DISSERTATION

# SAE J1939-SPECIFIC CYBER SECURITY FOR MEDIUM AND HEAVY-DUTY VEHICLES

Submitted by

Subhojeet Mukherjee

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2022

## ABSTRACT

## SAE J1939-SPECIFIC CYBER SECURITY FOR MEDIUM AND HEAVY-DUTY VEHICLES

Medium and heavy-duty (MHD) vehicles are part of the US critical infrastructure. In modern MHD vehicles, mechanical operations are regularly controlled by interconnected networks of electronic control units (ECU). Communication within and across these networks is typically governed by the SAE J1939 standards. It has been established that similar to their lighter counterparts (passenger vehicles), MHD vehicles expose remote and physically accessible interfaces through which arbitrary messages can be sent to ECUs with an intent to control and/or disrupt the vehicle's functions. For physical transport of information between ECUs, SAE J1939 utilizes the Controller Area Network (CAN) protocol. CAN is used extensively for in-passenger vehicle communication and its security features have been analyzed frequently. Albeit, the same cannot be said about SAE J1939. As such, in this dissertation, we investigate security methods for MHD vehicles that utilize specifics of SAE J1939. First, we research cyber-attacks that exploit weaknesses in the SAE J1939 standards. Along with the known attacks from related literature, these help in enhancing the current threatscape. Next, we research network-based security solutions that makes use of SAE J1939 specifications. Prior work on in-vehicle security identifies the necessity for a multi-layered security solution that can raise alarms even if the attack cannot be completely prevented. As such, we provide security in two layers. In the first layer we try to detect an ongoing attack and raise alarms. The method is designed to function in an online manner in the dynamic networking environment within an MHD vehicle. In the second layer we try to identify attacker injected messages using user provided rules in real-time as the message is being transmitted. The method is designed to classify a message (e.g. a command to unlock a door) as benign or malicious based on features other than its content (e.g. whether vehicle is in motion).

## ACKNOWLEDGEMENTS

*Every word of this dissertation is a reflection of the ideologies I learnt from you. Thank you Dr. Adele Howe, Dr. Sudhabati Mukherjee and Mrs. Sipra Ghatak*

To begin with, I want to thank God. As always, whenever I was in doubt my head tilted upwards and when tilted back to its normal position, I was breathing better.

I would like to thank Dr. Jeremy Daily for the unwavering support and continous guidance. Everything I learned about medium and heavy-duty vehicles is from you. Thank you Dr. Craig Partridge for helping me with my research and producing this dissertation. Thank you Dr. Joseph Gersch and Dr. Thomas Bradley, for your valuable inputs and suggestions. I would also like to acknowledge the three research grants which funded the reserch done during the course of this dissertation

- NSF EAGER: Collaborative: Toward a Test Bed for Heavy Vehicle Cyber Security Experimentation
- NSF SaTC: CORE: Small: Collaborative: GOALI: Detecting and Reconstructing Network Anomalies and Intrusions in Heavy Duty Vehicles
- Darpa Assured Micropatching (AMP)

I would like to thank my mother, Mrs. Soma Mukherjee. The substring "Ma" appears 2442 times in the raw files used to generate this document. I am pretty sure that is less than the times she has said "Don't worry son, everything will be fine", not just during these five years but throughout my life. This is for you Ma. I would like to thank my wife, Mrs. Antara Mukherjee. Every paragraph of this dissertation owes something to you. If not for your continous support and encourangement, this dissertation would not have been a reality. Thank you Babi. I would like to thank my father, Mr. Sanjoy Sankar Mukherjee. $(a+b)^2$ is something you taught me by hand and I believe that is the base on which I am standing here. Thank you Baba. I would also like to thank

my grandmother, Mrs. Sipra Ghatak. They said you were spoiling me but it was your constant love and support that brought me here. This is for you Dida.

TABLE OF CONTENTS

LIST OF TABLES

# LIST OF FIGURES

# 1 Introduction

## 1.1 Electronification of Medium and Heavy Duty Vehicles

Medium and Heavy-duty (MHD) vehicles are the lifeblood of the modern economy, delivering critical food, supplies, and freight throughout the world. As of 2020, there were about 15,000,000 [1] registered trucks and buses in the United States only. In decades past, these vehicles were heavy polluters with unchecked emissions. Modern technology has enabled more precise engine and emission controls. The technological advancement has largely been facilitated by embedded electronics and advanced communication methods. Figure 1.1 shows the mechatronic ecosystem of a modern MHD vehicle. At the heart of this ecosystem are low-power, embedded Electronic Control Units (ECU) that regulate the operations of the vehicle. ECUs read vehicle parameters (e.g., accelerator pedal position, engine speed, etc.) from sensors and other ECUs, use them to compute control signals (e.g. the duration for fuel injection), and effectuate the same through actuators. ECUs communicate with each other within or across network segments that follow a bus topology. The engine speed displayed on a vehicle's instrument cluster, for example, is generally obtained by communicating with an engine control unit on the powertrain network segment.

ECU-to-ECU communication within MHD vehicles is typically guided by a common set of specifications made in the SAE J1939 [2] standards. The standards are partitioned into layers based on the Open System Interconnect (OSI) [3] model. The common specifications allow vehicle vendors to assemble heterogeneously manufactured devices on the same physical network without worrying about interoperability, à la plug-and-play. Electronic data obtained on MHD vehicles are exfiltrated for logistics optimization, asset management, regulation compliance, and improving fuel economy. As such, many MHD vehicles have built-in wireless telematics devices, which effectively connect them to the Internet [4, 5]. This allows for remote administration and assistance when required.

