

Securing Vehicular Networks: A Rules-Based CAN Intrusion Detection System Using IoT Edge Architecture

by

Taylor Lee

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2024

© Taylor Lee 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The increasing interconnectivity of modern safety-critical embedded systems has led to an ever-increasing attack surface. The automotive and maritime industries are but two industries that use safety-critical embedded systems. A common protocol used in both industries is the Controller Area Network (CAN) protocol, which has been proven to have multiple security flaws.

This thesis proposes a novel rules-based CAN Intrusion Detection System (IDS) to protect against possible attacks via the CAN protocol and alert end users in real-time. A rules-based approach was chosen due to the ability to dynamically adapt to the varying state of CAN messages. Previous rules-based implementations use a small number of rules, leading to the potential to misclassify incoming CAN messages. This thesis expands on previous implementations by proposing 16 established rules in total.

The proposed rules-based CAN IDS leverages an IoT (Internet of Things) architecture to provide centralised management of the IDS and to give the capability of deploying the IDS at scale. This thesis tests the proposed rules-based CAN IDS on two real-world systems that use the J1939 and NMEA 2000 protocols, with the primary testing performed on a 2016 Peterbilt 579 truck. Interesting observations from testing the rules-based CAN IDS found that manufacturers do not follow J1939 standards and a five-millisecond per message limitation in the Azure IoT Edge infrastructure.

Acknowledgements

I would like to thank Sebastian and the whole ESG group for a wonderful two (and a bit) years at UW, it's been a blast.

Dedication

This is dedicated to my parents, Glen and Tory, and my partner, Angela, for helping me stay sane while writing this.

Table of Contents

List of Figures	ix
List of Tables	x
List of Abbreviations	xi
1 Introduction	1
1.1 Related Work	2
1.2 Contributions	3
1.3 Organisation of the Thesis	4
2 Background Information	5
2.1 CAN Bus Protocols	5
2.1.1 CAN	5
2.1.2 J1939 and NMEA 2000	7
2.2 CAN Bus Security	9
2.2.1 Vulnerabilities	9
2.2.2 Attack Methodologies	10
2.3 Intrusion Detection	10
2.3.1 Intrusion Detection Systems	11
2.3.2 Intrusion Prevention Systems	11

2.3.3	CAN Intrusion Detection Systems	12
2.3.4	J1939 and NMEA 2000 Intrusion Detection Systems	13
2.4	Azure IoT Edge	14
2.4.1	Downstream Devices	15
2.4.2	IoT Edge	16
2.4.3	IoT Hub	16
3	Rules-Based CAN Intrusion Detection System	17
3.1	Data Collector	18
3.2	Decoder	18
3.3	Rules Engine	18
4	Architecture	21
4.1	Requirements	22
4.1.1	Functional	23
4.1.2	Non Functional	23
4.2	Functional View	24
4.2.1	Vehicle Under Test	25
4.2.2	Interceptor	25
4.2.3	Aggregator	28
4.2.4	Central service	31
4.2.5	Analyzer	32
4.3	Deployment View	33
4.3.1	Off Site	33
4.3.2	On Site	33

5	Case Studies	36
5.1	Case Study 1: NMEA 2000	36
5.2	Case Study 2: J1939	38
5.3	Throughput	43
5.3.1	Single Interceptor	44
5.3.2	Multiple Interceptors	46
5.4	Lessons Learnt	48
6	Conclusion	50
6.1	Future Work	51
	References	52

List of Figures

2.1	Diagram representing a CAN frame	6
2.2	Diagram representing an extended CAN frame	6
2.3	Diagram representing the J1939 message format	8
3.1	Diagram representing the high-level flow of the CAN IDS	18
4.1	Functional block diagram of the CAN IDS architecture	24
4.2	Deployment view of an off-site deployment	34
4.3	Deployment view of an on-site deployment	35
5.1	Diagram representing the NMEA 2000 case study setup	37
5.2	Graph showing the frequency of CAN messages from the NMEA 2000 setup	38
5.3	Diagram representing the J1939 case study setup	39
5.4	Graph showing the frequency of CAN messages from the passenger door controller	42
5.5	Graph showing the frequency of CAN messages from the OBDII port . . .	43
5.6	Violin graph showing the time it takes for the interceptor to process CAN packets	44
5.7	Violin graph showing the time it takes for the aggregator to process CAN packets	45
5.8	Violin graph showing the time it takes for the analyzer to process CAN packets	46

List of Tables

5.1	Table containing the expected frequencies, true frequencies and jitter percentage of PGNs in the NMEA 2000 setup	37
5.2	Comparison of PGNs by interceptor location	40
5.3	Comparison of PGNs by SA for the OBDII port	40
5.4	Comparison of PGNs by SA for the passenger door controller	41
5.5	Table containing the maximum message frequency per number of interceptors	47

List of Abbreviations

ACK Acknowledgement [7](#), [19](#), [26](#)

CAN Control Area Network [1–10](#), [12](#), [14](#), [15](#), [17–19](#), [21–23](#), [25–29](#), [32](#), [37–39](#), [41–51](#)

CRC Cyclic Redundancy Check [7](#), [19](#), [26](#)

CRF CAN Receive and Forward [28–30](#)

DLC Data Length Code [7](#), [19](#), [25](#)

EET Electromechanical Emissions Tripwire [21](#), [22](#), [29](#), [31](#), [50](#)

IDE Identifier Extension Bit [7](#), [19](#), [25](#), [26](#)

IDS Intrusion Detection System [2–5](#), [9](#), [11–14](#), [17](#), [20](#), [21](#), [23](#), [44](#), [50](#), [51](#)

IoT Internet of Things [3–5](#), [14–16](#), [21](#), [22](#), [27](#), [28](#), [30](#), [31](#), [34](#), [47](#), [48](#), [50](#)

IPS Intrusion Prevention System [11](#)

MQTT Message Queuing Telemetry Transport [15](#), [22](#), [28](#)

NMEA National Marine Electronics Association [3](#), [5](#), [9](#), [18](#), [19](#), [28](#), [36](#), [38](#), [41](#), [45](#)

NTP Network Time Protocol [27](#), [31](#), [41](#), [42](#), [48](#)

OBD On-board Diagnostics [1](#), [38](#)

PGN Parameter Group Number [8](#), [9](#), [13](#), [19](#), [20](#), [28](#), [29](#), [36](#), [37](#), [39–41](#)

PTP Precision Time Protocol [48](#)

RTR Remote Transmission Request [6](#), [7](#), [19](#), [26](#)

SPN Suspect Parameter Number [13](#), [18–20](#), [28](#), [29](#)

SRR Substitute Remote Request [7](#), [19](#), [26](#)

VUT Vehicle Under Test [25](#)

Chapter 1

Introduction

Modern automobiles are safety-critical systems composed of individual safety-critical embedded systems that communicate with one another via the use of the Control Area Network (CAN) protocol. This interconnectivity makes automobiles vulnerable to attacks such as a CAN injection attack on a 2021 Toyota RAV4, which led to thieves being able to steal the car without needing the key [1]. Another example of using interconnectivity as an attack vector is the CAN bus attack found by Miller and Valasek that disrupted the driving of a 2014 Jeep Cherokee via the On-board Diagnostics (OBD) port [2].

Safety-critical systems are systems whose failure could result in loss of life, significant property damage or damage to the environment. Examples of safety-critical systems include medical devices, aircraft, automobiles, weapons and nuclear systems [3]. An embedded system is a combination of computer hardware and software, which may contain additional components that are either mechanical or electronic [4]. Examples of embedded systems range from personal systems such as cell phones and digital watches to industrial systems such as cruise missiles and gas pumps.

A safety-critical embedded system is part of an overarching safety-critical system. The safety-critical system relies on the correctness of the logical results and the timing at which results from the safety-critical embedded system are produced [5]. For example, an automobile relies on the reliability of its by-wire brakes; the by-wire brakes need to bring the car to a halt in due time. If there was an issue with the logical results of the by-wire brake system, it could lead to the car not stopping in time and lead to damage to property damage or loss of life [6].

Safety in the context of safety-critical embedded systems is defined as functional safety. As defined by the IEC-61508 standard, the definition of functional safety is that any safety-

related system must work correctly or fail in a predictable way [7]. If they do not work correctly or fail unpredictably, it can lead to consequences such as loss of life or property damage [8]. Examples of safety-related failures in safety-critical embedded systems include Therac-25 a computer-controlled radiation therapy machine that delivered high doses of radiation to patients due to a race condition, Ariane 5 Flight 501 disintegrating on launch due to an integer overflow error and the Mariner 1 spacecraft diverting from its intended path on launch due to the onboard computer miscalculating the rocket’s trajectory [9].

Security in embedded systems aims to provide mechanisms to protect the system from all types of malicious behaviour [10]. Security is important in safety-critical embedded systems, as insecure safety-critical embedded systems have an increased potential to fail via additional attack vectors. For instance, Miller and Valasek disrupted the driving of a 2014 Jeep Cherokee via remote wireless communication [2], Costin and Francillon found a vulnerability that allowed a malicious user to take control of a wireless pyrotechnic firing system [11] and in 2010, Stuxnet, a 500KB computer worm, infected 14 industrial sites in Iran, with one site being a uranium enrichment plant which led to substantial damage to the Iranian nuclear program [12].

While both safe and secure safety-critical embedded systems aim to protect the system from failure, there is a distinct difference between the two. A safe system aims to protect against random faults, while a secure system aims to protect against intentional faults caused by an attacker. While both protect against failure, the way the failure is caused is different.

To prevent failures in safety-critical embedded systems, both safety and security need to be a priority when building these systems. An important part of ensuring the safety and security of a system is to detect when a random fault or intentional fault is occurring. Ideally, the random fault or intentional fault should be detected before it occurs to allow the system or user to react accordingly.

One way to detect both random and intentional faults is via an Intrusion Detection System (IDS). An IDS is a device or software that detects the presence of suspicious activities and generates alerts when they are detected [13]. This thesis proposes an IDS for the CAN bus, which is part of a safety-critical embedded system.

1.1 Related Work

CAN IDS need to be designed so they do not block any critical processes, as it can undermine the safety of the vehicle [14]. Previous CAN IDS approaches have looked at machine

learning but have been unfeasible due to factors such as driver behaviour and environmental factors [15]. Instead, rules-based approaches are seen as an optimal alternative for CAN IDSs. This is due to the ability of a rules-based approach to adapt to dynamic system behaviours like driver behaviour and environmental factors [16].

Previous CAN IDSs, from my knowledge, have never been tested on a real system in real-time and instead have either used stored data or a lab setup [15] [17]. This creates an interesting research gap to explore in designing a CAN IDS that can collect CAN bus data from a vehicle in real-time and raise an alert in adequate time if a CAN message is suspicious.

When designing a CAN IDS, it is important to make sure the alert mechanism of the IDS is separate from the CAN bus itself [17]. This is to make sure no critical processes are blocked, and the safety of the vehicle isn't impeded by the IDS. An example of an alert mechanism given would be an LED light that triggers upon an alert.

One issue with CAN is the ability to decode proprietary messages. For this reason, the J1939 and National Marine Electronics Association (NMEA) 2000 standards are preferred in CAN IDS research [17] [15] [18]. The J1939 and NMEA 2000 standards are publicly accessible therefore, J1939 and NMEA 2000 messages can be easily decoded in comparison to proprietary messages. For this reason, both the J1939 and NMEA 2000 standards will be used to test the proposed CAN IDS.

1.2 Contributions

The main goal of this thesis is to propose a rules-based solution for a CAN IDS that uses real-time CAN data. This expands on previous work by Rogers et al, which proposed a real-time rules-based CAN IDS, but it was only verified via dataset manipulation using a lab setup [17]. This thesis instead tests the CAN IDS on a 2016 Peterbilt 579 truck to verify the CAN IDS works outside a lab setup. This thesis makes the following contributions:

- A novel rules-based CAN IDS using Internet of Things (IoT) Edge architecture, which collects and analyses CAN bus data in real-time.
- An analysis of the rules-based CAN IDS on two separate CAN bus systems, including an actual vehicle.
- Testing of the rules-based CAN IDS to make sure its throughput meets the baud rate of the CAN protocol.

- Improvements on the data acquisition pipeline previously proposed by the embedded systems group in prior work [19].

1.3 Organisation of the Thesis

The structure of this thesis is organised as follows: Chapter 2 contains background information on CAN, CAN security, IDSs and IoT. Chapter 3 proposes a rules-based CAN IDS that collects and analyses data from a live CAN bus system. Chapter 4 describes an IoT based architecture for the rules-based CAN IDS proposed in Chapter 3. Chapter 5 consists of two case studies testing the proposed architecture on different CAN bus systems. Finally, Chapter 6 concludes the thesis and suggests possibilities for future work.

Chapter 2

Background Information

Before describing the proposed CAN IDS some background information is helpful. This chapter provides background information to help understand the context of the CAN IDS. Section 2.1 describes details about different CAN protocols. Section 2.2 explains the security issues faced by the CAN protocol. Section 2.3 gives a detailed overview of IDSs. Section 2.4 gives an overview of the IoT backbone leveraged in the architectural design in Chapter 4.

2.1 CAN Bus Protocols

CAN and its variants, such as NMEA 2000 and J1939, are a conventional way for a vehicle's components to communicate. The rest of this section describes how the CAN, J1939 and NMEA 2000 protocols work.

2.1.1 CAN

CAN is a data bus protocol developed by Bosch in 1986 and is used by most modern automobiles [20]. It allows components in the vehicle, such as ECUs, to broadcast information in the form of CAN packets to communicate with other components in the system.

A CAN packet contains a CAN frame that can come in four types [21]: a data frame, a remote frame, an error frame or an overload frame. This thesis will focus on the data frame because it is the only frame for data transmission and is, therefore, used by the majority of attackers to perform CAN bus attacks.

