be maintained for a longer period of time.

To summarize, NRZ only encodes negative and positive signals and ignores zero signals. Figure 3.2, outlines how NRZ works in practice. It highlights the fact that a signal can remain in the same state for a longer period of time, which is a problem, because it can lead to a desynchronization of the communication.

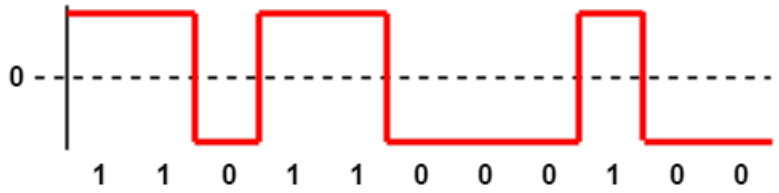

Figure 3.2: Non-Return-to-Zero encoding example

In order to ensure the accurate transfer of data, all participating nodes are synchronized and adjusted to the same clock rate. The synchronization of the clocks themselves is not achieved by a sovereign clock signal, but rather the CAN frames on the bus are utilized. All nodes connected to the CAN bus are listening for frames and synchronize their internal clocks to the clock of the transmitting node.

The point of reference for this *hard synchronization* is the *Start of Frame* bit of the CAN frame. Following differences in the polarities in the CAN frame will be used for the *soft synchronization*.

3.3.2 Bit stuffing

In order to thwart the aforementioned desynchronization of the communication, the CAN uses a practice called *Bit stuffing*. The central idea behind bit stuffing is to inject a bit of reversed polarity, after five bits of equal polarity have been determined in the communication, and thus enforce the correct synchronization. Figure 3.3 demonstrates how a bit is stuffed into the communication.

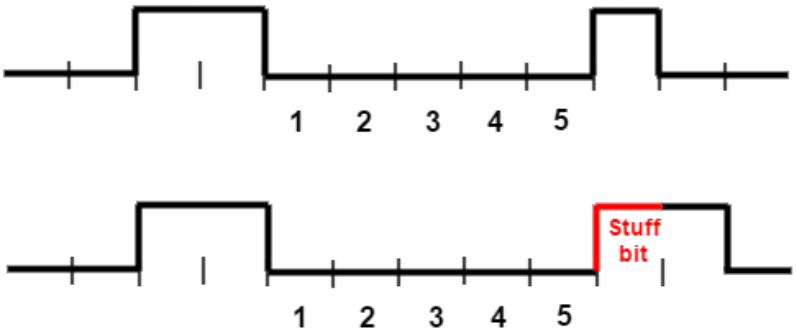

Figure 3.3: Bit stuffing example

In the context of bit stuffing and the individual fields of a CAN frame, it is important to mention that not all fields become stuffed in this way, for instance, the *CRC delimiter*, the *ACK* and the *EOF* field are all of a fixed size and are not allowed to be stuffed.

3.3.3 Dominant and recessive logical bus states

For the transmission of signals, CAN uses two distinct wires: the CAN High (CANH) and CAN Low (CANL) wire. When no signals are transmitted on the bus, these

wires are said to be in an *idle state* and the voltage amounts to 2.5V. After the first bit has been transmitted the CANH wire increases its voltage to 3.75V and the CANL wire decreases its voltage to 1.25V. Figure 3.4 highlights the transition from the *idle state* to the increase or decrease of the voltage on the wires. Note that the difference between these two states is exactly 2.5V.

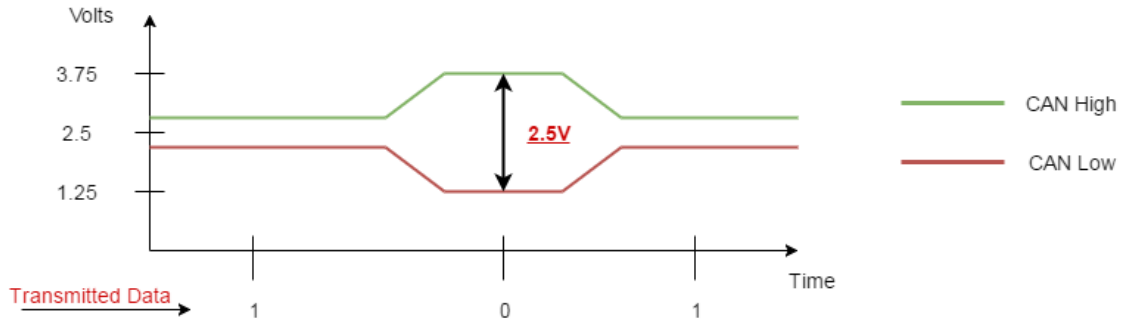

Figure 3.4: Dominant and recessive logical bus states

Following the differences in the voltage on the wires, the bus can be in two distinct logical states: recessive or dominant.

- The bus is said to be in a **recessive state** when transmitting a "1" if *"the differential voltage on CANH and CANL is less than the minimum threshold (less than 0.5V receiver input or less than 1.5V transmitter output)"* [20].
- The bus will enter the **dominant state**, i.e. transmitting a logical "0" if "*the differential voltage on CANH and CANL is greater than the minimum threshold*" [20].

3.4 Data link layer

The data link layer is further divided into two sub-layers. The Medium Access Control (MAC) and the Logical Link Control (LLC). In this section, we explain these sub-layers in greater detail and highlight important concepts and operations. We also give an overview on how we utilize the signal database files.

3.4.1 Medium access control

The MAC sub-layer has a set of the most important operations that handle several aspects of the transmission of CAN frames.

3.4.1.1 CAN frames

The communication inside CAN is comprised of four main frames or message types: the data frame, the remote frame, the error frame and the overload frame, all of

which are further divided into several fields [5].

The majority of the communication proceeds through data frames which constitute of the data field, the arbitration field, Cyclic Redundancy Check (CRC) field and acknowledge field. The arbitration field further contains an 11-bit identifier field and a Remote Transmission Request (RTR) field, which is used in the arbitration and must be set to a dominant bit in case of a data frame[2]. The data field then follows, which can be 8 byte in total, followed by the cyclic redundancy check field. The structure of the data frame and its distinct fields is illustrated in Figure 3.5.

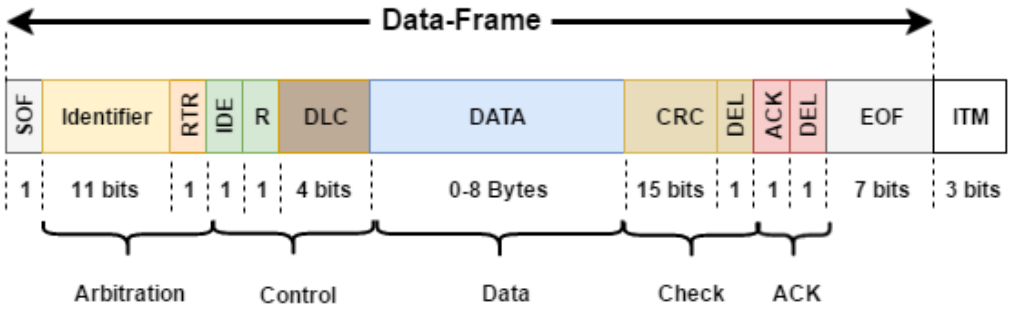

Figure 3.5: CAN data frame structure

The next important frame is the remote frame with the purpose of acquiring data from different nodes. It contains the same fields as the data frame, except that there is no possibility to attach data to this frame and the RTR bit is marked as a recessive bit. An overload frame will only be sent out if there is too much traffic originating from one node. The main function of this frame is to add an extra delay between messages.

If an erroneous frame is detected during the communication between nodes, a special error frame will be sent out. This frame has a distinct layout from normal frames and has the purpose to notify all nodes connected to the bus of the corrupted frame. The emitter of this frame is able to resend the message. One issue that could emerge with error frames is, the flooding of the network with error messages. To avoid the flooding, each CAN node implements an error counter.

3.4.1.2 Carrier sense multiple access with collision avoidance

A significant feature of CAN is the MAC protocol Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). It's responsibility lies in managing and avoiding communication collisions, in case several nodes are trying to access and send frames on the bus at the same time [8].

As some frames are of safety critical nature, e.g. frames which control the vehicle's speed, it is not acceptable to simply drop the frames or delay them for an extended amount of time. During the communication on the bus, only one node at a time is able to send frames, all other nodes have to wait and monitor the bus [2].

If the bus is busy and two CAN nodes want to send frames at the same time they have to wait until the bus is free again, this is achieved by observing the bus

for the *intermission frame*. The node that wants to send a high priority frame is capable of doing so. Low-priority frames can be sent after observing the intermission field again.

This arbitration of low and high priority frames is a crucial feature of CAN. Figure 3.6 exhibits how a multiple access is sensed and how CSMA/CA resolves this conflict.

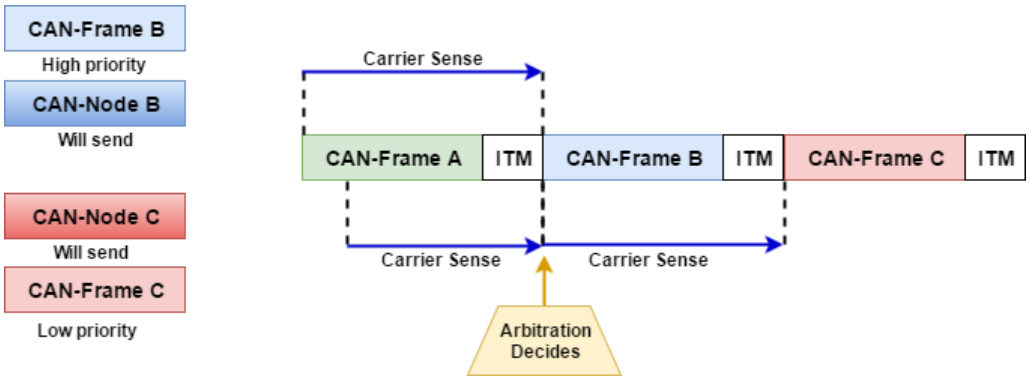

Figure 3.6: CSMA/CA example

3.4.1.3 Arbitration

Since all nodes share the same bus, there must be a correct procedure in case two nodes want to send frames at the same time.

CAN differentiates between different priority messages, the frame with the highest priority has the smallest ID and vice versa. Figure 3.7 illustrates the arbitration process. can be seen that the frames are identical up to the first three bits. Following, it is observable that node A is sending out a dominant bit ('**0**') and node B is sending a recessive bit ('**1**') and therefore loses the arbitration. Node A is allowed to continue sending frames while node B has to wait.

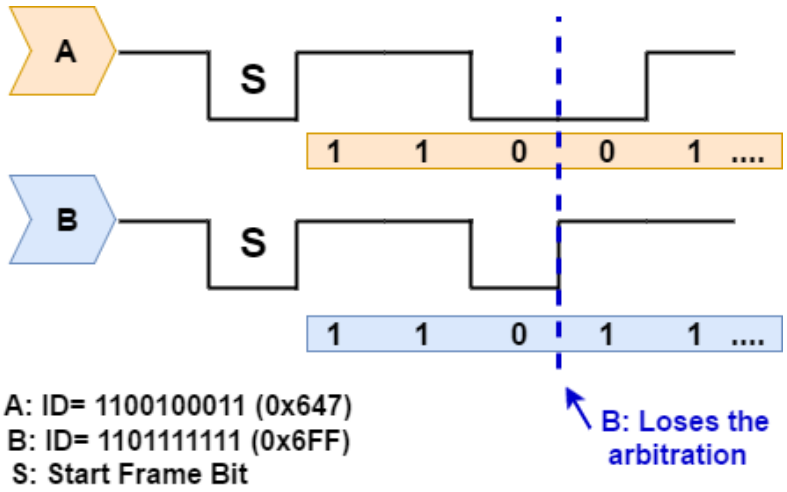

Figure 3.7: Arbitration process

3.4.1.4 Error handling and signaling

As stated before, if a frame does not follow the specification or is malformed, an error frame is generated to notify the nodes on the bus. This is happening if the frame is not passing certain error checks. There are five different error types distributed over the message and bit level.

A **bit error** is identified, if the node that is emitting a bit on the bus, discovers that the bit has changed since sending it out. The only exception to this rule is, when the arbitration is executed on the bus and the node is sending out a recessive bit but is noticing a dominant bit. In order for this not being detected as a bit error, the emitting node has to send out a passive error flag before participating in the arbitration.

Before a node sends out a frame onto the bus, it calculates the CRC and saves it in the CRC field of the frame. Upon receiving the frame, the receiving node also calculates the CRC. If the calculated CRC differs from the CRC value found in the frame, a **CRC error** is detected.

If it is detected that after five consecutives bits, the bit has not changed to the opposite polarity, e.g a dominant bit after five recessive bits, a **stuff error** is detected on the bus.

Some fields and frames have a fixed-format[2]. These fields are: ACK, CRC, EOF, intermission and overload frames. If the receiving node detects illegal bits in these fields, a **form error** is encountered.

The last error type is an **acknowledgement error**, which is detected if the bit inside the ACK field is not a dominant one. If a node detects one of these errors, it informs other nodes on the bus by sending out an ERROR flag. In the case of a CRC error, the flag is sent out after the ACK field. For all other error types, the flag will immediately be sent with the next bit following the erroneous bit.

