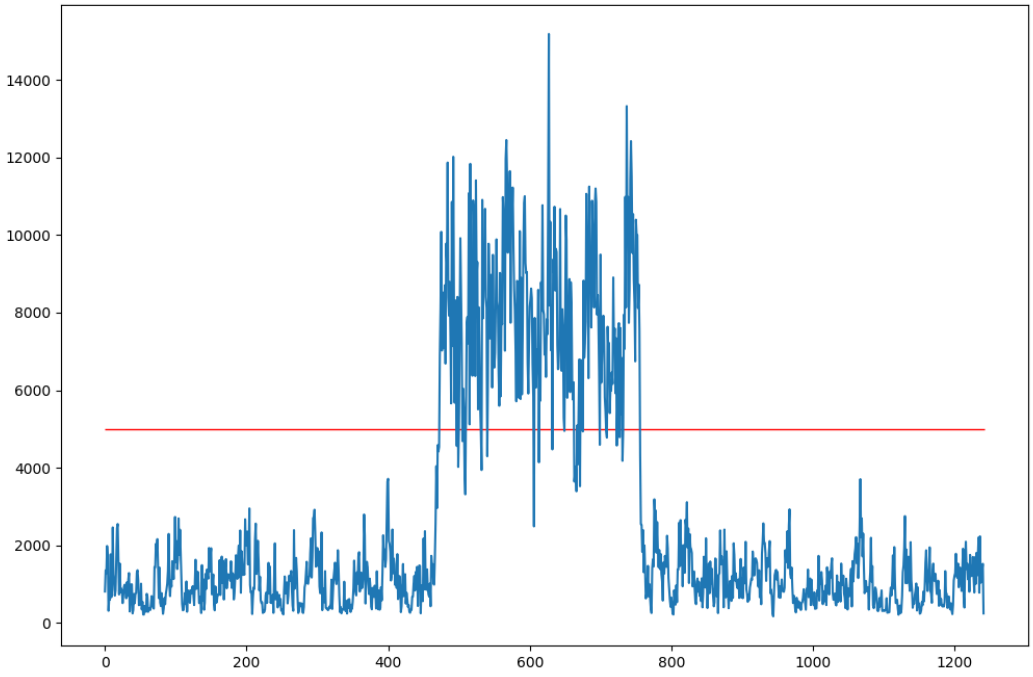
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



A Lightweight Intrusion Detection System for In-Vehicle Communication on CAN

Master's thesis in Computer Systems and Networks

SEBASTIAN KVARNSTRÖM

DAVID THIRINGER

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Master's thesis 2019

A Lightweight Intrusion Detection System for In-Vehicle Communication on CAN

SEBASTIAN KVARNSTRÖM
DAVID THIRINGER

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

A Lightweight Intrusion Detection System for In-Vehicle Communication on CAN
SEBASTIAN KVARNSTRÖM
DAVID THIRINGER

Supervisor: Magnus Almgren, Computer Science and Engineering
Advisor: Nasser Nowdehi, Volvo Car Corporation
Advisor: Wissam Aoudi, Computer Science and Engineering
Examiner: Tomas Olovsson, Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Graph of departure scores calculated by the intrusion detection algorithm, showing how a *fabrication* attack is detected *online* on a real vehicle.

Typeset in LaTeX
Gothenburg, Sweden 2019

A Lightweight Intrusion Detection System for In-Vehicle Communication on CAN
SEBASTIAN KVARNSTRÖM & DAVID THIRINGER
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In-vehicle networks (IVNs) are being equipped with an increasing number of electronic control units (ECUs) with each new generation of vehicles. This increase in ECUs contributes to a larger attack surface. Due to the lack of security mechanisms in the Controller Area Network (CAN) protocol, the most widely used communication bus for IVNs today, any ECU that is compromised can in turn compromise other parts of the network. As the attack surfaces of vehicles increase, so does the need for secure communications in the internal network to reduce the impact of attacks. One commonly proposed solution is the installation of an Intrusion Detection System (IDS) to detect attacks on the CAN bus.

In this thesis, we investigate if it is possible to implement a data-driven intrusion detection algorithm for IVNs on low end hardware. Furthermore, we investigate what optimizations need to be done to the IDS for it to be able to detect attacks in a realistic environment in real-time. Using the state-of-the-art detection algorithm CASAD, we test whether it is able to reliably detect online attacks in a realistic environment.

Having chosen four categories of attacks based on previous work within the field, the IDS was tested against them. The results of this thesis show that it is possible to detect at least three of the four attacks. The IDS was implemented on two different test benches where the first was used to verify our implementation, and the second to compare and evaluate the optimizations of the algorithm. The optimizations were done to meet the real-time requirements.

Keywords: In-vehicle network, Intrusion Detection System, Controller Area Network, Embedded Security

Acknowledgments

We extend a big thanks to our supervisor Magnus Almgren, and the advisors of our thesis, Nasser Nowdehi and Wissam Aoudi. Magnus, thank you for guiding us throughout the thesis and providing us with invaluable help regarding both the writing and the execution of the thesis. Nasser, thank you for helping us get started and providing us with the desired hardware. We also wish to thank you helping us test the algorithm on the in-vehicle network by helping us perform the attacks, and setting up the testing environment. Wissam, thank you for all of the help implementing and using CASAD, and training the algorithm for all of our experiments, and ensuring that our results were correct. We also wish to extend a thanks to Volvo Car Corporation, for allowing us to test the algorithm and hardware on their equipment. Lastly, we wish to thank Tomas Olovsson, for taking the time to be our examiner.

Sebastian Kvarnström & David Thiringer, Gothenburg, 2019

Contents

1

Introduction

As modern technology continues to advance, so does the complexity of the systems. This increase in complexity allows technology manufacturers to add a lot of interesting and innovative features. Indeed, in modern vehicle manufacturing, new functionality can come in many different forms. Smarter systems for controlling speed, self-driving vehicles and improved media systems are just a small subset of these features. These systems require small computing systems to work, which, in cars, are called Electronic Control Units (ECUs) [1]. These ECUs interface with sensors and actuators that control the various functions in the car, including the engine, doors, air-bags and brakes [2]. This means that parts that were traditionally controlled mechanically are becoming increasingly computerized. The ECUs, sensors and actuators communicate over multiple subnetworks in the vehicle, which are, in turn, connected via gateways. The different subnetworks use different communication technologies, often depending on their area of use. These include, but are not limited to, protocols such as CAN, CAN FD, LIN, MOST, FlexRay and Ethernet [3],[4].

While many different communication protocols are used, some even being specific to certain vehicle manufacturers, the majority of network communication between ECUs occur on a Controller Area Network (CAN) bus. Due to its extensive use in the automotive industry, CAN is an excellent target for security research, as any research that yields positive results on CAN may be widely implemented across numerous brands and types of vehicles. Cars, trucks and buses are all types of vehicles that implement the CAN protocol for ECU communication.

From a different perspective, CAN is also an interesting protocol for security research due to its inherent lack of security. CAN lacks much of what makes modern communication protocols secure, such as end-to-end encryption and authentication. Due to this, much of the existing research aims to fix these issues. The research typically follows one of two paths: security solutions that modify the protocol directly and solutions that aim to work with what already exists. Solutions that directly modify the protocol usually aim to fix the flaws that make the protocol insecure. However, such solutions come at a cost, as existing ECUs would

need to be modified to support the updated protocol. Even though solutions that do not modify the protocol may not be able to stop attacks directly, they do not come with this cost, and can still work to provide security.

The increased number of ECUs contribute to a lot of important functionality, but unfortunately opens up the vehicle to attacks by increasing the number of attack surfaces [1], [2], [5]. Not only are the ECUs connected to the internal networks, but they may also be connected to external communications, such as Bluetooth or mobile networks. If an attacker manages to take control of an ECU by exploiting its external communications and weaknesses, they might be able to perform attacks to the internal communications as well. In doing so, an attacker might even be able to perform a privilege escalation attack, and compromise other ECUs on the same network.

In recent years, the car manufacturing company Fiat Chrysler Automobiles was forced to recall 1.4 million cars when security researchers showed that they could remotely compromise the internal network of their cars [6]. The attackers exploited a weakness in an ECU that was open to a mobile network, and by compromising the internal CAN bus they were able to perform a large number of attacks, including steering the vehicle, turning the engine off, forcing the car to brake, and even disabling the brakes entirely.

With the security vulnerabilities that come with CAN and the increasing number of ECUs, it is clearly important that attacks on in-vehicle networks (IVN) are detected. Most individual ECUs have some form of input validation to protect the ECU from using invalid data, but malicious data can still be valid, and as a result be accepted by the ECUs. Unfortunately, a solution to these issues that requires heavy modification of the existing networks is not applicable to already existing vehicles, and will likely take a very long time to implement. Additionally, modifying a well-established network protocol requires rewriting documentation, and possibly years of planning. One such solution is the use of an intrusion detection system (IDS) spanning the network. This can add more security to the limited security of individual ECUs, without impacting the rest of the network.

1.1 Problem

Due to the aforementioned lack of authentication in CAN, several types of attacks are possible within the network. Choi et al. [7] identify attacks that they have divided into three categories: suspension, fabrication and masquerade. In their work, they have shown the ability to detect all three using the physical hardware characteristics of ECUs. Nowdehi et al. [8] identify one additional type of attack, which they call the conquest attack.

There are many methods for detecting intrusions on a CAN bus, which we have chosen to divide into two primary methods. The first, looking at the physical hardware characteristics, exploits unavoidable hardware imperfections that arise during the manufacturing process of the ECUs. The second, looking at the data, detects attacks by observing changes from the normal behaviour. Both of these methods have their own advantages and disadvantages. An IDS inspecting the physical characteristics may be able to detect the source of attacks, in addition to the occurrence of them [7], [9]. In contrast, data-driven IDS algorithms have not shown this ability of source detection. But instead, data-driven IDS have shown the ability to detect conquest attacks [8], unlike IDS based on physical characteristics.

Implementing an IDS on top of the CAN bus of a real vehicle is restricted by two important factors. The first is the real-time requirement of the network, as a CAN bus in a vehicle may be operating at near max capacity. This means that performing attack detection in real-time requires significant processing power, as messages must both be read and processed in time. If this is not done before new messages arrive, data is lost, and results may be inaccurate. The second limiting factor is the hardware that is available for each feature or application. Consequently, implementing an IDS should not impact or throttle the other ECUS.

With the real-time and lightweight requirements in mind, it is important that the IDS is verified live on a low-end computer similar to an ECU. Many IDS solutions have been proposed [2], [7], [9], [10], [11], [12], [13], and some of them have been tested on vehicles in real-time. However, many of them are only tested on the OBD-II port, which does not provide full access to the in-vehicle CAN bus, and even fewer have been tested on a low-end computer matching that of an ECU.

1.2 Goal

The goal of this thesis is to investigate whether the following research prototypes of IDS for IVNs can actually work in practice. To accomplish these goals, we need to cover and explore:

- Which algorithms have been suggested, and how they work
- The behaviour of normal CAN traffic
- Implementation on realistic hardware with limited resources (memory and CPU)
- Implementation of several realistic attacks

- Detection of the attacks in real time under aforementioned conditions

To achieve the aforementioned goals, we will (1) survey a number of research prototypes to see if they are suitable candidates to be evaluated in a realistic context. Through related literature, we will (2) aim to gain an understanding of the state of the art of attacks on vehicles. To evaluate the chosen algorithm, we will (3) build up several test benches to understand when the chosen algorithm works, and when it does not. Moreover, we will (4) suggest optimizations to the implementation to allow for real-time detection, based on several metrics, such as performance and accuracy of attack detection. To test the IDS and evaluate the results, we will first (5) implement the attacks in a realistic environment and, finally (6), evaluate the test bench systems in order to better understand how research prototypes can be brought to a real car, and what lessons the research community still need to address for future systems.

1.3 Scope

As mentioned earlier, the scope of this thesis is to choose an IDS for the CAN protocol and implement it. CAN is the de facto standard in the automotive industry for communication between ECUs, and as such is the most relevant protocol for our work. Other network protocols exist, such as FlexRay and Ethernet, but these are not as widely used as CAN is in the automotive industry, and as such we place greater importance on developing security solutions for CAN. Therefore, in order to evaluate the IDS, and its performance, we test it on a CAN bus of a real vehicle. The IVN gives us the opportunity to try the IDS on real data, with real-time requirements.

1.4 Outline

The report is structured as follows.

Chapter 1 presents the topic and goal of the thesis. Chapter 2 provides the required background knowledge about the CAN bus protocol and IDS. Following this, chapter 3 describes the attack surfaces of IVNs, as well as the attacker model and various attack scenarios. We summarize other works related to this thesis and to this field in chapter 4, and how our work differs from theirs.

Based on the first four introductory chapters, our approach and system design is covered in chapter 5. The implementation of this design is discussed in chapter 6. The experiments that we run on these implementations, as well as their results are covered in chapter 7.

We conclude our thesis by discussing the implications of our results in chapter 8. In the same chapter, we also discuss future work within in-vehicle intrusion detection, as well as the lessons we have learned and how this thesis relates to ethics and sustainability. Finally, we summarize the thesis and present our conclusions in chapter 9.

2

Background

To allow different parts of the vehicle to communicate with one another, there is a need for a communication network within the vehicle. The typical modern IVN is often comprised of a number of different communication buses, with different standards and technologies, that are often connected via different gateways [4]. The most widely used communication bus, and the focus of this thesis, is the CAN bus [14].

In this chapter, we describe how the CAN protocol works and what its security flaws are. Therefore, we make numerous claims about the CAN protocol throughout this chapter. The information presented in this chapter comes from the CAN version 2.0 specification [15], and from the overview given by Farsi, Ratcliff and Barbosa [16]. In addition to the CAN protocol, we also dedicate a section to describing intrusion detection systems, which details the differences between host- and network-based systems, and signature- and anomaly-based detection.

2.1 Controller Area Network (CAN)

In 1986, Bosch GmbH released a new communications protocol for controller networks. The aim of this protocol was to simplify the peer-to-peer communications that existed in vehicles at that time. This new standard was called the CAN protocol. The older communication models had employed dedicated communication lines between ECUs. With each new generation of vehicles, and the rapid computerization of previously mechanical parts, the existing solutions were not feasible anymore. With the release of the CAN protocol, car manufacturers were able to employ a single bus for communication between multiple control units, greatly reducing the number of physical wires required. Additionally, with CAN using a single twisted wire pair, the wiring that it does use is inexpensive.

CAN was designed to be able to quickly send small chunks of data between several different control units. Bosch had observed that most of the data that was being transmitted between control units contained very small payloads, and

designed CAN accordingly. This is reflected in the maximum payload size of a CAN frame, which is only eight bytes. In order to increase the speed further, they implemented a fast arbitration process that does not require a token.

There are several standards for CAN, but the main one is ISO 11898-1, which defines the data link layer and physical signalling layer of the protocol. The most commonly used CAN standard is the high speed standard, defined in ISO 11898-2, but there are other standards that may be used, including ISO 11898-3 defining low-speed (fault tolerant) CAN, and ISO 11898-4, defining time-triggered CAN. We will only cover the high speed standard, since it is the most widely used standard within vehicles.

The high-speed CAN standard runs at anywhere from 40 Kbit/s to 1 Mbit/s, but 500 Kbit/s is normal in vehicles. Assuming each CAN frame transmitted on the bus contains a full payload, and the bus is under 67% load (two reasonable assumptions for modern IVNs), then there are approximately 3000 messages being sent every second, or 24 000 bytes per second.