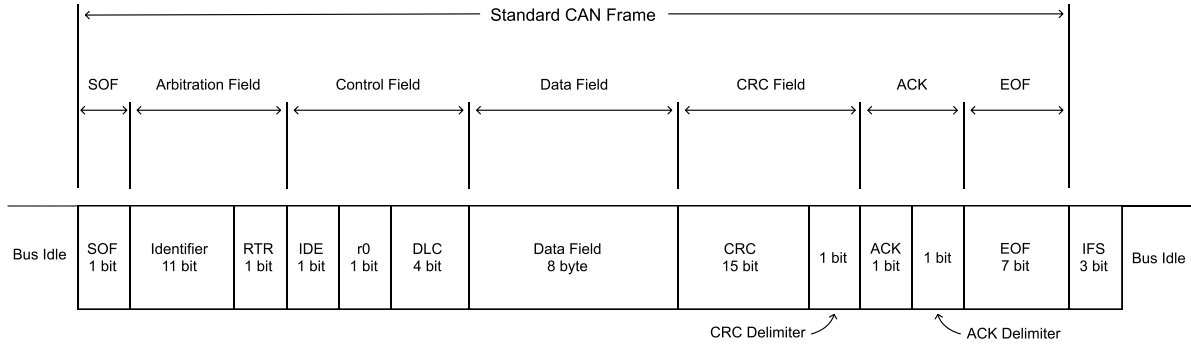


Figure 2.1: Diagram representing a CAN frame

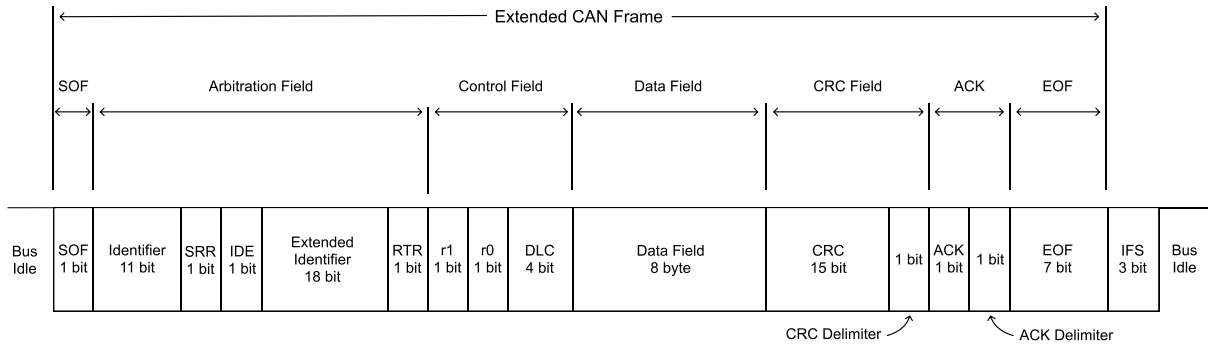


Figure 2.2: Diagram representing an extended CAN frame

A CAN data frame has a defined structure that differs based on the identifier length in the arbitration field. The identifier is 11 bits long for standard CAN frames, while the identifier is 29 bits long for extended CAN frames. Figure 2.1 shows the format for a standard CAN frame, and Figure 2.2 shows the format for an extended CAN frame. Each field in both frames is described below:

- Start of Frame (SOF): Denotes the start of a CAN data frame via a dominant bit.
- Arbitration Field: Contains the identifier and the Remote Transmission Request (RTR) bit. The identifier represents the frame's priority and is used to decode the

data field, while the RTR bit distinguishes between data (0) and remote (1) frames. The arbitration field for extended frames also contains a Substitute Remote Request (SRR) bit.

- Control Field: Contains the Identifier Extension Bit (IDE) and Data Length Code (DLC). The IDE is set to 0 if an 11-bit identifier is used, while it is set to 1 if a 29-bit identifier is used. The DLC, on the other hand, is used to determine the size of the data field.
- Data Field: This field contains the data to be transmitted, which is then decoded by CAN nodes to perform actions. The data field can be any length from 0 to 8 bytes and is only used by the data frame.
- Cyclic Redundancy Check (CRC) Field: The CRC field provides the CAN frame with a checksum. It is 15 bits in length and checks for packet validity.
- Acknowledgement (ACK) Field: The ACK field ensures that receiver nodes have received the CAN frame correctly. This is done via the ACK slot bit, which is set as recessive (1) by transmitters, and the receivers set the ACK slot to dominant (0) once received. If the transmitter doesn't read a dominant bit (0) in the ACK slot, it detects an ACK error and will attempt to send the frame again.
- End of Frame (EOF): Denotes the end of a CAN data frame via seven recessive bits.

This thesis will focus on the data frame's identifier and data field sections. The ID determines the message type, and the data field contains the data that CAN nodes need to perform actions. The other sections of the data frame tell us little about the content of the CAN packet, so they will not be discussed.

2.1.2 J1939 and NMEA 2000

J1939 is a standard developed by SAE and is the accepted industry standard for vehicle network communication in heavy-duty vehicles such as trucks [22]. J1939 is a high-layer protocol based on CAN, with CAN providing the physical and data-link layers for the J1939 protocol. J1939 uses 29-bit extended CAN IDs when sending messages; the J1939 standard describes how to interpret the 29-bits as seen in Figure 2.3. The 29 bits of the CAN ID can be broken down into three components as listed below:

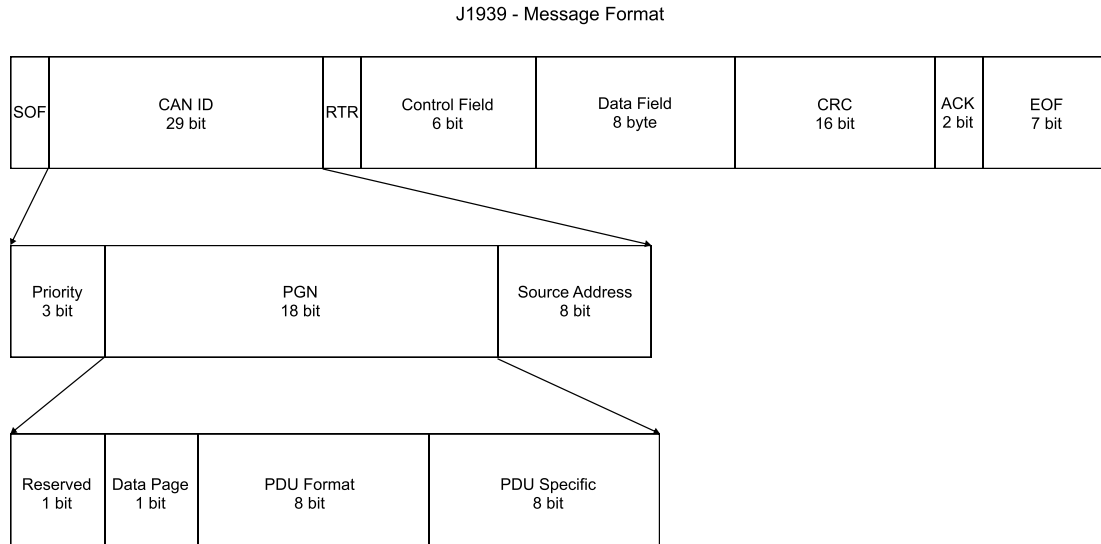


Figure 2.3: Diagram representing the J1939 message format

- Priority: 3-bits long and controls a message's priority during the arbitration process. The priority can range from zero to seven, with zero as the highest priority and seven as the lowest priority.
- Parameter Group Number (PGN): 18-bits long and serves as a unique identifier for the CAN frame. For example, PGN 61444 in the J1939 standard is the Electronic Engine Controller 1.
- Source Address: 8-bits long and contains the device's address that sent the CAN frame. Each device on the J1939 network has a unique source address with 254 different addresses available.

The 18 bits of the PGN can be broken down further as seen in Figure 2.3; the breakdown is as follows:

- Data Page and Extended Data Page: The data page and extended data page are both a bit long in size and allow the selection of different data pages. This effectively quadruples the total number of different parameter groups.

- PDU Format (PF): 8 bits long and determines if the message is a PDU1 (peer-to-peer) or a PDU2 (broadcast) message. If the value of the PDU format is less than 240, the message is a PDU1 message, and if the value of the PDU format is greater than or equal to 240, then the message is a PDU2 message.
- PDU Specific (PS): 8 bits long, and its meaning changes depending on if the message is a PDU1 or PDU2 message. For PDU1, the PS field represents the message's destination address, while for PDU2, the PS field represents a group extension for the PGN and is a global message broadcast to all nodes on the CAN bus.

NMEA 2000 is a standard for the maritime industry based on the J1939 protocol and uses the same 29-bit extended format [23]. The NMEA 2000 protocol allows communication between different components on boats and can be integrated with J1939. For example, J1939 would be used to send messages from the engine while NMEA 2000 would be used to send messages from sensors and the GPS present on the ship.

Unlike proprietary CAN messages, both J1939 and NMEA 2000 CAN messages follow a set standard and can be universally decoded. For this reason, the IDS proposed in the thesis will be trained on systems that use the J1939 and NMEA 2000 protocols.

2.2 CAN Bus Security

The CAN protocol has been proven to be vulnerable to attacks multiple times. For instance, Miller and Valasek steered a Jeep Cherokee off the road by targeting a flaw in the automatic park feature via the CAN bus [24]. Vulnerabilities were also found in the 2021 Toyota RAV4, which led to thieves being able to steal the car without needing the key via a CAN injection attack [1]. This section describes the vulnerabilities with the CAN protocol and the attack methodologies that are used to take advantage of the vulnerabilities.

2.2.1 Vulnerabilities

The CAN protocol was designed to minimise the need for wiring in an automotive system. When the CAN protocol was proposed, security wasn't considered. Due to this, several vulnerabilities are present in the CAN protocol. The vulnerabilities with the CAN protocol are as follows [25]:

- Lack of encryption: The CAN messages present on the CAN bus can be read by any node in the network.
- Lack of authentication: A node in a CAN network can impersonate any other node present in the network. This is due to CAN messages carrying the identity of the node.
- Message priority: The priority of CAN messages is determined by the unique identifier, with a lower identifier signalling a higher priority. This opens nodes up to flood attacks via CAN messages with lower identifiers, which take precedence over CAN messages with higher identifiers.
- Broadcast Communication: Any node on the CAN network can receive messages from any other nodes present on the CAN network. Any node on the CAN network can also send messages to any other nodes present on the CAN network.

2.2.2 Attack Methodologies

Nodes in a CAN network can be attacked in various ways. This section discusses the ways:

- Frame Sniffing: Due to a CAN network using broadcast communication, the CAN frames sent on the network can be sniffed by any node on the network. By analysing the CAN frames, functionalities of the components on the CAN network can be known [26].
- Frame Injection: Since nodes don't need authentication, a malicious node can inject CAN frames onto the CAN network [25].
- Dominant-Bit Assertion: By using the message priority function of the CAN protocol, a malicious user can set a dominant bit in the CAN message to stop messages from other nodes being processed [25].

2.3 Intrusion Detection

One way to help solve the security issues with the CAN protocol is via intrusion detection. Intrusion detection alerts users to possible intrusions on the CAN network by detecting suspicious activities [13]. The rest of this section describes the differences between intrusion detection and prevention systems and details previous work on CAN, J1939, NMEA 2000 and rules-based intrusion detection systems.

2.3.1 Intrusion Detection Systems

An IDS is a device or software that detects the presence of suspicious activities and generates alerts when they are detected. An IDS can detect intrusion activity in the following ways [13]:

- Signature-based: Signature-based IDS solutions use the fingerprint of known threats to identify and detect said threats. Once a known threat has been identified, it is added to the IDS list of known threats to be used for detection purposes. Signature-based detection has high detection rates with few false positives but is vulnerable to zero-day attacks and limited to detecting known attacks.
- Anomaly-based: Anomaly-based IDS solutions establish the expected behaviour of a model. This expected behaviour is then used as a baseline to detect any anomalies that differ from the baseline. While anomaly-based detection allows the detection of zero-day attacks and unknown attacks, anomaly-based detection can be vulnerable to false positives and negatives. So, reducing the false positive and false negative rates is extremely important when designing anomaly-based detection solutions.
- Hybrid-based: Hybrid-based IDS solutions use signature-based and anomaly-based detection to detect intrusions. Using signature-based and anomaly-based detection allows the IDS to have a lower error rate than the other detection methods in isolation.

2.3.2 Intrusion Prevention Systems

An Intrusion Prevention System (IPS) is similar to an IDS, as they both detect threats. However, unlike an IDS, an IPS takes action to block or remediate a threat when a threat is identified. To note is that an IPS can raise alerts, but its main goal is to prevent intrusions from occurring altogether [27]. While an IPS may make an IDS seem redundant due to dealing with threats automatically as they occur it may not be the best solution for every system. For instance, take a system with high availability requirements, such as an Industrial Control system or an automobile. The most essential requirement is to keep these systems running and blocking suspicious traffic via an IPS may impact the performance of the systems. Instead notifying a human operator via an IDS would be the best course of action as they can evaluate the current situation and can take action if necessary.

Safety-critical embedded systems also fall under this banner, meaning that due to their high availability requirements, an IDS solution is the preferred option over an IPS

solution. This thesis focuses on the security of safety-critical embedded systems, specifically automotive security, and therefore proposes an IDS solution.

2.3.3 CAN Intrusion Detection Systems

Conventional IDS undermine the safety of complex applications. Meaning a conventional IDS could erroneously block critical processes in safety-critical systems. As CAN is used in safety-critical systems, a CAN IDS must not be designed to block any critical processes [14].

Like conventional IDS, CAN IDS use the three approaches of anomaly, signature and hybrid. However, a fourth approach is also used, with that being specification-based [28]. Specification-based IDS detect attacks whenever a network's expected behaviour diverges from designated specifications or standards. For example, if a CAN bus is stated to use the J1939 standard, but the CAN IDS picks up a CAN message that diverges from the standard, it could be a potential intrusion.

Kang et al, proposed a deep neural network for a CAN IDS that generated 200,000 packets for the OCTANE simulator [29]. 100,000 of the generated packets were labeled as normal packets, while the other 100,000 generated packets were labeled as attack packets. 70% of the packets were used for training and the other 30% were used for testing. The results from the training lead to a false positive rate of 0.8% and a false negative rate of 1.4%. Overall giving the model an accuracy of 97.8%. While the model has a high accuracy rate it is not feasible for safety-critical systems as 1320 packets were incorrectly categorised and if just one of these packets go undetected it can lead to catastrophic failure. This model was also only trained on one specific simulator, meaning it was never implemented in a physical system.

Boumiza et al, proposed a CAN IDS based on a multi-layer perceptron neural network [30]. The IDS works by dividing data according to the CAN ID field using K-means clustering and trains each CAN ID separately, with the main aim of the IDS to detect frequency-appearance modification and data-content modification for each ID. The dataset used for this was 24 hours of CAN traffic from a 2012 Subaru Impreza captured over several months, this dataset had 20 IDs with 15 IDs used during testing and training. This means the model will have to be retrained if used on a new vehicle and due to 24 hours of data being used in the original model, it is to be seen how much data would be needed from the new vehicle to create an accurate model.