3.4.1.5 Fault confinement

If an error is detected on the bus, it is crucial to handle the error in a gracious way in order to avoid disturbing the normal communication and risk of further complications. A participating node on the CAN bus can be in one of three distinct failure states [2]: *error passive*, *error active* and a node can be completely taken off the bus by being in a "*bus off*" state.

Whenever a node is in a "*error passive*" state, it sends out a passive error flag after determining an error, and has to wait a certain time before the node can participate in the communication again.

A node in the "*error active*" state sends out an active error flag, but is still capable of engaging in the communication on the bus.

Finally, a node is in a "*bus off*" state. In this state, the node is completely restricted in participating in any kind of communication on the bus.

The decision when a node should enter a certain error state, is handled by two

counters that are implemented in every node on the bus: the transmit error counter and the receive error counter. Based on certain rules, these counters are incremented or decremented respectively. For instance, if the transmit error counter of a node is >= 256, it switches into the bus off state. The specification of CAN requires that a node which has an error counter greater than 96 is considered as highly disturbed [2].

3.4.2 Logical link control

The Logical Link Control-layer (LLC) is another sub-layer of the datalink-layer. In the context of CAN, the main responsibilities of this layer are described in the official specification by BOSCH [2]:

- Providing services for data transfer and for remote data request
- Deciding which messages received by the LLC sub-layer are actually to be accepted
- Providing means for recovery management and overload notifications

LLC and MAC are closely related to one another, as the MAC handles messages that were sent from the LLC. Furthermore the MAC approves of messages that are going to be transmitted to the LLC layer. In this regard, the LLC is engaged in "*Message filtering, Overload Notification and Recovery management*" [2].

3.5 Signal databases

In order to correctly identify and interpret the information inside the data field of a CAN frame, database files for the distinct signals exist. These databases can be distributed in a variety of file extensions, such as .xml or .dbc (which is a special database file used by Vector and is the focus in this thesis).

Each signal that is transmitted on the CAN bus is well defined and has specific conversion methods which are described inside the database. The information which is included in the database, is the following:

- Channel name
- Starting point and size of each channel inside the data field.
- Byte order
- Data type (e.g. signed or unsigned)
- Scaling and unit strings
- Data ranges
- Default value
- Comments

Based on this information, it is possible to convert bytes into interpretable and recognizable values. Figure 3.8 demonstrates an example conversion of raw bytes into human interpretable values. In this example, the *temp_value* of 1115, inside the data field of a CAN frame, is converted into the temperature value of 75.1 degrees.

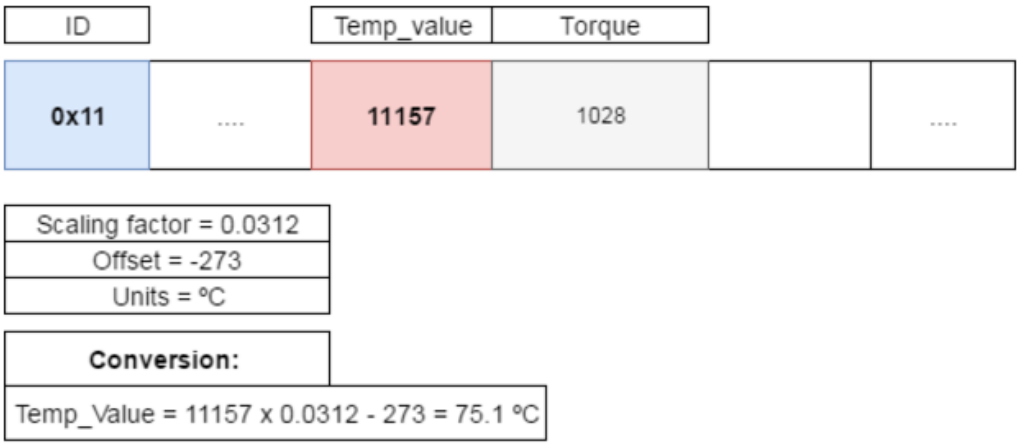

Figure 3.8: A conversion example using the signal database

These databases are usually restricted to specific CAN subnetworks and are vendor-specific. With the help of a CAN database it is possible to describe and reconstruct the entire communication and gain sensitive information about the inner functionality of the vehicle.

4

Attacking Controller Area Networks

The following section is concerned with the examination of the security mechanisms a CAN bus has to offer. As mentioned in the introduction, we assume a person with malicious intent has read and write access to the CAN bus and can try out different techniques to fully take over the network. In order to highlight the attack surface and domains an attacker can compromise, we adapt to the well-known CIA-triad, i.e. **Confidentiality**, **Integrity** and **Availability**. If an attacker manages to breach only one of these security properties, the system is said to be compromised and it can have unexpected and severe consequences for the involved user and technical components.

4.1 CAN security considerations

While looking at the original CAN specification published by BOSCH, it is surprising that there are no considerations on how to secure the network at all [2] . In order to give a basic overview of the problems that arise due to the lack of security mechanisms, we enumerate some of the potential attack vectors against the CAN bus with the aid of the aforementioned CIA triad.

Confidentiality in CAN is not achieved since all nodes broadcast their messages on the bus, where the receiver node picks the message up based on the message IDs it is configured to receive. This enables attacks such as message sniffing. Figure 4.1 shows how the sniffing could occur on the CAN bus.

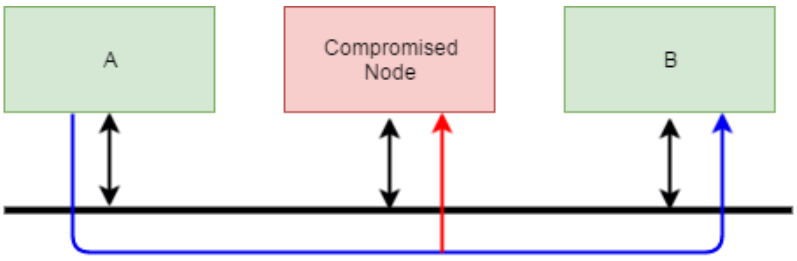

Figure 4.1: Sniffing CAN frames

Authenticity and **non-repudiation** are contravened due to lack of information about the sender in the message e.g signatures. This allows an attacker to send arbitrary CAN frames to any node in the network. One type of frame injection attack is masquerade attacks, where the attacker sends normal correct messages in order to trigger certain actions. Figure 4.2 exhibits the injection of arbitrary packets on the CAN bus from a compromised node.

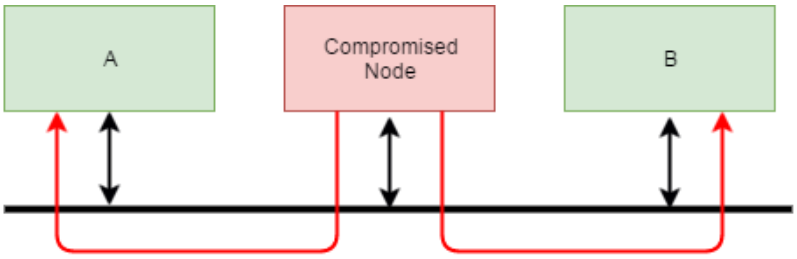

Figure 4.2: Injecting arbitrary packets on the CAN bus

If an attacker manages to take over a gateway that forwards messages from one domain to another, the attacker can add, remove or change any type of data that the relayed message carries. Even though CAN message has CRC checksums for error detection purposes, it does not prevent a malicious node or an attacker to tamper with transferred messages then adding valid checksums. Knowing this fact, we can say that there is a violation of **integrity** in CAN messages. Figure 4.3 demonstrates how an adversary can tamper the CAN frames in order to compromise the security of the communication.

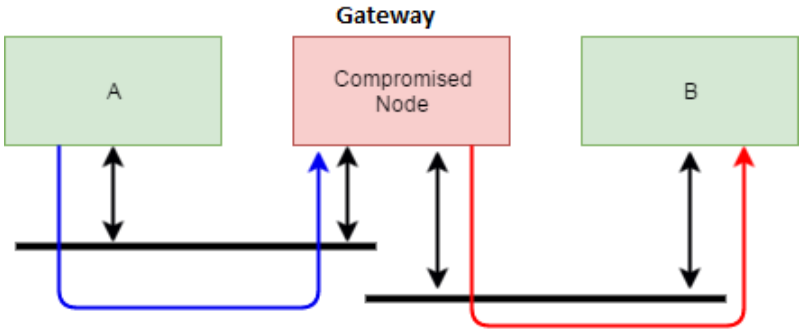

Figure 4.3: Tampering of legitimate CAN frames

Since an attacker can send any kind of data on the bus, sending high-priority messages or messages with error flags can cause nodes to stop responding causing a Denial of Service (DoS), thus affecting the **availability** of the system. Another scenario is where the attacker has control over a gateway node, this gives the ability to drop all or certain transferred messages between multiple sub-networks. Figure 4.4 highlights how a compromised node can drop packets from the on-going communication, where node A sends a message to node B, but it gets dropped by the compromised gateway before being relayed.

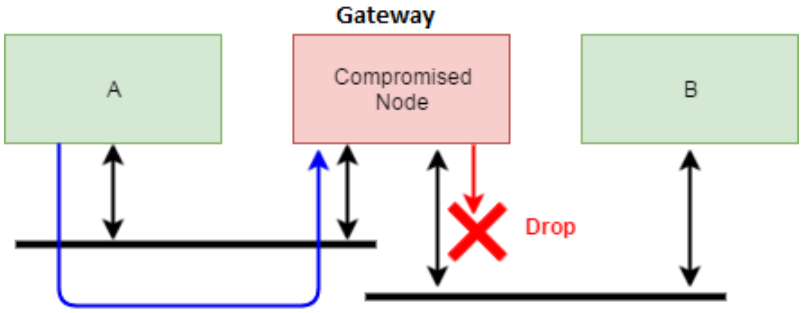

Figure 4.4: Dropping of CAN frames

4.2 Known security measures for in-vehicle networks

4.2.1 Message encryption and signing

The lack of confidentiality of transferred messages can be solved by applying "end-to-end" encryption. This solution, however, needs to be carefully designed. The nodes have limited processing power and cryptographic solutions need to be as lightweight as possible in order to prevent an extensive latency, which could impair the communication on the bus.

Adding cryptographic signatures to the transferred messages is one solution to ensure integrity while appending a layer of encryption ensures the confidentiality of the messages.

4.2.2 Node authentication

Ensuring the authenticity of a node can be achieved by using a public key encryption scheme, using a certificate [10], however, the authenticity check is not limited to this solution. Authenticity can also be ensured by observing a potential change in the message frequency of the transmitter ECU.

Node authentication, encryption and signing is a measure that can lower the risk of a possible attack on the bus. Preliminary research has been done and explained in [4], which states, that if a security measure has been taken and implemented in an ECU and the attacker is capable to compromise that ECU. The attacker then has access to the data stored in the memory, including all the data that is related to these security measures, such as encryption keys. Moreover, the attacker has the ability to disable such measures by flashing the firmware as demonstrated in [15].

4.2.3 Firewalls and policy enforcement

Monitoring the frames and filtering them before broadcasting ensures the correctness of messages with respect to the message content, ID and timing. This can be done either by installing hardware components such as firewalls or packet filters, or by embedding these functionalites programmatically by enforcing a policy; which defines what types of messages are allowed to be received and sent from the node, as well as when it is allowed to transmit [13].

4.2.4 Honeypots

Honeypots is an approach that has been used widely in traditional computer systems. The main concept is to have a fake service which emulates the behaviour of the real system. The information and resources from the honeypot system are separated from the actual system which provides a safe environment to study the behaviour of attackers. The concept of using honeypots in in-vehicle networks has also been a topic of recent research that is presented in [28].

5

Intrusion Detection Systems (IDS)

An intrusion detection system (IDS) is a system that monitors network traffic to detect abnormal behaviour and content. In case of a successful detection, an alarm is raised or a procedure to prevent the attack is taken. A system that takes appropriate actions upon a detection is also known as Intrusion Prevention System (IPS). The outcome with respect to the correctness of the decision made by the intrusion detection system, can be classified into four categories described in table 6.1.

Table 5.1: Intrusion detection system output categories

Output	Description
True Positive (TP)	Identifies an activity as an intrusion and the activity is actually an intrusion
True Negative (TN)	Identifies system behavior as normal and the activity is actually normal
False Positive (FP)	Identifies an activity as an intrusion but the activity is normal
False Negative (FN)	Identifies an activity as normal when the activity is an intrusion

False negative cases are considered to be the problematic, because of their given nature of not detecting an actual intrusion on the network, which can lead to a variety of unforeseen problems. When designing an IDS, it is desired to diminish the false positive rate to a minimum, in order to not flood the logs with erroneous results. Therefore, when evaluating an intrusion detection system it is recommended to aim for the true-positive and true-negative rate to be as high as possible, as well as the false positive and false negative to be as low as possible.

5.1 Intrusion detection system types and classification

This chapter gives a classification on the different types of IDS and how they perform. Figure 5.1 exhibits the major classification of IDS.