In 1991, CAN version 2.0, was released. This version added the optional use of extended identifiers, which increase the size of the identifier from 11 bits to 29 bits. More recently, in 2012, Bosch released a different CAN standard that supports a flexible data-rate, called CAN FD [17]. CAN FD allows for longer payloads and the ability to switch to a different bit-rate. Devices built to use CAN FD are also backwards compatible, and may be connected to work on a CAN 2.0 network. Due to its recent release, CAN FD has not seen a market-wide use yet. As such, it is not covered in this thesis.

2.1.1 Physical Layer

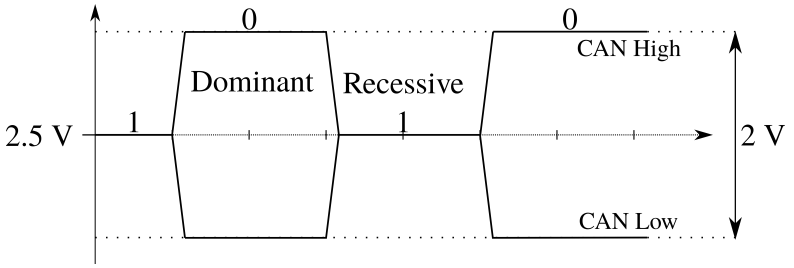
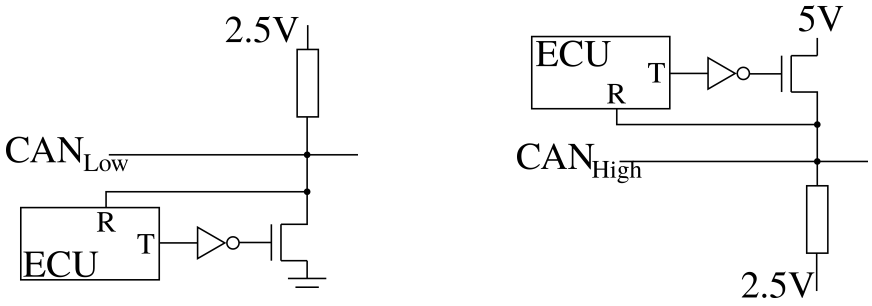


Figure 2.1: The voltage levels of CAN high and CAN low over time when ”1010” is being sent on a CAN bus, illustrating the dominant and recessive bits.

On the physical layer, a CAN bus uses differential signalling with a single twisted pair wire, with one of the wires referred to as CAN high and the other CAN low. Differential signalling and twisted pair wiring is used to protect the

communication against noise. In order to convert the signals on these wires into logical 0's and 1's that an ECU can read as a stream of bits, the difference in voltage is measured. Typically, a voltage differential between the wires of more than 2 V is read as a logical 0, and is called a *dominant* state, while a low voltage difference, below 2 V is read as a logical 1 and is called a *recessive* state. Both the recessive and the dominant states can be seen in Figure 2.1. The figure shows how the CAN low and CAN high always mirror one another, which is one of the defining characteristics of differential signaling.



(a) ECU connected to bus on CAN low.

(b) ECU connected to bus on CAN high.

Figure 2.2: Circuit diagram showing the pull-up and pull-down properties of CAN.

Since the CAN bus is a shared medium, meaning many ECUs share a single bus for communication, a sending ECU must be granted exclusive access to it. This is done using an arbitration process, which is the process of deciding which one of several competing messages to transmit first, based on their priorities. Arbitration on CAN is possible due to two important physical features. First, if a recessive bit and a dominant bit are both being broadcast by two different ECUs, the bus will be in a dominant state. This is shown in the figure 2.2a, which is a circuit diagram visualizing how ECUs are connected to CAN low. A pull-up resistor keeps the CAN low bus at 2.5 V, unless an ECU wants to send a dominant bit, which would ground the CAN low wire, setting the voltage to 0. Similarly, as shown in figure 2.2b, CAN high uses pull-down resistors to pull the voltage to 2.5 V when in recessive state.

Second, during transmission, an ECU is also able to read the state of the CAN bus. This allows the ECU to immediately see if it has been eliminated during the arbitration process. Looking back to figure 2.2a, we can also see that if the ECU reads 0 V from the bus and is not currently sending a dominant bit, it will know that some other ECU is sending.

2.1.2 Network Layer

In CAN, there are two different types of frames: the standard frame with an 11-bit identifier and an extended frame with a 29-bit identifier. The identifier sets the priority of the message, with a lower ID having a higher priority. The arbitration process is done when two or more ECUs are sending at the same time. During arbitration, the ID bits are compared and the one with the lowest ID is allowed to transmit, while the higher ID is not.

Typically, the ID is not used to identify the CAN node that is sending the data, but rather the type of data being sent. This means that a single CAN node may send CAN frames with several different IDs. A particular ID is typically not meant to be sent by multiple ECUs. Since the ECUs in the network do not have any IDs associated with them, messages on the CAN bus are not sent directly to the other ECUs. Instead, any ECU can snoop the bus, and is able to read all of the data that is transmitted over it. While no CAN node is sending, the bus is in a constant recessive state. In order to start transmitting data, a CAN node must start by sending a dominant bit on the bus.

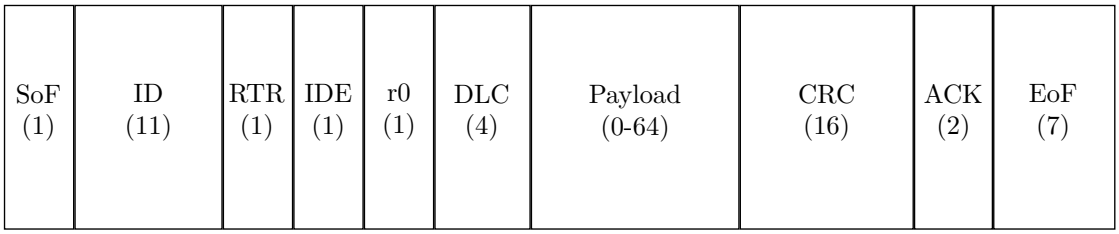


Figure 2.3: The different parts of a CAN frame. The number under the acronyms shows how many bits are used by that part of the frame.

The CAN standard frame can be 44 to 108 bits long, allowing for a maximum size payload of 64 bits (8 bytes). Figure 2.3 shows the structure of a CAN frame, and we describe each of its fields in this section, the first of which is the SoF (start of frame) bit. This, always dominant, bit lets the other ECUs know that a message is being transmitted, and that the bus is busy. Immediately after sending a SoF bit, the sending ECU starts sending its identifier.

The RTR (remote transmission request) bit allows a connected ECU to request data from another ECU on the bus. When the RTR bit is set, the identification bits now represent the identifier of the message that is being requested. By snooping the CAN bus, the ECU responsible for delivering messages with this ID is requested to send data. This feature allows ECUs to ask for information when it is needed, so that the bus is not filled with unwanted data. This is also the only way in the CAN protocol to directly request data from a specific ECU, or at least a specific

ID.

The IDE (identifier extension) bit is used to differentiate between the extended and standard CAN frame formats. A dominant bit in this field signifies the standard frame format, while a recessive signifies the extended frame format. The next bit is a reserved bit, which is always dominant. This bit is reserved for a later version of the protocol, allowing for backwards compatibility, by programming ECUs running the older version to ignore messages with recessive reserved bits.

Since the payload size can vary in the CAN frame, the transmitter first needs to specify how much data it is sending. This is done in the DLC (data length code) bits. The DLC contains four bits that specify the number of bytes of data that are being transmitted. While only three bits are required to describe the payload length (one to eight), an additional bit is required, as the payload length can also be zero. Although the DLC can describe payloads of lengths above eight (up to fifteen), this is not supported by the CAN 2.0 standard.

Following the payload is a checksum, sent in the CRC (cyclic redundancy check) bits. The checksum is used by receivers to verify the integrity of the data, and is calculated using a CRC function. A CRC is often used to detect accidental, transient, errors and involves using a mathematical algorithm to generate a checksum from the payload. This fixed-size checksum is typically smaller than the original input, and adds an extra layer of certainty that the data being sent has not been accidentally altered in transit. The CRC bits are used by the receivers, who compare it to their own CRC calculation of the payload data. If the CRC is the same as what the sender calculated, the receiving ECUs will output a dominant bit as the first ACK (acknowledge) bit. This means that no changes to the data have occurred. If the sending ECU reads a recessive bit, it will resend the message. The last bit of the CRC gives the receiving ECUs an additional time slot to fully compute and compare the CRC. Similarly, the last bit in the ACK field is used to let the sending ECU have the time to process whether it needs to resend the payload.

The last bits are the end of frame (EoF) bits, which signal the receivers that the message is fully transmitted, and that a new message may be transmitted on the bus. The EoF always consists of seven recessive bits, which forces the bus back to a recessive state. This utilizes a technique for error handling called *bit stuffing*, which is an important feature of the CAN protocol. Whenever six, or more, consecutive bits with the same state are transmitted, all the readers are programmed to reset and return to a recessive state. Since a series of the same consecutive bits may occur when ECUs are functioning properly, for example in the identifier or payload, a flipped bit is inserted after every five consecutive bits with the same state. This flipped bit is called a *stuff bit*, and is read by all the listening ECUs — as an indication to not reset — but is stripped from the message

once it has been received, so that the original message remains the same.

2.1.3 Security Considerations

When it comes to security, the CAN protocol leaves a lot to be desired. When it was originally designed, it was not done so with security in mind. From the perspective of the C.I.A (*Confidentiality*, *Integrity* and *Availability*) triad, the CAN protocol significantly lacks in all three.

Confidentiality is in many ways non-existent, as messages are not encrypted before being transmitted on the bus. This means that any ECU connected to the bus can read what any other ECU sends, which is an intended feature of the protocol. However, one could argue that some confidentiality does exist, since only ECUs connected to the bus can actually read messages. In that sense, more confidentiality is added when ECUs are split into different networks that are only connected via gateways.

Integrity exists for the data in the form of cyclic redundancy checks (CRCs), but is only used for error detection to detect bit errors and not for intentional data modification. In addition to not being able to detect intentional data modification, there is also a complete lack of message authentication. In the CAN frame, there are no fields that allow for sender identification beyond the ID of the CAN frame itself. As was mentioned earlier, a sender is often able to send messages with many different IDs, and there is nothing stopping any other ECU from sending a message with the very same ID. This allows for attacks where compromised ECUs can send messages with a low ID that does not belong to the compromised ECU.

Availability may go hand in hand with CAN at a first glance, since the protocol provides a low latency communication network, with different levels of priority. However, since there is no restriction in the priority an ECU might have, any ECU in the network can flood the bus with high priority messages at any given time. With the high load of in-vehicle CAN networks, this may seriously destabilize the communication.

2.2 Intrusion Detection Systems (IDS)

By using a system that is able to detect the presence of attacks, an IDS, users or administrators can be alerted of them, so that preventive action can be taken. In order to detect attacks to a system, an IDS will monitor the system and analyze aspects of it to determine whether an attack is occurring or not. This is done either by looking for specific patterns in the data — typically network traffic — or by analyzing the overall behaviour of the system. On a network, this typically

means monitoring the communication packets being transmitted, and on a host, the incoming and outgoing traffic, as well as the processes running on it.

There are many reasons why an organization would use an IDS. An IDS provides organizations with useful information regarding the system, providing an overview of what might need to be changed, and what needs to be updated. Furthermore, administrators that are alerted to attacks are able to analyze the attacks. This gives them useful information that they can use to further secure their systems, by identifying vulnerable systems, as well as potential targets.

IDSs are typically split into different categories, either depending on their target, or how they function. An IDS that is placed locally on a node with the aim of detecting attacks on the node is called a host-based IDS (HIDS). In contrast, an IDS that monitors network traffic between multiple nodes is called a network-based IDS (NIDS). While more classifications of IDS exist, such as wireless IDS and systems for network behaviour analysis, these fall very close to NIDS.

Since the difference between HIDS and NIDS is clear, it is often more common to differentiate different IDS depending on their functionality. IDS functionality is often divided into two categories: signature-based or anomaly-based [18], both of which we describe in section 2.2.2. Neither anomaly-based nor signature-based IDS are limited to HIDS or NIDS. Instead, both of them may be implemented as either.

2.2.1 Locations for Detection

When installing an IDS, an important consideration is the location of the IDS. For instance, if the goal is only to protect a single target, a HIDS is often a more suitable option. However, an administrator for an organization with a large network is more likely to install a NIDS to protect the entire network. Important systems may also be protected by both HIDS in addition to a NIDS, providing the system with multiple layers of security.

Since the aim of a NIDS is to protect a network and listen to its traffic, it is often placed in a strategic location, where it can monitor as much as possible, allowing the NIDS to monitor the data being sent from and to nodes in the network. In case a network is divided into multiple subnetworks, multiple NIDS are often used, with one for each subnetwork.

The traffic that the IDS monitors can vary depending on the IDS and its configuration. Typically, an IDS will look at the packets to determine which protocol is being used, and then use specific rules for each protocol. Much like modern firewalls, some IDS also have the ability to keep track of protocol states, such as whether there is an actual TCP connection established when TCP data is being

sent. However, an IDS may be much more powerful than this, and may even be able to determine intrusions based on application level data.

There are many different NIDS. Two examples of NIDS, both of them open source, are Zeek (formerly known as Bro) [19] and Snort [20]. Snort was developed by Martin Roesch, and is currently being developed by Cisco, and works both as an IDS and as an intrusion prevention system. Meanwhile, Zeek is currently being developed by Vern Paxson, and is described as a network analysis framework, rather than a traditional IDS.

In contrast to a NIDS, a HIDS only looks at the traffic and events of a single host. A HIDS may be run on high value hosts, or simply on all computers in the network, depending on the costs and requirements. Like NIDS, there are also a number of different HIDS, all with different functionalities. Two examples of these are OSSEC [21], created by Daniel Cid, and Open Source Tripwire [22], based on code from Tripwire, Inc. Open Source Tripwire is a HIDS that monitors the file system, and detects unexpected changes to the file system by comparing the file system with a known baseline. Though not necessarily better, OSSEC monitors logs, processes and performs ”rootchecks”, in addition to monitoring the file system.

2.2.2 Methods for Detection

In signature-based detection, data is analyzed in order to find specific patterns. This can include anything from looking for a particular series of bytes, to finding certain application level messages. A signature-based IDS needs to have the signature of malicious messages already stored in order to detect them. This can become a very expensive task, and new attacks with new signatures will go undetected until they are discovered and the IDS is updated accordingly with the signature of the attack. Therefore, while such an IDS may not be complex in itself, the set of rules it will use may grow, making the IDS more complex [18].

Anomaly-based detection involves training the IDS to understand what is normal behavior for the system, and then to trigger an alarm when it detects behavior deviating from the norm. Anomaly-based detection may be mathematically complex and harder to implement, but does not require maintaining a list of potential attacks, and has a higher probability of detecting undocumented attacks. Most of the cost lies in the training of these IDS, and they must be trained on an untainted target, or the malicious behavior could be seen as normal behavior [18].

Both categories of IDS have their own pros and cons, with signature-based IDS favoring simplicity, and can immediately be launched on a system out of the box. In contrast, anomaly-based IDS require a training period on the system before

being launched, meaning that there is some downtime where the IDS can not be used. Unfortunately, neither type of IDS is flawless, but anomaly-based IDS has the advantage of being sustainable, since it may continue to function properly without updates, in addition to potentially being able to detect zero-day exploits.