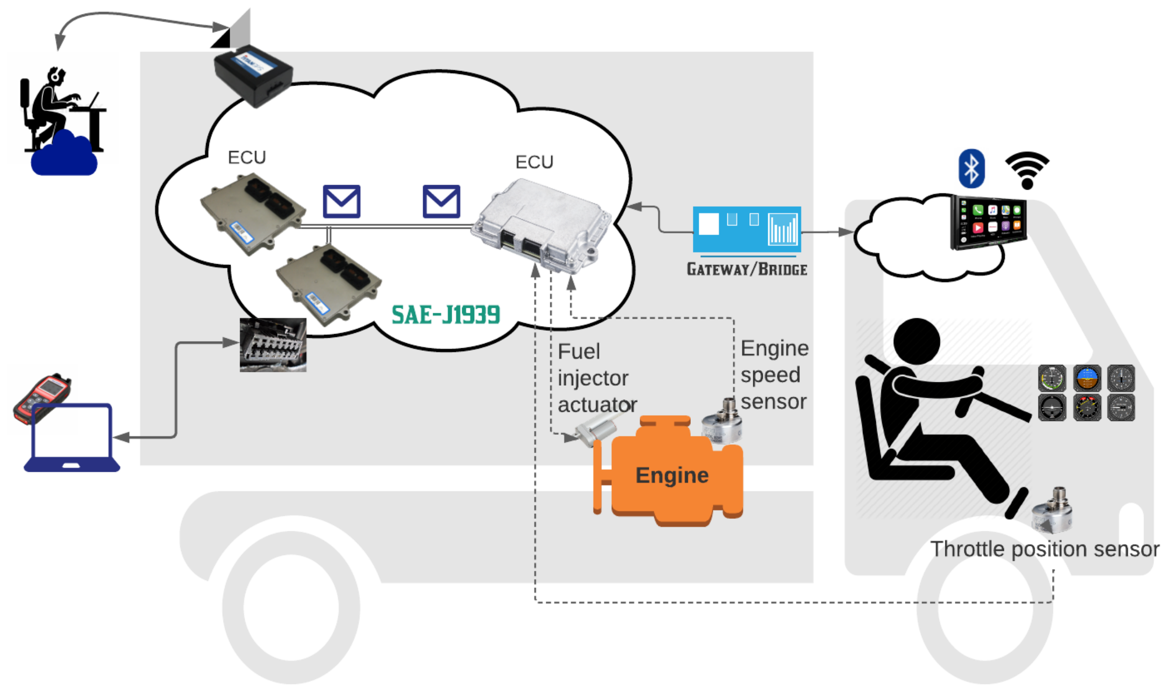


**Figure 1.1:** Mechatronic Ecosystem of a Medium and Heavy-duty (MHD) vehicle

## 1.2 Emerging Security Concerns

SAE J1939 refers to the Controller Area Network (CAN) specifications [6, 7] for transportation of messages. CAN is used heavily in passenger vehicles and there has been extensive analysis of its security [8]. Early research [9] recognized that physically proximate attackers can control and/or disrupt the vehicle's operations by injecting arbitrary messages on CAN through exposed entry points like the mandatory onboard diagnostic (OBD)-II port. The gravity of the concerns was amplified when it was discovered that physical access may not be required if one or more ECUs exposed remotely exploitable vulnerabilities [10, 11]. Preliminary heavy vehicle security research [5, 4] has confirmed that similar attack surfaces, as well as others such as exposed trailer wiring, can be found on MHD vehicles (refer to Figure 1.1 for examples) and are increasingly being used to facilitate better transportation management. With access to these attack surfaces, a malicious intruder can launch a variety of attacks on the internal network with the intent to control and/or disrupt the normal operations of an MHD vehicle. What's more, the openness of the SAE J1939 standards can be leveraged to ease the process of attack discovery [12, 13, 4]. Burakova et al. [12] and Murvay et al. [13] have already demonstrated such attacks at the application and network management layer of the SAE J1939 specifications.

To date, no known cyberattacks have been recorded on MHD vehicles, but professionals and researchers have speculated that MHD vehicles can quickly become lucrative targets for cyber attackers due to their daily involvement in societal and business-critical activities, as well as increased exposure of intruder entry points and the use of openly available standards. Recently, the perceived level of threat has been aggravated by a string of targeted cyberattacks [14] on truck manufacturing companies.

## 1.3 Existing Security Solutions

To ensure defense-in-depth, preliminary investigations into MHD vehicle security [4, 5] have recommended a combination of cryptographic authentication and intrusion detection and prevention to protect the in-vehicle networks. Cryptographic authentication needs to be built into the target and can be resource intensive [15]. It can incur communication overheads when exchaning digital signatures over CAN which only allows 64 bits of data per message. There may also be cases where ECUs process malicious messages irrespective of the sender's identity and, in some cases, a legitimate sender (like a body controller with wireless connectivity) may be compromised. Intrusion detection and prevention (IDP) is a viable alternative and can be used to detect and possibly thwart malicious behavior, even from a legitimate but compromised sender. To date, two IDP solutions [16, 17] have been published for SAE J1939 networks. The rest are designed for CAN networks in passenger vehicles. Albeit, there may be an opportunity to adapt them for MHD vehicles.

There are three types of IDP systems proposed for in-vehicle networks: anomaly-based, specification-based, and rule-based. In general, anomaly-based IDP systems learn the normal behavior of the network and label unknown behavior as malicious [18]. In this way, they can detect unknown attacks but can also observe false alarms [19]. Existing in-vehicle anomaly-based IDP systems learn normal behavior from data collected during past drive cycles. This data may not reflect the different network configurations and use cases that the vehicle may go through during its life-cycle [4, 20]. As such, they may need to be retrained frequently. Specification-based systems use manufacturer specifications to build a reference model for the target system and, like anomaly-based systems, detect deviations from the built model. Unfortunately, SAE J1939-specific attacks may abide by specifications and still succeed. Rule-based systems compare information on the network with pre-defined attack signatures and raise alarms if a match is found. Rule-based IDP systems for in-vehicle networks can be designed to identify malicious messages and prevent the target ECUs from processing them [21, 22]. Unfortunately, existing systems rely on signatures defined on message content only and this may not be enough to detect cleverly crafted attacks on SAE J1939 networks. For example, a message to unlock doors is perfectly normal but may be a safety hazard if transmitted when the vehicle is in motion. Existing rule-based solutions will treat the later situation as normal and let the message pass.