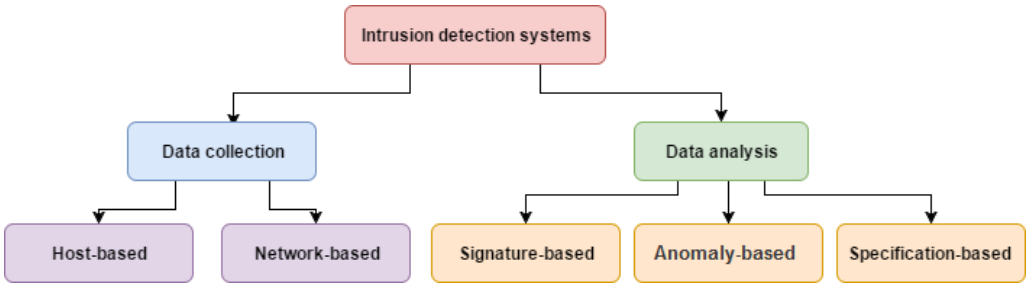

Figure 5.1: Intrusion detection systems classification

The first step of designing an IDS, is to think about the placement of the IDS and which data needed to be collected. The IDS can be host-based, which would collect information about the file system and the behavioural patterns of the user of the system. The second approach features a network-based approach, in which the main goal is to monitor the traffic that is entering and leaving the network.

A signature-based approach is simply looking for signatures of known attack vectors and tries to find them in the in collected traffic. The anomaly-based approach relies on a "baseline" condition of the system and reports everything that is striving away from this baseline. The specification-based approach is a relatively new attempt to somehow connect the aforementioned approaches. The main idea is to define a "legal behaviour" of the communication which is following a certain protocol.

5.1.1 Data collection techniques

Intrusion detection systems can be classified according to the type of data they collect. There are two main categories for these types of IDSs: Network-based Intrusion Detection System (NIDS) and Host-based Intrusion Detection System (HIDS). The network-based IDS, monitors the network traffic and it is usually placed on the traffic routing component, such as a network gateway, which is connecting multiple networks together. This enables the NIDS to intercept and analyze the traffic before it is entering a sub-network, e.g the local area network (LAN).

A host-based IDS, contrary to network-based, is placed on the host computer itself where it monitors the host's system behaviour, e.g. which processes are accessing which resources. In addition to these main classifications of IDS, other sub-categories exist [6], such as:

- Stack-based: monitors the exchange of data between different layers of the protocol's stack.
- Protocol based: Monitors the protocols that are used by the host system
- Graph-based: monitors connections between several nodes or hosts

5.1.2 Data analysis techniques

As mentioned in the introduction to this chapter, there are three main types of intrusion detection approaches: Anomaly-based detection, signature-based detection and specification-based detection. In this section, these different paradigms are explained in greater detail.

5.1.2.1 Signature-based detection

Signature-based detection observes the messages and tries to find pre-defined patterns or sequences. These patterns are also known as signatures.

The Signatures can mainly define two kinds of lists, a white-list or a black list. In the white list, we only define the kind of messages that are allowed in the system and in the blacklist we define the types of patterns and messages that are not allowed to access the system.

Both have their advantages and disadvantages. When using white-listing , we need to to have a prior knowledge of the exchanged messages. This might have huge constraints on open systems , for example computers that use application layer protocols that have very diverse contents. On the other hand, this would be the best choice closed systems and for the lower level protocols that have very limited capabilities and fewer types of messages with predictable content. Correspondingly, blacklisting is suitable for more complex systems. However, the challenge with blacklisting is the need to define the patterns and messages that are not allowed in the system.

5.1.2.2 Anomaly-based detection

In anomaly detection, we observe the behaviour of the system. For this, we need to define an abnormal attitude that will be considered as suspicious. This is similar to the blacklisting approach in the signature based, however, in anomaly based we define a behaviour that can have a much broader scope and result in detecting behaviour that has never been seen before.

In order to model the behaviour of the host system or the network traffic observed, there is a need to construct a view which has a certain level of abstraction, that leads to extraction of features. These features are then used together with machine learning techniques that can construct a hypothesis that models the behaviour of the system e.g Neural networks, support vector machines, time series , standard deviation and more. Upon the IDS run-time, features similar to the one used to train the model, have to be extracted again before passing them to the hypothesis that will perform the classification.

5.1.2.3 Specification-based detection

In specification based intrusion detection system, a set of properties, that are extracted from the protocol design are defined for the monitoring purpose i.e. we know the correct system behaviour and can specify it in rules. The classification and detection is then performed by observing a deviation of the execution from the defined properties.

5.1.3 Stateful vs Stateless Intrusion detection systems

Another general sub-classification for intrusion detection systems are: stateless and stateful. A stateless intrusion detection system is a system that does not keep track of previous data it has already seen. While stateful intrusion detection systems, keep temporary information about previously seen data in order to use it for possible further detections.

5.2 Intrusion detection system architecture

The following section focuses on the basic system architecture of a network-based intrusion detection system. This will aid in gaining a deeper understanding of how to plan and build the prototype IDS [21].

When implementing an Intrusion detection system, it is not possible to completely resort to a common standard due to varying requirements when utilizing the gathering of the data and its analysis. Nevertheless, almost all of them have some basic components/modules that they all share [21]. These components are:

- **Data gathering:** used for monitoring the source environment. The data gathering is performed using different sensors that observe a specific application or protocol.
  A pre-processing module can also be included [3] that performs basic classification of the data type received from the source.
- **Detector (Detection engine):** is a module that performs the comparison between the gathered data and the defined rules set and raises alarms in case a deviation is found.
- **Database(Knowledgebase):** is a storage module that contains the rule-sets or the IDs which the detector uses when comparing the received data.
- **Output (Response):** When an alarm is raised a proper action is taken. This could be an active response where the IDS performs a predefined action such as drop the packet, or an inactive response such as logging for later inspection by a human factor to determine the appropriate response.

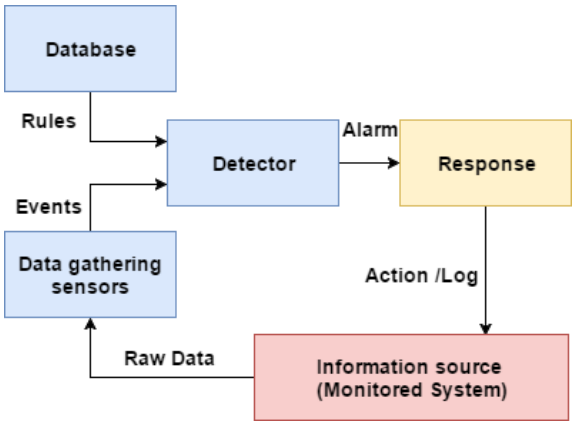

Figure 5.2: Intrusion detection system architecture

We use Snort, an IDS/IPS [3] to explain how a generic IDS works. Figure 5.3 shows the architecture of Snort. We can see that the packet arrives from the network where it is collected by the sniffer module. The packet is then sent to the pre-processor that inspects the type of the packet the detection engine will deal with. This information together with the packet are then forwarded to the detection engine that compares the internal component of this packet with the predefined rule set stored. Based on the decision of the detection engine an alert is raised and logged in the log database.

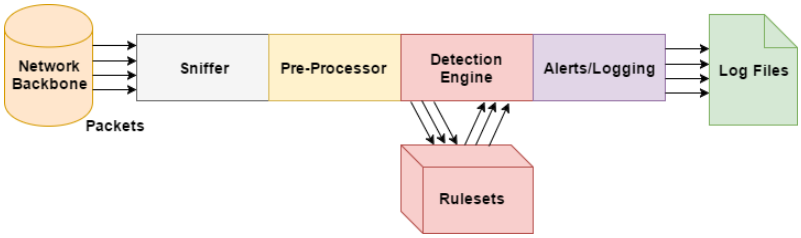

Figure 5.3: Snort IDS architecture

6

In-vehicle intrusion detection systems

6.1 Related work

The literature review revealed a variety of different approaches for IDS in in-vehicle networks. Previous research has shown to be heavily dominated by an anomaly-based approaches.

When an ECU broadcasts messages over the CAN bus, these messages are identified by message IDs. Furthermore, these messages have a predefined frequency, where the messages are regularly transmitted with an interval separating them. Researchers [23, 15] have shown that when an attacker injects legitimate messages, to perform a spoofing attack or a Denial of Service (DoS) attack, these frequencies will increase. This observation has been considered for many suggested IDS for CAN such as [23] where alarms are raised when the interval of the messages passes a certain threshold.

Since in-vehicle network messages have a very strict and predefined behaviour, [16] suggested that a normal state of the network can be modeled and the randomness entropy of all the messages can be calculated. A low entropy of the network can act as an indicator for a compromised network, because an attacker could send a huge number of high priority messages, that lower the overall entropy. Whereas, high or normal entropy can be used as an indicator for a normal network.

Muter et al. [17] suggested an approach based on different sensors, which verify the structure and the content of transmitted messages, to gain information about any misuse of the "normal" structure and behaviour. The sensors are listed in Table 6.1.

Table 6.1: In-vehicle IDS sensors [17]

Sensors	**Description**
Formality	Checks the size of different fields of the data frame.
Location	Checks whether a frame is allowed in a certain sub-network.
Range	Checks the range of the data in the frame.
Frequency	Checks whether the frequency of a frame in compliance with the frequency stated in the signal database.
Correlation	Checks whether the frames from distinct bus systems correlate in compliance to the specification.
Protocol	Checks whether the communication on the bus following the protocol.
Plausibility	Checks whether the transmitted information plausible. E.g the speed not going from 10km/h to 100km/h in 0.5 sec.
Consistency	Checks if the data is consistent when they are sent from the same source.

Deep learning and neural networks are also a common approach for developing anomaly-based IDS which is adopted by [12] and deployed for in-vehicle networks.

In [13], the authors discuss specification-based detection, where the policies are enforced directly on the nodes and rules are extracted from the protocol specification and compared with the transmitted messages. This is very similar to the formality, location, range sensors discussed by [17] and listed in table 6.1.

Masquerading attacks, where the message carries legitimate and legal values are attacks that are not easily detectable in in-vehicle networks, due to the lack of authenticity and signatures in the sent messages. The Clock-based Intrusion Detection System (CIDS) [4] uses an approach which monitors every ECU's clock offset and clock skew in order to fingerprint the transmitting ECU. The fingerprints are then used to model the standard behaviour that is used later for detection.

6.2 Challenges and constraints

Design and implementation of an intrusion detection systems for in-vehicle networks coheres to a variety of challenges and constraints. During the initial literature review, it became apparent that very few papers mention the challenges that are involved in the development and deploying process. This section serves as a basic overview over these constraints.

6.2.1 Hardware constraints

Traditional computer systems and relatively expensive hardware do not have any problem using a lot of processing power and memory to run programs [24]. Most of the traditional computer systems can handle and run dozens of applications and services at the same time.

Unfortunately, this is not the case in the vehicle hardware or any embedded system in general. A small embedded system usually takes care of a small computing task that does not require a lot of processing power or memory which makes the cost of production low. The tasks performed by such embedded systems have to be executed within a certain time frame with no buffering.

For this reason, when designing a software for an embedded system in general, the software has to be made in a way that can adapt to the previously mentioned constraints.

6.2.2 Detection method and data selection

The CAN bus has a strict well defined protocol specifications, which makes it a complex task to extract and select the correct information to be used in the IDS. Furthermore, the high message exchange rate adds another layer of complexity to the detection. Both anomaly based and specification based approaches have been suggested. According to the literature, a combination of these both approaches is the most effective and can detect a wide range of attack vectors [6].

A signature based approach for an IDS in the CAN bus, has scarcely been mentioned in the literature. One of the main reasons why researchers have not focused on the signature based approach is, because there are a few, if not none, attack signatures to be used in an IDS for the CAN bus.

6.2.3 Detection accuracy and performance

An important aspect of the intrusion detection system development's life cycle is the evaluation of the detection accuracy. A known problem when designing an IDS system is the false-positive and false-negative rates. This is a difficult problem to overcome especially for anomaly based solutions and for the zero-day attacks.

Network based intrusion detection systems have to cope with an excessive amount of data processing when installed on networks with high bandwidth, especially those which connects multiple networks. This stretches the processing and hardware limits of the ECU on which the IDS is implemented on. Therefore it is crucial to think about these constraints in the early phases of the development cycle, to ensure good performances.

6.2.4 Placement

The placement of the in-vehicle intrusion detection system is a very important aspect to think about when designing the system. Previous work such as [13, 18] studied the impact of different attacks performed by ECUs that were placed in distinct sub-networks.

The impact of a compromised gateway ECU is larger than a normal compromised ECU [13], owing to the fact that a gateway ECU has control and access to more packets, because the packets have to traverse through the gateway ECU in order to reach remote domains.

Additionally, it is important to think about the domain in which the ECU is placed and the functionalities that can be controlled through the ECU. Nilsson et.al [18], study the effect of ECU failures concerning passenger safety. For example, the risk of an attacker being able to send frames to the telematics train can cause distraction of the driver, thus endangering his or her life. Whereas the same attack on the vehicle safety train may trigger a safety functionality, such as the air bags while driving. The most dangerous domain, according to Nilson et.al, is the power train, where messages can possibly control engine functionalities and breaks.

6.2.5 Response to an attack

Autonomous decision about how to act upon an intrusion or the way to notify the driver comes with a big safety risk [25] as improper reactions can endanger the life of passengers. Even if a potential comprise is not endangering the passenger, how should the system react? Should it react at all? Is it necessary to drive the car to a repair shop? Even if a solution with a lower risk is found, there is a necessity to think about potential false-positive results.