Though an anomaly-based IDS may be able to detect attacks a signature-based IDS cannot, it may also incorrectly classify some behavior as anomalous and trigger false positives. This may be even more prevalent if the IDS is poorly trained. Another risk with an anomaly-based IDS is that it may be trained on an already compromised system, and be unable to identify the attack at all.

3

Security of In-Vehicle Networks

With an understanding of the vulnerabilities of the CAN protocol from the previous chapter, it is important to understand how attackers can abuse them. Therefore, this chapter first looks at the attack surface of a vehicle, and how an attacker may gain access to the CAN bus. Following this, we present the attacks to which the CAN protocol is vulnerable.

3.1 Attack Surface

Much of the research that has been done in the field of vehicle security has focused on the growing attack surface of vehicles, and the attacks that can be performed on them. Checkoway et al. [23] have done an extensive analysis of different attack surfaces, wherein they identify four different vulnerability classes:

- Direct physical
- Indirect physical
- Short-range wireless
- Long-range wireless

Direct physical attacks involve plugging some malicious hardware directly into the on-board diagnostics (OBD-II) port, which is a standardized port for on-board diagnostics that is directly connected to a CAN bus. In addition to a CAN bus, the OBD-II port usually grants access to several other communication buses. The OBD-II port provides access to a limited part of the CAN traffic, which the manufacturers have intentionally made available for diagnostic purposes. This port is required by law, in both the EU and the US, to exist in modern vehicles. As a result, these attacks are not unique to specific brands of vehicles [24],[25].

Indirect physical vulnerabilities are the second class of vulnerabilities identified. The channels in this class still require physical access, but are those that do not

plug directly into the communication buses. Examples of these include the use of J2534 hardware devices, colloquially known as Pass-Thru, which are hardware devices that allow computers to communicate with ECUs [26]. In this case, the computers may contain malware that would make use of this connection. This class also includes CD's or DVD's containing malware, which exploit vulnerabilities in the media device.

The last two classes are both wireless, and are the *short-range* and *long-range wireless* classes. The short-range wireless class includes Bluetooth, which is a feature many modern cars have, allowing drivers to connect their devices to the car wirelessly for a multitude of reasons, such as audio playback or diagnostics. The long-range wireless class includes cellular, which is, again, a feature many modern cars have access to.

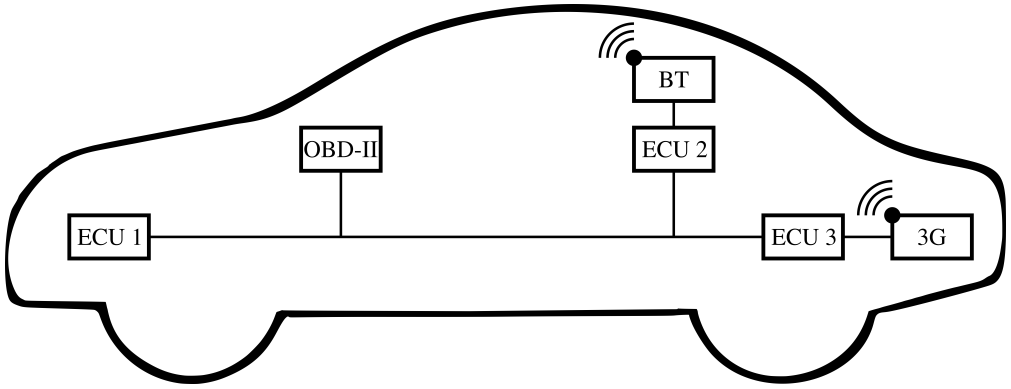


Figure 3.1: A simplified IVN. ECUs 1, 2 and 3 are connected through a CAN bus within the vehicle. ECU 1 is not exposed to anything but the CAN bus. ECUs 2 and 3 are exposed to the outside via Bluetooth and 3G communication, and are external attack surfaces.

These four vulnerability classes are depicted in figure 3.1, where ECU 1 can only be reached internally through the CAN bus. In this simplified figure we can see that the OBD-II port is directly connected to the CAN bus, and can be used to either launch a direct or indirect physical attack against the network. ECU 2 is vulnerable to short-range wireless attacks, since it has a Bluetooth component that might contain security flaws. Lastly, we see that ECU 3 is communicating both on the CAN bus and via a 3G cellular component, exposing it to long-range wireless attacks. Due to the insecurities of the CAN bus, even ECU 1 may be vulnerable to attacks, since the other ECUs are exposed externally.

As previously mentioned, there is nothing stopping a compromised ECU from sending a message with an ID that does not belong to it. Determining which ECU is sending a message is not easy, as there is no built-in CAN functionality to identify

the sender, and not enough space in a CAN frame to add custom functionality for this without also adding overhead in the traffic [9], [27]. This means that replay attacks — attacks where previous transmissions are resent by a different sender acting as the original — are possible.

3.2 Threat Model

We assume that an attacker gains access to the network by compromising at least one of the ECUs, and is then able to either re-program it or, at the very least, stop its transmissions. We assume that this is done by the attacker in order to perform some malicious activity. However, we do not concern ourselves with how this access is gained, nor what this malicious activity actually does to the car.

Borrowing notation from Cho and Shin [7], we call an attacker that is able to re-program an ECU in order to send arbitrary messages on the CAN bus a *strong attacker*, and an attacker only able to suspend an ECU's transmissions a *weak attacker*. In a similar vein, we say that an attacker that is a strong attacker of an ECU A has *strong access* to ECU A, and a weak attacker of an ECU B has *weak access* to ECU B.

In this thesis, we make use of an anomaly-based IDS. As mentioned previously, an anomaly-based IDS is trained on the normal behavior. When training the IDS, we assume that the environment that it is trained on is in no way compromised. Moreover, during detection, an attacker may have access to multiple ECUs at once, but we assume that an attacker is not able to compromise the IDS. The access the attacker has to each ECU may vary, having weak access to some of them, and strong access to others. During our verification of the IDS, we use the fewest number of compromised ECUs possible to launch the attacks, since a real adversary would not have the luxury of compromising all the ECUs. By assuming that the attacker compromises as few ECUs as is possible, the footprints of the attacks are reduced, making them potentially harder to detect.

3.3 Attack Scenarios

Connecting back to the goals of this thesis, we will now analyze what attacks CAN is vulnerable to, so that we can later implement realistic attacks that we test the IDS against. In this section, we will describe the attacks that we will later implement for the evaluation of the IDS. Our implementations of the chosen attacks will be described in chapter 7

As discussed in the previous section, we assume an attacker is able to, in some

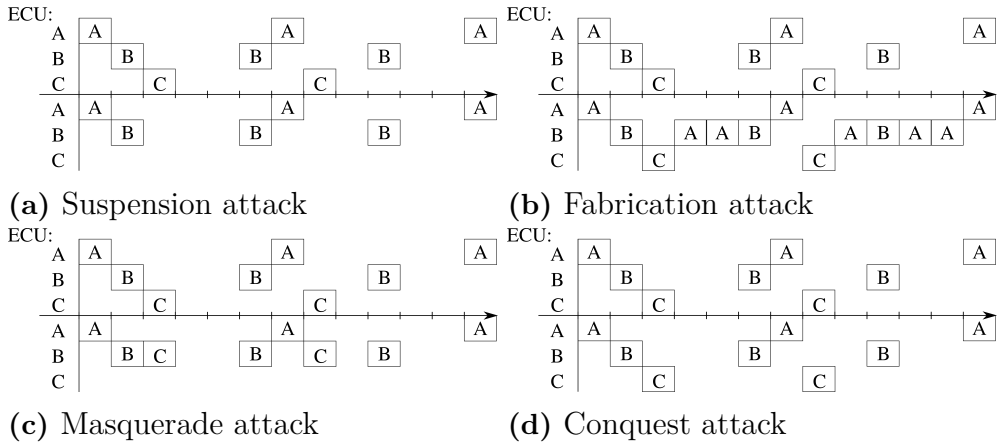


Figure 3.2: Illustrations of each attack. The top part of each figure represents a system during normal use, while the bottom part represents the same system under attack. The values in each box are the IDs of the messages. For simplicity, all IDs in this example also indicate which ECU is the real sender of the ID.

way, compromise one or more ECUs as either a strong or weak attacker. In their paper, Cho and Shin [7] discuss three types of attacks: *suspension*, *fabrication* and *masquerade*. We also include an attack, called *conquest*, described by Nowdehi et al. [8] in this thesis. This classification is done since it covers the possible attack vectors that modify the communication of the bus.

Suspension attacks involve suspending all transmissions of an ECU. This only requires that the attacker has weak access to an ECU. If the targeted ECU relies on communication, this attack can disable the ECU. As the functionality of some other ECUs may depend on data acquired from the compromised ECU, this attack can also affect more ECUs than just one. Figure 3.2a shows the attack in action, where the ECU C is under attack. As can be seen, no messages from ECU C are sent while it is being suspended.

Fabrication attacks are those where an attacker with strong access to an ECU aims to confuse one or several other ECUs by fabricating messages of particular IDs, with different values. If the listening ECU performs any actions based on these values, the attacker may cause the listening ECU to perform a faulty action. Such a faulty action may be due to the wrong value having been read at the moment of an action being performed. Figure 3.2b shows the attack in action. In this case, if ECU C uses values from A, it would receive multiple messages to use, some from ECU A and some from ECU B, without any idea of which ones actually belong to ECU A. In addition to confusing other ECUs, fabrication attacks can be used as denial of service attacks by flooding the network.

Masquerade attacks can be seen as a combination of suspension and fabrication attacks. If an attacker only has weak access to the ECU it wishes to exploit, then the attacker may suspend its transmissions. With strong access to another ECU, it may then fabricate messages of the suspended ECU, effectively masquerading as the weakly compromised ECU. Figure 3.2c shows the attack in action. From the perspective of ECU A, messages B and C arrive at the same time as without the attack, so it is completely unaware of any changes.

Conquest attacks can be performed when the attacker has strong access to the ECU it wishes to exploit. It may re-program this ECU to change its behaviour, making this the stealthiest of the four attack categories. If an attacker wishes to do this without easily getting detected, they may maintain the periodicity of the original message, and only change the content of the message. An example of this can be seen in figure 3.2d, which shows how the communication does not change during an attack. Without looking at the actual payload, the attack would not be possible to detect.

4

Related Work

Recent research has revealed that many vehicles suffer greatly when it comes to security. Security of cars is something many people take for granted, and something that is important, as the consequences of an attack can result in fatal accidents. This makes security research in this field especially important. Much of the work related to the security of IVN have been to investigate attacks and to analyze attack surfaces of vehicles. More recently, several in-vehicle IDS have been proposed, in order to increase the security of vehicles, without the need to redesign the parts of the vehicle that are vulnerable.

To our knowledge, no IDS has yet been proposed that is both lightweight enough to be implemented in a car, while being able to detect all of the attacks we have previously discussed. By analyzing the work that has been done so far, we gain the knowledge to choose an appropriate IDS. We also summarize some of the field of IVN security, especially concerning IDS for IVNs.

4.1 Vulnerabilities of Vehicles

In 2014, Miller and Valasek [6] showed many different attacks on a line of Jeep cars. As a result of their work, Fiat chrysler was forced to recall 1.4 million vulnerable vehicles. In their work, they exploited the cellular attack surface of the vehicle, and were able to remotely compromise vehicles across the entire USA. The attacks they performed involved sending different signals, many across the CAN bus, from a compromised ECU. The attacks ranged from harmless, like activating the turn signal, to extremely dangerous, like disabling the brakes. Although the latter could only be done by triggering the car to enter a diagnostics mode, which required a speed of 5-10 Km/h, they did not rule out the possibility of tricking the ECU into thinking the speed would be different from the actual speed. In addition to disabling the brakes, they were also able to remotely steer the vehicle, as well as force the car to brake. Other attacks they performed were changing the radio channel, turning up the volume, and disabling the volume control.

More recently, in 2018, Cho and Shin [28] found that a vehicle may be compromised while its ignition is turned off. By exploiting the wake-up functionality that some ECUs have, they show how they can remotely activate ECUs in the vehicle. Doing this, they show how an attacker would be able to drain the battery of a vehicle or lock the driver out of the car using a so-called *Denial-of-Body-control* attack. The wake-up functionality of ECUs is implemented in order to provide standby functions and to increase energy efficiency. However, this work shows that ECUs with these features can be vulnerable, even when they are offline.

Checkoway et al. [23] show more insecurities in vehicles, and have performed analyses of their attack surfaces, in which they note that there is a number of different points of attack not limited only to physical, but also remote. In their article, they mention being able to hijack the car's telematics unit by calling it via mobile phone, making it an attack that may be executed from anywhere in the world. Carsten et al. [5], as well as Song et al. [10], were also able to confirm, in more recent articles, that vehicles do indeed lack a lot of security and point out some of the same flaws. In their work, Carsten et al. also describe numerous potential attacks on the CAN network, including data stealing, control override, vehicle degradation, data falsification and external sensor attacks. Attacks such as data stealing do not directly interact with the CAN bus, and so can not be detected by a network-based IDS.

4.2 Security of Vehicles

There have been proposals to reinforce the CAN protocol itself. One of these is shown by Woo et al. [27] who propose a security protocol that uses the extended CAN frame to create a message authentication code that can verify that the sender is the correct sender. This proposal would add some additional security to IVNs, since it would eliminate some of the possible attacks. Not all attacks are stopped, however, since the proposed IDS only stops ECUs from acting as other ECUs, and does not stop suspension or flooding attacks.

To counteract the insecurities of the CAN bus, most of the research that has been done has been focused on developing intrusion detection systems that can detect anomalies in the network. There have been a lot of different articles published regarding this topic, but we will give an overview of the different approaches and the ones we find to be the most promising. Most IDS that we have studied base their work on the same assumptions, which is that in-vehicle CAN messages are periodical and that a lightweight system is preferred since it would be easier to embed in a real vehicle.

An intuitive way to create an IDS for vehicles is to only look at when the

messages are sent and how many messages are being sent during a specified time interval. By using this approach, one can build a lightweight IDS, since there is not a lot of data that needs to be stored. An example of this is a simple IDS presented by Song et al. [10], who have created an IDS that looks at the timing of the CAN messages. Their IDS looks at the frequency of messages with a certain ID and the interval at which they are being received. Song et al. show that it is possible to detect attacks with a very lightweight IDS. Their approach only works for attacks that change the message frequency.

Another similar approach is given by Kuwahara et al. [2] where they analyze how many messages are being sent with each ID in a given time interval. By looking at this metric and, again, assuming periodicity, they are able to detect both suspension and fabrication attacks just by counting the number of times each ID is sent during each interval. This is an elegant and simple solution, but it also lacks the ability to detect attacks that do not change the frequency of the messages.

Inspecting and analyzing the payloads of the CAN frames that are sent can also be used as a method for attack detection, as has been shown with the anomaly-based intrusion detection algorithm CASAD (CAN-aware stealthy attack detection) by Nowdehi et al. [8]. This algorithm is a CAN-specific version of the algorithm PASAD (Process-aware stealthy attack detection), developed by Aoudi et al. [29]. We refer to this algorithm as being *data-driven*, as it looks at the data that is being sent, unlike many of the other algorithms described in this chapter. CASAD has shown the ability to detect all of the four attacks on CAN that we described in section 3.3. CASAD has been shown to detect attacks in vehicles from logged data. It is our opinion that CASAD is currently one of the most promising works in the field.