2.3.4 J1939 and NMEA 2000 Intrusion Detection Systems

J1939 and NMEA 2000 IDS need special attention due to the existence of two specific procedures in the standard, that being address claims and multi-frame transmission. Jichici et al [31], proposed a two-layered approach, the first layer checks for the validity of the source and destination addresses, this is achieved by re-mapping the source and destination addresses to encrypted values that are known to legitimate nodes on the network. The second layer then detects anomalies within the data field, this is done by encrypting the data field to stop attackers deducing the structure of the data field. If an intrusion is detected by either of the two layers, the J1939 frame is destroyed with error flags. As this solution involves re-mapping and encryption it cannot be placed on an already existing system as each node on the network would have to understand the encryption protocol. This makes this solution only feasible if an IDS is considered in the system's design from the start.

Shirazi et al, proposed a J1939 machine learning based-IDS that detects intrusions using a PGN's Suspect Parameter Number (SPN) values [18]. The machine learning algorithm uses the history of the SPN value along with the average, standard deviation, and slope of the last n values to determine if an intrusion is present. The machine learning model was trained using five previously captured logs from a truck, each log was then converted to multiple datasets with each dataset containing data on one PGN. The PGNs used in the datasets had at least 500 messages, meaning if an attack happens on a PGN which has less than 500 messages it would go undetected. The IDS also fails to consider J1939 messages that weren't decoded successfully, as it fully relies on decoded SPN values as the only way to detect intrusions.

Mukherjee et al, states that machine learning J1939 IDS approaches are infeasible as in-vehicular communication traffic depends largely on internal mechanics such as the driver's behaviour and environmental factors [15]. Meaning an IDS trained on a sample dataset gained from one driving condition may not be effective in detecting attacks under different driving conditions.

Rogers et al, proposed a rules-based IDS that detects attacks on both J1939 and NMEA 2000 networks [17]. The proposed IDS uses both timing analysis and data analysis to detect the presence of attacks. The timing analysis detects J1939 and NMEA 2000 messages with irregular timing. The data analysis allows the detection of J1939 and NMEA 2000 messages that transmit incorrect IDs or data, this is done by comparing the output of a data ECU with the flow of data over the bus and if they do not match there is an anomaly.

Rogers et al, also states that when an intrusion is found and an alert is triggered, any communication of the alert to the vehicle operator must be done through an external

mechanism separate from the compromised CAN bus [17]. The example provided by Rogers is a simple LED light that triggers upon an alert, though they do state an external alert mechanism with more context could be more helpful than the proposed LED light.

While Rogers et al's rules-based approach solves issues faced with previous machine learning approaches, the proposed IDS was never implemented on an actual vehicle due to safety reasons. This thesis fills a gap in the field by proposing a rules-based IDS that collects CAN data in real-time from a vehicle and raises external alerts when an anomaly is detected.

2.4 Azure IoT Edge

One issue with designing a system to collect real-time CAN data is the ability to capture data across multiple CAN buses at the same time. One way to achieve this is via an IoT approach that allows the aggregation of data from multiple sensors [32]. Each sensor can be connected to an individual CAN bus, which sends the data from the sensor upstream for later processing. Due to this, the architecture proposed in this thesis uses IoT in its foundation.

Azure IoT Edge is an internet of things service developed by Microsoft that allows the management of devices from the cloud [33]. Azure IoT Edge is made up of three primary components, which are listed below:

- **Downstream Devices:** Downstream devices are embedded systems with little power and processing resources that collect telemetry data. An example would be a temperature sensor in a weather station. Downstream devices can upload their data upstream to an IoT Edge instance which processes the data.
- **IoT Edge:** IoT Edge devices run general-purpose operating systems such as Linux or Windows, which have more processing power than downstream devices. This gives machine learning and data analysis the ability to run on the IoT Edge.
- **IoT Hub:** The IoT Hub manages both downstream devices and IoT Edge by establishing connections between downstream devices and IoT Edge instances as well as providing remote commands to both downstream devices and IoT Edge.

Downstream devices have a one-to-many relationship with the IoT Edge. This helps with scalability as a new IoT Edge device does not have to be setup for each individual

downstream device. Instead, a single IoT Edge device can process data from multiple downstream devices. This allows a single IoT Edge device for a single vehicle with aggregation of CAN data from multiple CAN buses inside the vehicle by placing a downstream device on each individual CAN bus.

The rest of this section talks about the three Azure IoT Edge components in detail.

2.4.1 Downstream Devices

Downstream devices communicate with the IoT Edge using Message Queuing Telemetry Transport (MQTT), which is a standard messaging protocol for IoT [34]. Downstream devices publish MQTT messages to topics, for example a temperature topic which is then subscribed to by an IoT Edge instance via the use of an MQTT broker. Anytime a message is published to the subscribed topic the IoT Edge reads the message and processes it accordingly. Overall the use of MQTT means that limited messages are dropped between the IoT Edge and downstream devices, meaning a consistent stream of data is available to the IoT Edge for processing.

One feature of downstream devices are device twins which allow the configuration of a downstream device remotely. Device twins consist of three features that are listed below:

- **Tags:** Tags can only be read and wrote from the IoT Hub but contain important information about the downstream device, such as the serial number or device model which can be used for device categorisation.
- **Desired Properties:** Desired properties are set by the IoT Hub and read by the downstream device with the downstream device using the desired properties for configuration purposes. An example of a desired property would be a boolean value to enable or disable data transmission.
- **Reported Properties:** Reported properties are set by the downstream device and represent the configuration present on the actual device. This allows the end user to confirm the configuration of the device was successful.

A feature similar to device twins but primarily used for one-off communications are direct methods. An example of a direct method would be a reboot command sent to the downstream device. Another feature of downstream devices is device updates which allow a user to deploy over-the-air updates to a downstream device.

Overall device twins, direct methods and device updates help with the maintainability of downstream devices and means physical access to the downstream devices once set up is rarely required.

2.4.2 IoT Edge

The IoT Edge has many of the same features as a downstream device but differs in its approach. Instead, the IoT Edge uses a module based approach compared to the device based approach present in downstream devices.

Modules are containerised microservices present on the edge that communicate with one another via the Edge Hub module. Modules are containerised via the use of the docker runtime. The Edge Hub module and the Edge Agent module are present on all IoT Edge instances. The Edge Hub acts a bridge between all modules on the IoT Edge as well as the downstream devices attached to the IoT Edge. The Edge Agent on the other hand is responsible for module deployment and management.

Apart from the Edge Hub and Edge Agent users can create their own custom modules that run their own code such as a module for processing data from downstream devices. The benefits of this containerised approach via the use of modules means the IoT Edge can be ran on multiple different systems with a small amount of setup.

Similar to device twins present in downstream devices modules have twins of their own called module twins but instead of setting configurations for the device itself the module twins allow the configuration of each individual module. This like the device twins present in downstream devices improve maintainability of the IoT Edge as physical access is rarely required after initial setup.

2.4.3 IoT Hub

The IoT Hub is the central hub for both the IoT Edge and downstream devices that monitors, manages and controls IoT Edge and downstream devices. One important feature of the IoT Hub is device provisioning that automatically registers an IoT Edge or downstream device as soon as it comes online, this helps greatly with scalability due to there being no need to manually register a device.

The IoT Hub can also update the desired properties of a module or device twin, as well as invoke direct methods and device updates to a downstream device. Predominantly the IoT Hub is the central base for all downstream device and IoT Edge instances allowing the remote management of millions of IoT devices.

Chapter 3

Rules-Based CAN Intrusion Detection System

Previous work with CAN IDSs found that machine learning approaches were not feasible [15]. Instead, rules-based approaches were found to be a more preferable option [17]. Previous rules-based approaches focused on generating rules for timing, identifiers and signal values.

This thesis expanded on previous rules-based approaches by proposing additional rules to generate a more practical system. The thesis also proposes a rules-based CAN IDS that works with real-time data while previous work has only worked with stored data [15] [17].

A rules-based IDS uses predictive rule learning to help detect the presence of anomalies. Predictive rule learning automatically updates rules to accommodate dynamic system behaviours. Predictive rule learning also helps achieve higher interpretability, due to symbolic representation of learned rules [16].

Figure 3.1 shows the high-level flow for the proposed rules-based CAN IDS. It is primarily made up of three separate steps: acquisition, decoding and analysing. First, we collect data from a vehicle under test using a data collector. The data collected from the data collector is then sent to the decoder to be decoded. Finally, once the decoder has decoded the data it sends the decoded data to the rules engine for analysis, which then raises a flag if it detects an anomaly.

The rest of this chapter talks about the data collector, decoder and rules engine components in more detail.



Figure 3.1: Diagram representing the high-level flow of the CAN IDS

3.1 Data Collector

The data collector attaches to the CAN bus on a vehicle under test. It reads CAN electrical signals and translates them to CAN data frames. Once a CAN electrical signal is processed, the data collector forwards the resulting CAN data frame to the decoder.

3.2 Decoder

The decoder receives a CAN data frame from the data collector and produces the signal values for the received data frame. In the case of J1939 and NMEA 2000 the signal values are the data frame’s SPN values. Once the signal values for the data frame are produced, the signal values along with the data frame are sent to the rules engine for analysis.

The decoder produces the signal values via the use of a CAN DBC file. A DBC file is a text file that contains information for decoding raw CAN bus data to physical values [35]. DBC files are mostly proprietary, except for J1939 and NMEA 2000 DBC files which are publicly accessible. For this reason, this thesis primarily focuses on J1939 and NMEA 2000 data frames as they can be easily decoded.

3.3 Rules Engine

The rules engine receives the decoded data frames from the decoder and produces a set of rules using the data frames. The rules engine has two states, one state being a training state and another being an active state.

The training state is used when the rules engine is producing new rules for the vehicle under test. During training it is expected for the vehicle under test to be in a ”normal”

operative state. This means all CAN traffic seen on the bus follows the manufacturer's expected behaviour.

The active state is used when the vehicle is actively protected. During the active state all new data frames are compared to the rules generated during training. If any information from data frames differ from what was established during the training state an anomaly flag is raised.

The rules engine generates a number of rules. For example, a PGN rule based on the PGNs saw during training. If the rules engine is in an active state and a PGN differs from what was experienced during training, an anomaly flag will be raised. Previous work only focused on establishing rules for PGNs, SPNs and message frequency [17]. This thesis expands on previous work by establishing 13 additional rules, for 16 rules in total. The various rules established for a J1939 or NMEA 2000 message sent to the rules engine are listed below:

- PGN: The PGN must have been seen during training.
- SPN: The SPN must have been seen during training.
- Frequency: The time between each occurrence of a PGN and source address combination must match what was seen during training.
- Source Address: The PGN and source address combination must have been seen during training.
- Destination Address: The PGN and destination address combination must have been seen during training.
- Reserved and DataPage: The Reserved bit must equal zero for both J1939 and NMEA 2000 messages. The data page must equal zero for a J1939 message and one for a NMEA 2000 message.
- DLC: The DLC for each PGN and source address combination must have been seen during training.
- Priority: The priority for each PGN and source address combination must have been seen during training.
- Formality: Checks the SRR equals one, the IDE equals one, the RTR equals zero, both reserved bits equal zero, the CRC delimiter equals one, the ACK slot is either zero or one, and the ACK delimiter is one.

- Bit Stuffing: A value of zero means bit stuffing was successful. A value of one means bit stuffing was unsuccessful and violates this rule.
- Max SPN Value: The maximum value seen for an SPN during training. If this value is exceeded when the rules engine is in an active state the rule is violated.
- Min SPN Value: The minimum value seen for an SPN during training. If this value is subceeded when the rules engine is in an active state the rule is violated.
- SPN Positive Rate of Change: The positive rate of change for an SPN between occurrences of a PGN and source address combination. If this value is exceeded when the rules engine is in an active state the rule is violated.
- SPN Negative Rate of Change: The negative rate of change for an SPN between occurrences of a PGN and source address combination. If this value is subceeded when the rules engine is in an active state the rule is violated.
- Availability: The availability of an SPN that can be either true or false. The availability cannot differ from what was seen during training.
- State Value: The state of a non-linear SPN such as the current gear of a vehicle's transmission. Every time a new state value is seen during training it is saved in a list. This rule is violated if during an active state, a state value is seen that is not present in the list.

If any of these rules are violated when the rules engine is active, an anomaly flag will be raised. By establishing a greater number of rules compared to previous work [17], we aim to create a more practical rules-based IDS that can help detect intrusions in real-world systems.

Chapter 4

Architecture

This chapter proposes an architecture for the rules engine described in chapter 3. The architecture proposed in this chapter builds on previous work by the Embedded Systems Group and Palitronica Incorporation with the Electromechanical Emissions Tripwire (EET) project [19].

Architecture, in the context of software, is the set of structures needed to reason about a system. Structures comprise software elements, the relations among them and the properties of both [36]. One issue with designing systems is the presence of abstract goals that can be interpreted differently by individuals. Defining an architecture with structures is important as it removes the abstract aspects of system design and instead translates them to formalised structures. Therefore, we can say architecture provides a formal translation of the abstract goals defined for a system.

Defining an architecture also requires formalising the abstract goals of the project in the form of requirements. Requirements can be either technical or non-technical and help inform the architect about the main objectives of the project. Previous studies have shown that defining requirements is critical in establishing a successful system architecture [36].

The architecture proposed for the EET project was designed for an IDS that used power consumption data from multiple devices in real-time. As our CAN IDS also requires processing CAN bus data from multiple CAN bus in real-time, the architecture proposed for EET will lay the foundations for the architecture proposed in this chapter. The main difference between the previous architecture and the architecture proposed in this thesis is the change from a decentralised approach to an IoT approach.

The two main flaws found with the architecture for EET were maintainability and reliability due to the architecture's decentralised approach [19]. Maintainability requires

the system to have the capabilities to be easily updated, modified and maintained over time. Reliability requires the system to perform its functions consistently and accurately over time [37]. To improve maintainability and reliability, a more centralised approach in the form of the Azure IoT Edge was chosen as a foundation for the new architecture. Features of the Azure IoT Edge, such as over-the-air updates and the use of the MQTT protocol, aim to fix the maintainability and reliability issues faced with the EET architecture.