6.2.6 Log storage, post analysis and updates

When an IDS for an in-vehicle network detects an intrusion, there will be a need for storing log files for further analysis. It's necessary to think about storage solution for these log files and ways to forward them to the manufacturer for further inspection.

A further complication that arises, especially in the case of a signature-based IDS, is the process of keeping the given attack signatures up-to-date and adapt to newly discovered ways of compromising the network. The average lifetime of a car can exceed 10 or more years, which constitutes the problem of delivering the updated signatures to the targeted car.

6.2.7 Cost

Implementing a security solution for the in-vehicle network will, in most cases, require adding extra hardware. For instance, imagine adding a small ECU that takes

care of message verification and encryption. Let us say that this ECU production cost is 1 dollar. In a modern car we have about 100 ECUs, that is 100 dollars for each vehicle. Say the manufacturer produces 1 million cars, that is 100 million dollars in revenue loss just for adding message verification and encryption nodes.

7

Setup and Design

In this chapter, we give a description of our experimental setup that simulates a simple CAN network in a vehicle and we also explain the setup required to interact with a real vehicle. Both setups have been used to perform attacks on different nodes in these networks. Furthermore, we show and motivate our design choice for in-vehicle IDS system. We propose two approaches for detection: A specification based approach with rules extracted from the signal database, and an anomaly based approach driven by algorithms based on the analysis of the normal system behaviour.

7.1 Experimental setup

This section presents both the software and the hardware used to build our simulation setup and the real vehicle setup.

7.1.1 CANoe

CANoe is a software, made by Vector [27], that is used for designing, simulating and analyzing ECU networks with support for a wide variety of communication technologies used in commercial vehicles, including CAN.

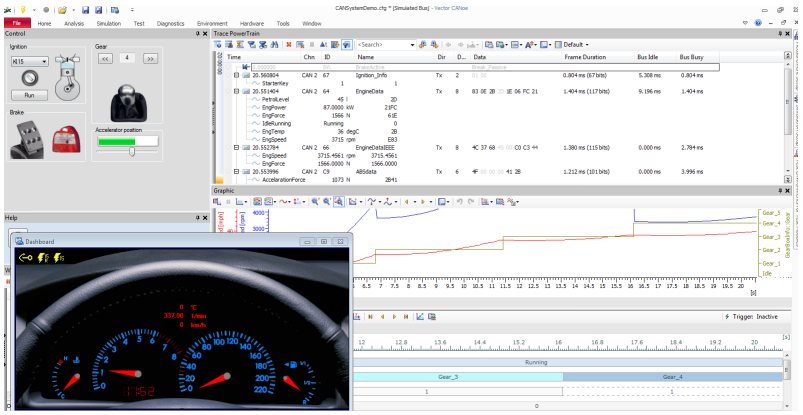

Figure 7.1: Snapshot of the CANoe software GUI

In addition to the software, we use a hardware component made by Vector

with the model number *VN8912*. This hardware component acts as an interface for a PC and provides a real time parallel access to different buses like CAN, FlexRay and LIN, which can be used in a real in-vehicle network when connected to the OBD port.

7.1.2 Signal database

The Distributed System Backbone Communication (DBC) for the CAN bus [26] is a database, which describes the entire CAN network communication. It can be used to translate bits and bytes to meaningful messages, e.g. diagnosis request messages.

CANoe provides an editor for such files that is called the CANdb++ Editor [26], where the developer can easily view, construct and edit these kinds of files.

Name	ID	ID-Format	DLC [...	Tx Method	Cycle Time	Transmitter
Console_1	0x1A0	CAN Standard	4	not_used	20	Console
Console_2	0x1A1	CAN Standard	2	not_used	500	Console
DebugMsg1	0x100	CAN Standard	8	not_used	2	-- No Transmit...
Diag_Request	0x700	CAN Standard	8	not_used	2	-- No Transmit...
Diag_Response	0x600	CAN Standard	8	not_used	2	-- No Transmit...
DiagRequest	0x606	CAN Standard	8	not_used	2	-- No Transmit...
DiagResponse_DoorLeft	0x607	CAN Standard	8	not_used	2	DOOR_le
DiagResponse_Motor	0x601	CAN Standard	8	not_used	2	Gateway
DOOR_l	0x1F0	CAN Standard	1	not_used	30	DOOR_le
DOOR_r	0x1F1	CAN Standard	1	not_used	30	DOOR_ri
Gateway_1	0x110	CAN Standard	3	not_used	100	Gateway
Gateway_2	0x111	CAN Standard	8	not_used	2	Gateway
NM_Console	0x41A	CAN Standard	4	not_used	2	Console
NM_DOORleft	0x41B	CAN Standard	4	not_used	2	DOOR_le
NM_DOORright	0x41C	CAN Standard	4	not_used	2	DOOR_ri
NM_Gateway	0x41D	CAN Standard	4	not_used	2	Gateway
TP_Console	0x604	CAN Standard	6	not_used	2	Console
TP_Dashboard	0x605	CAN Standard	6	not_used	2	Dashboard

Figure 7.2: The signal database in CANoe

Figure 7.2 shows an example of such database. It defines a number of messages that have unique IDs mapped to human interpretable names. Further, it defines which node is the sender of a specific message, together with additional information that defines a CAN data frame, e.g the cycle time. A signal is a specific data value that can represent sensor data or operation information. As explained in Chapter 3, each CAN data frame has 8 bytes (64 bits). A CAN data frame is capable of carrying more than one signal at a time. The message can carry one signal with 1 bit to 64 bits or as many as 64 binary signals.

Figure 7.3 shows a list of signals carried by the message *Gateway_2* from Figure 7.2. Each signal has a set of properties defined in the signal database, that includes the start bit, the length and other useful information. Figure 7.4 highlights the signal mapping in a more clear manner.

Name	Message	Startbit	Leng...	Byte Order	Value Type	Initial Value	Factor	Offset	Mini...	Maxi...	Unit
EngineTemp	Gateway_2	0	8	Intel	Unsigned	0	1	0	0	0	degC
CarSpeed	Gateway_2	8	16	Intel	Unsigned	0	0.5	0	0	300	mph
EngSpeed	Gateway_2	24	16	Intel	Unsigned	0	1	0	0	0	rpm
PetrolLevel	Gateway_2	40	8	Intel	Unsigned	0	1	0	0	0	l
Voltage	Gateway_2	48	10	Intel	Unsigned	12	0.1	0	0	102.3	V

Figure 7.3: Example *Gateway_2* message signals

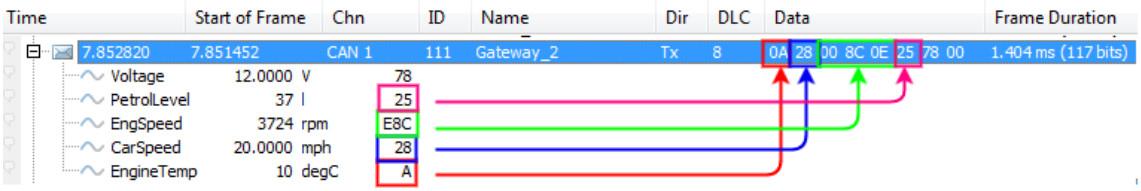

Figure 7.4: CAN frame data to signal mapping (Signals)

7.1.3 Simulation setup

To comprehend and test the behaviour of an in-vehicle network in a secure environment, we decided to use a simulated setup for the early experimental stages of this work.

The setup is closely related to the default environment in CANoe for simulating a CAN network. The network consists of two CAN buses emulating the power train and the comfort train connected through a gateway. The gateway forwards all the messages going from one bus to another. The power train has one ECU that listens to messages that are carrying information about the state of the engine. Doors, dashboard and the console are controlled by the ECUs located in the comfort train. The dashboard ECU collects and sends information regarding engine performance, engine temperature, petrol level, car speed and gear shifts to display it to the driver. Messages regarding mirrors and light are managed by the console ECU.

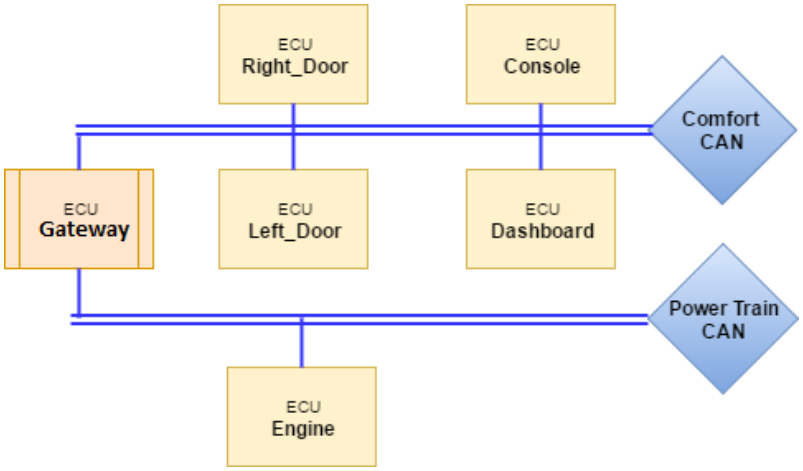

Figure 7.5: The project's simulation setup

All ECUs in the setup are flexible and their code and behaviour can be modified. New additions to the network are possible by adding extra ECUs which extend the network with further functionalties.

7.1.4 Real vehicle setup

While our simulation setup consists of two domains and less than ten ECUs, real in-vehicle networks consist of more than 100 ECUs.

We had the privilege to access the box car lab at Volvo Cars and test the different approaches for our experiments. The setup uses the same CANoe hardware to act as an interface for a PC, but this time we had to connect it to the On-Board Diagnostics (OBD) port on a car and configure the channel for the domain we want to access. At this point, we are able to listen and trace the messages after adding the corresponding signal database for the domain. However, in order to send messages we had to add a virtual node using the CANoe software. A virtual node can act exactly as a normal node: it can send, receive and process CAN frames. Figure 7.6 illustrates our setup.

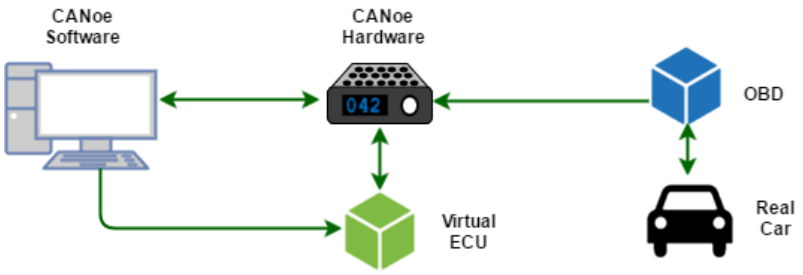

Figure 7.6: Real vehicle experimental setup

Both the simulated and the real car setup consist of the same basic functionalities, regarding the implementation of our prototype IDS. The only difference is the signal database we have used and the different IDs and properties of the messages.

The placement of the attacker and the IDS node depend on which type of attack or detection we want to perform. In the real car setup it is not possible to neither overwrite the existing functionality nor add a new (physical) ECUs and we had to rely on adding virtual nodes. Meanwhile, in the simulated setup we have the ability to add, remove and edit every functionality on any ECU, regardless of it's type (normal node or a gateway node).

While the structure and design of the simulated network was rather straightforward and manageable, the propagation of our setup to a real network was not as easy. The real network consists of several network standards interconnected with each other. Figure 7.7 gives an abstract overview of the real-architecture of the in-vehicle network we had to deal with. In order to access parts of the network, we had to rely on a single access-point through the OBD port. This access point allowed us to connect to special CAN domains, that were the *Propulsion CAN* and

the *Chassis CAN*. Access to other sub-networks was either restricted by the OBD port or was out-of-scope.

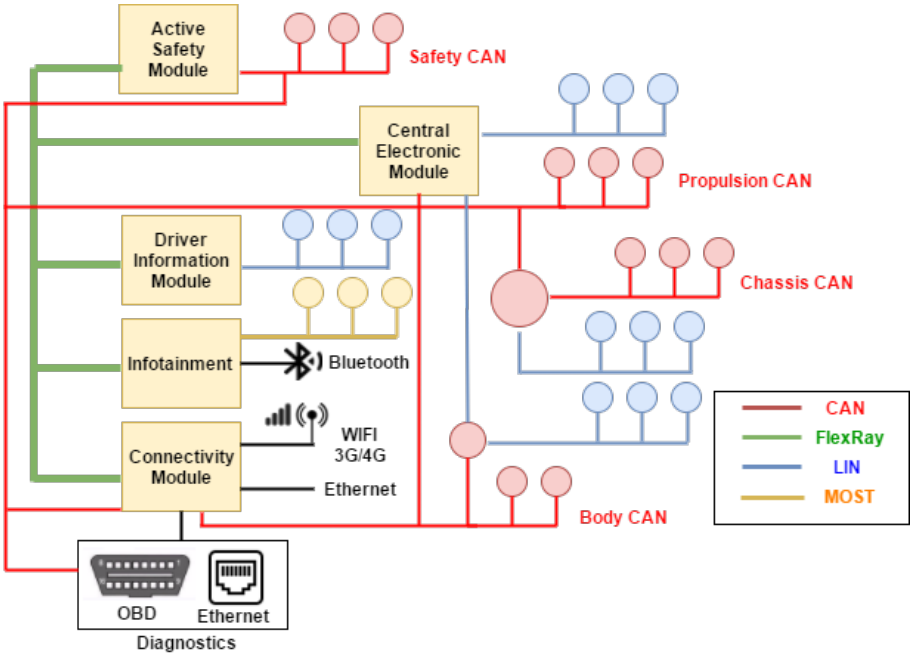

Figure 7.7: Real experimental vehicle network architecture

7.2 System design and structure

In a previous chapter, we explained the architecture of computer based IDS. In a similar way and with the resources available, we propose the use of software-based IDS embedded in a single ECU.