4.3 Fingerprinting

Fingerprinting is a technique that monitors the system and creates a profile (a fingerprint) for each ECU. This profile can be used to detect some attacks, but, more importantly, can in many cases be used to detect the source of an attack. Source detection is done by matching the fingerprint of the messages involved in an attack with a stored set of fingerprints. Fingerprinting and source detection can be done both as separate processes or as part of the IDS itself, and is especially important in CAN due to the lack of sender authentication. Most, but not all, of the state of the art research for in-vehicle CAN IDS now incorporate fingerprinting. In some cases, research only targets source detection and not intrusion detection.

Choi et al. [9] developed an IDS with fingerprinting abilities based on the use

of extended CAN identification and the signal's characteristics. The standard identification bits in the frame are used as regular CAN identification, while the extended identification bits are all dominant. Only the extended identification bits are gathered for analysis, since no other ECU will be sending at that time, as the arbitration process is complete. The reasoning behind only sending dominant bits is that they were able to show that the recessive bits are not different from ECU to ECU, while the dominant bits are more unique in their physical traits. However, this method and the security changes of CAN proposed by Woo et al. [27] have a common drawback. They both require that every ECU in the network use the extended CAN protocol. This would require that updates be applied to ECUs, which can be costly for the car manufacturers. Additionally, using the extended CAN protocol adds additional overhead, as longer messages cause longer transmission times. In the end, this might not be a feasible approach since the CAN buses are often running at near max capacity.

Cho and Shin [7] also looked at the physical traits of the signal. More precisely, they looked at the clocks of the individual ECUs. The authors utilize the fact that all the ECUs have independent clocks, with slightly different oscillation frequencies, and only synchronize when an ECU is broadcasting. This results in every ECU adjusting their clock every time they see a rising or falling edge on the CAN bus. Cho and Shin show that the clock skew is a characteristic of each ECU, which arises from imperfections in the manufacturing process. By analyzing how the clocks of the sending ECUs were increasing or decreasing in speed, they could pinpoint which ECU is connected to which ID(s). Their proposed IDS detects if one ID is suddenly sent with a different clock skew, and the IDS alerts that the ID is being broadcasted by another ECU than it is supposed to, and also which ECU it believes is doing so. This is a very unique way of utilizing the characteristics of the ECUs, though it has some drawbacks. Firstly, clock skews can be tweaked by adversaries [11]. Secondly, it requires going through a learning process for each individual car, and can not be trained for all cars of the same model, since each clock skew is unique to each physical clock.

In addition to their work on analyzing clock frequencies, Cho and Shin [7] [12] propose a similar solution to the work by Choi et al. [9] called VIDEN. Like the work by Choi et al., VIDEN is able to fingerprint ECUs based on their voltage characteristics. However, it is not capable of intrusion detection, which means that VIDEN can only be used as a complement to an existing IDS. Like the IDS by Choi et al., VIDEN only looks at the dominant bits, and ignores all of the recessive bits, since they do not reflect the characteristics of a single ECU, but rather the bus as a whole. Unlike the former, VIDEN samples voltages during the whole frame, including during the arbitration process. One problem that can occur during voltage sampling is when sampling the ACK bits. In CAN, more than one

ECU is transmitting during an ACK, and may impact the voltage level. VIDEN handles this problem by removing outliers in the voltage levels, and motivates this by stating that the majority of dominant bits that are read are not ACK bits.

Lastly, we want to mention Scission, which is an IDS with source detection designed by Kneib and Huth [13], two researchers at Bosch. This IDS is very similar to VIDEN [12] as it, too, uses the voltage levels to fingerprint ECUs to match them to their IDs. They, however, have two big differences to both VIDEN and the IDS by Choi et al. [9] in which parts of the signal are being analyzed, and at which sampling frequency. Choi et al. use a very high sampling frequency of 2.5GHz, and analyze the extended CAN ID. VIDEN only uses the dominant bits of the frame, and has a much lower sampling frequency of 50kHz. Scission uses a frequency of 20Mhz, and analyzes 3 parts of the signal separately. These three parts of the signal are the dominant bits, rising edges and falling edges. By categorizing all the samples in one of these three categories, Scission can compare them separately, which they show give more accurate results. It is important to note that Scission is not as vulnerable to noise or changes in its surroundings as VIDEN is, since it looks at the differential voltages, while VIDEN compares CAN high and CAN low separately. Along with VIDEN and CASAD, we think Scission is one of the most promising contributions to the field.

4.4 Our Contribution

Compared to the work that we have done, most of the recent research has been done within unrealistic environments, and may not always be applicable in real world scenarios. For instance, most other work has not mentioned trying their IDS against the real-time requirements of the bus, but have rather only tried their algorithms against logged traffic from a real CAN bus. Moreover, most of the research we have looked at neglects the realistic and limited resources that would be available in a car.

In our work, we strive to fill the existing gap in state of the art research by using realistic hardware within a real vehicle, and verifying that an IDS can be used in real-time. To verify our work, we used different test benches and created four different attacks against each test bench. These attacks were created to mirror the four attack categories that we found in our literature review. We believe that these four categories of attacks are a solid way to categorize the attacks on CAN, providing a foundation that captures many of the conceivable attacks. As a comparison, we note that in a lot of research that we found on the topic, many of the attacks seem almost random.

5

Building the Evaluation Framework

Most of the proposed IDS for CAN have, as mentioned in Chapter 4, proven to detect attacks in vehicle. However, most of this research can be divided into two groups that we call *online* and *offline* testing. Online testing is done in real-time in a vehicle to test whether attacks can be detected at the same time they are being performed. In contrast, offline testing is done on the data that has been logged from a vehicle during an attack. Some of the IDS discussed in the related work run in an offline setup and use proof-of-concept algorithms for detection, and may not be optimized to run in real-time. Meanwhile, the IDS that have been tested online seldom use realistic hardware similar to that found in vehicles.

In this chapter, we will first motivate why we have chosen to use CASAD [8] as the algorithm for our IDS, and how it connects to the goals of this thesis. Following this, we describe the system that CASAD is implemented on. Then, we will describe the two test environments that we evaluated CASAD on. The first is a smaller test setup that we used to try CASAD. The second is the environment that is more similar to the real system.

5.1 Choice of Algorithm

Having surveyed several different intrusion detection algorithms, we have chosen to use the data-driven IDS CASAD. This choice was made since we believe that this IDS has the most potential to fulfill the goals of this thesis. Firstly, the algorithm has been shown to detect attacks from the four previously mentioned categories, corresponding to the goal of detecting realistic attacks. Secondly, CASAD has been used — with great results — on data collected from a real vehicle, proving that it is capable of detecting attacks on IVNs. Furthermore, since the authors claim that it is a lightweight algorithm, it makes sense that it is more likely to work in a live environment, where resources are constrained.

CASAD is an algorithm that consists of two phases: a training phase and a detection phase. Training is performed by looking at time series of CAN values, and creating a representation of the normal behavior. The detection phase compares the latest time series of CAN values to the normal behavior from the training phase.

The training is done in several steps, where CASAD replicates the first two steps of singular spectrum analysis, namely the embedding step — creating a trajectory matrix of lagged vectors — followed by calculating its singular value decomposition. This extracts a set of the best representative eigenvectors, and a lower-dimensional representation of the signals, called the signal subspace, can be constructed with minimal information loss. Before the signal subspace is used, CASAD calculates the sample mean (the centroid) of all lagged vectors in the trajectory matrix. The centroid is then projected onto the signal subspace, and is later used for distance tracking. These steps make up the training phase of CASAD.

The detection phase is the part of the algorithm that we implement and optimize. It involves tracking the distance of each new lag vector from the centroid. In order to do this, each new value that is obtained by CASAD updates the lag vector by adding it at one end, and removing the oldest value at the other end, meaning that it will always contain the latest values. The lag vector is projected onto the signal subspace and the L2-norm is calculated from the centroid to the projected vector. If the distance of any lag vector passes some predetermined threshold based on the normal behavior, CASAD sends an alert due to the departure from the normal behaviour.

5.2 System Overview and Design Choices

It is important to test the IDS in an environment that is as close to the intended end product as possible. In our case, the IDS would ideally be deployed in the network of a real vehicle. The proposed IDS running CASAD is depicted in figure 5.1, which shows the IDS connected to the CAN bus which connects ECUs A, B, C and D.

One IDS is required for each CAN bus within the vehicle, since the IDS only listens to one CAN bus at any time, making sure that no attacks go unnoticed. This is yet another important motivation for building a lightweight system, since there should exist several instances of it within the vehicle.

Even though there is more than one CAN bus, we will only be testing the IDS against one. In our experiments, the IDS runs as a freestanding node within the CAN bus, allowing us to extract an accurate representation of the resource usage. By analyzing the resource usage and verifying that the IDS works, we also aim to show that there is a possibility of implementing the IDS on an existing ECU

within the network in the future. The choice of which test bus we use will be motivated and expanded upon in chapter 7.1.2.

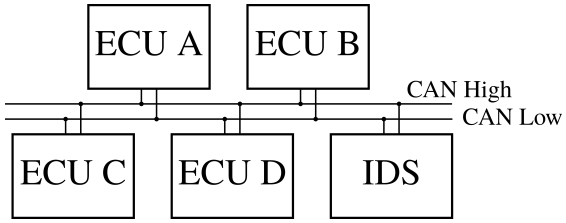


Figure 5.1: Overview of the system with the proposed IDS connected to the same CAN bus as ECU A, B, C, and D.

5.3 Testing Environments

As part of our experiments, a network of programmable microcontrollers are connected to form one of our test benches. By connecting several microcontrollers, it is possible to directly interact with the CAN bus by controlling which nodes communicate, and what data they communicate. The purpose of experimenting on a set of connected microcontrollers is that attacks are easily programmed and simulated, and that it is not difficult to control the number of messages on the bus. This gives us an insight into how different attacks may work, and how they are detected by the IDS. By experimenting on this test bench first, we are able to verify our implementation of the algorithm CASAD before moving on to the following test benches.

In order to ensure that our IDS works in a real environment, it is also tested against a real vehicle. The environment we use is that of a box car, see 7.1.1, which is provided by Volvo Cars. By testing the IDS in a real vehicle, we can see whether it is able to detect attacks despite the larger amount of dynamic values in the network, as opposed to a microcontroller network. Also, by testing the IDS in a box car, we can see if it is able to keep up with the traffic when run online. The box car is a fully wired car, though it is not fully assembled. This means that, unlike a real car, some components are missing but all of the electronics are there.

The box car provides us with a rare opportunity to test the IDS in a more realistic setting, where an attacker may have access to an ECU that has direct access to internal CAN networks. The full access to the CAN networks is different from that of a real car, which only provides an OBD-II port. As previously stated, the OBD-II port only provides CAN bus traffic that is necessary by law, and does not completely represent all the actual communication that is done between ECUs internally.

6

Implementation

In this chapter we explain and motivate the different implementation choices that we do in this thesis. We first motivate the hardware that we use, both for the test benches and for the IDS itself. Following this, we explain how CASAD is implemented, and which optimizations have to be done to the code in order to meet the real-time requirements.

6.1 Choice of Hardware

The target for the implementation for the IDS is a Raspberry Pi 3 Model B [30]. This model of Raspberry Pi comes with a Quad Core Broadcom BCM2837 CPU, with a clock frequency of 1.2 GHz. Additionally, it also comes with 1 GB of RAM, which is more than enough to run the algorithm on. It also comes with an HDMI connector, and four USB 2 ports, which allows us to connect it directly to a monitor, mouse, and keyboard. We used the official Linux distribution for Raspberry Pi, Raspbian Stretch running the June 2018 version. Raspbian is one of the most widely used Raspberry Pi operating systems, with a big community and good documentation. This means that programming does not require any special tools or hardware, and can be done directly. Connecting back to our goals, we mention implementing the algorithm on realistic hardware with limited resources. The Raspberry Pi provides us with a suitable balance between realistic, limited, hardware and programming friendliness.

On the Raspberry Pi, data is logged through a CAN bus interface to the CAN network. The interface used is called PiCAN2 [31], and can be connected to the CAN bus via screw terminals. This interface allows us to interact with the CAN bus via the SocketCAN interface, which is an implementation of the CAN protocol for the Linux kernel. Once the interface is enabled, and a socket is opened, it is possible to store new messages inside a C struct by using the Linux command `recvfrom`. This struct contains the information sent in a CAN frame, including the identifier, the payload length, flags like the RTR bit and, of course, the payload

itself.

6.2 Algorithm Implementation

Our implementation of the CASAD IDS is written in the C programming language. This gives us fine-grained control over the structure of the program, and how it is optimized. By using a low level language, it also enables us to work closer to the hardware, and to reduce the amount of extra processing power required. Writing it in C also makes it highly portable between different low end microcontrollers. If, in the future, we would like to compile this code on a microcontroller with even fewer resources, it is likely to be programmable in C. This means that large chunks of code can be copied, as much of the existing code is not specific to the operating system.

The IDS is implemented in such a way that it may both read directly from a SocketCAN interface, or from a text file. There are two reasons for why we do it this way. First, this allows us to focus on getting CASAD to work offline and then move to make it work online. Second, and more importantly, by allowing it to read from a text file, we could run the same data against different parameters in order to compare the results and how the parameters affect them.

To increase the performance of the IDS, several speed-up techniques are used in the implementation. A rotating buffer is used for storing the lag vector. This means that every new byte that is added does not shift all the values of the lag vector, and is instead done in constant time, as opposed to being bound by the size of the lag vector. In practice, this only requires two operations when adding a new value to the array. One for overwriting or adding a value to a certain slot in the array, and one for incrementing the current pointer value.

Another improvement that is done to CASAD is to multithread the application. We use one thread as a *producer*, whose sole task is to listen to the CAN bus and store the values in what we call a read vector. Using the main thread as a *consumer*, keeping the lag vector up to date with the new CAN values, the matrix multiplications can be split up into different threads. The producer/consumer model is implemented so that no data that arrives in the middle of a departure score calculation is lost. In addition, the multithreading of the matrix multiplications further improves the time it takes to perform the departure score calculations.

In addition to producing individual distance values, we also calculate a rolling average over a fixed number of distance values. By calculating a rolling average, we reduce the number of spikes in the departure score calculations. This is used to reduce the number of false positives, and the number of false negatives. Introducing this additional calculation is an optimization of the result, and not an optimization

of the processing time or memory requirements. This optimization should therefore be used sparingly, together with the other optimizations mentioned earlier.

6.3 Algorithm Optimization

One of the biggest challenges during this project is the need to optimize the algorithm to meet the real time constraints that are present. The problem arises from the fact that CASAD uses a large number of matrix multiplications for detecting anomalies. Since the Raspberry Pi lacks a GPU, the multiplications have to be done on a CPU. Our observations, as mentioned earlier, showed an average bus load of around 67%, which turns into 24 000 incoming bytes per second. As an example, with a lag vector of 10 000, and 24 eigenvalues spanning the subspace, this amounts to 5.76 billion multiplications per second. Without even taking into account cache misses, listening to the CAN bus and interleaving to other processes, this alone is too much — even using all four cores — for the Raspberry Pi.