The architecture for EET featured three components called the interceptor, aggregator and analyzer. The interceptor is responsible for the collection of power consumption data from a device. The aggregator is responsible for transferring and processing data between the interceptor and analyzer. The analyzer is responsible for analysing the data received from the aggregator.

The architecture proposed in this chapter uses the three components from the EET project but adds an additional component called the central service. The role of the central service is to manage the interceptor and aggregator via the use of over-the-air commands and updates. The central service aims to fix the maintainability and reliability issues faced with the EET architecture.

Chapter 3 described three components called the data collector, decoder and rules engine. The three components relate to the architecture proposed in this chapter by acting as the interceptor, aggregator and analyzer. The data collector is the interceptor and is responsible for collecting CAN data from a CAN bus. The decoder is the aggregator and is responsible for receiving CAN data from the interceptor, decoding the data and then forwarding it to the analyzer. The rules engine is the analyzer and is responsible for analysing the decoded data from the aggregator and raising an alert if an anomaly is detected.

The first section of this chapter outlines the architecture's requirements. The second section provides a function view of the architecture. Finally, section three gives a deployment view of the architecture.

4.1 Requirements

When designing an architecture for a system defining requirements is essential [36]. Requirements can be placed into two categories which are functional and non-functional requirements [37]. Functional requirements specify how a system behaves and the use cases of the system. Non-functional requirements define how a system should perform the use cases and the functionality.

The following subsections define the functional and non-functional requirements of the proposed rules-based CAN IDS.

4.1.1 Functional

Functional requirements define the standard functions and behaviours of a system [37]. The functional requirements are detailed as follows using the abbreviation FR.

FR-1 *The system shall support capturing data from multiple CAN buses.*

FR-2 *The system shall provide mechanisms to control and manage the data collection process and data storage.*

FR-3 *The system shall support attaching meta information to data.*

FR-4 *The system shall provide access to performance metrics of the pipeline and storage.*

FR-5 *The system shall provide role-based access control to data and management functions.*

FR-6 *The system shall support seamless updates of components.*

FR-7 *The system shall provide non-intrusive access to the CAN bus.*

FR-8 *The system shall provide real-time analytics on the state of the CAN bus.*

FR-9 *The system shall provide the ability to automatically configure rules.*

FR-10 *The IDS can be run with previously captured CAN bus data.*

4.1.2 Non Functional

Non-functional requirements of a system focus on items concerning the performance of a system [37]. Unlike functional requirements, non-functional requirements do not focus on the behaviours of the system.

NFR-1 *The system shall support capturing data from a minimum of two CAN buses simultaneously.*

NFR-2 *The throughput of the system should handle the maximum CAN busload.*

NFR-3 *Over-the-air commands and updates shall trigger successfully within 3 minutes.*

4.2 Functional View

Formalising a proposed architecture is essential in providing a mutual understanding between all stakeholders [36]. In the case of this thesis, the architecture has been formalised in the form of a functional block diagram. The functional block diagram is featured as Figure 4.1.

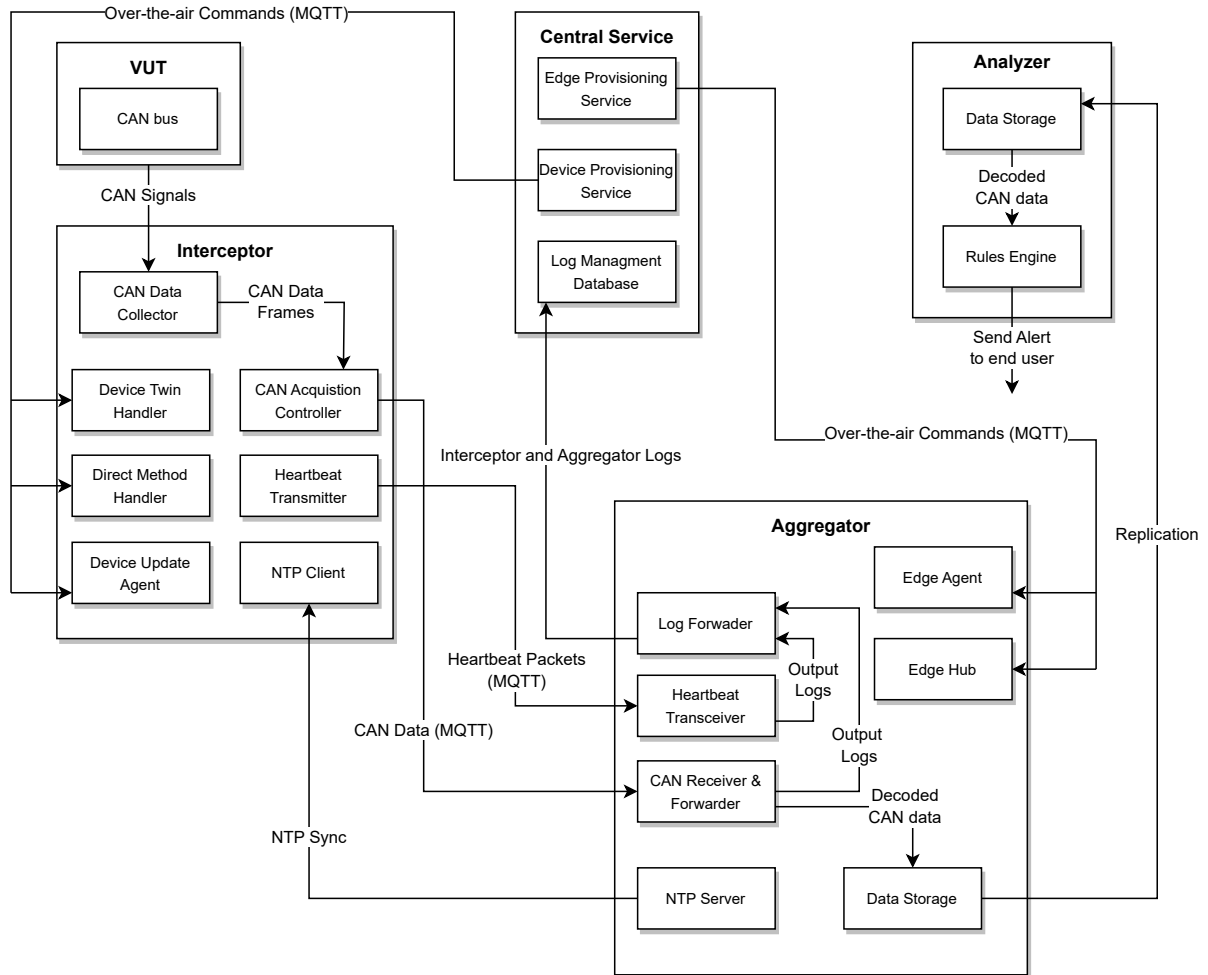


Figure 4.1: Functional block diagram of the CAN IDS architecture

Figure 4.1 contains five components called Vehicle Under Test (VUT), interceptor, aggregator, central service and analyzer. The following subsections explain each of the components in detail.

4.2.1 Vehicle Under Test

The vehicle under test is the target vehicle from which CAN bus data is collected. The CAN bus in the vehicle is connected to the interceptor, as seen in Figure 4.1.

When an interceptor is connected to the vehicle under test, it should be done in a non-intrusive manner, as stated in FR-7. This is achieved via the use of a contactless J1939 reader [38] and keeping the interceptor in a listen-only state.

4.2.2 Interceptor

The interceptor is responsible for collecting data from CAN buses in the vehicle under test and forwarding it to the aggregator. The Interceptor consists of various sub-components as seen in Figure 4.1. The sub-components are described in detail in this section.

CAN Data Collector

The CAN data collector is an embedded system that reads CAN data from the CAN bus and forwards the CAN data to the CAN acquisition controller. The CAN data collector is made up of three components linked together: a CAN transceiver chip, an opto-isolator and an FPGA.

First, the CAN bus is read via a CAN transceiver chip with the CAN transceiver chip set to passive mode meaning no CAN messages can be sent to the CAN bus. The CAN transceiver chip is connected to the opto-isolator to ensure full electrical isolation of the system from the CAN bus with the isolated signal sent to a pin on the FPGA.

The FPGA reads the isolated signal and triggers when the first falling edge that marks the start bit of a CAN message occurs. The FPGA then calculates the desired bit-sample position using a given baud-rate. From here, the FPGA checks the IDE to find if the CAN message is standard or extended CAN and checks the DLC to get the length of the data field. Finally, using the IDE and DLC values, the FPGA reads the bits on the bus and reconstructs the CAN data frame, which is then stored in local registers.

CAN Acquisition Controller

The CAN acquisition controller reads the CAN data frame stored in the FPGA's local registers and generates a timestamp for it. For each CAN data frame, the CAN acquisition controller contains the following data:

- Timestamp: The timestamp the CAN data frame occurs at.
- CANID: The 11-bit or 29-bit identifier of the CAN data frame.
- DLC: The data length code of the CAN data frame.
- Databytes: The 0-8 bytes of data for the CAN data frame.
- Extrabits: A byte that contains the following data for the CAN data frame: SRR, IDE, RTR, Reserved bit 0, Reserved bit 1, CRC delimiter, ACK slot and ACK delimiter.
- Stuffingbyte: A byte that signals if bit stuffing has been performed without errors, all CAN data frames should have the stuffingbyte equal to zero, otherwise a stuffing error has occurred.

Once the CAN acquisition controller has generated a CAN data frame, it stores the CAN data frame in memory and increases a buffer counter by one. The buffer counter limit is set to a value of 68. Once the buffer counter limit is reached, the CAN data frames in memory are prepared for transmission to the aggregator. Once the CAN data frames are forwarded to the aggregator the buffer counter is reset and the process starts again.

Heartbeat Transmitter

The role of the heartbeat transmitter is to check the connectivity of the interceptor. It aligns with [FR-2](#) as it helps us manage the data collection pipeline. The heartbeat works by sending a heartbeat packet to the interceptor every 10 seconds. This is to assure the messaging routing for Azure is correct and shows the interceptor has an active internet connection.

NTP Client

One important thing the interceptor needs to do is timestamp each CAN message accurately. Due to the possibility of time drift on the interceptor, a form of time synchronisation is needed. To help with time synchronisation, Network Time Protocol (NTP) is used. NTP is a protocol that syncs an interceptor's clock with that of an NTP server.

In the case of the proposed architecture, the NTP server is located on the aggregator and is polled every 60 seconds by the NTP client on the interceptor. By time syncing we make sure the timestamps are in real-time therefore addressing [FR-8](#).

Device Twin Handler

The interceptor is a downstream device in the context of the Azure IoT Edge and hence can be configured via the use of the device twin. The device twin handler's purpose is to process the device twin configuration. Features that are configured via the device twin are enabling or disabling CAN transmission, changing the CAN bus baud rate and changing the NTP server hostname. By configuring values at runtime the device twin handler addresses [FR-2](#).

Direct Method Handler

Similar to the device twin, the interceptor can be configured via the use of direct methods. Direct methods, unlike device twins, are rarely used but help with maintainability. Features that are configured via direct methods include soft resetting the interceptor and updating the hostname of the aggregator the interceptor is connected to. By configuring values at runtime the direct method handler addresses [FR-2](#).

Device Update Agent

The interceptor uses the device update feature of the Azure IoT Edge to provide over-the-air updates. The device update agent handles the incoming updates and flashes the interceptor with new firmware. Over-the-air updates improve maintainability and help with [FR-6](#).

4.2.3 Aggregator

The aggregator handles the incoming data from the interceptors and prepares the data for analysis. The aggregator is also designed to handle data from multiple interceptors as defined in [FR-1](#). The IoT Edge runtime is used to provision the aggregator. As the IoT Edge runtime requires a general-purpose operating system to run, we selected Debian 11 due to its support by Microsoft. Multiple components make up the aggregator, which are detailed in this section.

CAN Receiver and Forwarder

The CAN Receive and Forward (CRF) receives MQTT messages that contain CAN data frames from the interceptor. After receiving the CAN data frames they need to be decoded via the use of a DBC file. Once decoded the CRF uploads the decoded CAN data frames to data storage.

The CRF is run as a module, meaning the CRF has access to module twin functionality. Values that can be configured via the module are the endpoint for data storage, enabling debug logs and setting the queue size for upload to data storage.

The primary function of the CRF is to decoded incoming CAN data frames from the interceptor. The decoding process is done by searching the DBC file by the unique identifier for the data frame. In the context of the proposed architecture the unique identifier is the PGN as we are decoding J1939 and NMEA 2000 messages. Once a PGN has been found in the DBC it is stored in cache for later use as searching the DBC again takes unnecessary time.

Each DBC entry contains SPN values for the respective PGN. SPNs contain several properties to help decode the databytes of the CAN data frame, the properties are as follows:

- Name: The unique name of the SPN to help identify it in the context of the message.
- Type: A value that indicates the signedness of the SPN this can be either unsigned or signed.
- Byte Order: This indicates if the bytes read by the SPN are 'Motorola' (big-endian) or 'Intel' (little-endian) format.
- Start Bit: This is the bit position where the SPN starts from, this bit is the least significant bit.

- Length: This is the length of the SPN in bits and determines how many bits from the start bit are used for the SPN.
- Factor and Offset: These two values are used to convert the raw data from the bits to a physical value. The raw data is first multiplied by the factor and then the offset is added to the result to get the physical value.
- Minimum and Maximum: The values define the expected range of the physical value.
- Unit: The unit of the SPN such as the RPM, km/h and degrees Celsius.

From the SPN properties we can extract a physical value. As an example, we have the following databytes (FF-FF-FF-68-13-FF-FF-FF) and are trying to extract a physical value which has a starting bit of 24, length of 16, factor of 0.125, offset of 0 and a byte order of Intel. First, we get the bytes 68-13 due to a starting bit of 24 and a length of 16. We then flip the bytes due to the SPN being little endian giving us the bytes 13-68. Next, we convert the bytes to decimal which gives us a raw value of 4968. Finally, we can use equation 4.1 with a raw value of 4968 to give us a physical value of 621.

$$physicalvalue = offset + scale * rawvalue \quad (4.1)$$

Once a physical value for each SPN has been calculated the name, physical value and unit for each SPN are appended to the CAN data frame received from the interceptor. For each CAN data frame we also append the priority, PGN, source address, data page, extended data page, PDU format and PDU specific values.

Once the CAN data frame is decoded, it is stored in a queue with a queue size configured by the module twin. Once the queue size is reached, the CAN data frames stored in the queue are uploaded to data storage in the form of a compressed .csv file.