## 1.4 Research Questions

Given the status of the current research on SAE J1939-specific security for MHD vehicles and the challenges to extending solutions from the passenger vehicle domain, this dissertation aims to address questions related to both the offensive and defensive sides of security for SAE J1939 networks in MHD vehicles. In particular, the following three research questions are asked

- Given that current research of offensive technologies has focussed on the application and network management layers of the SAE J1939 standards, this research asks: **can weaknesses in the data-link layer specifications of SAE J1939 be exploited to attack in-MHD vehicle ECUs?**

- Given that current research on anomaly-based intrusion detection techniques for CAN networks require frequent offline training, this research asks: **can a system be designed to detect network anomalies on an SAE J1939 network in an online manner?**

- Given that current research on rule-based intrusion detection and prevention systems cannot detect threatening SAE J1939 messages using message content only, this research asks: **can a rule-based system be designed to detect threatening SAE J1939 messages as they are being transmitted using features other than message content only?**

## 1.5 Dissertation Contributions

In general, this dissertation tries to elucidate the relatively new research topic of MHD vehicle security. In a broader sense, it also provides an incentive for future research and demonstrates that higher-level specifications on CAN (like SAE J1939) can be utilized for security research. In particular, this dissertation makes the following three contributions

- Three denial-of-service attacks are introduced on ECUs by leveraging specifications made in the SAE J1939 standards. For each attack, the research hypothesis is described, followed by the results of testing the hypothesis on different testbeds and a discussion on their physical impacts and possible defense strategies.

- An online anomaly-based intrusion detection system is presented that models network behavior through SAE J1939 specified concepts and flags abnormal deviations from normal behavior as security infringements. In the process, two new features have been introduced that model the network's behavior through SAE J1939 parsed network data. A hypothesis for attack detection is laid down based on these two features and an algorithm is presented that implements the hypothesis and tries to detect attacks as messages are received on the network.

- A rule-based intrusion detection and prevention system is presented that provides security personnel with ways to flag messages in transit that can and cannot be identified solely on the basis of their content. In the process, a novel rule structure is described that captures multiple attack detection features, example rules are demonstrated to detect real-world attacks, and a scheme is presented to enforce user-specified rules in real time, i.e. as messages are being transmitted.

## 1.6 Document Organization

Chapter 2 provides the background knowledge required to comprehend the content of the remaining chapters in this document. This includes a primer on SAE J1939 and CAN, an overview of the trucks used for experimentation, a detailed description of the threat model, and the types of attacks that can be executed on an SAE J1939 network within an MHD vehicle. In the same chapter, a review of the existing directions in in-vehicle security is also presented.

In chapter 3, the first contribution of this dissertation is described: three denial-of-service attacks that exploit weaknesses in the SAE J1939 data-link layer specifications.

In chapter 4, the second contribution of this dissertation is described: an online anomaly detection system that models network behavior through SAE J1939 specified concepts and flags abnormal deviations from normal behavior as security infringements.

In chapter 5, the third contribution of this dissertation is described: a rule-based intrusion detection and prevention system that presents the security personnel with ways to flag messages in transit that can and cannot be identified solely on the basis of their content.

In chapter 6, this dissertation is concluded with a summary of the accomplishments and future directions of work.

# 2 Background

This chapter first provides a primer on SAE J1939 and Controller Area Network (CAN), the two communication specifications that form the backbone of in-vehicle networking in MHD vehicles. In the primer, fundamentals of message exchange, timing aspects of message transmission, and critical J1939 protocols are discussed. Next, a brief description of the two trucks that have been used (directly or indirectly through collected data) throughout this research is provided. Following this, a description of our threat model is provided along with the different types of attack strategies on SAE J1939 networks. Finally, a review of the existing directions in in-vehicle security is provided with an analysis of their ability to be adopted for SAE J1939 networks in MHD vehicles.

## 2.1 Primer on SAE J1939 and Controller Area Network (CAN)

SAE J1939 [2] standard is organized in layers, much like the ISO/ISO standards [3] for IT networking systems. Each layer is described through one or more standard documents as shown in Table 2.1. This section describes information in these documents that is critical to understanding the contents of this dissertation. It begins by describing the fundamentals of message processing in SAE J1939. It then describes the low-level frame formats used in message exchange. This is followed by a description of parameter placement in SAE J1939 messages as well a brief overview of transmission scheduling. Finally, it describes three SAE J1939 specified protocols that are referred to on multiple occasions in this dissertation.

### 2.1.1 Message Processing

Information on SAE J1939 networks is exchanged using messages. The structure of an SAE J1939 message is shown in Figure 2.1. It consists of four fields. Starting from the right, the

| **SAE J1939 Documents** | **Layer** |
|---|---|
| SAE J1939-11, SAE J1939-13, SAE J1939-14, SAE J1939-15, SAE J1939-16 | Physical |
| SAE J1939-21 | Data-Link |
| SAE J1939-31 | Network |
| SAE J1939-71, SAE J1939-73, SAE J1939-74, SAE J1939-75, SAE J1939 Digital Annex | Application |
| SAE J1939-81, SAE J1939-82, SAE J1939-84 | Network management |