In order to combat the issue of an excessive number of multiplications, we use three methods:

- Downsampling the input data (input downsampling)
- Reducing the size of the lag vector
- Downsampling the number of calculations (output downsampling)

The first method that we use is input downsampling. Since every byte of the payload is used, we typically receive eight new values with every new frame. When using input downsampling, we reduce the number of multiplications by simply reading fewer values from the bus. We use two different downsampling values, 15 and 63, for comparison. Downsampling by 15 reduces the number of multiplications to 384 million per second, and downsampling by 63 reduces the number of multiplications to 91.4 million multiplications per second. The downsampling rates are chosen following the formula $8n - 1$ so that every byte of the frame is read over time, assuming each payload is 8 bytes long.

Another optimization that we experiment with is the reduction of the size of the lag vector. Much like downsampling, by reducing this value we are able to drastically reduce the number of multiplications that are needed. However, by reducing this value, we also reduce the precision of the algorithm. This is because by reducing the size of the lag vector, the resolution of the projection matrix is also reduced.

The final step we take to reduce the number of multiplications is to downsample the number of departure score calculations. The output of CASAD does not depend on earlier outputs, but only on the input values stored in the lag vector. Taking advantage of this, we could conclude that it is more important to read every value than it is to perform every matrix multiplication. Instead of calculating every output value, our implementation of CASAD can be configured to only calculate every n-th departure score. We call this optimization output downsampling, or downsampling of the output.

7

Experiments & Results

In this chapter, we discuss our experimental setup, how we performed our experiments and the results of these. First, we describe the test benches that we used and the attacks that were performed to verify the IDS. With each test bench increasing in its complexity and requirements, the algorithm is updated between each test bench to manage these new requirements. With our goal of implementing several realistic attacks in mind, we implemented the four previously mentioned attack categories on the test benches. Lastly, we explain the results of these experiments and how they were used to optimize CASAD, in order to reach our goal of implementing a lightweight IDS that works on a realistic environment, in real-time.

7.1 Experiments Setup

During the project, two test benches were used, a microcontroller network and a box car. The former is simply used to verify that the algorithm works, while the latter is a more realistic testing environment used to evaluate the IDS and its performance. The box car tests were divided into two parts: running the IDS offline on logged data, and running it online in real-time. Implementations of the four attacks — suspension, fabrication, masquerade and conquest — were used to test the IDS on each of the test benches.

7.1.1 Test Benches

The microcontroller test bench consists of two Arduino nodes, henceforth called the Arduino network. Both nodes are connected to two common wires that act as CAN high and CAN low, interfaced through the CAN-bus Shield V2 from Seeed Studio [32]. Figure 7.1 shows how the IDS — installed on the Raspberry Pi — is connected to the Arduino test bench. In this setup, both Arduinos are programmed to have a prespecified normal behavior until an attack is triggered. The normal behavior is used to simulate traffic with several unique messages. In order to

further challenge the IDS, some of the values in CAN frames change dynamically, via some hard-coded pattern. Examples of these are values that increment for each message, and some that cycle through a number of randomly selected values. The Arduino test bench was used to verify that the attacks work, and whether the IDS can detect them or not.

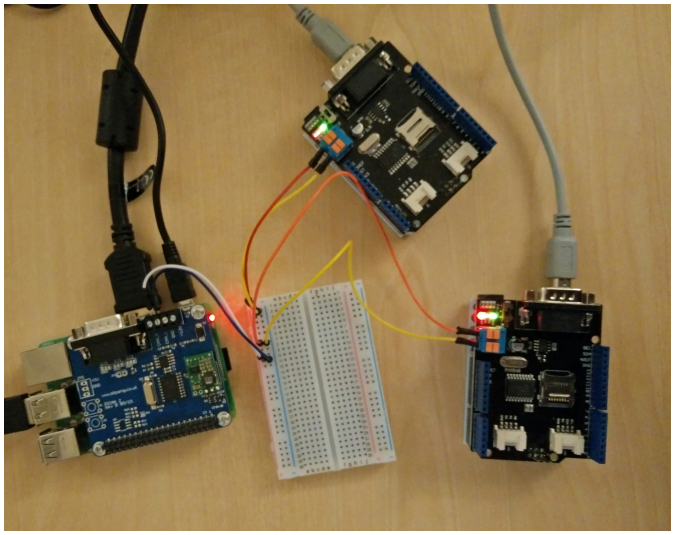


Figure 7.1: Arduino test bench and IDS hardware

The second test bench that we used is a box car. A box car is a testing rig of a real vehicle, with the chassis and wheels stripped away. Test rigs like these allow vehicle manufacturers to run tests and experiments with the electronic components of a vehicle, while also being able to easily access each individual part. By using the box car, we are also able to plug in directly to the internal communication buses, which are not easily accessible in a normal car. By gaining access to these, we gain access to communication buses that attackers might also gain access to through a compromised ECU. The box car that we use is provided to us by Volvo Cars.

Verifying that the IDS works in a simpler system such as the Arduino network does not verify that it works in a real IVN. The IVN of the box car is more complex and has a significantly higher bus load than the Arduino network. This allowed us to verify that the IDS works as it should in the targeted environment. More specifically, it allowed us to verify that the IDS is able to detect attacks in real time, on a realistic network.

For both test benches, a baseline was logged for 2 minutes so that there was enough data to train CASAD, especially when using input downsampling. The baseline was used to produce a threshold value that is used to signal when an

anomaly is detected.

7.1.2 Attack Design

Two implementations were made for each of the four attacks: suspension, fabrication, masquerade and conquest. The attacks were first implemented for the Arduino network and then for the box car. Although the IDS was tested both in an offline and online setting for the box car, the attacks remained the same between the tests. All of the attacks that we performed were logged for 1 minute each, with each being divided into three parts: 20 seconds normal traffic, 20 seconds traffic under attack, and concluding with yet another 20 seconds of normal traffic. This was done to see how the IDS detected the attacks, and also to see what happens after an attack has ended.

The attacks against the Arduino network were performed by sending control signals to the Arduino nodes in the network, with different control signals signalling the start of the different attacks. These attacks were merely used as a proof of concept to test CASAD, verifying that we had implemented it correctly, and were not designed to be realistic.

While experimenting with attacks against the box car, we wanted attacks that matched the realism of the environment. Since the purpose of testing the IDS on the box car environment was to see how the IDS fared in a realistic environment, it made sense that the attacks would behave realistically too. In order to implement the attacks against the box car, the software Vector CANoe is used. CANoe is an application that is often used to analyze CAN traffic, and acts as a direct interface to the bus. It can also be used to simulate ECUs, that can receive and transmit data to the CAN bus. Within CANoe, the attacks against the box car were written in the programming language CAPL (Communication Access Programming Language), a proprietary programming language developed by Vector. Other in-house diagnostics tools were also used, in order to suspend some ECUs altogether, imitating weak access.

Having selected the CAN bus responsible for communication related to the chassis of the vehicle, we logged a baseline of uncompromised traffic. The CAN bus we used was selected primarily for demonstration purposes, as there were certain messages that were possible to fabricate that allowed for visual results. The same ECU was targeted for suspension, fabrication and masquerade attacks. This ECU was responsible for displaying the RPM on the dashboard. An additional ECU was simulated with a basic periodic message that we targeted while performing the conquest attack. This message was sent with a low priority, so as to not disrupt or cause any disturbances to the normal traffic.

7.1.3 Box Car: Online and Offline

The box car test bench was, as mentioned before, split into two different test environments: offline and online. For the offline experiments, data was logged before and during an attack, and was processed by the IDS afterwards, without any real-time requirements. In contrast, the online implementation of the IDS processed data as it arrived, which required the IDS to detect attacks in real-time without missing any data on the bus.

By performing the offline experiments, we were able to benchmark different optimizations of the algorithm against a baseline implementation without any optimizations. By comparing the speed improvements between each optimization, as well as an increase or decrease in detection accuracy, we were able to determine which optimizations were the most suitable for running the algorithm in the online setting.

In order to fulfill the goals of the thesis, we performed the final online experiments. By performing the online experiments, we were able to show whether it is possible to detect attacks in real-time in a realistic environment or not. This meant being able to keep up with the traffic, without missing any data, as well as being able to compute the departure scores to detect the attacks in a reasonable time frame.

7.2 Experiments on the Arduino Network

The first attacks that were performed were performed against the Arduino network with two nodes, A and B. As previously mentioned, the attacks are triggered externally by sending specific serial signals from a computer to one of the Arduinos. In case of an attack where multiple participants need to act as if they are compromised, the same node sends a zero length CAN message with a certain ID to trigger the other participant to start. This makes it simpler from a testing point of view, and since only the data is read by the IDS, it does not change the results.

Suspension Attack

The suspension attack is implemented on the Arduino network by using a global flag, which is set when receiving a serial signal. Both nodes A and B are programmed to send data normally, until the attack is triggered. When that happens, the compromised node A immediately stops sending CAN frames, while node B continues sending data as before. The results of this experiment is shown in figure 7.2, which shows how CASAD detects the attack. In addition, the threshold of

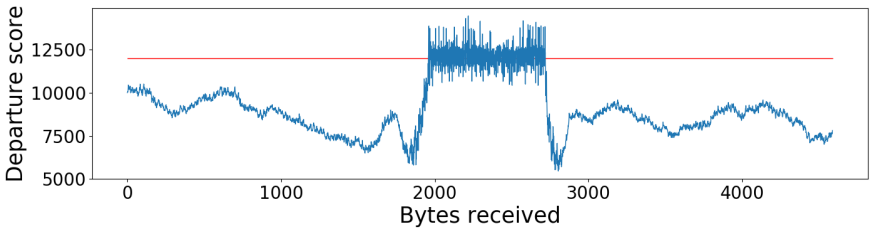


Figure 7.2: Suspension attack on the Arduino test bench.

12 000 does not trigger any false positives.

Fabrication Attack

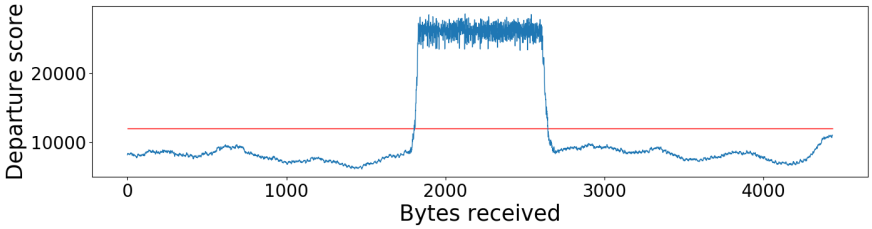


Figure 7.3: Fabrication attack on the Arduino test bench.

Practically, a fabrication attack is performed by sending CAN frames with an ID that belongs to another ECU, at a higher frequency. On the Arduino network, the attack is implemented on node B, and its aim is to fabricate a message that belongs to node A, at a higher frequency. The result of this attack is shown in figure 7.3, which shows that the attack has a much higher impact than the suspension attack. Using the same threshold as before, the attack is clearly detected.

Masquerade Attack

A masquerade is a combination of both a fabrication and a suspension attack. As such, for the Arduino network, the same techniques are re-used for this attack, suspending messages belonging to node A, and letting node B fabricate one of A's messages, with the same frequency. Looking at the result of this, as shown in figure 7.4, we can see that the attack has a significant impact on the network, greater than both fabrication and suspension, and is clearly detected.

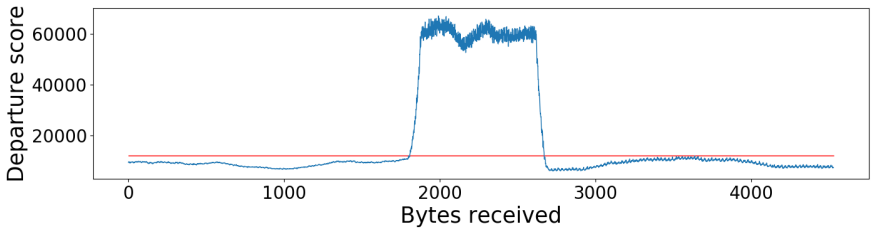


Figure 7.4: Masquerade attack on the Arduino test bench.

Conquest Attack

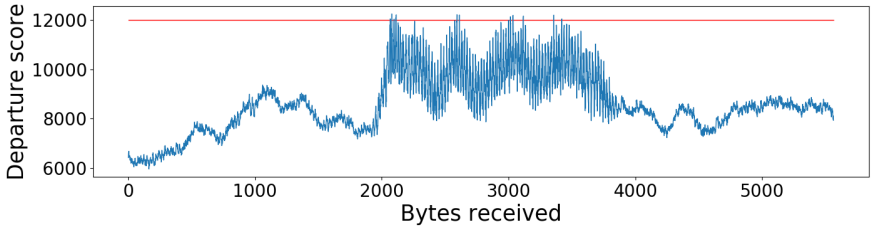


Figure 7.5: Conquest attack on the Arduino test bench.

In the Arduino network, the target of the conquest attack is node A. From a attacker's point of view, the goal of this attack is to be as stealthy as possible, and the attack should have the smallest footprint possible, while still remaining an attack. For this reason, the attack only changes a few of the bytes of a single message when triggered. We can see from figure 7.5 that this attack is much stealthier than the previous three, and is barely detectable using the same threshold used for the previous attacks. Even so, it is still detected several times during its run.

7.3 Offline Experiments on the Box Car

In the offline experiments on the box car, all of the data was logged with CANoe then run on CASAD, from a file, to see if the attacks were detected. We ran the logged data through CASAD five times with different settings to see how well it performed. The parameters and optimizations used in these five cases can be seen in table 7.1.

The first test case was our baseline, with neither input nor output downsampling. This case was used as a reference for the accuracy, which we compared the other cases against. For the first case, we used the parameters $L = 10\,000$ and

Table 7.1: The four CASAD settings that were used during the box car offline experiments.

Case	L	r	Input downsample	Output downsample	Threshold
1	10 000	24	1	1	4000
2	5000	24	1	1	6000
3	10 000	24	1	1000	4000
4	8000	58	15	1	10 000
5	8000	20	63	1	30 000

$r = 24$. In contrast, we reduced the size of the lag vector for the second case, to $L = 5000$, to see how this affected the accuracy of CASAD.

In the third case, we continue using the same parameters as the first, baseline, case. However, we run this case using an output downsampling rate of 1000. Similarly, we run the fourth and fifth case using input downsampling. The fourth case, with an input downsampling rate of 15, uses the parameters $L = 8000$ and $r = 58$. Meanwhile, the fifth case, with an input downsampling rate of 63, used the parameters $L = 8000$ and $r = 20$.

Baseline

As mentioned before, the baseline consists of normal traffic from the car, with the addition of our simulated ECU. This behaviour is highly regular and the departure score does not diverge much, as can be seen in figure 7.6a. When downsampling the output by 1000, we noticed that the precision of CASAD is not affected, as seen in figure 7.6c. Since the parameters and input were not changed, we did not have to change the threshold. However, when we change to a lag vector of 5000, we have to change the threshold of CASAD, as can be seen in figure 7.6b. The threshold also needed to change for both of the input downsampling cases. When we downsampled the input by 15, seen in figure 7.6d, we had to change the threshold of CASAD in order for it to not result in any false positives. In the same way, we increased the threshold further when downsampling by 63, seen in figure 7.6e, so that the threshold does not cause any false positives. Though the thresholds change between each test case, the threshold will remain the same between the detection of each attack.