Data Storage

The data storage component comes in the form of MinIO, a high-performance S3 compatible storage [39]. The use of MinIO was suggested by previous work with EET and side-channel data storage systems [19]. MinIO also deals with **FR-3** and **FR-5** due to the ability to attach meta information to data and give role-based access to data.

The CAN data is uploaded to MinIO from the CRF in the form of timestamped compressed .csv files. Compressed .csv format was chosen due to previous work on EET that

found compressed .csv files efficient as a file format due to their file size, S3 query capability and human readability [19].

Data storage like the CRF is run as an IoT Edge module. This can be easily achieved as MinIO has a docker release which allows easy integration with the IoT Edge runtime.

The main benefit of using MinIO is the data replication feature, which allows data replication from one MinIO instance to another. MinIO data replication allows data storage on the aggregator to replicate to the data storage instance in the analyser. By having two instances of data storage it gives the ability to store data when there's no connection to the analyzer and then replicate it to the analyzer once a connection is reestablished.

Heartbeat Transceiver

The heartbeat transceiver receives the messages sent from the heartbeat transmitter on the interceptor. The heartbeat transceiver logs all incoming messages from the heartbeat transmitter to help inform the user of interceptor connectivity. Like the CRF and data storage, the heartbeat transceiver is also run as an IoT Edge module.

Log Forwarder

The log forwarder accumulates the logs from the CRF, data storage and heartbeat transceiver and forwards them to the log management database in the central service. By collecting various logs from the aggregator, it gives users more clarity on the state of the system which helps with [FR-4](#).

Edge Hub

The edge hub is part of every Azure IoT Edge deployment [40]. The edge hub acts as a proxy for the IoT hub in the central service. By acting as a proxy the edge hub allows modules to be ran without a direct connection to the iot hub.

The edge hub also has a Prometheus endpoint that contains detailed metrics about the IoT Edge. Useful metrics include the message queue length and the total number of messages received for each interceptor. Overall the access to metrics helps deal with [FR-4](#).

Edge Agent

The edge agent, like the edge hub, is part of every Azure IoT Edge deployment. The edge agent is responsible for instantiating modules, ensuring that they continue to run, and reporting the status of the modules back to IoT Hub [40].

Like the edge hub, the edge agent also has its own metrics. Useful metrics from the edge agent include the current CPU use and the current uptime of the edge. These additional metrics also help deal with **FR-4**.

NTP Server

The NTP server helps make sure the interceptors attached to the aggregator have accurate timestamps by communicating with the NTP client on the interceptor. Unlike the rest of the components in the aggregator, the NTP server is run directly on the operating system instead of a module. This is due to NTP already being a Debian 11 package and the NTP server needing little configuration once set up.

4.2.4 Central service

The purpose of the central service is to act as a device management system for all aggregators and interceptors. The central service goal is to solve **FR-2** and **FR-5** while also solving maintainability issues faced with EET [19]. The central service contains two components called the IoT hub and log management database which are explained in the following sections.

IoT Hub

Direct methods, module twins, device twins, update agent, provisioning The main goal of the IoT hub is to manage interceptors and aggregators remotely. The IoT hub manages interceptors via the use of direct methods, device twins and device updates. While the IoT hub manages aggregators via the use of module twins and edge deployments. By leveraging the IoT hub for remote management, it helps solve the maintainability issues faced previously [19].

Log Management Database

The log management database is a central collection of logs from all aggregators. Keeping a central collection of logs for each aggregator helps with visibility at scale and if issues occur in the infrastructure, the logs in the log management database can be consulted.

4.2.5 Analyzer

The analyzer is responsible for analysing the CAN data received from the aggregator. The analyzer is made up of two components called data storage and rules engine, which are explained in this section.

Data Storage

The data storage component, like the data storage component on the aggregator, is a MinIO instance. The data from the MinIO instance on the aggregator is replicated to MinIO on the analyzer.

The data storage component allows the rules engine system to not only be used with live CAN data but past CAN data as well. This flexibility allows the rules engine to be optimised and tweaked by repeating the training process with past CAN data. The use of live and stored data meets both [FR-8](#) and [FR-10](#).

Overall, keeping a centralised data storage allows users to use the rules engine with data from any aggregator. Otherwise, users would be limited to analysing data from just one aggregator.

Rules Engine

The rules engine component implements the rules engine described in [3.3](#). The rules engine first reads CAN data from the data storage component using the MinIO API. If the rules engine is in a training state, the data read from data storage is used to establish the rules defined in [3.3](#). If the rules engine is in an active state, the data read from data storage is compared against the rules established in the training state, and if an anomaly is detected, an intrusion flag is raised. The rules engine automatically configures its rules depending on the incoming data, which solves [FR-9](#).

Once an instruction flag is raised, the system will alert the end user of the detected intrusion. This alert system is flexible depending on the end user's needs. For instance, the end user could be alerted in the form of emails or text messages. The main point of the intrusion alert system is to give flexibility to the end user so they can be alerted to possible intrusions in the most efficient way that suits their use case.

4.3 Deployment View

A deployment view translates an abstract architecture to real-world use cases [41]. By translating to real-world use cases, we can help understand how the architecture works in reality. This section contains two deployment views, one for an off-site deployment and another for an on-site deployment.

4.3.1 Off Site

The off-site deployment is featured in Figure 4.2. The off-site deployment is the primary deployment for the architecture and is used when the aggregator has a constant connection to the central service and analyzer.

The main goal of the off-site deployment is to reduce processing on the aggregator as much as possible and keep the analyzer in a centralised place. By centralising the analyzer, it gives the rules engine easy access to data from all interceptors as well as increased processing power. An off-site approach also gives room for machine learning development in the future if needed by leveraging the increased processing power.

4.3.2 On Site

The on-site deployment is featured in Figure 4.3. The on-site deployment is the primary deployment for when the architecture doesn't have a constant connection to the central service and analyzer.

In the on-site deployment, all the functions of the analyzer are moved to the aggregator. The rules engine now reads directly from the data storage on the aggregator instead of the replicated data storage on the analyzer. This is so the rules engine can perform its functionalities without a constant connection to the analyzer.

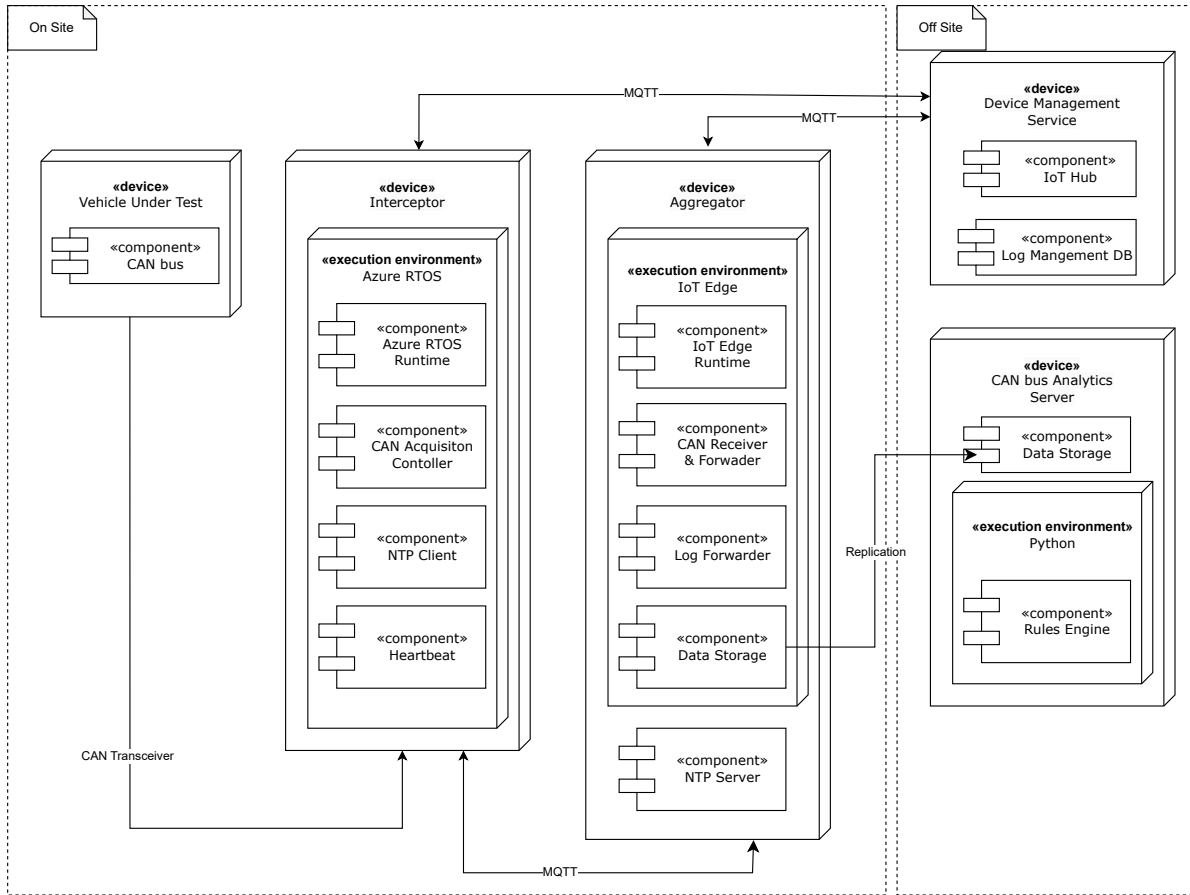


Figure 4.2: Deployment view of an off-site deployment

A benefit of using the on-site approach is the rules engine becoming an IoT Edge module due to being run on the aggregator. Due to this, we can leverage the IoT Hub functions, such as the module twin to remotely manage the rules engine.

As seen in Figure 4.3 the IoT Hub is still off-site in this deployment. This is because there is only one IoT Hub that manages all deployments. Fortunately, once set up the interceptor and aggregator do not need a connection to the central service. This is because the Edge Hub acts as a proxy when there is no connection to the IoT Hub and stores the configuration values for both the interceptor and aggregator.

Overall the off-site deployment is preferable for most deployments, but if a deployment

is likely to be in a place with little or no internet connectivity, the on-site solution is more than viable.

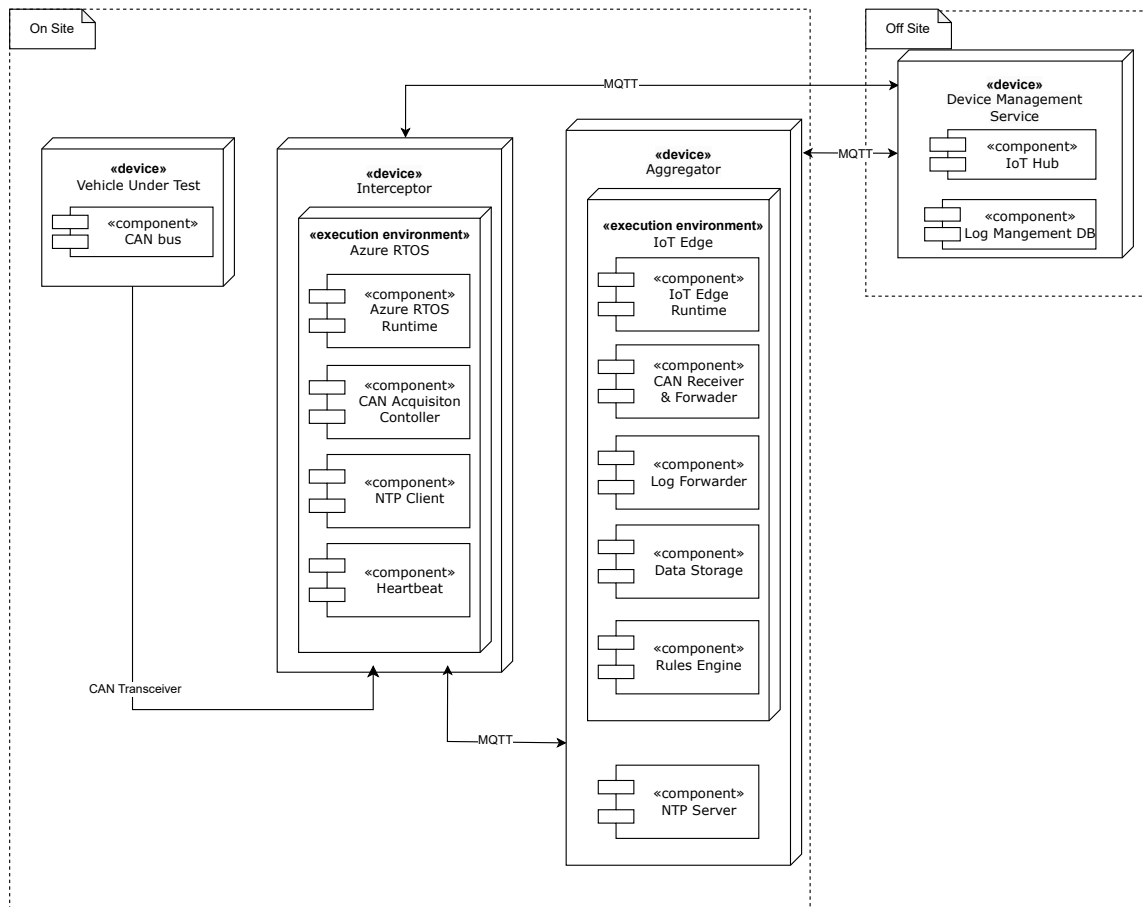


Figure 4.3: Deployment view of an on-site deployment

Chapter 5

Case Studies

This chapter tests the off-site deployment featured in Figure 4.2 on two separate case studies. Each of the case studies are systems used in industry making them reflective of a real-world deployment. The first case study is a deployment on a NMEA 2000 temperature sensor manufactured by Yacht Devices. The second case study is a deployment in a 2016 Peterbilt 579 truck that uses the J1939 protocol.

The structure of this chapter is as follows: Section 5.1 defines the NMEA 2000 case study and provides an analysis of the system. Section 5.2 defines the J1939 case study and provides an analysis of the system. Section 5.3 tests the throughput of the system from both a single and multiple interceptor perspective. Finally, section 5.4 describes the lessons learnt from testing the architecture with the case studies.