An *IDS node* is a dedicated ECU that listens to messages and monitors one domain or more. Recall that the gateway is capable of receiving all messages from the domains it is connected to. On the other hand, a regular node can only receive the messages sent in its own domain. The capability of the IDS node to listen to several domains relies on the choice of whether to deploy the IDS system on a gateway or on a regular node.

An IDS node is expected to have a list of generic functionalities regardless of the approach it follows to perform the detection. We only consider logging of an actual attack and we leave the research on how to prevent or how to warn the user for future work. The generic functionalities for an IDS node are:

1. Sniffing: Receive all the messages that were sent on the bus.
2. Filtering.
3. Decision making: Process each message based on its ID.
4. Logging.

Figure 7.8 highlights, when a data CAN frame (message) is sent by an ECU on the bus, any other ECU in the same domain can receive and inspect its content, due to the broadcasting nature of CAN communication. After the message is received by the IDS node, it needs to be categorized according to its ID. With the message ID we can identify the properties and the behaviour of the message. The message then goes through several checks, a normal message should follow the signal database specification and pass the anomaly-based algorithms without being flagged. Upon a successful detection of a malicious message, the information can be logged with more relevant information e.g. timestamp of the detection.

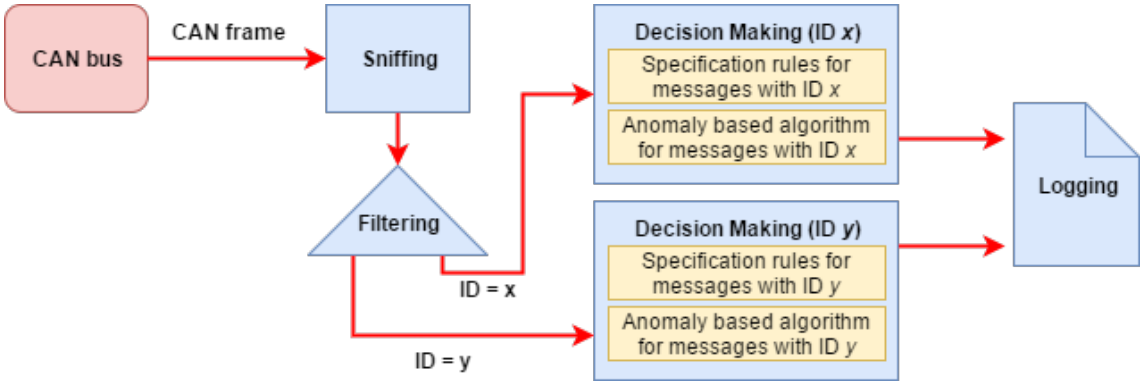

Figure 7.8: The proposed architecture of the in-vehicle IDS

Comparing our proposed architecture with a computer-based IDS architecture, it is noticeable how similar both architectures are. Nevertheless, the set of rules in a computer-based IDS uses a separate component to define generic rules or according to each packet's protocol. While in the case of our proposed IDS, each set of rules or algorithm has properties which only work for messages with a specific ID.

Implementation wise, CANoe offers the *on Message* event-based function, which allows to listen and filter messages according to the ID of the message or its predefined name in the signal database. Using this function, for each message that is sent in the domain where the IDS node is placed, we are able to match and detect any abnormal message in that domain. Thus, each block (*on Message* function) can represent a sensor for a unique message and an IDS node can have a number of blocks that are equivalent to the number of messages expected in the domain. Listing 1 presents a template for this sensor block.

```
1 on Message SOME_MESSAGE {
2   // Specification-based checks
3
4   // Anomaly-based checks
5 }
```

Listing 1: A sensor block

8

Implementation of an IDS system

This section explains the process of creating both the specification-based detection rules and anomaly-based detection algorithms. While these rules can be all gathered in one sensor block for one message ID, we show the rules and algorithms in separate blocks for clarity.

8.1 Specification-based detection

Specification based rules are extracted from the design specification of the messages. The properties of each unique message is defined in the signal database in each domain. This makes the task of extracting such rules straightforward but challenging at the same time, due to the huge number of messages that are usually transmitted in an in-vehicle network. A major challenge is to be able to cover all the messages and signals defined in the signal database. For this we examined the signal database from CANoe with the dbc format and we found that the file can be simply opened in a text editor and we can see the content. Still we had to parse the file to extract the information in a structured order, which can be done with any programming or scripting language. With this we were able to extract the information needed and define rules accordingly.

We categorized the specification-based detection methods into two types according to their objectives. The first one, inspects the message properties and compares it to the signal database specification, while the second one detects any unauthorized messages in the monitored domain.

8.1.1 Message parameter misuse

An attacker is able to inject malformed frames onto the CAN bus by sending a message that another ECU is expecting to receive. Furthermore the attacker is able to change the message properties that are defined in the signal database, such as data length or changing the bit length of the signal values. This can force the CAN bus to behave in unexpected ways.

Since we know the properties of each CAN message, we define a rule set that inspects the structure of messages and their properties that the ECU should accept or reject before processing the message.

The steps to write such rules depends on the *message ID* we monitor. We first need to identify the *domains (sub-networks)* in which this message originates and is sent to. Knowing in which domain the message is going to be transmitted, we can define where to place the IDS node which listens for this message ID, i.e the sensor blocks for this message ID.

For example, assume we want to write a rule set for the message with the ID *0x110*. We know that this message is present on the comfort train domain. We now know where to place the sensor block. However, since our intention is to inspect *any* deviation for all messages received within any domain, we can think about placing the IDS on the gateway as illustrated in Figure 8.1. Moreover, if we inspect the signal database for information about this message we learn that it carries three signals. The first signal (x), has the maximum of 8 bits, subsequently the second signal (y) has 8 bits and the third signal (z) has 5 bits. With this information we can extract a list of specification-based rules presented in Table 8.1.

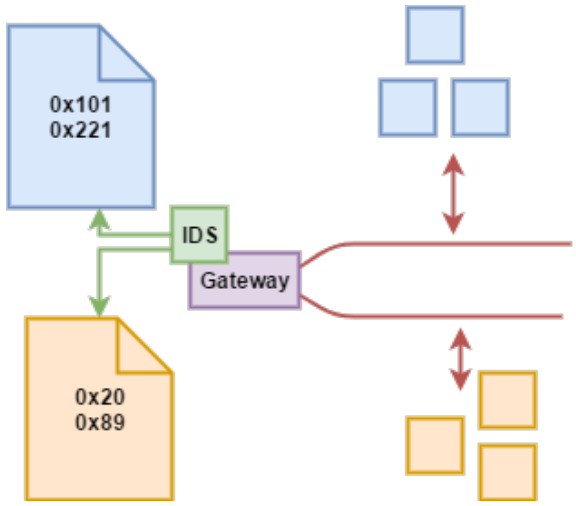

Figure 8.1: Placement of an IDS node with malformed frame detection rules

Table 8.1: An example of specification and rule pairs for message parameter misuse detection

Specification	**Rule**
The message carries three signals each signal is 8 bits or less	$DLC = 3$
Signal x is 8 bits maximum	$0 \leqslant x.value \leqslant 255$
Signal y is 8 bits maximum	$0 \leqslant y.value \leqslant 255$
Signal z is 5 bits maximum	$0 \leqslant z.value \leqslant 31$

In a similar way, we can extract other rules for the rest of the properties that are defined in the signal database for a specific message. The code implementation is straight forward and can be done using direct *if-else* conditions. The implementation for the previous example with the message 0x110 is presented in Listing 2.

```
/*MALFORMED FRAME DETECTION ID=0x110 */
on message 0x110{
  if (this.DLC != 3 || this.x>255 || this.y>255 || this.z>31){
    write("MALFORMED FRAME with ID  %x",this.id);
  }
}
```

Listing 2: A simple code example for message parameter misuse detection

8.1.2 Unauthorized message detection

In the signal database, each message has defined sender and receiver nodes. The gateway has the additional task of forwarding the messages from one domain to another. Nevertheless, an attacker can still send:

- Messages that may not be received by any node in the domain.
- Messages from one domain to another through the gateway.
- Messages to cause denial of service, for example high-priority messages with ID 0x00.

To detect this, a specification-based white-list is required for each domain where this white-list contains all the authorized messages in that domain

Let us consider a small network in which a local ECU receives the messages with IDs *0x20,0x101,0x221* . The messages *0x101,0x221* originate from the same domain while the message *0x20* comes from another domain, forwarded by a gateway. The IDS node uses the white-list to flag unauthorized messages. Figure 8.2 illustrates the placement of the IDS node with an unauthorized message detection rule for the aforementioned example.

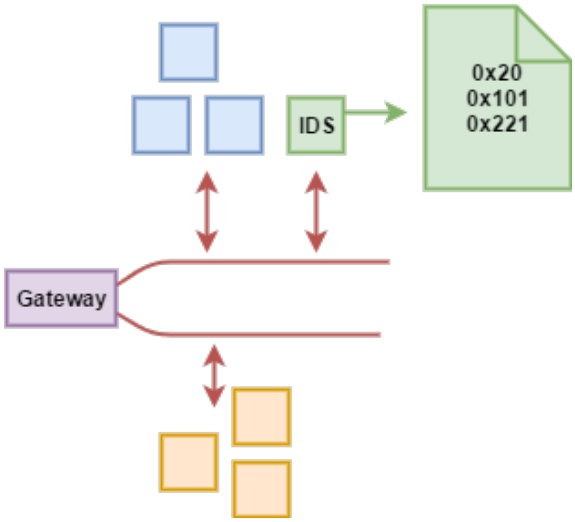

Figure 8.2: Placement of an IDS node with unauthorized message detection rule

The code in Listing 3 shows how this while-listing approach for the previous example can be implemented in CAPL. However, due to the limitation of the CAPL

programming language as it does not provide any search algorithm, an implementation of such algorithm is required to reduce the code complexity cost.

```
variables{
  /*An array containing a white list of the messages exchanged in the domain*/
  int messages_allowed[]={0x20,0x101,0x221}
}

on message * {

  int i;
  int count=0;

 /* Search */
  for(i=0;i<length(messages_allowed);i++){
    if(this.id==messages_allowed[i])
      count++;
  }

  /* Raise an alarm (log) */
  if(count!=1){
    write("Unauthorized message with ID=\%d",this.id);
  }
}
```

Listing 3: Unauthorized message detection example

8.2 Anomaly-based detection

In order to build an anomaly-based algorithm, careful analysis of the normal system behaviour is required. In this section, we present two types of anomaly-based detection algorithms. The first type monitors the the plausibility of the speed value sent from an ECU to the dashboard, in order to detect abnormal speed shifts due to injected traffic by an attacker. The second type monitors frequency changes of messages with a specific ID, this approach has been widely discussed by previous research [23, 4, 16] due its simplicity and effectiveness.

8.2.1 Speedometer plausibility detection

For this type of detection, we consider the speed values sent to the dashboard. The main criteria for plausibility detection, is to detect an abnormal shift in the speed. For example, it is not expected to see a frame that contains the speed signal value that is equivalent to 30 km/h followed by another frame containing the value 200 km/h. Reviewing more recent literature, we notice that only a few, such as [17] discuss attacks and defence techniques that relate to the plausibility of CAN signal values.

In our setup, the raw speed values received by the ECU should be multiplied

by a factor of 0.5 mph. This conversion is displayed in Figure 8.3 and is archived by using Equation 8.1 with (**x**) being the **raw speed data**.

The main idea behind anomaly-based detection is the detection of abnormal behaviour by extracting the normal behaviour of the system and then using this for monitoring purposes. Therefor, a simple analysis to derive a threshold on speed shifts is required. The key idea behind this analysis is to be able to extract an upper limit for the maximum value the speed value is allowed to shift between two consecutive messages.

$$real\_speed = x * 0.5(mile/h) \Rightarrow km\_speed = real\_speed * 1.60934(km/h) \quad (8.1)$$

Name	Message	Startbit	Leng...	Byte Order	Value Type	Initial Value	Factor	Offset	Mini...	Maxi...	Unit
EngineTemp	Gateway_2	0	8	Intel	Unsigned	0	1	0	0	0	degC
CarSpeed	Gateway_2	8	16	Intel	Unsigned	0	0.5	0	0	300	mph
EngSpeed	Gateway_2	24	16	Intel	Unsigned	0	1	0	0	0	rpm
PetrolLevel	Gateway_2	40	8	Intel	Unsigned	0	1	0	0	0	l
Voltage	Gateway_2	48	10	Intel	Unsigned	12	0.1	0	0	102.3	V

Figure 8.3: Speed signal value conversion in signal database

In order to perform the analysis, we use the simulation setup. The simulation setup allows us to perform a scenario where the car accelerates and stops repeatedly. Each round of acceleration and breaking runs for about 20 seconds. In each round, the car accelerates to reach the fifth gear then starts breaking gradually. The speed signal in the simulation setup is carried by a message named *Gateway_2*. This messages is originated by the engine ECU. This message passes through the gateway in the network and is received by the dashboard ECU. While testing, we notice that in each acceleration round, around 3700 to 3900 messages are sent to the dashboard ECU. Finally, to perform the analysis:

- We run the simulation and gather 4000 samples.
- Each sample represent the difference between the speed value of two consecutive messages.