**Table 2.1:** SAE J1939 Document Organization

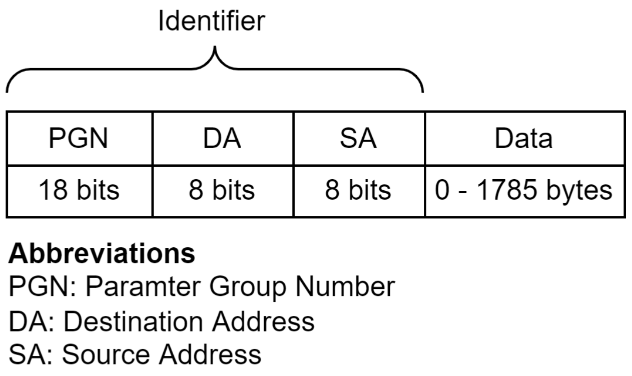


**Figure 2.1:** SAE J1939 Message Format

data field carries a parameter group (PG). A PG is a collection of vehicle parameters that are related to the same function. For example, the vehicle cruise control function utilizes wheel-based vehicle speed and cruise mode, among others, as defined in SAE J1939-71 under the Cruise Control/Vehicle Speed parameter group. In SAE J1939 parameter groups are assigned an 18-bit unique number called the Parameter Group Number (PGN) that is also present in the message. Aside from this, the message also carries a source address (SA) and destination address (DA) identifying its sender and receiver. CAN is a broadcast medium but nodes on an SAE J1939 network are assigned addresses. Every controller application connected to the bus must have a

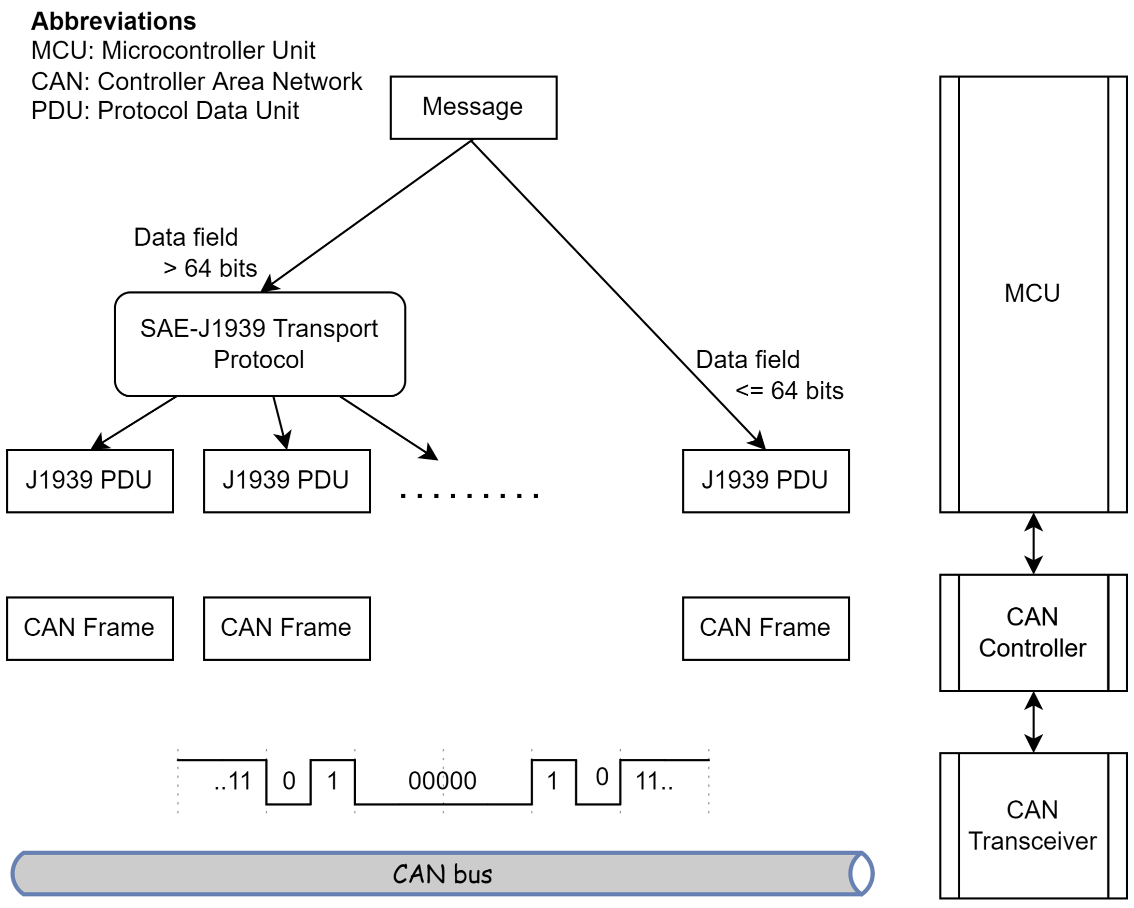


**Figure 2.2:** SAE J1939 Message Processing

unique 8-bit address, some of which are standardized by J1939. For example, the engine controllers are assigned standard addresses of $00_{16}$ and $01_{16}$. Addresses can also be claimed dynamically at vehicle startup time through the address claim protocol described later in section 2.1.4.

SAE J1939 messages are communicated on the CAN bus through J1939 protocol data units (PDU) that are in turn placed into CAN frames as shown in Figure 2.2. The format of the J1939 PDU and CAN frame is described in the next subsection. CAN allows a maximum of 64 bits of data to be transmitted at one time. As such, J1939 PDUs carry 64 bits of message data only. If the size of the message is greater than 64 bits, SAE J1939 specifies the use of the transport protocol (defined in the SAE J1939/21 document [23]) to split the data field into multiple PDUs and communicate each PDU reliably from source to destination. Details on this protocol are described later in section 2.1.4. Currently, only 9% of the non-proprietary messages defined in the SAE

J1939-71 document are allowed to have a size greater than 64 bits. These messages are typically used to exchange configuration information, not physical parameters like speed, torque, etc. To that end, this research does not consider messages of size greater than 64 bits, otherwise referred to as “multipacket messages” [24]. It only considers messages whose data content is less than or equal to 64 bits, including special messages used by the protocols described later in this section.

Usually, SAE J1939 messages are handled by the microcontroller unit of the ECU. The CAN controller of the ECU handles CAN frames and ensures that CAN communication abides by the specifications [6, 7]. The CAN controller communicates bits of the CAN frame with a transceiver using non-return-to-zero (NRZ) line encoding as shown at the bottom of Figure 2.2. In CAN terminology a 0 bit is referred to as dominant while a 1 bit is referred to as recessive. The width of a bit depends on the baud rate of the CAN bus. For example, a baud rate of 250 kbps implies a bit width of $\frac{1000}{250}$ = 4 $\mu$sec. SAE J1939 recommended baud rates are 250 and 500 kbps. To ensure time synchronization for bit sampling, a bit with the opposite polarity is always transmitted after five consecutive bits with the same polarity. This procedure is referred to as bit stuffing.