5.1 Case Study 1: NMEA 2000

The NMEA 2000 case study aims to test the system's capability to handle NMEA 2000 messages. The setup for the NMEA 2000 case study consists of a NMEA 2000 temperature sensor manufactured by Yacht Devices attached to an Arduino that acts as a receiver, with an interceptor placed between the temperature sensor and Arduino. The setup for the case study can be seen in Figure 5.1.

An Arduino is used as the NMEA 2000 temperature sensor cannot function when placed on a bus by itself and requires a listener. The Arduino's software sends a NMEA 2000 message with the PGN 13012 every two seconds and the PGN 126993 every 60 seconds to replicate the functionality of an actual NMEA 2000 device.

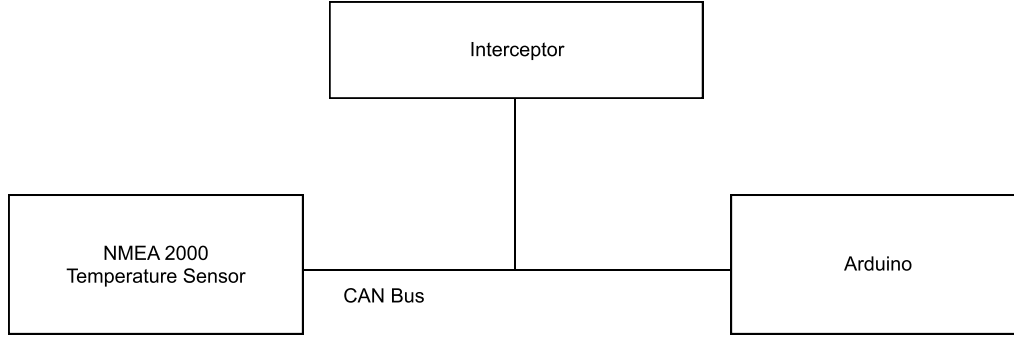


Figure 5.1: Diagram representing the NMEA 2000 case study setup

First, we checked the number of unique PGN and source address combinations found in the setup by analysing the data collected from the interceptor in a period of ten minutes. In total, we found seven unique PGN and source address combinations, which are listed in Table 5.1. It is also worth mentioning that all of the five unique PGNs in the table were successfully decoded by the aggregator.

PGN	Source	Expected Frequency (s)	True Frequency (s)	Jitter (%)
126993	Arduino	60.0	59.98319	0.028
126993	Temp Sensor	60.0	59.6678	0.554
130310	Temp Sensor	0.5	0.49724	0.552
130311	Temp Sensor	0.5	0.49724	0.552
130312	Arduino	2.0	1.99927	0.037
130312	Temp Sensor	2.0	1.98895	0.553
130316	Temp Sensor	2.0	1.98895	0.553

Table 5.1: Table containing the expected frequencies, true frequencies and jitter percentage of PGNs in the NMEA 2000 setup

Interestingly the true frequency of each PGN captured by the boxes wasn't the same as the expected frequency found in the temperature sensors manual and the code of the Arduino. The true frequency is the mean of the frequencies seen over a ten minute period of CAN data captured with the interceptor. An interesting observation from these results

is the clock in the temperature sensor is not as accurate as the clock in the Arduino, with an average jitter of 0.55 % present in the temperature sensor and an average jitter of 0.032 % present in the Arduino. Jitter, in this context, means the deviation of the true frequency from the expected frequency.

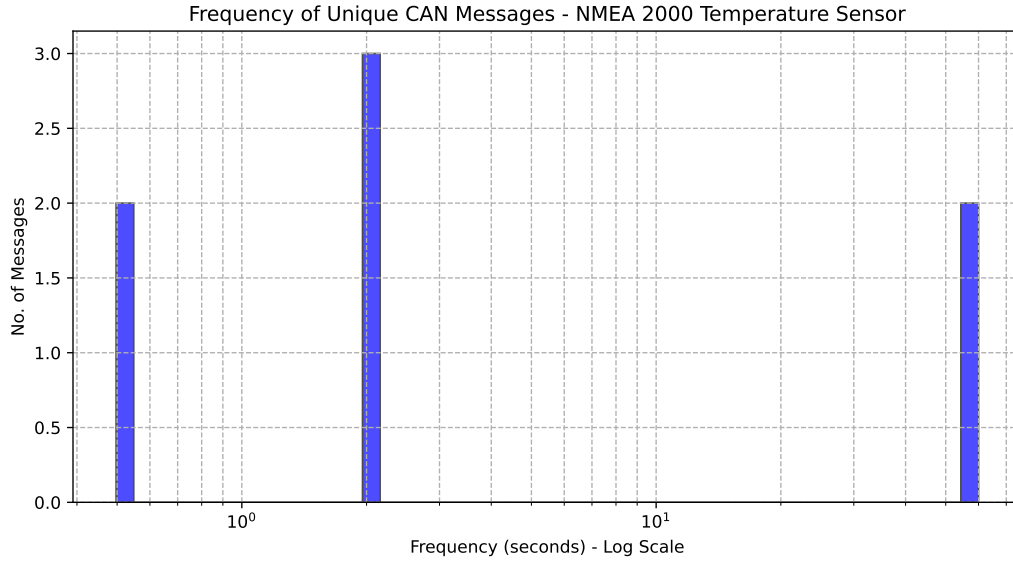


Figure 5.2: Graph showing the frequency of CAN messages from the NMEA 2000 setup

Figure 5.2 reflects the frequency distribution of the PGNs in Table 5.1. The fastest messages on the bus are sent every 0.5 seconds, while the slowest are sent every 60 seconds. Overall, this case study helps prove the system decodes NMEA 2000 messages successfully.

5.2 Case Study 2: J1939

The J1939 case study aims to test the system’s capability to handle J1939 data and CAN data from a real automotive vehicle. This is achieved via tapping into the CAN bus in a 2016 Peterbilt 579 truck. Figure 5.3 shows the setup for this case study with two interceptor boxes attached to the OBD connector and the passenger door controller.

Table 5.2 features the unique PGNs seen across ten minutes of data capture by the interceptors in the truck while idle. In total, there were 114 unique PGNs seen across both

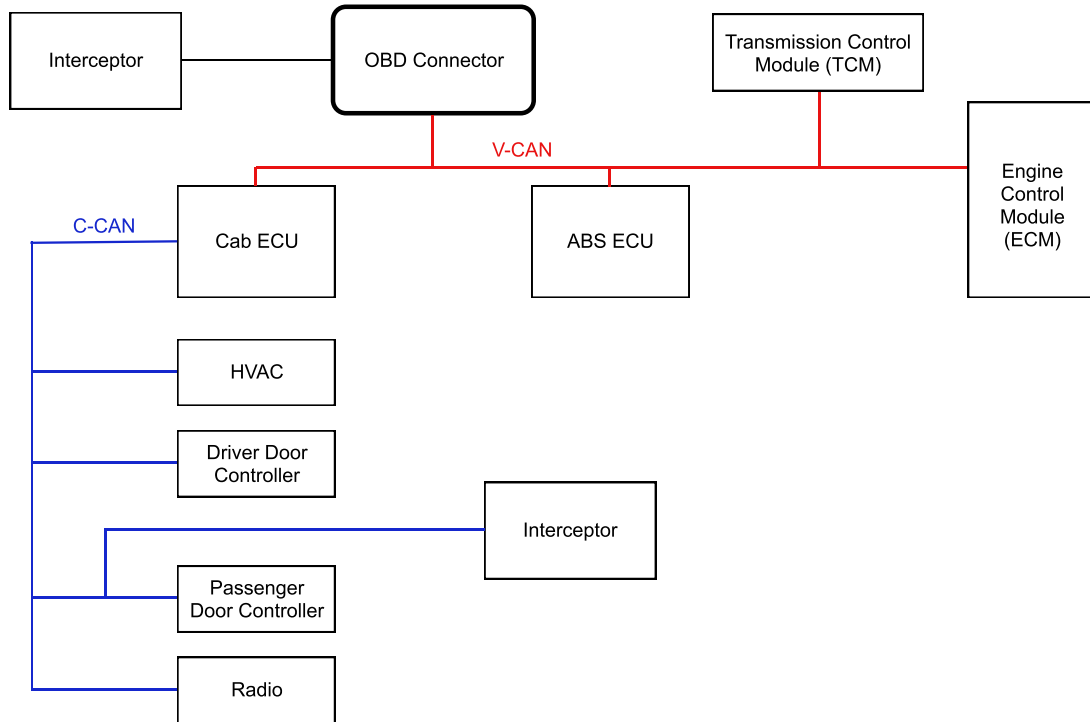


Figure 5.3: Diagram representing the J1939 case study setup

the OBDII port and the passenger door controller. The OBDII port has 99 unique PGNs by itself, and the passenger door controller has 27 unique PGNs by itself. Between the OBDII port and passenger door controller, twelve PGNs were shared, with eleven of them being decodable. The undecodable PGN was PGN 65226 for active diagnostic trouble codes.

In total, there were 23 undecodable PGNs found in the truck. One of the undecodable PGNs was PGN 65226, and another was PGN 64695 for vehicle electrical power, which was found on the OBDII port but is now obsolete. The other 21 undecodable PGNs were all proprietary and unique to Peterbilt. The eleven decodable PGNs found via the passenger door controller were also present in the 91 decodable PGNs found via the OBDII port. The passenger door controller also had more proprietary PGNs, which makes logical sense due to the passenger door controller being part of the internal CAN bus, while the OBDII port is external.

Interceptor Location	Total PGNs	Decoded PGNs	Undecoded PGNs
OBDII Port	99	91	8
Passenger Door Controller	27	11	16
OBDII Port & Passenger Door Controller	114	91	23

Table 5.2: Comparison of PGNs by interceptor location

Table 5.3 contains the number of PGNs per source address for the OBDII port. In total seven unique source addresses were found via the OBDII port. The majority of PGNs came from the engine, with 67 unique PGNs in total. The least number of PGNs came from the passenger operator climate control, with only one PGN in total.

SA	SA Name	Total PGNs	Decoded PGNs	Undecoded PGNs
0	Engine #1	67	65	2
3	Transmission #1	13	11	2
11	Brakes - System Controller	14	11	3
15	Retarder - Engine	5	5	0
25	Passenger-Operator Climate Control #1	1	1	0
42	Headway Controller	11	8	3
49	Cab Controller - Primary	20	17	3

Table 5.3: Comparison of PGNs by SA for the OBDII port

Table 5.4 contains the number of PGNs per source address for the passenger door controller. In total seven unique source addresses were found via the OBDII port. The majority of PGNs came from the cab controller, with twelve unique PGNs in total. The least number of PGNs came from the global source address, with only one PGN in total.

Interestingly, the global source address 255 in the J1939 is never permitted in the source address field of J1939 according to the J1939 standard and should only be used as

a destination address. The truck seems to still send messages with the source address 255, meaning it technically does not conform to the J1939 standards.

SA	SA Name	Total PGNs	Decoded PGNs	Undecoded PGNs
0	Engine #1	3	3	0
25	Passenger-Operator Climate Control #1	8	5	3
49	Cab Controller - Pri- mary	12	3	9
76	Radio	2	0	2
236	Door Controller #1	2	0	2
237	Door Controller #2	2	0	2
255	Global	1	0	1

Table 5.4: Comparison of PGNs by SA for the passenger door controller

Figure 5.4 reflects the frequency distribution of the PGNs for the passenger door controller. The fastest message on the bus is sent every 20 milliseconds, while the slowest message is sent every five seconds.

An issue that wasn't present in the NMEA 2000 setup was an issue with NTP syncing. The fastest CAN messages in the truck are sent in the tens of milliseconds, compared to the fastest messages in the NMEA 2000 setup being sent every 0.5 seconds. The accuracy of NTP is only within ten milliseconds over the public internet [42]. Due to this accuracy, the frequency of CAN messages that send in the tens of milliseconds can vastly differ when the NTP server syncs with the interceptor. For example, the PGN 61444 is sent every 20 milliseconds but due to the syncing issues can have a frequency range from 10 to 30 milliseconds. Overall this shows that NTP is accurate enough for CAN messages that send at least every 100 of milliseconds, like the messages from the NMEA temperature sensor. Though NTP is not accurate enough for messages that send in the 10s of milliseconds, such as the messages from the door controller.

Figure 5.5 reflects the frequency distribution of the PGNs for the OBDII port. The fastest message on the bus is sent every ten milliseconds, while the slowest message is sent every 60 seconds. Unlike the distribution for the door controller in Figure 5.4, more messages have a frequency within tens of milliseconds. Similar to the door controller, the

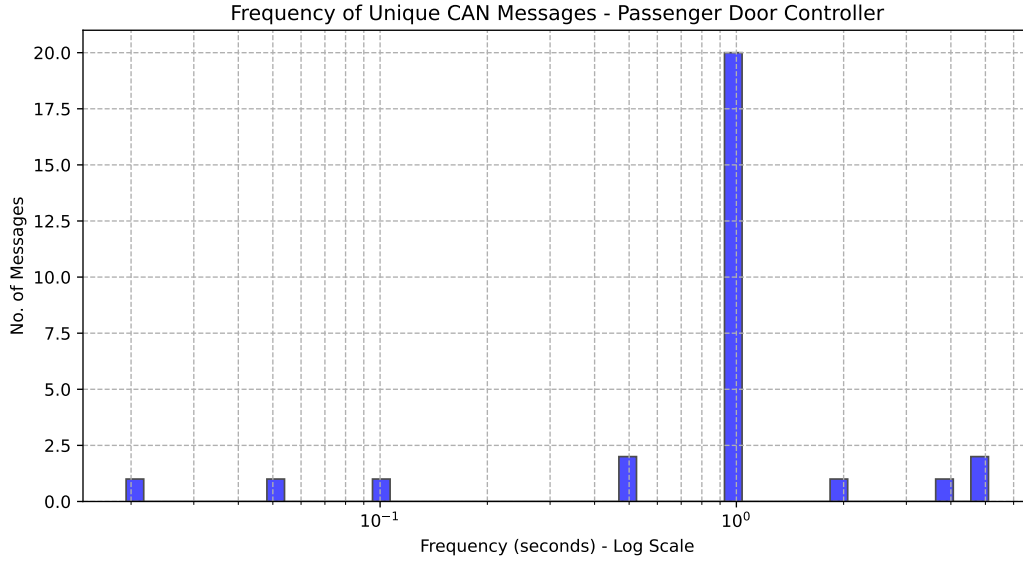


Figure 5.4: Graph showing the frequency of CAN messages from the passenger door controller

OBDII port also suffers from the NTP syncing issue but has more inaccuracies due to the increased number of CAN messages being sent within 10s of milliseconds. Overall this shows that a more accurate alternative to the NTP protocol would be preferable for frequency accuracy in production.