Suspension Attack

The suspension attack on the boxcar was performed by suspending an ECU with the internal diagnostic tools. Several CAN frames are affected by this since one

ECU controls several functionalities. We found this to be a realistic approach, since an outside hacker would not be able to cherry-pick which messages to disable if they only had weak access to an ECU.

In figure 7.7a, we can see that the attack is easily detected by CASAD when no downsampling is used. The lower value of L produced a similar result to the first case but was not as precise, as can be seen in figure 7.7b. Using output downsampling by 1000, seen in figure 7.7c, gives almost the same results as the first case. The attack is also detectable with input downsampling, and is seen in figure 7.7d and figure 7.7e. In the cases where the input has been downsampled, the attacks can not be identified as clearly as in the other cases. In addition, it is also less clear when the attack starts.

Fabrication Attack

The fabrication attack is performed by simulating an extra ECU that launches the attack. This simulated ECU does not send any frames in normal behaviour. When the attack is started, the ECU starts sending a falsified message frame with an ID that belongs to the ECU that was suspended during the suspension attack.

The attack can be detected without any problems by CASAD running without downsampling, as can be seen in figure 7.8a. It is worth noting that this fabrication attack is not as detectable as the suspension attack. In figure 7.8b, a precision loss can be seen when compared to figure 7.8a. Figure 7.8c shows that CASAD still detects the fabrication attack even after output downsampling by 1000. It can be seen, in figure 7.8d with input downsampling by 15, that it is still possible to detect the attack, even when the attack is not as clearly visible as the regular CASAD and CASAD with output downsampling. An unexpected behaviour of returning below the baseline during the attack can also be seen. However, with a input downsampling rate of 63, the fabrication attack is no longer visible, as the departure score is steadily under the threshold, as can be seen in figure 7.8e.

Masquerade Attack

The masquerade attack was performed by disabling the same ECU that we suspended during the suspension attack, and at the same time simulating an ECU that fabricates frames with the same ID as one of the IDs belonging to the now suspended ECU.

Similar to the suspension attack, masquerade was detectable by all five test cases of CASAD, which can be seen in figure 7.9. What is notable is the same slope which appears in both figure 7.9d and figure 7.9e. This slope is also present in the suspension attack with input downsampling, see figure 7.7d and 7.7e.

Conquest Attack

As mentioned earlier, we simulated an extra ECU for the conquest attack. This extra ECU only sent messages with one ID, which was dynamically changed. The attack changed the repeating pattern of bytes composing the payload, but left the interval and ID the same as before. Unfortunately, the attack was not detected by CASAD, as can be seen in figure 7.10.

Rolling Average

In the previous experiments, we noticed that there were significant spikes in the graphs. These spikes could potentially lead to false positives, and certainly leads to false negatives, as we saw in the fabrication attack in figure 7.8c. Since the third case (downsampling the output by 1000) presented in table 7.1 gave us the best results in our opinion, these were the parameters that we chose to test using the rolling average. The results can be seen in figure 7.11, where we used a rolling average of the latest 50 values. Most notable is the impact the rolling average had on the fabrication attack. As can be seen in figure 7.11b, once the attack had begun, and been detected, there were no more false negatives. In contrast, there were a significant number of false negatives in the original case. In addition, we were able to reduce the threshold from 4000 to 2000 in this particular case, without introducing any false positives.

The decision to implement and test a rolling average was done after we had performed the online experiments on the box car, and as such we did not have the time to test whether the rolling average worked in real-time as well.

7.4 Online Experiments on the Box Car

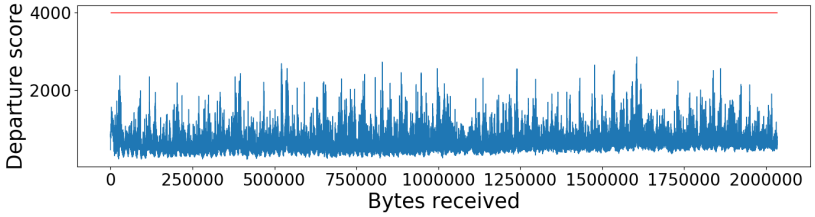
Observing the results, we chose the third case from table 7.1 (no input downsampling, output downsampling of 1000 and with a lag vector of 10 000), since it produced the best results compared to the other cases. Our observations are mainly based on the fabrication attack, where we noted the largest difference between the test cases.

In order to meet the real-time requirements CASAD has to be able to both read all of the CAN traffic without any gaps and and at the same time perform departure score calculations. Each departure score took more time to calculate than it took for new data to arrive on the bus, causing data to be discarded. For this reason, a single thread was dedicated to reading and producing the CAN values for the other threads, while they continued performing matrix multiplications.

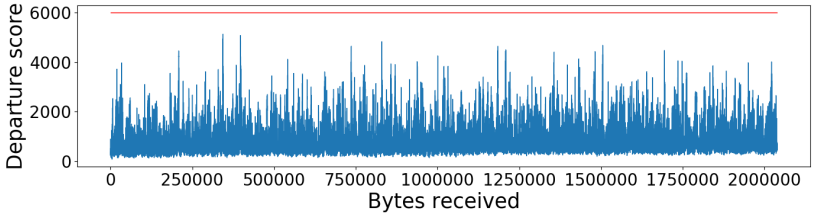
Since one less thread was used to perform matrix multiplications, we chose to double the output downsampling rate, using a value of 2000 instead. Since we observed no loss in accuracy between the use of no output downsampling and an output downsampling of 1000, we argued that using an output downsampling of 2000 would not significantly reduce the accuracy either.

The results from the online implementation of CASAD can be seen in figure 7.12. The results are very similar to the results from the offline experiments, with the IDS being able to detect the same three attacks. This is not surprising, since the attacks and test bench remain the same. Nevertheless, the fact that the IDS can detect the attacks in real-time is the important take-away from this experiment.

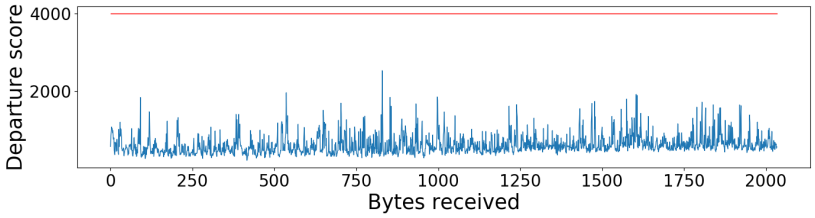
In figure 7.13, the output of the CASAD application is shown, which outputs a textual alert when an attack is detected. In this particular case, a suspension attack has been detected. CASAD does not print anything if the threshold is not exceeded, but as soon as the attack is activated, and the threshold is passed, CASAD will announce it.



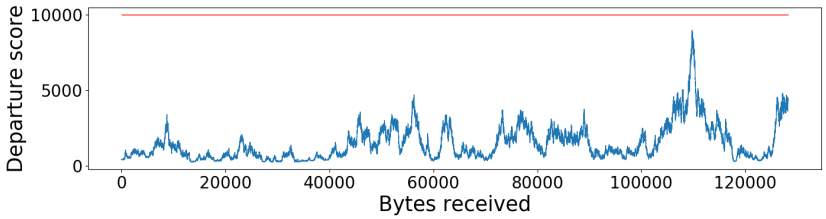
(a) No downsample.



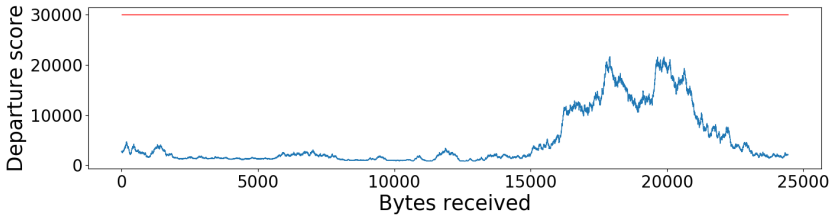
(b) No downsample and L=5000



(c) Departure score calculations downsampled by 1000.

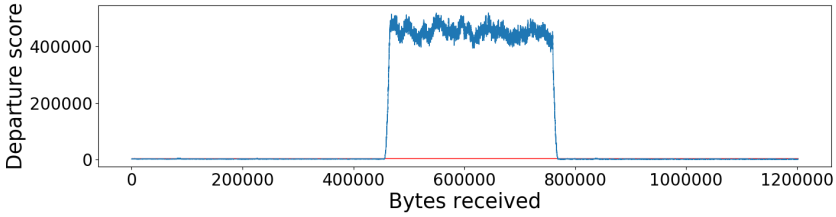


(d) Input downsampled by 15.

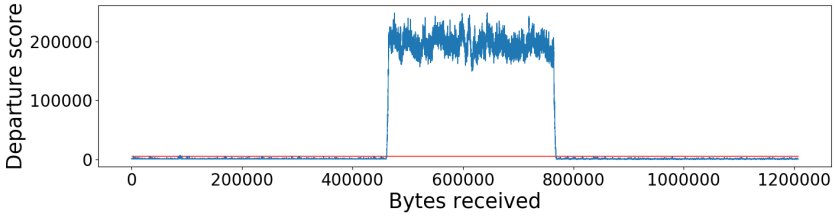


(e) Input downsampled by 63.

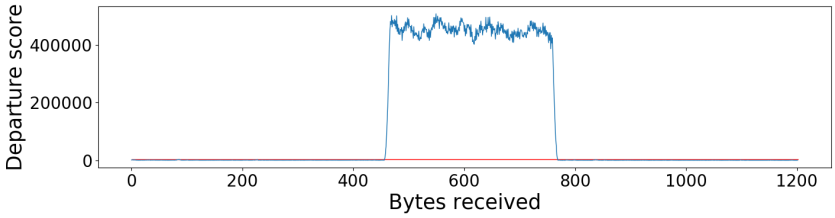
Figure 7.6: CASAD departure score for baseline CAN traffic while no attack is being performed.



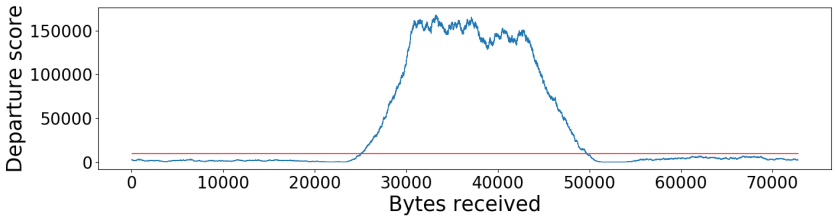
(a) No downsample.



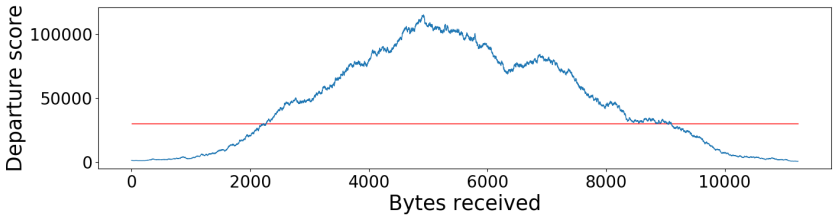
(b) No downsample, L = 5000.



(c) Output downsampled by 1000.

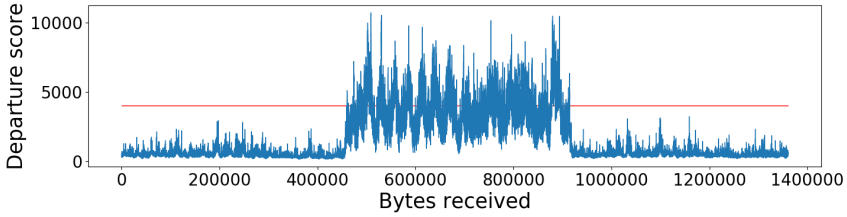


(d) Input downsampled by 15.

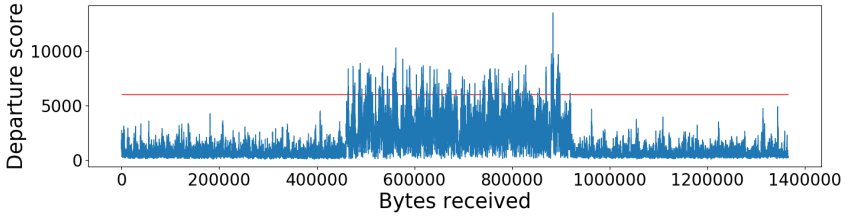


(e) Input downsampled by 63.

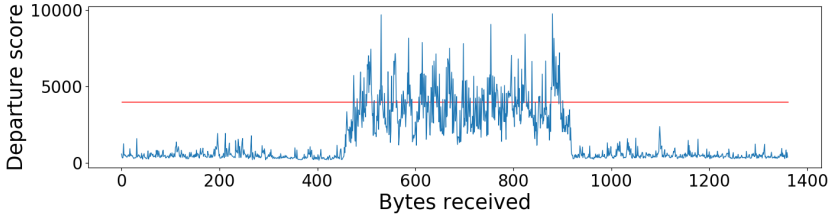
Figure 7.7: CASAD departure score for CAN traffic while a suspension attack is being performed.



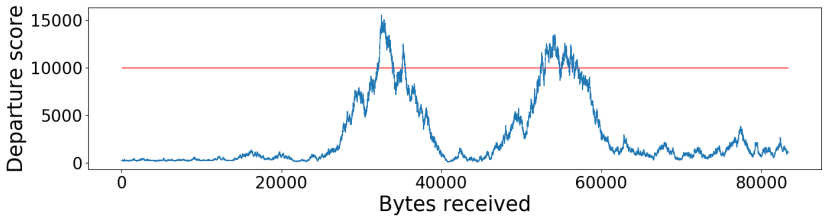
(a) No downsample.



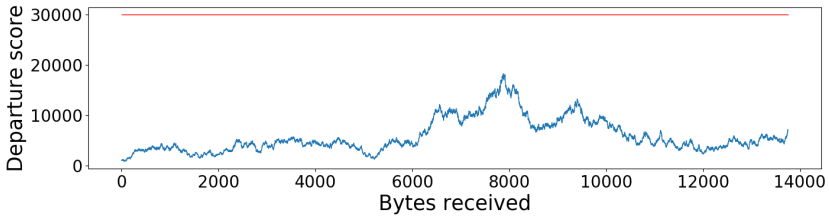
(b) No downsample, L = 5000.



(c) Output downsampled by 1000.

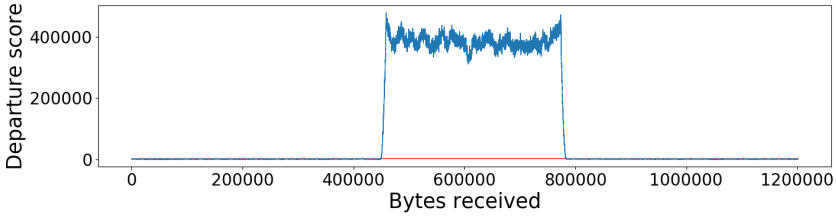


(d) Input downsampled by 15.

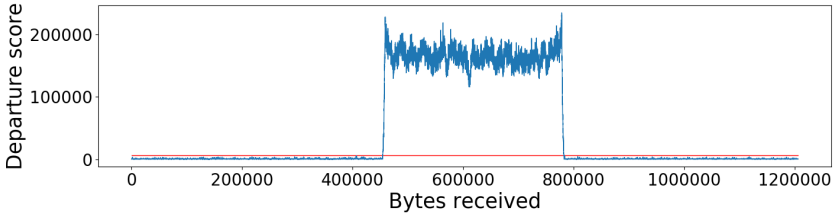


(e) Input downsampled by 63.