### 2.1.2 Data-Link Frame Formats

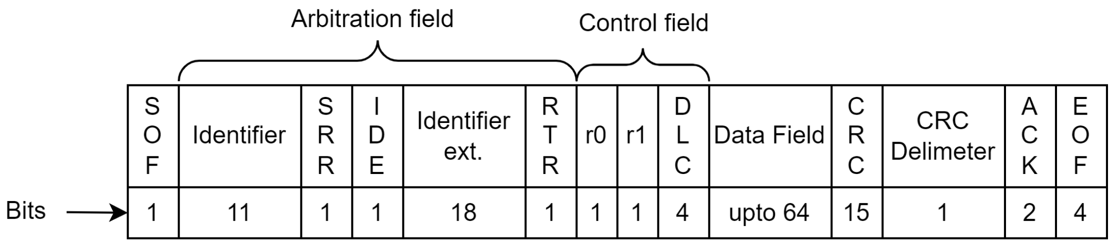
Frames for communication between ECUs are mentioned in the data-link layer of the SAE J1939 standards [23]. There are two levels of frames used, both of which have been introduced in the previous subsection. The first is the SAE J1939 protocol data unit (PDU) which is shown in Figure 2.3. The SAE J1939 PDU is further transmitted using the controller area network (CAN) frames, the format for which is shown in Figure 2.4.

An SAE J1939 PDU consists of 6 fields namely priority(pr), extended data page (EDP), data page (DP), PDU format (PF), PDU specific (PS), and source address (SA). (Pr)iority bits are used for message arbitration described later in this subsection. EDP (Extended Data Page) and DP (Data Page) are 1-bit values and can together assume only a pair of standardized values $00_2$ and $01_2$. These two fields consist the first two bits of the PGN from the most significant bit side. The next eight bits of the PGN are transmitted in the PF field of the J1939 PDU. The PS field either

| Pr | EDP | DP | PF | PS | SA | Data |
|---|---|---|---|---|---|---|
| 3 bit | 1 bit | 1 bit | 8 bits | 8 bits | 8 bits | 0 - 64 bits |

**Abbreviations**
Pr: Priority
EDP: Extended Data Page
DP: Data Page
PF: PDU format
PS: PDU Specific
SA: Source Address

**Figure 2.3:** SAE J1939 Protocol Data Unit Format



**Abbreviations**
SOF:Start of Frame
SRR: Substitute Remote Request
IDE: Identifier Extension
RTR: Remote Transmission Request
DLC: Data Length Code
CRC: Cyclic Redundancy Check
ACK: Acknowledgement
EOF: End of Frame

**Figure 2.4:** CAN Frame Format

carries the last eight bits of the PGN or the destination address field depending on the value of the PF. According to the SAE J1939 standards, if the value of the PF field is less than 240, PS is equal to the DA of the message, else it is equal to the last eight bits of the PGN. This, in turn, implies that messages with PGN 61440 to 65535 and 126976 and 131071 are not destination specific, i.e. they are broadcasted. The rest can be sent to specific destinations identified by their address or to the broadcast address 255. The SA field of the J1939 PDU carries the source address from the SAE J1939 message, while the data field carries part or whole of the data content of the SAE J1939 message, depending on its length (refer back to the previous subsection).

The content of the SAE J1939 PDU is communicated on the CAN bus through the CAN data frame, referred to hereafter simply as the CAN frame. The CAN frame begins with a recessive Start-of-Frame (SOF) bit, followed by an eleven-bit identifier field. The identifier field carries the SAE J1939 priority, the EDP and DP bits, as well as the first six bits (from the most significant side) of the PF field of the SAE J1939 PDU. The Substitute Remote Request (SRR), Identifier Extension Bit (IDE), and Remote Transmission Request (RTR) bits are always set to recessive, while the bits r0 and r1 are always set to dominant [6]. The identifier ext. field carries the last two bits of the PF field (from the most significant side) and the entirety of the PS and SA fields. Together, the identifier and identifier ext. fields form the 29-bit CAN ID. The ID, SRR, IDE, and RTR fields form the arbitration field for CAN. This field is critical for collision resolution on the CAN bus. Usually, an ECU transmits on the CAN bus when at least 11 consecutive recessive bits are seen consecutively. This indicates that the bus is idle. However, if two ECUs try to transmit at the same time, a collision may ensue. In this case, the ECU that transmits a recessive bit (1) stops transmitting and the ones that transmit dominant (0) keep transmitting. At the end of transmission of the arbitration field, the ECU that sends the lowest value for CAN ID obtains exclusive access to the bus. After this point, the control field is transmitted and contains the number of bytes ($\leq$ 8) in the data field. The trailer of the CAN frame contains a Cyclic Redundancy Check (CRC) code for error checking, a CRC delimiter, and an Acknowledgment (ACK) field that is used to acknowledge the correct receipt of the frame by at least one node on the network. Notice that bit-stuffing, introduced in the previous subsection, is only done up to the CRC field of the frame. The final field of the trailer is the End of Frame (EOF) and, as its name describes, signifies the end of the frame. Together with 3 bits of interframe space (IFS), the EOF field and the second bit of the ACK field denote the 11 consecutive recessive bits that signify an idle CAN bus.