According to the results presented in Table 8.2, the majority of the speed messages have small shifts in values more precisely 99.55% of the consecutive samples have a value shift of 1, 2 or 3 raw speed. The maximum value shift we encountered during the simulation was 19 raw speed. With this, we defined our threshold to be 20 raw speed. Recall that the real value need to be multiplied by 0.5*1.6 to get the corresponding value in km per hour. That means that the threshold 20 raw speed, is equivalent to about 16 km/h. This value is very unrealistic because of the time difference between two consecutive messages is too short, i.e. it is almost impossible to have such a big increase in speed in such a short time. However our intention is only to show how extraction a threshold could be done.

Last but not least, we use the threshold in a light-weight algorithm which we present in Listing 4. The code is very simple and includes the threshold and a variable that holds the shift value between two consecutive messages. Both values

are then used in a direct comparison to make a decision weather the received signal value is allowed or not.

Table 8.2: Speed value shift analysis

Speed value difference (raw speed)	Samples (message)	Total percentage
1	3114	77.85%
2	638	15.95%
3	230	5.75%
4	6	0.15%
5,6,7,8,9	0	0.0%
10	1	0.025%
11	1	0.025%
12,13	0	0.0%
14	1	0.025%
15	1	0.025%
16	1	0.025%
17	3	0.075%
18	3	0.075%
19	1	0.025%

```
Variable{
  int carspeed_monitor;     /*This is the value we calculate and monitor*/
  int speed_diff_threshold;
}

on start{
  carspeed_monitor=0;
  speed_diff_threshold=20; /*The derived threshold*/
}

on message Gateway_2 {
  /*SPEED PLAUSIBILITY DETECTION*/
 carspeed_monitor=this.CarSpeed - previous_speed;
 previous_speed=this.CarSpeed;

 if (abs(carspeed_monitor)>=speed_diff_threshold){
   write("FALSE READINGS");
  }
}
```

Listing 4: Speedometer plausibility detection

8.2.2 Frequency change detection

A more generic way to detect message injection is to observe the message frequency behaviour in the system. Previous works have mostly focused on the change in message frequency when an attacker tries to inject periodic messages. Similarly,

we show the effectiveness of such detection mechanism and the analysis required to derive an IDS rule.

The main idea behind this approach is that, while the compromised node injects messages on the bus, the original node keep sending the same message with the same ID with the original frequency, a noticeable change in the message frequency will appear.

Our first thought was to use a specification-based-like rule, since we know that the cycle time for each message is specified in the signal database. However, since messages on a CAN bus are transmitted asynchronously, a deviation from the defined cycle time may appear but is rare.

To show this, we gathered 4000 messages from the simulation setup without any attack simulations. The gathered messages all have the same ID *Gateway_2*, and are defined in the signal database with a cycle time of 2 milliseconds. We analyzed the message arrival time difference between each two consecutive messages.

The plot in Figure 8.4 shows the result of this experiment. It can be seen that the majority of these messages follow the defined cycle time and have a 2 milliseconds distance from the previous message, while others have a higher difference which is expected due to collision avoidance and priority message handling.

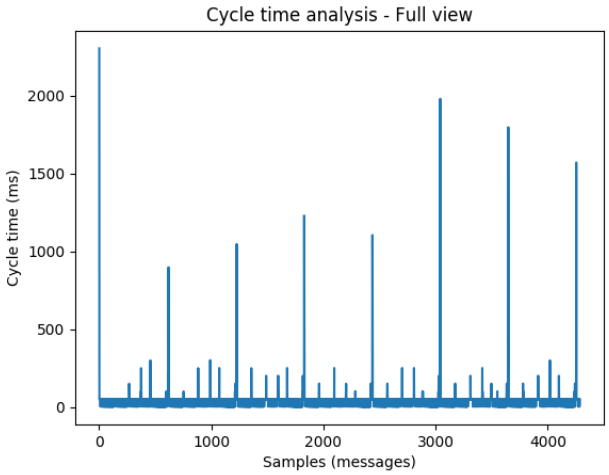

Figure 8.4: A plot showing the cycle times between consecutive messages

According to Figure 8.5 that holds the lower values of the plot in Figure 8.4, two of these 4000 messages have a cycle time lower than 2 milliseconds. The reason can be clock skew of the sending ECU. This example shows that it is very important to think about all possible deviations when writing the rules to avoid false positive alarms.

The simplest solution we can think of is to write a double check rule that eliminates such false positives. In order to do that, a temporary counter is used that holds the number of consecutive messages that exceed the threshold. The final decision is then made by checking whether this value counts more than one message or assigning a zero value otherwise. The final code is presented in Listing 5.

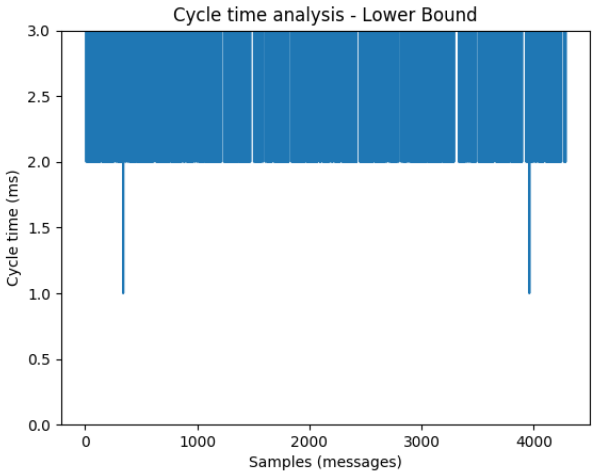

Figure 8.5: The lower bound of the cycle time difference analysis experiment

```
variables{
  int posibleTimeRateAttack;
  int consecutive;
  message Gateway_2 prevMessgTimeRate;
  //cycle time for message Getawy_2 in the database is equal to 2
  int cycle_time_threshold=2;
}

on start{
  consecutive=0;
}

on message Gateway_2 {
  /*TIME RATE CHANGE DETECTION*/
  posibleTimeRate=0;
  if (timeDiff(prevMessgTimeRate,this)<cycle_time_threshold){
    posibleTimeRateAttack=1;
    consecutive++;
    if (consecutive>1){
      write("ATTACK_TIME RATE CHANGE!!");
    }
  }
  /*False alarm , reset the counter*/
  if(posibleTimeRateAttack==0 && consecutive>0){
    consecutive=0;
  }
  prevMessgTimeRate=this;
}
```

Listing 5: Time rate change detection example

9

Results

In this chapter, we present an overview of the performance of the developed intrusion detection system. The process we followed for this evaluation is closely related to the approaches the authors of the related work have taken. We perform a set of attacks and observe the behaviour of the system, while having an active intrusion detection system node.

The test were performed on both the simulated setup and a the realistic setup which is a box car[1]. The evaluation covers detection of unauthorized messages (specification-based), plausibility detection (anomaly-based) and frequency change detection (anomaly-based).

We want to highlight that there are almost no differences between the attack or detection results while performing the tests on the different setups. The only difference is that we had to adjust the message IDs that we inject and detect based on the target environment. Note that we describe the attacks in a theoretical manner, whereas a more practical description of the attacks can be found in the Appendix of this report.

9.1 Specification-based attacks

9.1.1 Unauthorized message detection

The specification-based rules extracted from signal database are used to detect malicious messages that deviate from the original specification. These rules are also capable of detecting unauthorized messages in the domain, based on ID look ups in a white-list that contains the authorized messages in the monitored domain.

We perform two tests in which we simulate an attack by injecting messages in a specific domain. The injected messages are combinations of normal messages that follow the signal database specification and malformed messages that have one or more parameters changed. The type of changed parameters can be seen in the first four rows in Table 9.1.

[1]A box car includes all the car electronic components and it's relative body parts except for the engine. It is used for testing the functionality of the different in-vehicle network as well as the functionality of the in-vehicle network and its electronic components.

As for the unauthorized message detection, we inject a combination of messages: messages that belong to other domains and messages that are not defined at all in the signal database.

In order to perform these tests on the box car, in each test we add a virtual attacker node and a virtual IDS node. Finally, the domains we chose to test on are the propulsion CAN and the chassis CAN trains.

As presented in Table 9.1, the result of this evaluation show a 100 present detection rate in both setups (simulated network and box-car). Such high detection rate is due to the nature of these rules, as they perform a direct comparison between the message properties and the rules extracted from the signal database. This approach leaves no possibility for any false positive occurrence.

Table 9.1: Evaluation results for message parameter misuse and location change detection

Parameter changed	**Detection rate**
Data length (DLC)	100% detection rate
Signal bit length	100% detection rate
Constant signal byte value	100% detection rate
Unauthorized messages	100% detection rate

9.2 Anomaly-based attacks

9.2.1 Plausibility detection

In this section, we perform two tests in which we launch identical attacks on the speedometer that differ in the data values. The attacks are then used to evaluate the efficiency of our speedometer plausibility detection approach and showing the behaviour of the system when such attacks happen.

The target ECU is the dashboard ECU which processes the speed signal messages. We place an IDS node in the domain where this ECU is placed, *Comfort CAN*, in order to be able to monitor all of the transmitted messages to this ECU.

This is the only test that we were not able to perform on the box car. This was due to the design of the system where the speed information was transferred over FlexRay instead of CAN, which was outside the scope of this work.

9.2.1.1 Constant speed injection

In the first test, we inject 1,000 messages that hold a constant high speed value. The chosen number of messages to inject were chosen only due to the fact that 1,000 messages were enough to see the change when observing the speedometer by eye. Since the system observes abnormal shifts in speed values, the detector is only able to detect the start of the attack, where the value shifts from the normal speed

to the injected speed, and the end of the attack, where the value shifts from the injected speed value to the normal value sent from the original ECU.

For example, if the car is driving at speed of *30 km/h* and we inject messages that contain signals with constant speed of *200 km/h*, the speed shift from the first message of the attack and the real speed will be *(200-30=170 km/h)*.

Since speed is a constant *200 km/h* during the attack, the expected speed shift values will always hold the value of zero while the attack is performed *(200-200=0 km/h)*. In the end of the attack, assuming the real speed is still at *30 km/h*, the shift will also be *(200-30=170 km/h)*.

Figure 9.1 shows a sample of 4,000 messages with two attacks performed (1,000 messages each), we can see the speed shifts change to a constant value during the attack. We can also identify the exact start time of the attack (marked with the red circles) and the end of the attack (marked with the green circles).

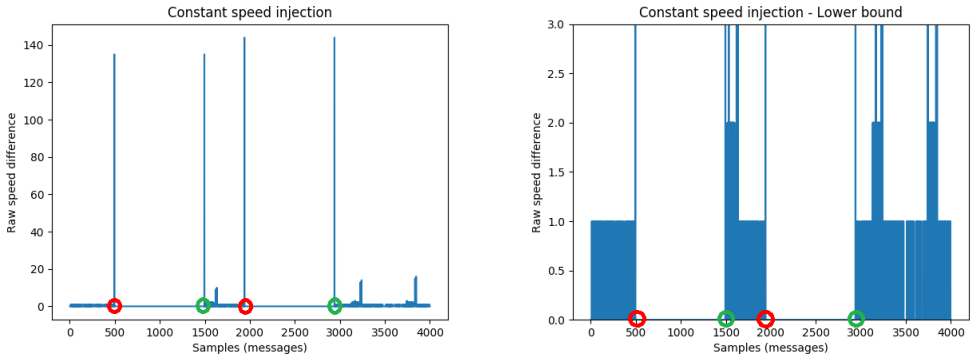

Figure 9.1: Constant speed message injection detection

9.2.1.2 Stealthy changing speed injection

In this test we again inject 1,000 messages that hold speed signal values. However, this time we gradually increase or decrease the values to the upper and lower speed boundary. Recall that the maximum and minimum values of a specific signal is defined in the signal database. In a similar way to the previous test, we can detect the start and the end of the attack. However, since the injected messages contain changing speed values with an increase or decrease of one, the shift will instead have a value of one as a constant shift during the attack. For example, when the injected speed value is 150 km/h, the next injected message will have the value 151 km/h, thus the shift is one. The previously described behaviour can be seen in Figure 9.2.

Note that injecting speed signals with start values that have shifts lower than the defined threshold in the IDS node will not be detected by this rule, instead it will be detected due to the frequency change of the messages that hold the speed signals.

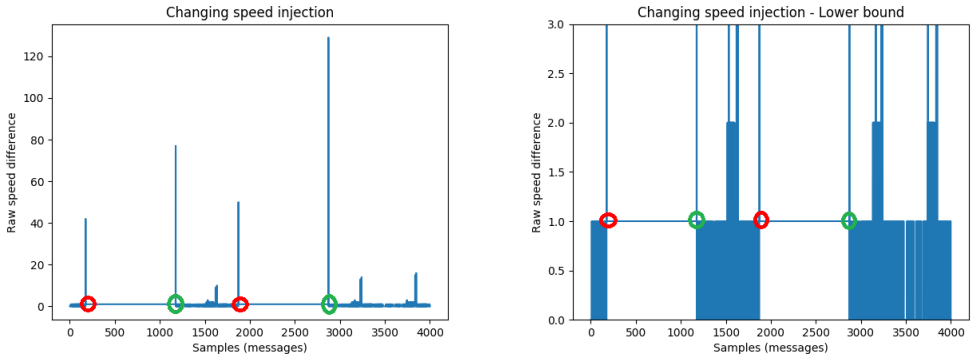

Figure 9.2: Changing speed message injection detection

9.2.2 Frequency change detection

We presented earlier how we performed message frequency analysis in order to extract the threshold for a specific message. This was then used to detect the frequency change between two consecutive messages while a message injection attack is being performed. The extracted rule had a safe check i.e. detecting at least two messages with deviated frequency, in order to eliminate any possible false positives.