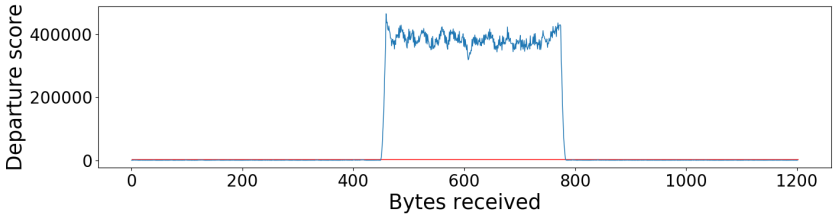
Figure 7.8: CASAD departure score for CAN traffic while a fabrication attack is being performed.



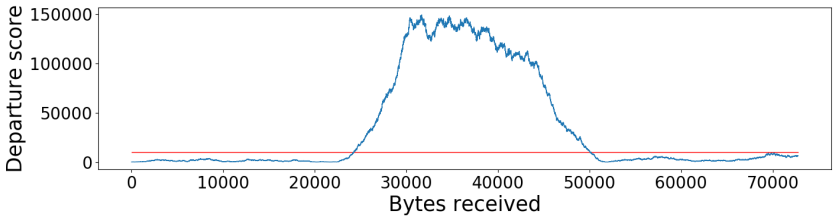
(a) No downsample.



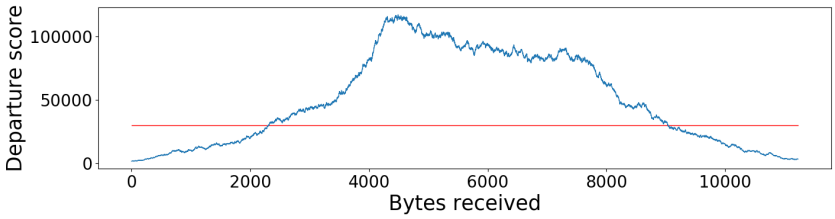
(b) No downsample, L = 5000.



(c) Output downsampled by 1000.

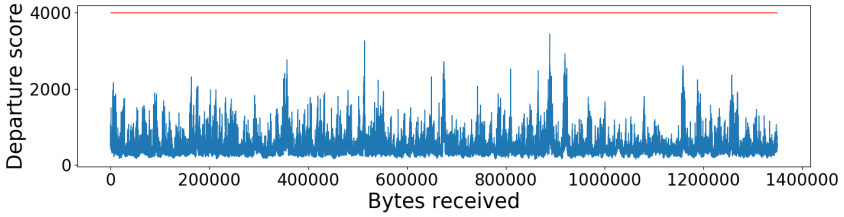


(d) Input downsampled by 15.

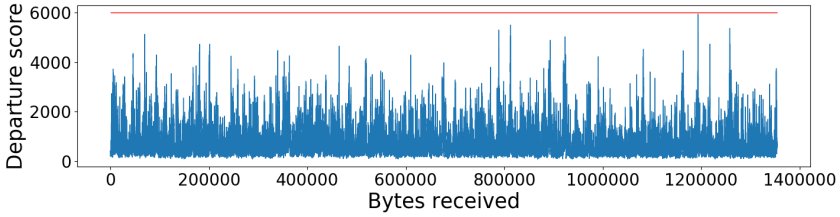


(e) Input downsampled by 63.

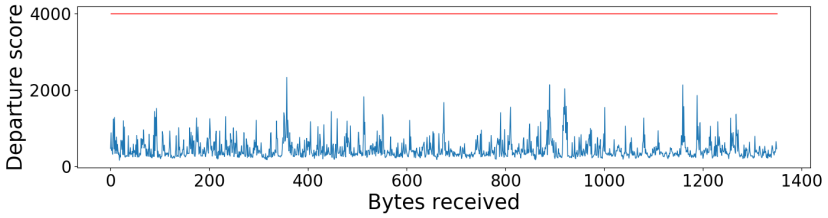
Figure 7.9: CASAD departure score for CAN traffic while a masquerade attack is being performed.



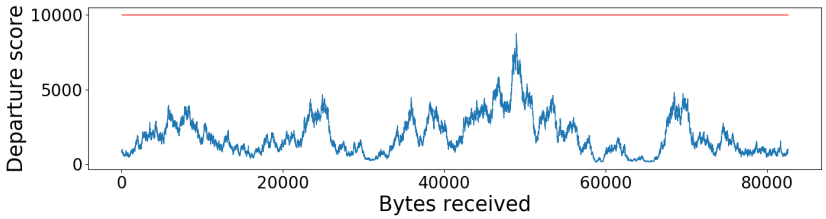
(a) No downsample.



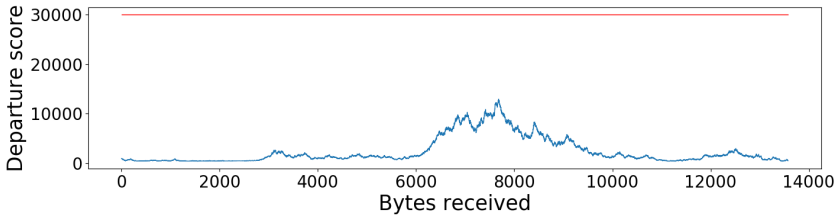
(b) No downsample, L = 5000.



(c) Output downsampled by 1000.

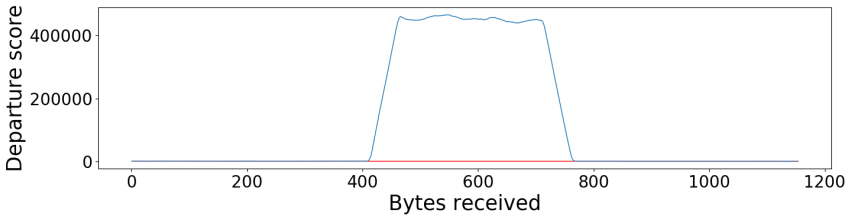


(d) Input downsampled by 15.

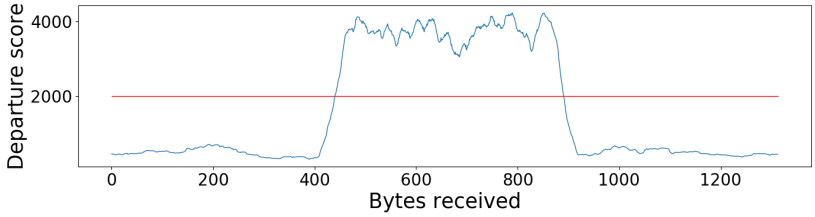


(e) Input downsampled by 63.

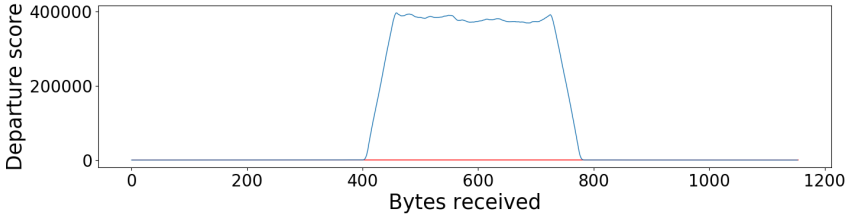
Figure 7.10: CASAD departure score for CAN traffic while a conquest attack is being performed.



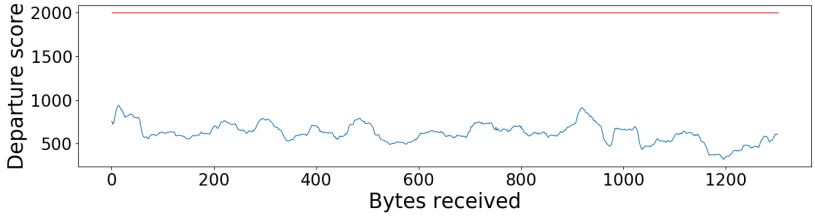
(a) Suspension attack



(b) Fabrication attack

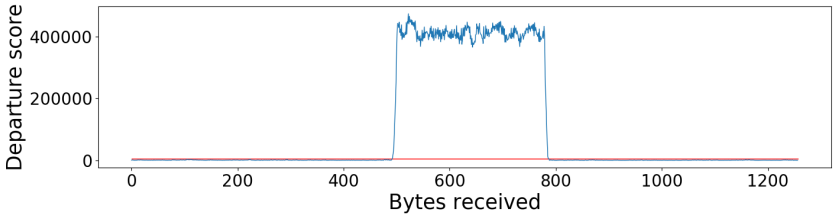


(c) Masquerade attack

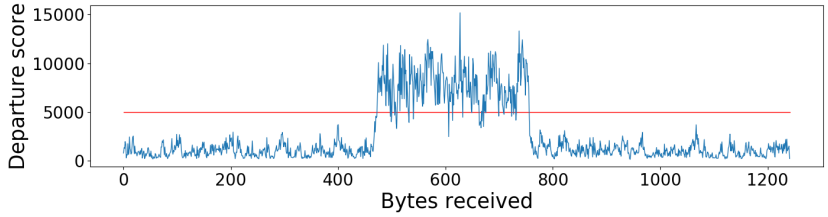


(d) Conquest attack

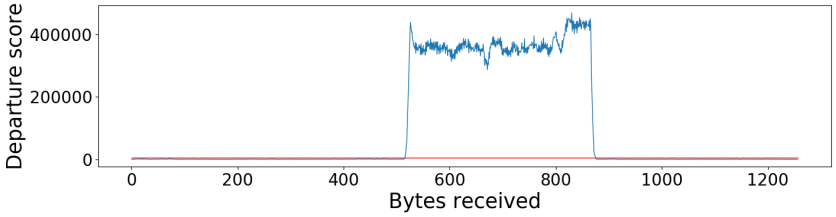
Figure 7.11: CASAD departure score for the four attacks using rolling average of 50 to reduce the number of spikes that might trigger false positives and negatives. For these experiments, output downsampling by 1000 is used.



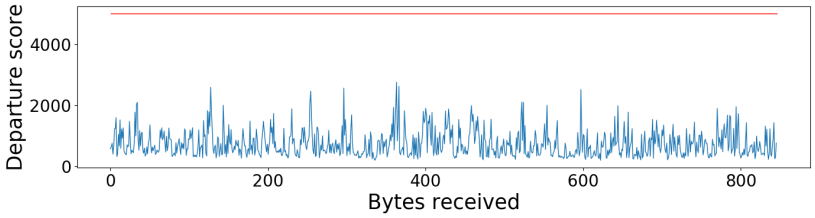
(a) Suspension attack



(b) Fabrication attack



(c) Masquerade attack



(d) Conquest attack

Figure 7.12: CASAD departure scores for the four attacks used on a vehicle in real-time. Output downsampling by 2000 is used in all cases.

```
2019-01-25_11-27
ALERT! - Threshold of 4000 passed. Departurescore: 9073.084604
ALERT! - Threshold of 4000 passed. Departurescore: 28845.550652
ALERT! - Threshold of 4000 passed. Departurescore: 49330.163643
ALERT! - Threshold of 4000 passed. Departurescore: 94550.525270
ALERT! - Threshold of 4000 passed. Departurescore: 142589.821186
ALERT! - Threshold of 4000 passed. Departurescore: 185252.925399
ALERT! - Threshold of 4000 passed. Departurescore: 261231.875970
ALERT! - Threshold of 4000 passed. Departurescore: 331861.132433
```

Figure 7.13: An alert message from the CASAD application, that is displayed when the departure score passes the selected threshold. In this case, CASAD is witnessing a suspension attack.

8

Discussion

Many of the results gathered from the experiments were unexpected, and the thesis took many unexpected twists and turns on its way to completion. In this chapter, we will discuss the implications of the results we gathered, and what conclusions we can draw from them. Following a lengthy discussion of our steps and challenges, we also discuss some of the future work that may be done in the footsteps of this thesis. We conclude the chapter by discussing the thesis from the perspective of ethics and sustainability.

8.1 Implication of our Results

In this thesis, we have looked at whether or not it is possible to implement an intrusion detection system for in-vehicle networks, with the real-time requirements of keeping up with the traffic of the bus to quickly detect attacks, and with the constraints of implementing it on a limited hardware. To investigate if an IDS could work on a limited hardware, we chose an IDS algorithm that had already been proven to work, CASAD, and implemented it on a Raspberry Pi 3. To further test if the IDS could detect attacks, we identified four different attack categories which we tested the IDS against. The IDS was tested on two different test benches: an Arduino network and the IVN of a box car.

8.1.1 Test Benches

Having started on a simple Arduino network, we showed that the IDS that we chose to implement, CASAD, successfully detected four out of the four attack categories we identified. We saw that attacks that impacted the number of messages — suspension, fabrication and masquerade — on the CAN bus were especially detectable, but we also confirmed that attacks that did not — conquest — could be detected. Although this setup was far more simple than a real IVN, this setup confirmed to us that our implementation of the IDS worked, and that we could

move on to a more advanced target, the box car.

Detecting attacks on the box car presented a new set of challenges we had to overcome, including a higher bus load, significant variability of the CAN frames and the real-time requirement for our online experiments. However, even with these challenges present, CASAD successfully detected three out of four attacks. Not only can it do this in an offline environment, but with the addition of downsampling on the output, it can successfully do this in real time, too. Suspension and masquerade attacks were clearly detectable, since all of the ECUs connected to the CAN bus sent out a large number of different messages. This meant that suspending any one of them leads to a significant change in the traffic, and this is why they are also the only two attacks detectable when downsampling the input. However, fabrication was harder to detect, but could still be detected in four out of five test cases for the offline version. Meanwhile, conquest attacks were not detected at all on this test bench.

It is unfortunate to see that our implementation of the conquest attack, which is known to be stealthy, could not be detected on the box car. Even with relatively large parameters, it is hard to detect such subtle changes to the system. With roughly 3000 messages per second, only a tiny fraction of the traffic changes as the conquest attack is activated, making it a truly stealthy attack. Regardless, we do not believe it is impossible to detect this attack against a box car, as the authors of CASAD were able to do so. We were also able to detect the conquest attack when it was performed on the Arduino test bench, which indicates that it might be possible to detect in the future. As a last note on conquest, we believe that it might be more noticeable in a real scenario, since other ECUs may be dependant on the ID. Since the conquest attack we created did not have any ECUs listening, the attack did not have a cascading effect across the bus.

8.1.2 Optimizations

In order to reach our final goal of implementing CASAD to work in real-time, much of our work was spent finding and implementing different optimizations to the algorithm. Two types of optimizations were done, where the first focused only on improving our implementation of CASAD, while the other focused on reducing the number of matrix multiplications that had to be done per second by CASAD to detect attacks. We will first cover the overall improvements to the algorithm, and then look at the optimizations for reducing the number of matrix multiplications per second.

General Implementation Improvements

The input to the algorithm consists of a long lag vector, where each new input shifts the values of the vector. This means that the lag vector will always contain the latest L values, but shifting every value of the vector for each new input is not a feasible strategy. Therefore, the first optimization we made to our implementation was to use a rotating buffer, which is a fixed-size array with a shifting pointer to start and finish. By using this method, we only have to overwrite old data, and move a pointer, instead of shifting every single value in the array. This improved the complexity of this part of the algorithm from $O(L)$ to $O(1)$, a significant improvement.