In the truck, we also tested the viability of the CANCrocodile contactless J1939 reader to prove [FR-7](#) [38]. A third interceptor was attached to the wiring from the OBDII port via the CANCrocodile, with the data from this third interceptor compared to the data from the original interceptor connected to the OBDII port. In theory, both interceptors should read the exact same CAN traffic.

When tested, the third interceptor dropped 0.8% of the CAN packets when compared to the data from the original interceptor, the data was collected across a period of ten minutes while the truck was idle. The CANCrocodile was then retested using the NMEA 2000 setup across a period of ten minutes and no packets were dropped.

Overall, it shows that while the CANCrocodile can work fine in a lab setup, such as with the NMEA 2000 setup. The CANCrocodile does not perform optimally in an industrial setting. This means if we want accurate CAN bus data, the interceptor has to directly tap into the CAN bus, otherwise we run the risk of dropping CAN messages.

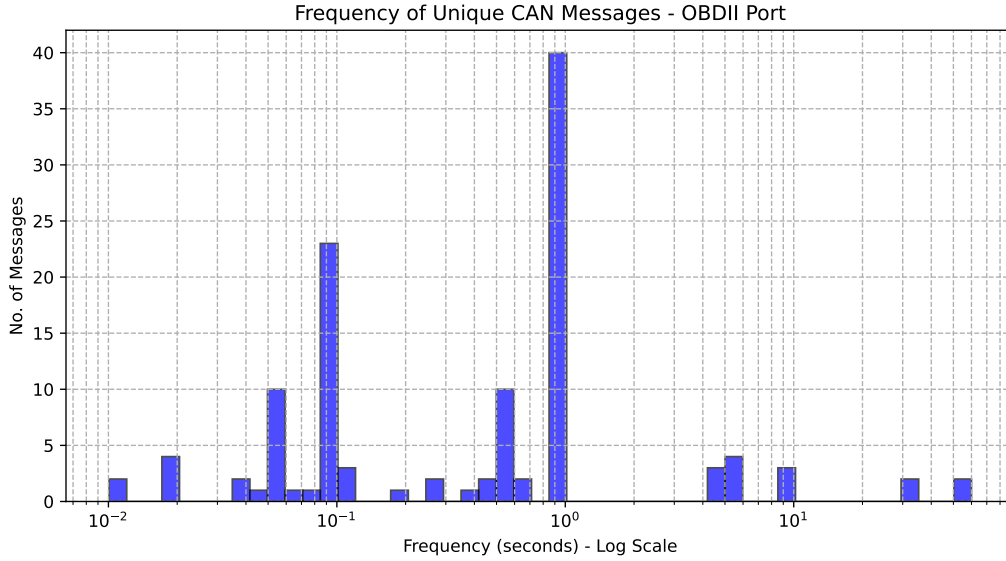


Figure 5.5: Graph showing the frequency of CAN messages from the OBDII port

5.3 Throughput

The goal of this section is to test the real-life throughput of the system and make sure it performs adequately under stress. This helps achieve **NFR-2** as the worst-case scenario is reading CAN data from a system with a maximum busload.

The section also measures how long data takes from being read at the interceptor to being processed by the rules engine on the analyzer. For this reason, the queue size of the aggregator is set to the same queue size as the interceptor, with a queue size of 68 CAN messages. This means CAN messages pass through the system as quick as possible. In turn, this puts more stress on the network due to every single packet from the interceptor being uploaded to MinIO individually, instead of in larger batches.

The rest of this section is split into two subsections. The first subsection tests the throughput of the system at a single device level using the NMEA 2000 and J1939 case study setups. The second subsection tests the throughput of the system at a multiple device level using a simulator.

5.3.1 Single Interceptor

The main objective at a single interceptor level is making sure the rules engine can process a full CAN packet before the next CAN packet is received. To solve this objective, we test the IDS on three devices from the case studies, the NMEA temperature sensor, the door controller and the OBDII port.

First, we test the time it takes for the interceptor to fill up its queue size of 68 with CAN messages. The results for this can be seen in Figure 5.6. The mean processing for the NMEA setup is every 12.061 seconds. For the door, the mean processing time is every 0.716 seconds. For the OBDII port, the processing time is every 0.074 seconds. Overall this shows that fast bus rates, such as the ones present in the truck, fill up the packet quicker than slower bus rates, such as the bus rate for the NMEA setup.

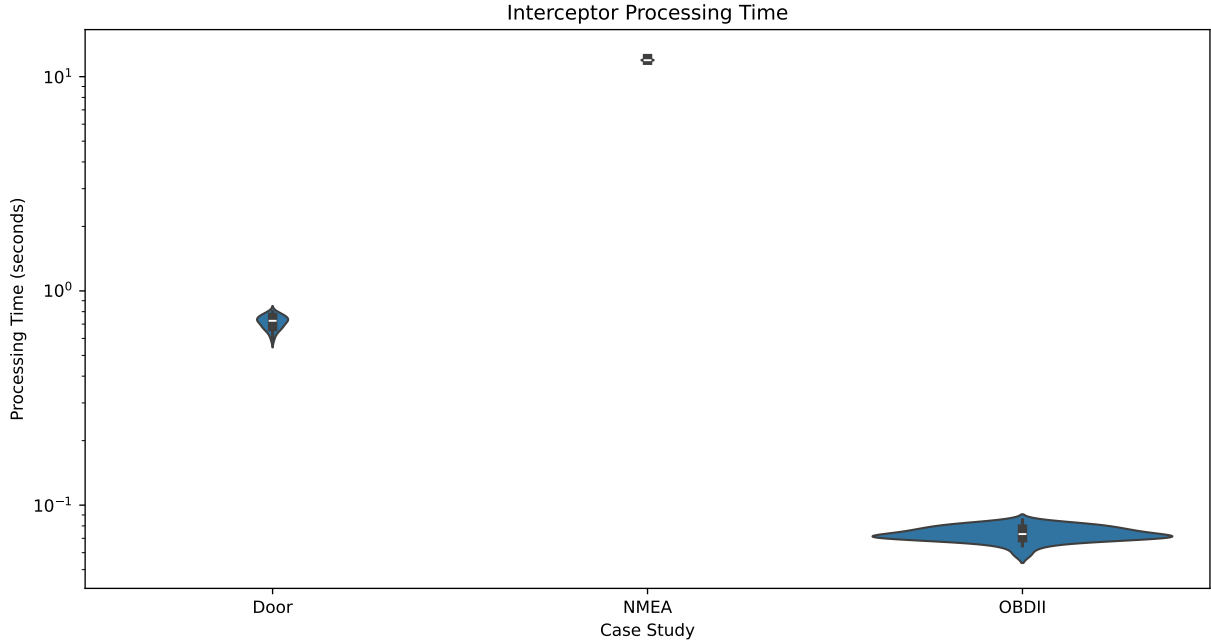


Figure 5.6: Violin graph showing the time it takes for the interceptor to process CAN packets

Second, we test the time it takes for the aggregator to process the 68 CAN messages from the interceptor. The results for this can be seen in Figure 5.7. The mean processing for the NMEA setup is every 0.363 milliseconds. For the door, the mean processing time is every 0.348 milliseconds. For the OBDII port, the processing time is every 0.299 milliseconds.

Overall the aggregator is efficient in processing and can easily process incoming messages from the interceptor due to having a sub-millisecond processing time on average.

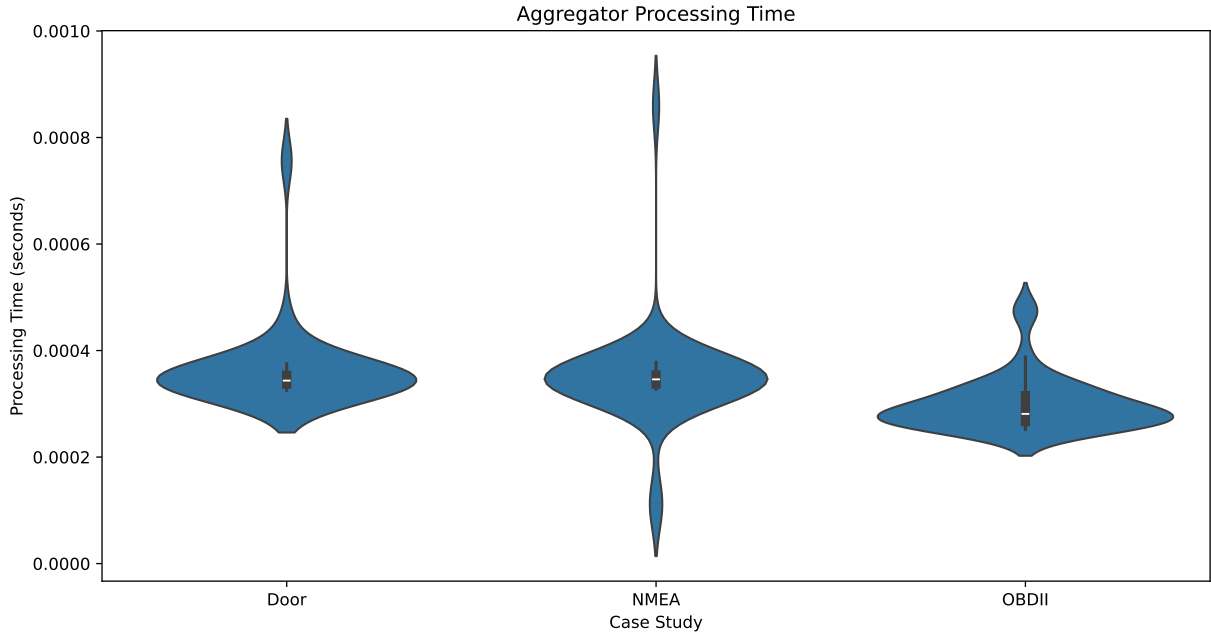


Figure 5.7: Violin graph showing the time it takes for the aggregator to process CAN packets

Finally, we test the time it takes for the analyzer to process the 68 CAN messages sent from the aggregator. The results for this can be seen in Figure 5.8. The mean processing for the NMEA setup is every 0.032 seconds. For the door, the mean processing time is every 0.025 seconds. For the OBDII port, the processing time is every 0.069 seconds. Overall the interceptor is inefficient in processing due to the 0.069 second processing time of the OBDII port being near equal to the interceptor processing time of 0.074 seconds, which leads to a potential bottleneck forming. Due to this future, reversions should focus on improving the efficiency of the analyzer to hopefully achieve a sub-millisecond processing time similar to the aggregator.

While we have talked about the inefficiency of processing messages from a CAN bus with a high bus rate the system also struggles from processing messages from a CAN bus with a low bus rate. For example, the interceptor takes 12.061 seconds to process CAN data from the NMEA setup. Regardless of the efficiency of the aggregator or analyzer the first packet in the queue will always be processed around twelve seconds after it occurred.

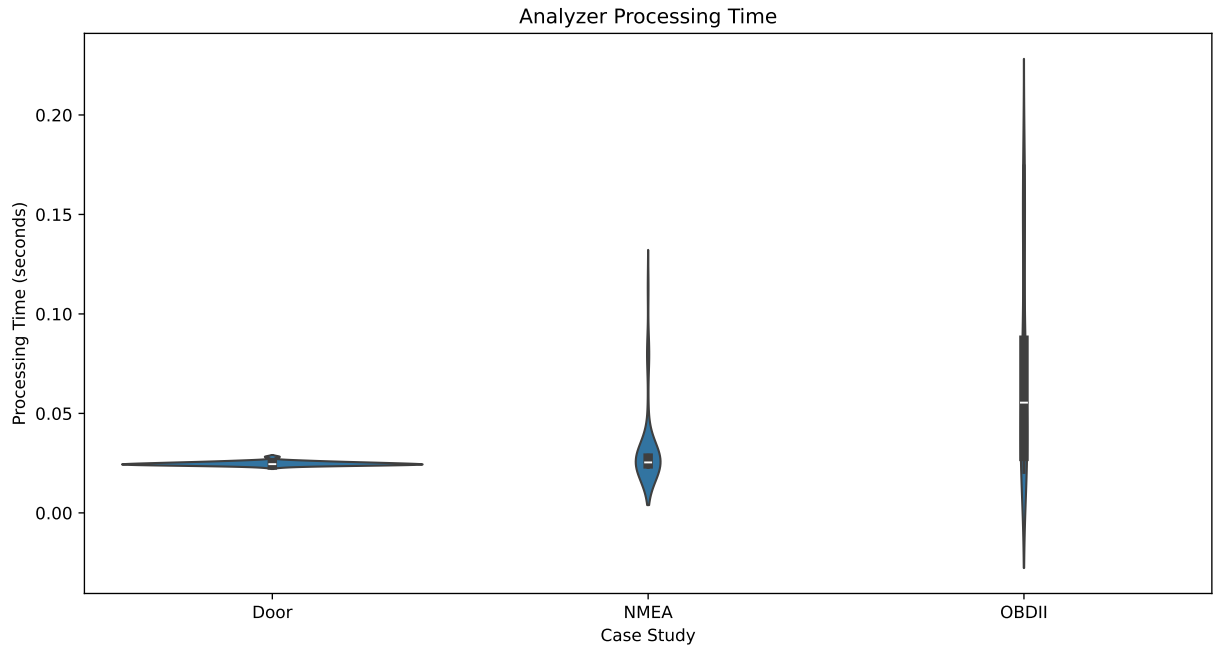


Figure 5.8: Violin graph showing the time it takes for the analyzer to process CAN packets

To improve this, we could potentially send CAN packets from the interceptor at set times even if the queue size has not been reached. Overall, this makes sure CAN data is always being processed as quick as possible.

5.3.2 Multiple Interceptors

This section tests the ability of the aggregator to handle data from multiple interceptors and the upper limit on how many interceptors can be connected to one aggregator.

First, we must calculate the worst-case scenario for the interceptor. The worst-case scenario would be the interceptor being connected to a CAN bus with a bus load of 100 percent. With a bus load of 100 percent a CAN message would be sent every 0.27 milliseconds with a baud rate of 500 kbps and every 0.54 milliseconds with a baud rate of 250 kbps [43]. This means it will take 18.36 milliseconds to fill an interceptor packet of 68 CAN messages with a baud rate of 500 kbps. While it will take 36.72 milliseconds to fill an interceptor packet of 68 CAN messages with a baud rate of 250 kbps.