In this section, we evaluate this approach using three different attacks. The attacks performed have different message count and frequencies. We also show the system's behaviour change while an aggressive message injection is being performed.

Also, note that any cyclic message can be used for test regardless of the message's functionality, cycle time or data. The detection works as long as the message is received by the same domain we place the IDS node in, and as long as the IDS node has a defined rule which monitors this message.

These tests were performed on the box car, more specifically, on the chassis CAN domain. The necessary changes to the setup were to add two virtual nodes where one performs the attacks and the other monitors the domain.

9.2.2.1 Injecting limited number of messages

In this test, we inject a limited number of messages. The injected messages have no predefined time distance in between and are transmitted on the bus as fast as they are allowed.

In Table 9.2, we can see the results of this test. The results show that with this rule, it is possible to detect n-2 of n injected malicious messages. This is due to the safe check of the *two consecutive message rule* that eliminates the false positive detection in attacks with longer periods.

Table 9.2: Evaluation results for injecting limited number of messages

Message count	Detection rate
1	No attack detected
2	No attack detected
3	1 malicious message detected (33.33%)
4	2 malicious messages detected (50%)
n	n-2 malicious messages detected $\frac{(n-2)*100}{n}\%$

9.2.2.2 Injection with identical cycle time

So far we have shown the number of messages that the rules were able to detect. In this test, we aim to study the relationship between the threshold and the system detection rate. The goal is to be able to find the threshold using the highest cycle time possible with the best detection rate.

To perform the attack we use the original cycle time as a threshold and periodically inject messages with different frequencies, starting from the defined cycle time that the original messages have.

Table 9.3: Evaluation results for message injection with identical cycle time

Original cycle time	Injection cycle time	Detection rate
15 (ms)	15 (ms)	Average detection (14.32%)
5 (ms)	5 (ms)	Average detection (96.67%)
2 (ms)	t $\leq$ 2 (ms)	Average detection (99.98%)

Table 9.3 shows the results of this experiment. We can see that injection attacks with a message having 15 ms as cycle time have a very poor detection rate. The 15 ms cycle time should not be considered a frequency change attack because the original ECU is still sending the messages with a frequency of 15 ms or higher, results in some messages having a frequency difference lower than the threshold. Figure 9.3 illustrates the disadvantage of this rule when used with messages with high cycle time. However, when message with lower cycle time were tested, this problem did not persist due to the smaller time gaps between each two consecutive messages which cause the injected messages to overlap with the original messages.

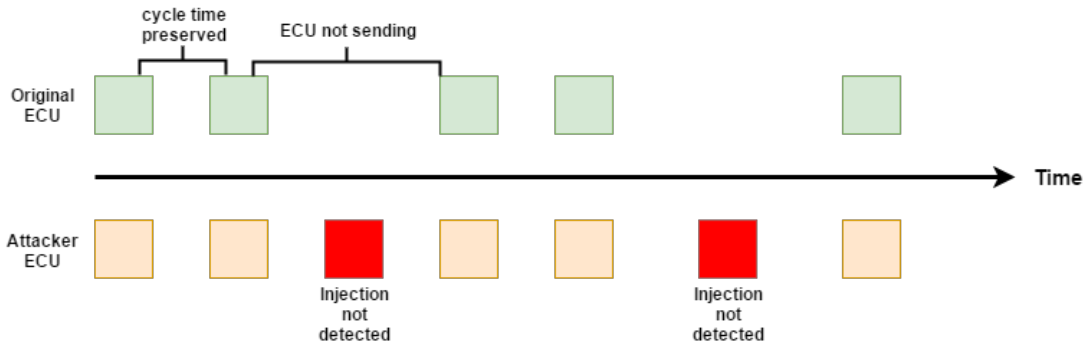

Figure 9.3: Same cycle time injection detection behaviour

The next part of this test was to gradually lower the cycle time of the injected messages and observe the detection rates. The results of this test are presented in Table 9.4. The injection of n messages with a cycle time which is lower than the original message cycle time was detected successfully, with 2 messages being a safe check to eliminate false positives as previously described. This shows how this type of behavior can be used by an attacker as a way of arbitrary message injection that will take effect for a short period. For a continuous effect, the attacker is required to inject messages with a higher frequency (lower cycle time) than the original in order for the malicious messages to be effective [15, 23].

Table 9.4: Evaluation results for message injection with less than identical cycle time

Original cycle time	Injection cycle time	Detection rate
15 (ms)	$t \leq 14$ (ms)	n-2 of n injected messages
5 (ms)	$t \leq 4$ (ms)	
2 (ms)	$t \leq 2$ (ms)	

9.2.2.3 Aggressive message injection

This test is very similar to the first test in this category. The main difference is to study the ability of the system in handling high number of messages. We test the ability to detect them by only filtering them according to their IDs and observing the frequency, regardless of their content.

Message injection with high priority messages can cause denial of service when injected aggressively (as fast as possible). To test this, we used messages with 0x00 ID in order to cause a denial of service attack in the chassis CAN domain on the box car.

Table 9.5, shows the conducted test. The results remained unchanged, if we compared the outcome to the test where we injected only a limited amount of messages in Section 9.2.2.1. However, due to the collision avoidance protocol we encountered message dropping on the box car when sending messages with lower priority. However, this case did not appear when transmitting messages with high priority.

Table 9.5: Evaluation results for aggressive message injection

Injected messages	Detection rate
1000	998 (99.8%)
10000	9998 (99.98%)
100000	99998 (99.998%)
n	n-2 $\frac{(n-2)*100}{n}\%$

To visualize the effect of such attack for a deeper understanding, we performed one more test. In this test, we log 4,000 messages while performing two attacks with 1,000 messages each. The injected messages had 1 millisecond cycle time while being injected. The original message that was logged had a predefined cycle time of 15 milliseconds.

Figure 9.4, visualizes the behaviour of the system while running this test. We can clearly see the attack phases and mark the start and end of such attack. We can also see that the frequency between two consecutive messages being dropped to 1 millisecond. This also shows the lack of randomness or *entropy* as demonstrated by [16].

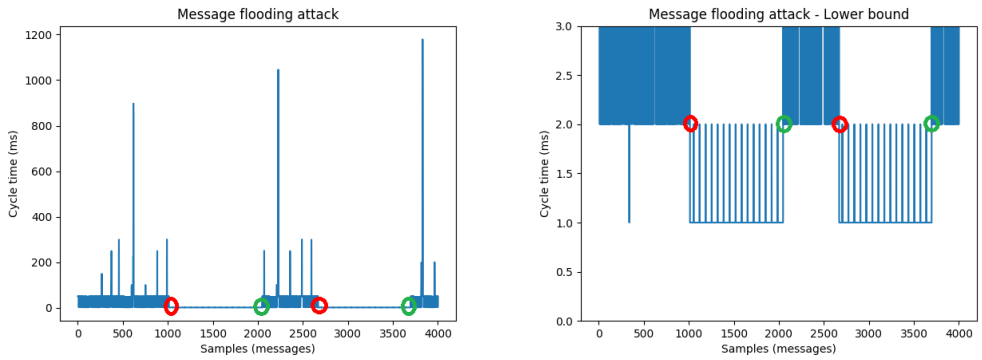

Figure 9.4: Flooding attack detection

10

Discussion

In this chapter we discuss the results of our work. For each point, we describe our findings and interpret them accordingly. The main questions to answer are, how are we able to overcome the challenges we have faced, and how our solutions and findings can contribute to future research.

Technical constraints and limitations

Previous work highlighted the fact that ECUs have limited capabilities when it comes to processing power and memory handling. The use of Vector software and hardware gave the experimental setup enough processing power which we usually don't not get in a real car. For example, when it comes to logging, we have not faced any issues simply because there was no storage issue in our setup. Sufficient storage for logging events is a serious limitation, even on gateway ECUs that have more resources in comparison to normal ones. Similarly, we did not face any processing power or memory limitations due to the configuration of our setup.

Software-wise, the implementation of the unauthorized message detection rule, which performs a search query in the white-list every time a message is received, turned out to be rather problematic for us since our setup was lacking a flexible programming language. While CAPL is a good language for implementing proof of concepts, the lack of search algorithms and return-oriented functions, raised the question whether there is an efficient search algorithm that can perform look-ups within lists that contain a very big amount of messages.

Contrary to our expectations, some of our attempts to simulate attacks failed due to the diversity of the communication technologies used in-vehicle network. For example, the speed value injection failed and messages got dropped when trying the attack on the box car, since it was designed to be transmitted over FlexRay. Therefore, we want to highlight the fact that investigating communication standards other than CAN is very important for a wider coverage to monitor the network.

Detection method

In the first stages of the work, we had the goal of extracting signatures from log files in order to use them as rules as a way to implement a signature-based IDS. During

the literature review phase we found no reference in previous work, to the best of our knowledge, that suggested a signature-based IDS. So we stopped following this approach. Instead, in our proposed solution we combine specification-based rules and anomaly-based algorithms in order to efficiently detect attacks on the CAN bus. Specification-based rules enable us to monitor every deviation of the CAN frame properties from the original specifications. It also helps us to monitor messages and determine whether a message is allowed in a sub-network . Furthermore, our anomaly-based detection algorithms try to detect shifts from the normal behaviour of the system, including changes in message frequency and plausibility.

We believe that considering only one approach for detection will not be efficient enough to cover different kind of attacks. For instance, the detection method for plausibility attacks can only detect the start and end of the attack and not the messages in between. However, the cycle time of the messages changes, because it is necessary for an attacker to inject messages with a lower cycle time in order to take a continuous effect and block the original messages [15]. Thus, the messages during the attack are detected by a frequency change algorithm instead.

Rule data selection

The frequency change detection was the most efficient approach to detect periodically injected cyclic messages. The implementation of this rule for a specific message only requires the knowledge of the cycle time for this message. While other detection mechanisms either require a full knowledge of the message specification or value shift analysis to extract appropriate rules.

We were able to extract a threshold for the speed plausibility detection by analyzing the speed value shifts between every two consecutive messages in a simulation setup. This threshold is based on a an artificial driver behaviour in our experiment. It is highly recommended to reassess the tests with a real driver and a real vehicle. Furthermore, another approach is to use the maximum acceleration capability of the car.

Detection accuracy and performance

Our results displayed a 100% detection rate for the specification-based rules. This was due to the nature of these rules, as they solely depend on a direct comparison. Even though our anomaly-based algorithms resulted in no false positives, the dynamic nature of these algorithms establish the possibility to miss certain frames.

In general, the combination of approaches we take for performing the detection, improves the overall detection rate. For instance, if an attacker injects a small number of malformed messages, they can not be detected by our basic frequency change algorithm, instead, they are detected by the specification-based rules.

While injecting a higher number of legitimate messages, the attack can not be

detected by the specification-based rules and because the attacker has to inject the messages with a cycle time that is less than the original cycle time [15]; they will be detected by the frequency change detection algorithm.

Placement and cost

In order to protect all CAN domains in a vehicle from an impending attack, we advise to have at least one IDS node in each domain, where more than one can be considered for redundancy in safety-critical domains. However, domains that have shared gateways have the possibility of monitoring these domains using gateways, thus, reducing the number of IDS nodes. This leads to a computational trade off as this can cause performance issues on the gateways due to the large number of messages processed, in addition to the regular routing task they have to perform.

On the other hand, the cost of implementing an extra IDS node in every sub-network for every newly produced vehicle can get very expensive.

Log Storage, post analysis and update

Modern car manufacturers are able to manage firmware updates and after market support remotely. With these technologies already implemented, we believe pushing rule updates or even major design updates for the IDS will not be an issue. Moreover, reports and logs can be periodically sent to the backbone, in similar way to sending diagnostic reports, for post analysis purposes. Despite the fact that we did not face any log storage challenges due to the configuration of our setup, we believe that the existence of the aforementioned technologies can help to overcome the log storage problem.

Future Work

The central idea of our work was to study, design and implement a prototype for an intrusion detection system for in-vehicle networks. As the developed prototype works only as a proof-of-concept, several questions remain unanswered at present. There is an abundant room for further progress in determining the following research questions:

- How can the driver be safely alerted of an intrusion on the network?
- To what degree can an attack be prevented?
- How to advance the research on other in-vehicle network bus standards, e.g. FlexRay or MOST?
- How to identify the most important signal IDs that could have a severe impact on the vehicle when misused?

- Can event based messages be detected and is it possible to detect all event-based signals? For example, activating the lights or honking the horn?

11

Conclusion

In this thesis, we design and implement a lightweight in-vehicle Intrusion Detection System (IDS) with a combination of specification-based rules and anomaly-based algorithms. We conclude that it is of highest importance to implement a defense mechanism, due to the attack vector that modern cars are prone to.