Since we had observed that the detection phase of CASAD consisted of a large number of matrix multiplications, the second optimization we applied to the algorithm was threading. This works naturally well with matrix multiplications and led to significant improvements, since matrix multiplication is an embarrassingly parallel problem. Though the Raspberry Pi 3 has four cores, only three of them were used for matrix multiplication. The change from four to three threads had to be done when moving from the offline experiments to the online one, as we needed one additional thread to read data from the CAN bus. Threading solved both the problem of incoming data being lost and simultaneously sped up the algorithm.

Having observed a number of spikes in the output of the algorithm, the third general improvement that we did was to calculate a rolling average of the last 50 departure scores. As we saw from the resulting graphs, this greatly reduced the number of spikes, and produced much smoother and consistent results. These results show that this is an important optimization for improved accuracy of the algorithm. While the system was not under attack, the rolling average did not approach the threshold nearly as much as for the normal departure score calculations. Similarly, the rolling average never went below the threshold during attack, meaning that we could not observe any false negatives. These results imply that it might be possible to decrease the threshold further to detect stealthier attacks, without introducing any false positives and only adding a small amount of overhead.

Reduction of Matrix Multiplications

The first optimization done to reduce the number of matrix multiplications per second was to reduce the size of the lag vector. By reducing the size of the lag vector from 10 000 to 5000, the total number of matrix multiplications would also be reduced by half. Even with this optimization in place, CASAD managed to detect the attacks almost as well as for the benchmarking case. However, during

the fabrication attack we were able to see a loss in accuracy. We found that using only this optimization, the IDS was not able to run in real-time. This method could be used in combination with other optimizations if the memory of the ECU is scarce.

The second optimization we used was to downsample the input, and not use every incoming byte. The number of calculations needed are reduced linearly with regards to the input downsampling rate. For our two test cases using input downsampling, we chose to downsample on every 15th and 63rd bytes. These values were chosen following the formula of $8n - 1$, since this would capture every byte. This method seemed to work well, since we were able to significantly reduce the number of multiplications needed. However, this change was not enough to make the algorithm work in real-time in a realistic environment. In addition, when we reduced the input downsampling beyond 15, we started to lose a lot of accuracy. We noticed that the attacks took a longer time to be detected, since the downsampling likely caused CASAD to miss the first couple of attacked bytes. This effect can be seen in the slopes of the input downsample graphs. Another flaw with the input downsampling was that the number of false negatives increased, which resulted in CASAD not detecting the fabrication attack for every departure score calculation, even though the attack was ongoing. As a final reflection, we believe that input downsampling is not the best solution for improving the speed of the algorithm. Instead, it might serve a purpose in training the algorithm on a longer time series.

The last method for reducing the rate of matrix multiplications was to use output downsampling, where we did not calculate the output for every incoming value. By downsampling the output, we observe a much more discrete and well formed output in the graph, which is much more similar to the output when using no optimizations at all. In comparison to doing input downsampling, this method does not miss any bytes, and each window is a continuous series of bytes. This means that no data is lost, and there is a significantly lower delay in detection. Using an output downsampling rate of $L = 2000$, and a lag vector of $L = 10\,000$, every byte sent is used five times in the calculations.

We found that output downsampling was the single most efficient way to reduce the number of calculations, since we showed that this could be done with downsampling rates as high as 2000, without reducing the accuracy of CASAD. Not only was it the most effective, but it was the only method that we tried that could detect attacks in real-time. We did not try higher output downsampling rates than 2000, but observing the results, we estimate that the output downsampling rate could be almost as high as the lag vector size. Since fewer calculations would be needed for this, we would see a speed up of the algorithm, allowing for a bigger lag vector.

Whether or not CASAD can be implemented on a real car remains to be seen, but it is our belief that it can. Since the best optimization we found was to implement output downsampling, we believe that the requirements of cache memory outweigh the requirements of the CPU. Since increasing the size of the lag vector significantly increases the accuracy of the algorithm, this is preferable. However, it will also add an extra strain on memory requirements. A larger lag vector also allows for a higher rate of output downsampling, which reduces some stress on the CPU. This leads us to believe that the requirement that might be the hardest to meet in order to run the IDS is that of the memory requirements.

8.2 Future Work

We find CASAD to be a strong IDS for IVN communication on CAN, and have some suggestions for future work. Some of the suggestions that we see as promising are regarding the algorithm CASAD, while other suggestions are complements to CASAD, to create a more holistic IDS.

While CASAD is able to detect changes to the periodic behaviour of the system, it is not as good at detecting small changes that occur in CAN frames with low frequencies, or potential control messages that may only be triggered once. By using a signature-based IDS alongside CASAD, these issues may be solved by combining these types of IDS into one.

Communication networks in modern cars are not limited to simply using the CAN protocol. Indeed, modern versions of CAN, such as CAN-FD, implementing a flexible data-rate, are increasingly being implemented. Future work would likely involve trying to implement CASAD on these communication protocols. Assuming many of the messages in these protocols are periodic, we do not see why this could not work. Indeed, we only looked at the data being transmitted, and not at any properties that are specific to CAN.

Detection is the first step towards attack prevention. Ideally, we would not just detect that an attack exists, but also prevent it. In order to prevent attacks, beyond simply shutting down the whole system and requiring the user take the vehicle to service, it may also be possible to detect which ECU has been compromised. By detecting which ECU is compromised using source detection techniques, the compromised ECU can potentially be reset or have their communications muted. This would prevent them from causing further harm to the system. While source detection techniques exist, such as VIDEN [12], it would be interesting to see how a combination of techniques like CASAD and VIDEN could be made to work in unison.

One interesting case that we did not take into account in this thesis is the case where an attacker compromises the IDS itself. We see future work in this area

involving the deployment of multiple IDS that communicate separately and compare their results using agreement algorithms. When a majority of the IDS detect an attack, there would be an agreement that there is an attack being performed, and prevention measures may be taken. Compromised IDS that are used for prevention could be used to either hide an attack by claiming there is no attack, or to forcefully reset or mute ECUs, simply by claiming that one of them is compromised. Therefore, it is especially important when developing intrusion prevention mechanisms to verify that the ECU is actually under attack, especially if resetting or muting the ECU could play to the advantage of the attacker.

Due to the large number of matrix multiplications we observed when investigating how CASAD works, it would be interesting to investigate the use of graphics processing units (GPUs) instead of the CPUs we used. We believe that this could lead to significant speed-ups. GPUs have thousands of processing cores, with special instructions for matrix operations. With modern cars having features such as driving assistance systems like parking assistance and automatic braking which require sensors and cameras, we think that more and more GPUs will be available in vehicles in the future. Indeed, as self-driving cars are approaching reality, they will definitely require the use of GPUs.

During the project, we decided that an appropriate downsampling rate of the output was every 1000 departure score calculations for the offline version, and 2000 departure score calculations for the online version, since both of these gave well defined outputs with no false positives or loss in accuracy. Though these provided satisfying results, it would be interesting to further investigate if it is feasible to increase the rate of output downsampling even further, in order to increase the lag vector and the precision of the algorithm. In addition to this, we believe this is the way to go forward in order to test CASAD on even more resource strict hardware, to test the limits of CASAD.

8.3 Ethics and Sustainability

Having shown that implementing an IDS on a vehicle is, indeed, feasible, we believe that the only ethical course of action is to implement one as soon as possible. With the large attack surfaces of modern vehicles, manufacturers have a responsibility to work to reduce the attack surface, and to implement intrusion detection, and even prevention, systems in their vehicles. A vehicle that can be hacked, and controlled, is a vehicle that is unsafe for society. This is especially important given the lifespans of cars, which may be used for years, if not decades. We refer back to the work by Miller and Valasek [6], where they managed to turn off the brakes of the car entirely. A car, or worse, a truck, unable to brake is a danger to everyone

in front of it, and to everyone in the vehicle. Attacks that are perceived as being less ”harmful” may also be dangerous. Drivers of vehicles where the tachometer suddenly starts acting erratically — an attack we have performed — may react unpredictably and therefore dangerously.

Equipping a vehicle with an IDS makes it, in many ways, more sustainable. By improving the security of vehicles, there is a lower risk of car manufacturers having to recall cars. Actions such as recalling cars may have any number of impacts on the environment, especially due to transportation of the vehicles. Attacks that can severely affect the normal operations of the car may increase its gas emissions and fuel consumption. Even newer cars using batteries, and not gasoline or diesel, may have their batteries drained entirely, as was shown by Cho and Shin [28]. Since it may not always be possible to recover drained batteries, batteries of attacks such as these may have to be thrown away, which also has an environmental impact.

9

Conclusion

This thesis has investigated the possibility and feasibility of implementing an IDS in an IVN for vehicles with real-time requirements. CASAD, an anomaly-based algorithm for attack detection which has been shown to work in IVNs, was chosen and implemented. In addition to detecting attacks, we have shown that it can be optimized to the point where it can detect attacks in real time on a highly loaded CAN bus using a small low-resource computer.

Having first identified four different types of attacks, we implemented each type of attack on two different test benches that we decided to use: an Arduino network and a box car. The IDS was first tested against the Arduino network, and confirmed that CASAD was able to detect all types of attacks while implemented on a low end hardware. On the box car, we compared different optimizations of the algorithm to make it work in real-time. We performed a comparison of different optimizations — reducing the lag vector, downsampling on the input and downsampling on the output — and found that downsampling on the output was the most efficient method for running CASAD in real-time. This method also retained the most accuracy, compared to the baseline. By using this method, we were able to detect three out of the four attacks that we created.

In conclusion, we have shown that is indeed possible to implement an IDS for IVN that is lightweight while also fulfilling the real-time requirements. The IDS we have implemented for this thesis is able to detect several attacks with high accuracy on a CAN bus with a high network load, while running on low end hardware.

Bibliography

[1] F. Sagstetter, M. Lukasiewycz, S. Steinhorst, M. Wolf, A. Bouard, W. R. Harris, S. Jha, T. Peyrin, A. Poschmann, and S. Chakraborty, "Security challenges in automotive hardware/software architecture design," *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 458–463, 2013.

[2] T. Kuwahara, Y. Baba, H. Kashima, T. Kishikawa, J. Tsurumi, T. Haga, Y. Ujiie, T. Sasaki, and H. Matsushima, "Supervised and unsupervised intrusion detection based on CAN message frequencies for in-vehicle network," *Journal of Information Processing*, vol. 26, pp. 306–313, 2018.

[3] J. Huang, M. Zhao, Y. Zhou, and C. C. Xing, "In-vehicle networking: Protocols, challenges, and solutions," *IEEE Network*, pp. 1–7, 2018.

[4] J. H. Kim, S.-H. Seo, N.-T. Hai, B. M. Cheon, Y. S. Lee, and J. W. Jeon, "Gateway framework for in-vehicle networks based on CAN, FlexRay, and Ethernet," *IEEE Transactions on Vehicular Technology*, vol. 64, no. 10, pp. 4472–4486, 2015.

[5] P. Carsten, T. R. Andel, M. Yampolskiy, and J. T. McDonald, "In-vehicle networks: Attacks, vulnerabilities, and proposed solutions," *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, 2015.

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

[7] K.-T. Cho and K. G. Shin, "Fingerprinting electronic control units for vehicle intrusion detection," *Proc. 25th USENIX Security Symposium*, pp. 911–927, 2016.

[8] N. Nowdehi, W. Aoudi, M. Almgren, and T. Olovsson, "CASAD: CAN-aware stealthy-attack detection for in-vehicle networks," N.D.

[9] W. Choi, H. J. Jo, S. Woo, J. Y. Chun, J. Park, and D. H. Lee, "Identifying ECUs using inimitable characteristics of signals in controller area networks," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 6, pp. 4757–4770, 2018.

[10] 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," *International Conference on Information Networking*, vol. 2016-March, pp. 63–68, 2016.

[11] W. Choi, K. Joo, H. J. Jo, M. C. Park, and D. H. Lee, "VoltageIDS: Low-level communication characteristics for automotive intrusion detection system," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 2114–2129, 2018.

[12] K.-T. Cho and K. G. Shin, "Viden: Attacker identification on in-vehicle networks," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1109–1123, 2017.

[13] M. Kneib and C. Huth, "Scission: Signal characteristic-based sender identification and intrusion detection in automotive networks," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 787–800, 2018.

[14] M. Wolf, A. Weimerskirch, and C. Paar, "Security in automotive bus systems," *Proceedings of the Workshop on Embedded Security in Cars*, no. July, pp. 1–13, 2004.

[15] Robert Bosch GmbH, "CAN specification version 2.0," 1991.

[16] M. Farsi, K. Ratcliff, and M. Barbosa, "An overview of controller area network," *Computing & Control Engineering Journal*, vol. 10, no. 3, pp. 113–120, 1999.

[17] F. Hartwich, Robert Bosch GmbH, "CAN with flexible data-rate," 2012.

[18] T. Kaur and S. Kaur, "Comparative analysis of anomaly based and signature based intrusion detection systems using PHAD and Snort," 2013.

[19] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[20] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks.," *Proceedings of LISA '99: 13th Systems Administration Conference*, vol. 99, no. 1, pp. 229–238, 1999.

[21] Daniel Cid, *OSSEC: Open Source HIDS SECurity*, 2019 (accessed February 10, 2019). [Online]. Available: `https://www.ossec.net/about/`.

[22] Tripwire, Inc., *Open Source Tripwire*, 2018 (accessed February 10, 2019). [Online]. Available: `https://www.tripwire.com/`.

[23] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, *et al.*, "Comprehensive experimental analyses of automotive attack surfaces.," *USENIX Security Symposium*, pp. 77–92, 2011.

[24] EU Commission, "Directive 98/69/EC of the European parliament and of the Council of 13 October 1998 relating to measures to be taken against air pollution by emissions from motor vehicles and amending Council Directive 70/220/EEC," *Official Journal L*, vol. 350, no. 28, p. 12, 1998.

[25] United States Environmental Protection Agency, Office of Transportation and Air Quality, *On-board diagnostic (OBD) regulations and requirements: Questions and answers (EPA-420-F-03-042)*, 2003.

[26] P. Subke, "Internationally standardized technology for the diagnostic communication of external test equipment with vehicle ECUs," *SAE Technical Paper*, 2014.

[27] S. Woo, H. J. Jo, and D. H. Lee, "A practical wireless attack on the connected car and security protocol for in-vehicle CAN," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 993–1006, 2015.

[28] K.-T. Cho, Y. Kim, and K. G. Shin, "Who killed my parked car?" *arXiv preprint arXiv:1801.07741*, 2018.

[29] W. Aoudi, M. Iturbe, and M. Almgren, "Truth will out: Departure-based process-level detection of stealthy attacks on control systems," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pp. 817–831, 2018.

[30] Raspberry Pi Foundation, *Raspberry Pi 3 Model B*, 2016 (accessed February 10, 2019). [Online]. Available: `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`.

[31] SK Pang Electronics Ltd, *PiCAN2 CAN-bus Board for Raspberry Pi*, 2016 (accessed February 10, 2019). [Online]. Available: `http://skpang.co.uk/catalog/images/raspberrypi/pi_2/PICAN2DSB.pdf`.

[32] Seeed, *CAN-BUS Shield V2.0*, 2018 (accessed February 10, 2019). [Online]. Available: `http://wiki.seeedstudio.com/CAN-BUS_Shield_V2.0/`.