To account for the worst-case scenario the aggregator should be capable of processing

a message from an interceptor every 18.36 milliseconds.

Previous work with the Azure IoT Edge has stated that messages from downstream devices can only be sent every five milliseconds [44]. To verify this we used a simulator to act as a proxy interceptor and adjusted the frequency depending on how many proxy interceptors are connected.

In Table 5.5 you can see the maximum frequency per number of interceptors. The maximum frequency is calculated by lowering the frequency until an interceptor message queue forms in the Edge Hub metrics. Overall the Table shows that the mean frequency across all interceptors divided by the total number of interceptors cannot exceed five milliseconds.

Interceptor Message Frequency (ms)	Number of Interceptors
5	1
10	2
15	3
20	4
25	5
30	6
35	7
40	8
45	9
50	10

Table 5.5: Table containing the maximum message frequency per number of interceptors

As the worst-case scenario for the interceptor is sending a CAN packet every 18.36 milliseconds, the maximum number of interceptors that can be connected to the aggregator in a worse-case scenario is three, according to Table 5.5.

An option to bypass the three interceptor limit is sending messages from the interceptors more dynamically. For instance, we would limit the message frequency depending on how many interceptors are connected to the aggregator. For example, if ten interceptors are connected each interceptor can only send a message every 50 milliseconds according to Table 5.5. This would require changes to the interceptor by increasing the queue size.

5.4 Lessons Learnt

A few lessons were learnt from conducting this case study. One lesson learnt was that the internal clocks inside multiple components on the CAN bus can vastly differ in accuracy, as seen with the Arduino and temperature sensor in Table 5.1.

Another interesting observation was the use of the global source address 255 in the source address field of J1939 messages in the door controller. As stated before using source address 255 in the source address field is forbidden under the J1939 standard. This shows manufacturers don't fully obey the J1939 standard even if they say they do.

Another lesson learnt was the large number of undecoded PGNs inside real-life systems. The 2016 Peterbilt 579 truck had a total of 23 undecodable PGNs. Due to the rules engine requiring decoded data to work optimally it gives potential attackers the option to attack via undecodable PGNs. Future work could potentially look at ways to detect attacks via undecoded PGNs.

One large issue found was the inaccuracy with using NTP to timestamp packets on the interceptor. Due to NTP being accurate within ten milliseconds, it led to messages that are sent within tens of milliseconds having inaccurate frequencies. Alternative avenues, such as using the Precision Time Protocol (PTP) may be more favourable [45].

The CANCrocodile contactless J1939 reader was also tested and was found to drop 0.8% of its total messages. This impacts [FR-7](#) as we cannot ensure both complete security, as packets are dropped, and non-intrusive access to CAN bus.

Another issue found was with the processing time of the rules engine. Due to the interceptor processing time matching the analyzer processing time for the OBDII port. This led to the rules engine being bottlenecked and it could not handle the incoming CAN data. The rules engine is currently written in Python and could be optimised by using an alternative language.

We also experienced issues with slow bus loads such as the interceptor for the NMEA setup sending every twelve seconds. This twelve seconds delay led to some messages being processed by the rules engine twelve seconds later. To improve this the interceptor boxes should send at a constant rate instead of waiting for the queue to fill.

One issue faced was the limitations of the IoT Edge messaging service. This led to messages from the interceptor being limited to certain frequencies according to Table 5.5. To solve the issue with the IoT Edge message limitations and the slow bus load issue, we propose sending CAN data from interceptors at a constant rate according to Table 5.5.

Finally, we found it hard to test the accuracy of the rules engine due to the real-time constraints of the CAN bus. It would be optimal to test it with a real-time security framework such as Metasploit. Metasploit offers the hardware bridge API that allows real-time security testing with the CAN bus [46].

Chapter 6

Conclusion

The protection of automobiles and by extension safety-critical systems is important. This thesis provides a novel rules-based solution to detect attacks that leverage underlying security flaws with the CAN protocol. The proposed rules-based CAN IDS uses IoT Edge architecture to optimally process CAN data and develop intelligent rules around the CAN data.

The IoT Edge architecture allows the rules-based CAN IDS to be easily deployed at scale. This is due to the easy management and provision of new devices via the centralised IoT Hub. In comparison to the decentralised approach for EET that required more manual management [19]. The proposed IoT Edge architecture can also be applied to the previous work with EET, allowing the collection of power trace data while solving the maintainability issues previously faced.

The rules-based CAN IDS correctly processes CAN bus data from both the NMEA 2000 and J1939 protocols in real-time. Though the rules engine is currently hard to test without leveraging a real-time security framework such as Metasploit [46]. One point to note is that the rules engine has the ability to protect any device that sends CAN messages, and not just vehicles. This gives the potential to perform security testing at a component level before a component is placed inside a vehicle.

Overall the proposed rules-based CAN IDS has been proven to work on real-world systems but will require minor tweaks to optimally work, which are listed in section 5.4. The following section describes potential avenues for future work.

6.1 Future Work

One route that future work can go down is the testing of various other CAN protocols. Examples of other CAN protocols include ISOBUS for tractors and standard CAN for cars. Though it would require the ability to decode the incoming CAN messages.

Testing the rules-based IDS with various driving conditions would be another avenue. Currently, the truck was only tested while idle. It would be suitable to test the IDS with the truck in different states. For example, while highway driving or in various weather conditions such as snow.

Undecoded CAN messages are a big issue for CAN IDS. This is because without decoding the message it is hard to grasp the physical values of the CAN message. Due to this, it would be optimal to research how intrusion detection can be performed on undecoded CAN messages.

As stated before the next step would be to test the rules-based IDS with Metasploit. This is due to the real-time security testing framework it provides. By testing the rules engine with a real-time security testing framework we can easily test the accuracy of the rules engine [46].

Finally, the rules-based IDS could be used as a verification tool for cyber security testing and not just intrusion detection. This is due to its ability to detect anomalies within the CAN bus. Due to the ability to detect anomalies the rules-based IDS can detect successful CAN bus security tests and inform the end user.

References

- [1] ken. CAN Injection: keyless car theft. <https://kentindell.github.io/2023/04/03/can-injection/>, April 2023.
- [2] Chris Valasek and Charlie Miller. Remote Exploitation of an Unaltered Passenger Vehicle.
- [3] J.C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550, May 2002.
- [4] Michael Barr and Anthony Massa. *Programming Embedded Systems: With C and GNU Development Tools*. "O'Reilly Media, Inc.", October 2006.
- [5] Shobha S Prabhu, Hem Kapil, and Shashirekha H Lakshmaiah. Safety Critical Embedded Software: Significance and Approach to Reliability. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 449–455, September 2018.
- [6] E.A. Bretz. By-wire cars turn the corner. *IEEE Spectrum*, 38(4):68–73, April 2001. Conference Name: IEEE Spectrum.
- [7] Ron Bell. Introduction to IEC 61508.
- [8] William Yeager. Safety-Critical Systems - Introduction to CIS. <https://sites.google.com/site/cis115textbook/safety-critical-systems>.
- [9] Simson Garfinkel. History's Worst Software Bugs. *Wired*. Section: tags.
- [10] Embedded Systems Security | Ultimate Guides | BlackBerry QNX. <https://blackberry.qnx.com/en/ultimate-guides/embedded-system-security>.

- [11] Andrei Costin and Aurélien Francillon. Short paper: A dangerous 'pyrotechnic composition': Fireworks, embedded wireless and insecurity-by-design. July 2014.
- [12] David Kushner. The real story of stuxnet. *IEEE Spectrum*, 50(3):48–53, March 2013. Conference Name: IEEE Spectrum.
- [13] What is an Intrusion Detection System (IDS)? <https://www.checkpoint.com/cyber-hub/network-security/what-is-an-intrusion-detection-system-ids/>.
- [14] Christopher W. Johnson. Why We Cannot (Yet) Ensure the Cybersecurity of Safety-Critical Systems, 2016. Conference Name: 24th Safety-Critical Systems Symposium ISBN: 9781519420077 Meeting Name: 24th Safety-Critical Systems Symposium Pages: 171-182 Place: Brighton, UK.
- [15] Subhojeet Mukherjee, Jacob Walkery, Indrakshi Rayz, and Jeremy Daily. A Precedence Graph-Based Approach to Detect Message Injection Attacks in J1939 Based Networks. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, pages 67–6709, August 2017.
- [16] Qi Liu, Veit Hagenmeyer, and Hubert B. Keller. A Review of Rule Learning-Based Intrusion Detection Systems and Their Prospects in Smart Grids. *IEEE Access*, 9:57542–57564, 2021. Conference Name: IEEE Access.
- [17] Matthew Rogers, Phillip Weigand, Jassim Happa, and Kasper Rasmussen. Detecting CAN Attacks on J1939 and NMEA 2000 Networks. *IEEE Transactions on Dependable and Secure Computing*, pages 1–15, 2022. Conference Name: IEEE Transactions on Dependable and Secure Computing.
- [18] Hossein Shirazi, Indrakshi Ray, and Charles Anderson. Using Machine Learning to Detect Anomalies in Embedded Networks in Heavy Vehicles. In Abdelmalek Benzekri, Michel Barbeau, Guang Gong, Romain Laborde, and Joaquin Garcia-Alfaro, editors, *Foundations and Practice of Security*, Lecture Notes in Computer Science, pages 39–55, Cham, 2020. Springer International Publishing.
- [19] Goksen Guler. Decentralized Data Acquisition Pipeline with Machine Learning For Side-Channel Information. Master's thesis, University of Waterloo, September 2021. Accepted: 2021-09-17T20:34:55Z.
- [20] CAN in Automation (CiA): History of the CAN technology. <https://www.can-cia.org/can-knowledge/can/can-history/>.

- [21] Hyunsung Lee, Seong Hoon Jeong, and Huy Kang Kim. OTIDS: A Novel Intrusion Detection System for In-vehicle Network by Using Remote Frame. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, pages 57–5709, August 2017.
- [22] Wilfried Voss. *A Comprehensible Guide to J1939*. Copperhill Technologies Corporation, 2008.
- [23] NMEA 2000 Explained - A Simple Intro [2023]. <https://www.csselectronics.com/pages/nmea-2000-n2k-intro-tutorial>.
- [24] Andy Greenberg. The Jeep Hackers Are Back to Prove Car Hacking Can Get Much Worse. *Wired*.
- [25] Jiajia Liu, Shubin Zhang, Wen Sun, and Yongpeng Shi. In-Vehicle Network Attacks and Countermeasures: Challenges and Future Directions. *IEEE Network*, 31(5):50–58, 2017. Conference Name: IEEE Network.
- [26] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462, May 2010. ISSN: 2375-1207.
- [27] IDS Vs IPS. <https://www.checkpoint.com/cyber-hub/network-security/what-is-an-intrusion-detection-system-ids/ids-vs-ips/>.
- [28] Siti-Farhana Lokman, Abu Talib Othman, and Muhammad-Husaini Abu-Bakar. Intrusion detection system for automotive Controller Area Network (CAN) bus system: a review. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):184, July 2019.
- [29] Min-Joo Kang and Je-Won Kang. Intrusion Detection System Using Deep Neural Network for In-Vehicle Network Security. *PLOS ONE*, 11(6):e0155781, June 2016. Publisher: Public Library of Science.
- [30] Safa Boumiza and Rafik Braham. An Anomaly Detector for CAN Bus Networks in Autonomous Cars based on Neural Networks. In *2019 International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–6, October 2019. ISSN: 2160-4894.

- [31] Camil Jichici, Bogdan Groza, Radu Ragobete, Pal-Stefan Murvay, and Tudor Andreica. Effective Intrusion Detection and Prevention for the Commercial Vehicle SAE J1939 CAN Bus. *IEEE Transactions on Intelligent Transportation Systems*, 23(10):17425–17439, October 2022. Conference Name: IEEE Transactions on Intelligent Transportation Systems.
- [32] Muhammad Shafiq, Zhaoquan Gu, Omar Cheikhrouhou, Wajdi Alhakami, and Habib Hamam. The Rise of “Internet of Things”: Review and Open Research Issues Related to Detection and Prevention of IoT-Based Security Attacks. *Wireless Communications and Mobile Computing*, 2022:e8669348, August 2022. Publisher: Hindawi.
- [33] Microsoft. What is Azure IoT Edge. <https://learn.microsoft.com/en-us/azure/iot-edge/about-iot-edge?view=iotedge-1.4>, February 2023.
- [34] MQTT - The Standard for IoT Messaging. <https://mqtt.org/>.
- [35] Martin Falch. CAN DBC File Explained - A Simple Intro [+Editor Playground]. <https://www.csselectronics.com/pages/can-dbc-file-database-intro>.
- [36] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice: Software Architect Practice_c3*. Addison-Wesley, September 2012.
- [37] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer US, Boston, MA, 2000.
- [38] Contactless reader CAnCrocodile. <https://jv-technoton.com/products/contactless-readers/cancrocodile/>.
- [39] MinIO Inc. MinIO | High Performance, Kubernetes Native Object Storage. <https://min.io>.
- [40] Microsoft. Learn how the runtime manages devices - Azure IoT Edge. <https://learn.microsoft.com/en-us/azure/iot-edge/iot-edge-runtime?view=iotedge-1.4>, March 2023.
- [41] Zheng Qin, Jian-Kuan Xing, and Xiang Zheng. *Software Architecture*. Springer, November 2010.
- [42] 5. How does it work? <https://www.ntp.org/ntpfaq/NTP-s-algo/>.

- [43] SAE J1939 Bandwidth, Busload And Message Frame Frequency. <https://copperhilltech.com/blog/sae-j1939-bandwidth-busload-and-message-frame-frequency/>, January 2019.
- [44] Aaron Propp. Secure Data Acquisition for Physics-based Side Channel Cybersecurity. Master’s thesis, University of Waterloo, August 2023. Accepted: 2023-08-18T13:12:21Z.
- [45] John C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*. Springer Science & Business Media, May 2006.
- [46] Sulav Lal Shrestha, Taylor Lee, and Sebastian Fischmeister. Metasploit for Cyber-Physical Security Testing with Real-Time Constraints. In *Science of Cyber Security: 4th International Conference, SciSec 2022, Matsue, Japan, August 10–12, 2022, Revised Selected Papers*, pages 260–275, Berlin, Heidelberg, August 2022. Springer-Verlag.