### 2.1.3 Parameter Placement and Transmission Scheduling

All SAE J1939 parameter placement can be represented using the notation “$R.x$ - $S.w$”. Here $R$ and $S$ are byte positions and $x$ and $w$ are bit positions in the 64-bit data field. If $R$ is equal to $S$,

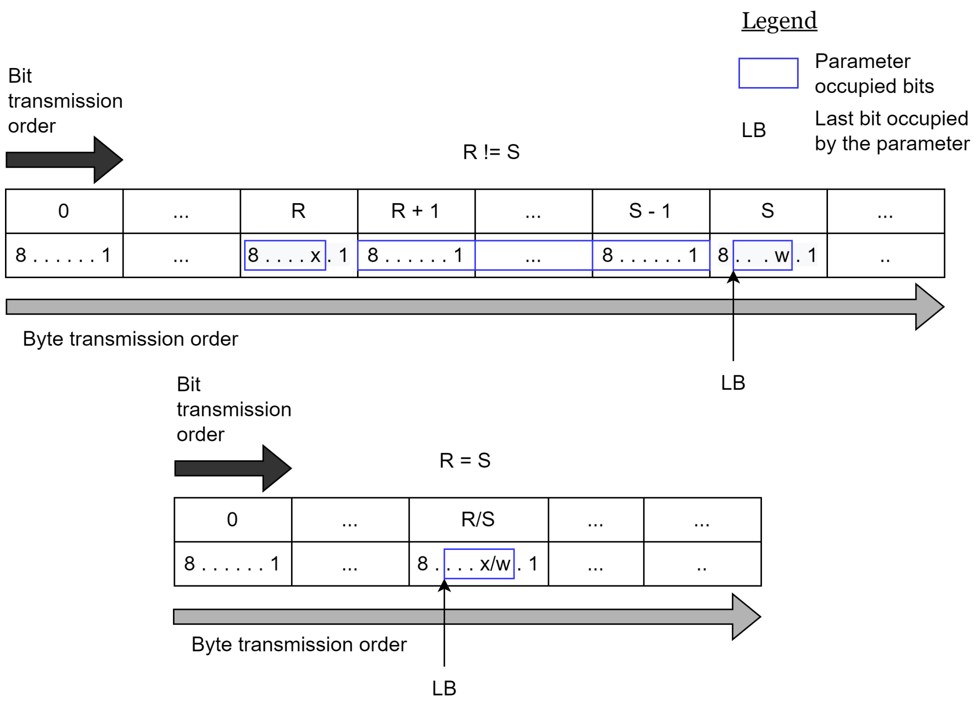


**Figure 2.5:** SAE J1939 Parameter Placement from Position *R.x* to *S.w*

then *x* is also equal to *w*. This signifies that the parameter is placed entirely in byte *R* starting from bit *x* and ending at bit *x* + *length* -1, where *length* is the total number of bits that the parameter occupies. If *R* is not equal to *S*, then the parameter occupies bits *x* through 8 of byte *R*, all bits of bytes *R*+1 through *S*-1, and the remaining bits starting from bit *w* of byte *S*. This is shown in Figure 2.5. Non-proprietary placement specifications are made in the SAE J1939 digital annex [25]. An example placement representation from the digital annex is "4.6 - 6.7" for a parameter that occupies 12 bits. This parameter is placed in bits 6 through 8 of byte 4, the entirety of byte 5, and bits 7 and 8 of byte 8. It must also be noted that SAE J1939 parameters are transmitted least significant byte first but most significant bit first i.e. byte *R* is transmitted first on the network, while byte *S* is transmitted last and within a byte, bit 8 is transmitted first while bit *x* is transmitted last.

Before being placed into the aforementioned positions, parameters are transformed using an SAE J1939 specified resolution and offset factor. In the transformation procedure, the offset is first subtracted from the value of the parameter after which the result is divided by the resolution. At the receiving end, the actual value is obtained by multiplying the resolution with the raw value obtained from the message data field and adding the offset afterward. For alphanumeric parameters, SAE J1939 does not specify a resolution and offset. As such, this transformation procedure is not required for those parameters.

An example of the SAE J1939 parameter parsing process is demonstrated below using a message with CAN ID 18FDE131$_{16}$.

$$ID\ (18FDE131_{16}) \rightarrow \underbrace{110}_{p}\overbrace{\underbrace{0}_{edp}\ \underbrace{0}_{dp}\ \underbrace{11111101}_{pf}\underbrace{11100001}_{ps}}^{pgn}\underbrace{00110001}_{sa}$$

$$Data\ (64FDFFF....) \rightarrow \underbrace{01100100}_{spn2609}111111\ \underbrace{01}_{spn7853}\ 11111111111111..$$

Since the value in the PF field is greater than 240, the PGN is calculated using values in the EDP, DP, PF, and PS fields. In this case, the binary to hexadecimal conversion yields a PGN of $0FDE1_{16}$. This PGN identifies the parameter group “Cab A/C Climate System Information" and is associated with 2 separate SPNs:

- **2609** “Cab A/C Refrigerant Compressor Outlet Pressure" is conveyed in byte 1 (from left) and calculated by multiplying the decimal equivalent with the resolution of 16 and adding to it an offset of 0.
- **7853** “Air Conditioner Compressor Status" is conveyed in byte 2 (from left) bits 0 to 1 and expressed in its raw binary form.

The actual value of the “Cab A/C Refrigerant Compressor Outlet Pressure" is calculated by multiplying the decimal equivalent of the extracted bits with the resolution and adding the offset to it i.e. $01100100_2 * 16_{10} + 0_{10} \rightarrow 100_{10} * 16_{10} \rightarrow 1600$. The unit for this particular parameter is kilo pascal (kPa). Similarly, the “Air Conditioner Compressor Status" is set to 1 which denotes that the “Air Conditioner Compressor is ON".

Most of the messages on an SAE J1939 network are transmitted periodically from ignition on to off [26]. SAE J1939 specifications provide default transmission intervals for most parameter groups [25], even though manufacturers may opt for different transmission intervals. For example, the parameter group carrying transmission controller information is specified to be transmitted at intervals of 10 milliseconds. SAE J1939 messages can also be transmitted on an ad hoc basis. These include request responses, commands, parameter groups that must be reported on a change of one or more parameter values, etc.