Using our IDS, different types of attacks can be detected, including message with properties deviating from the signal database and arbitrary message injection. The proposed solution successfully detected attacks on the CAN bus, that were conducted on a real in-vehicle network environment. The solution has some limitations, as it is still in the prototype phase, e.g. it is not able to detect attacks on networks that do not use the CAN standard. Additionally, for a broader coverage of the network and for a more efficient performance, our work suggests having a dedicated ECU that performs the monitoring in each domain, which on the other hand results in a very high production cost for the manufacturer. Therefore, the cost constrain remains an open issue that needs to be addressed in future works.

Despite the mentioned limitations and challenges, the prototype can be used as a starting point for further research. Meanwhile, manufacturers have to make the decisions and evaluate whether to invest money and adopt such solutions or postpone this for a few more years.

Bibliography

[1] G. Baribaud, H. J. Burckhart, W. Heubers, P. Van de Vyvre, D. Brahy, L. Jirdén, F. Perriolat, D. Swoboda, R. Barillère, A. Bland, et al. Recommendations for the use of fieldbuses at cern in the lhc era. 1996.

[2] R. Bosch et al. Can specification version 2.0, 1991.

[3] B. Caswell and J. Beale. *Snort 2.1 intrusion detection*. Syngress, 2004.

[4] K. Cho and K. G. Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 911–927, 2016.

[5] S. Corrigan. *Texas Instuments*, 2002.

[6] A. Deepa and V. Kavitha. A comprehensive survey on approaches to intrusion detection system. *Procedia Engineering*, 38:2063–2069, 2012.

[7] M. Di Natale, H. Zeng, P. Giusto, and A. Ghosal. *Understanding and using the controller area network communication protocol: theory and practice*. Springer Science & Business Media, 2012.

[8] I. et al. Can-bus: Ausarbeitung zum can bus. 2009.

[9] I. Foster and K. Koscher. Exploring controller area networks. *USENIX Usenix Magazine*, 40(6), 2015.

[10] T. Hoppe and J. Dittman. Sniffing/replay attacks on can buses: A simulated attack on the electric window lift classified using an adapted cert taxonomy. In *Proceedings of the 2nd workshop on embedded systems security (WESS)*, pages 1–6, 2007.

[11] D. Hristu-Varsakelis and W. S. Levine, editors. *Handbook of Networked and Embedded Control Systems*. Birkhäuser, 2005.

[12] M. J. Kang and J. Kang. A novel intrusion detection method using deep neural network for in-vehicle network security. In *IEEE 83rd Vehicular Technology Conference, VTC Spring 2016, Nanjing, China, May 15-18, 2016*, pages 1–5, 2016.

[13] U. E. Larson, D. K. Nilsson, and E. Jonsson. An approach to specification-based attack detection for in-vehicle networks. In *Intelligent Vehicles Symposium, 2008 IEEE*, pages 220–225. IEEE, 2008.

[14] U. Lee and M. Gerla. A survey of urban vehicular sensing platforms. *Computer Networks*, 54(4):527–544, 2010.

[15] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015, 2015.

[16] M. Müter and N. Asaj. Entropy-based anomaly detection for in-vehicle networks. In *IEEE Intelligent Vehicles Symposium (IV), 2011, Baden-Baden, Germany, June 5-9, 2011*, pages 1110–1115, 2011.

[17] M. Müter, A. Groll, and F. C. Freiling. A structured approach to anomaly detection for in-vehicle networks. In *Sixth International Conference on Information Assurance and Security, IAS 2010, Atlanta, GA, USA, August 23-25, 2010*, pages 92–98, 2010.

[18] D. K. Nilsson, P. H. Phung, and U. E. Larson. Vehicle ecu classification based on safety-security characteristics. In *Road Transport Information and Control-RTIC 2008 and ITS United Kingdom Members' Conference, IET*, pages 1–7. IET, 2008.

[19] K. Pazul. Controller area network (can) basics. *Microchip Technology Inc*, page 1, 1999.

[20] P. Richards. A can physical layer discussion. *Microchip Technology Inc*, 2002.

[21] F. Sabahi and A. Movaghar. Intrusion detection: A survey. In *Systems and Networks Communications, 2008. ICSNC'08. 3rd International Conference on*, pages 23–26. IEEE, 2008.

[22] M. Shavit, A. Gryc, and R. Miucic. Firmware update over the air (fota) for automotive industry. Technical report, SAE Technical Paper, 2007.

[23] H. M. Song, H. R. Kim, and H. K. Kim. Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network. In *2016 International Conference on Information Networking, ICOIN 2016, Kota Kinabalu, Malaysia, January 13-15, 2016*, pages 63–68, 2016.

[24] T. Stapko. *Practical embedded security: building secure resource-constrained systems*. Newnes, 2011.

[25] I. Studnia, V. Nicomette, E. Alata, Y. Deswarte, M. Kaâniche, and Y. Laarouchi. Survey on security threats and protection mechanisms in embedded automotive networks. In *43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop, DSN Workshops 2013, Budapest, Hungary, June 24-27, 2013*, pages 1–12, 2013.

[26] VECTOR. Dbc communication database for can. `https://vector.com/vi_candb_en.html`, 2017. [Online; accessed 02-may-2017].

[27] VECTOR. Ecu development & test with canoe. `https://vector.com/vi_canoe_en.html`, 2017. [Online; accessed 12-may-2017].

[28] V. Verendel, D. K. Nilsson, U. E. Larson, and E. Jonsson. An approach to using honeypots in in-vehicle networks. In *Vehicular Technology Conference, 2008. VTC 2008-Fall. IEEE 68th*, pages 1–5. IEEE, 2008.

[29] T. L. Willke, P. Tientrakool, and N. F. Maxemchuk. A survey of inter-vehicle communication protocols and their applications. *IEEE Communications Surveys and Tutorials*, 11(2):3–20, 2009.

A

Appendix 1

A.1 Attack simulation

This section demonstrates a practical way to implement attacks on the CAN bus which were feasible with our setup and permissions. The research of the different attack vectors was important to evaluate our detection rules of our prototype IDS.

We assume that the attacker has direct access to the nodes and is able to modify the code and behaviour of any node of his choice. This assumption is necessary because our focus is on the in-vehicle network itself and not how an attacker may gain access to it. Previous research [15, 9] has already explained that in a clear manner.

The main method of data communication on a CAN bus is performed by exchanging CAN data frames (messages). We begin by demonstrate how message sniffing can be done. This is possible due to the lack of any confidentiality measures in the CAN protocol. The second type of attack is injecting arbitrary messages. Message injection attacks can be performed in different ways on the CAN bus. This is due to the different properties that CAN data frames consists of. For example, an attacker can change the sending rate of messages, to force the receiver node to listen to the injected message instead of the original nodes messages. Finally, we show an addtional form of message injection which has the intention of causing a Denial of Service (DoS), thus affecting the availability of the system.

A.1.1 Message sniffing

Sniffing frames on the CAN bus does not require any modification of existing ECUs, due how the CAN protocol is specified. The communication on the CAN bus works through broadcasting the message on the bus and any ECU can listen to any message. However, it is important to note that this feature of CAN can let the attacker get a good overview of the system. The attacker could be able to reconstruct some parts the signal database [15] which is considered confidential for a great variety of car manufacturers.

In order to perform the message sniffing from any node within CANoe, it is enough to use the event based function "**on Message**" while specifying all types of messages. Listing 6 demonstrates a practical example.

```
on Message * {
  // do something with the messages or just log them
}
```

Listing 6: Sniffing messages with CANoe

A.1.2 Message injection

There are several details to consider about how the messages are sent on the bus according to the CAN specification:

- The CAN message are identified by a unique ID.
- The receiver node listens to the CAN bus and picks up the ID that is specified in its code.
- The message specifications, like data length and signal fields length are defined in the signal database.
- A collision avoidance protocol can put nodes which are sending messages with higher IDs on hold until there is no other node sending messages with lower IDs at the same time.

A.1.2.1 Injection of a single message

In this attack, the compromised ECU sends *one* message. The data in this message is in accordance with the specification in the signal database. However, it will contain arbitrary values, for example, sending a speed value of 300km/h when the actual value is 20km/h. While performing this attack, we observed that it does not make a difference when exactly this message was received, as it is received periodically by a node, which is listening for e.g. speed information. This is due to the fast recovery time when receiving the next message, this time will be at most the specified cycle time for this message in the signal database.

```
singleMessageInjection(message * msg){
    output(msg);
}
```

Listing 7: Injecting a single message

Nevertheless, for other types of nodes it is enough to send one non-cyclic (event-based) message to keep the changes persistent. For example, when sending one message with text information to the radio, this text will be displayed until the passenger interacts with the radio.

```
changeRadioDisplay(char[] malicious_text. char[] malicious_text2){
 SysSetVariableString(sysvar::ComfortBus::SetRadioChannelDisplay,malicious_text);
 SysSetVariableString(sysvar::ComfortBus::RadioInfoDisplay,malicious_text2);
}
```

Listing 8: Injecting a single message to the radio

Figure A.1: Radio information text change by injecting a none-cyclic message

A.1.2.2 Injection of messages with identical cycle time

For this attack, instead of sending a single message, we simulate the behaviour of a node by periodically sending the massages with a identical cycle time.

The cycle time is a time value that a node should preserve between two consecutive messages when transmitting on the CAN bus. Listing 9, highlights that the cycle time for the chosen messages *Gateway_2* is 2 ms, which is defined in the signal database.

```
Variable{
  msTimer injection_timer;
}

on start{
  setTimer(injection_timer,2);
}

on timer injection_timer{
  message  Gateway_2 msg;
  msg.CarSpeed=200;
  output(msg);
  setTimer(injection_timer,2);
}
```

Listing 9: Injection of messages with identical cycle time

This attack will have two possible outcomes:

1. The node will listen to the node of the compromised node.
2. The message sent by the compromised node will create a noticeable effect on the target node.

The first outcome will happen only if the messages of the attacking node are able to be received and processed by the target node before the messages sent by the original node. While the second outcome is more probable, especially with messages with a small cycle time. Note that achieving synchronization is a complex task because the cycle time is being sent in a millisecond rate.

A.1.2.3 Aggressive message injection

To make a node listen to our messages we have to send the CAN frames in a rate that is higher than its original rate i.e. *use a lower cycle time* [15, 23] , this is the fundamental idea of many papers we presented in the related work. When there is a fractional gap between the transferred messages, it will lead to a congestion on the CAN bus, causing the other nodes to not be able to submit any messages. This will be discussed in more detail in a subsequent section.

For this attack we consider a regular "for-loop" over a timer, to make the gaps between the messages as small as possible. We repeat that with a high value of of messages (10,000 in our case) to make the attack have a longer impact on the targeted receiver. The corresponding code is presented in Listing 10.

```
1 sendAggressive(message * msg){
2   int i;
3   for(i=0;i<10000;i++){
4     output(msg);
5   }
6 }
```

Listing 10: Aggressive message injection

A.1.3 Denial of service attacks

Denial of Service (DoS) attacks are another form of injection attacks.Due to its devastating effect on the network we chose to put it in a category of its own.

A.1.3.1 Malformed message injection

We consider a message as malformed, if the layout of the message does not follow the signal database specifications. Malformed messages rarely appear on the CAN bus, especially in real vehicles, due to the significant effort in code reviewing and testing. However, we want to show that it is possible and easy to send such messages and demonstrate the effect on the communication.

Lets take the message with the *ID 0x110* from our simulation setup. This message has the data length field value of 3. In Listing 11, we try to send a message with the same ID but with one extra byte. Even though it had no side effect on the system, this example shows that its still possible.

```
1 malformedGateway1(){
2   message 0x110 malformed_msg;
3   malformed_msg.dlc=4;
4   malformed_msg.byte(0)= 0x1;
5   malformed_msg.byte(1)= 0x3;
6   malformed_msg.byte(2)= 0x3;
7   malformed_msg.byte(3)= 0x7;
8   output(malformed_msg);
9 }
```

Listing 11: Message injection with bigger frame data length (DLC)

The other example is trying to overflow the reading mechanism of a signal in a message. Each signal has a defined bit length in the signal database. This value is used by the receiver node when trying to interpret the signals inside the message. In Listing 12, we define a value that has a significantly larger bit length than the defined 4 bit value in the signal database. The result was a crash of the simulation in CANoe, without any error messages.

```
1 malformedGateway1(){
2   message 0x110 malformed_msg;
3   malformed_msg.byte(0)= 0xFF; // When changed to 0xFF the system crashes
4   output(malformed_msg);
5 }
```

Listing 12: Message with oversized signal bit length

A.1.3.2 Flood attack

The reason why this attack works so flawlessly, is because of the specification of the collision avoidance mechanism of the CAN bus. It is stated that the message with the lowest ID has always the highest priority. The main idea of the attack is to not let any other message take part in the communication. We tested this by flooding the network with high priority messages, both in the comfort train and power train. Our approach was to inject 1000 messages into the communication, with an ID of 0x00.

Listing 13 shows a code-snippet on how to implement this attack in CANoe, where line 2 declares a message of the highest priority and this message will be send in a high frequency on the bus.

```
floodAttack(){
  message 0x00 flood_message; // highest priority message
  sendAgressive(flood_message);
}
```

Listing 13: Aggressive flood attack with the lowest ID message