### 2.1.4 Protocols

This dissertation refers to three critical protocols that SAE J1939 specifies. These are used for requesting parameters, transferring multipacket messages, and dynamic address allocation.

#### Parameter Requesting

While some parameter groups are transmitted programmatically, some may have to be requested. A request is sent from one ECU to another using the SAE J1939 specified request message. This message has a PGN of 59904 ($0EA00_{16}$) and carries the requested PGN in reverse byte order in the first three data bytes. Requests can be directed to a specific device or to the broadcast domain.

A destination-specific request is answered by the receiver with the requested parameter group and/or an acknowledgment. Acknowledgment messages in SAE J1939 are assigned the PGN 59392 ($0E800_{16}$). As with the request messages, acknowledgments can be destination-specific or broadcast. For use with requests, SAE J1939 specifies four types of acknowledgment: positive acknowledgment (ACK), negative acknowledgment (NACK), access denied, and cannot respond. The value in the first data byte of an acknowledgment message determines the type of acknowledgment.

#### Multipacket Message Transfer

As already discussed, J1939 PDUs allow a maximum of 64 bits in the data field, but message data can be larger than that. In those cases, message data is transported using a different set of messages, each of which can fit into a J1939 PDU. In SAE J1939 terminology, this process is

| **Label** | **PGN** | **Parameter Placement in Data Bytes** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| Request To Send (RTS) | 60416 | 16 | Number of bytes to send | | Number of packets to send | Maximum number of packets that can be sent in response to one CTS. | PGN of the message being transferred | | |
| Clear to Send (CTS) | 60416 | 17 | Number of packets that can be sent | Next packet number to send | $FF_{16}$ | | PGN of the message being transferred | | |
| End of Message Acknowledgment (EoMA) | 60416 | 19 | Number of bytes received | | Number of packets received | $FF_{16}$ | PGN of the message being transferred | | |
| Abort | 60416 | 255 | Reason for abort | Role of Sender | $FF_{16}$ | | PGN of the message being transferred | | |
| Data Transfer (DT) | 60160 | Sequence number | Data bytes to send | | | | | | |

**Table 2.2:** Formats of Messages Used in Multipacket Message Transfer

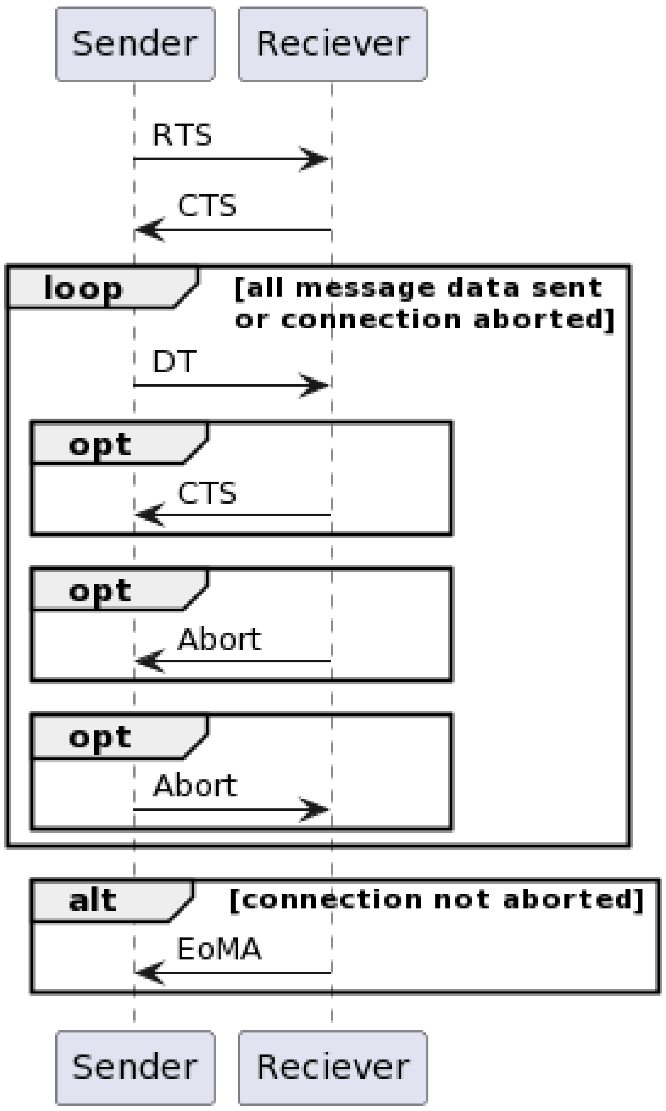


**Figure 2.6:** Multipacket Message Transfer Protocol

referred to as the multipacket message transfer. Formats of messages used in multipacket message transfer are given in Table 2.2. The actual protocol is shown in Figure 2.6 using standard UML 2.5 sequence diagram notations [27]. In there, arrows represent messages, blocks *opt* and *alt* stand for optional and alternative and block *loop* is self-explanatory. This dissertation does not deal with multipacket transfers to the broadcast address.

At first, the sending party attempts to open a connection by sending a Request to Send (RTS) message. An RTS message carries information about the total number of packets and the number of bytes to be sent. At the receiver's end, this information can be used to allocate resources to store

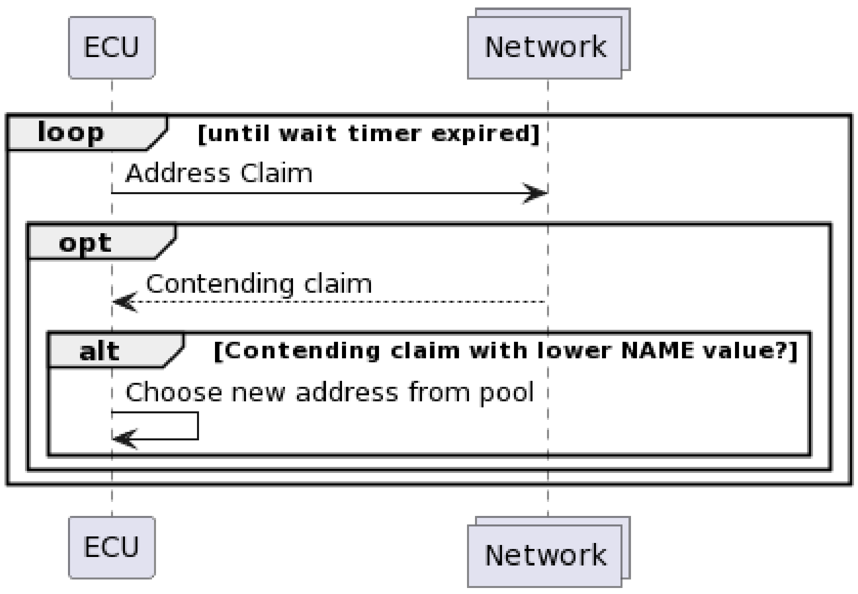

ECU
Network
loop
[until wait timer expired]
Address Claim
opt
Contending claim
alt
[Contending claim with lower NAME value?]
Choose new address from pool
ECU
Network

**Figure 2.7:** Address Claim Protocol

the incoming data. In response to the RTS, the receiver sends a Clear to Send (CTS) message. CTS messages enforce flow control by limiting the number of packets that can be sent after each CTS is transmitted. Both RTS and CTS messages use a PGN of $0EC00_{16}$ and include the transported PGN in the last three data bytes in reverse byte order. Upon receiving the CTS, the sender can send message data using the data transfer (DT) messages until all data bytes are sent. The PGN of a DT message is $0EB00_{16}$. The first byte of this message is reserved for a sequence number. Sequence numbers are used to reassemble the incoming data bytes. The remaining seven bytes of data in a DT message are used to transport the data bytes from the multipacket message being transferred. The flow of DT messages can be aborted by optionally sending an Abort message that includes information stating the reason for the abort. On successful completion of the transfer (i.e. if not aborted), an End of Message Acknowledgment (EoMA) is sent by the receiver to indicate closure of the connection.

### Address Claiming

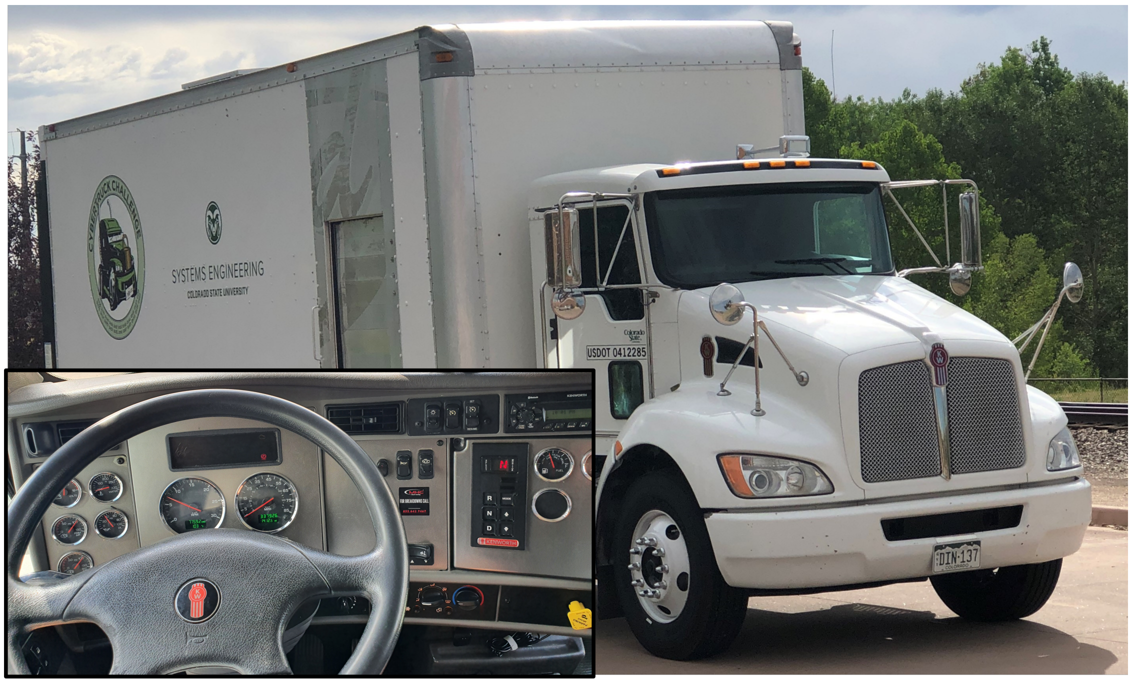
Every node on an SAE J1939 network is required to have a unique address and a unique name which, as specified in the SAE J1939 network management layer standards, is a 64-bit value referred to as NAME. At startup, every node is required to execute the address claim protocol shown in Figure 2.7.

The address claim protocol requires an ECU to send an address claim message at startup showcasing the address they claim in the SA field of the message. This message has a PGN of 60928 and carries the NAME parameter in the data field. In the SAE J1939 standards it is referred to an the address claim message. After sending the message the ECU waits until a pre-specified timer expires. During this period it may receive a contending address claim with the same value in the SA field of the message. If the value of NAME in the contending claim is lower than the ECU's NAME, the current address is relinquished, a new address is chosen from a pool of addresses, and the protocol is repeated from the start. Upon successful completion of this protocol, all ECUs on the network are assigned unique addresses.

## 2.2 Research Trucks

This dissertation performs experiments on two trucks, directly or indirectly through collected data logs. Both these trucks include(d) at least one SAE network. The first truck is a PACCAR PX-7-powered 2014 Kenworth T270 which we currently have access to. There is a Cummins CM2350 engine ECU, Bendix EC-60 brake ECU, and Allison RDS-2000 transmission ECU on the truck. All of these ECUs communicate with each other on a 250 kbps CAN bus. A picture of this truck along with its dashboard is provided in Figure 2.8a. The picture of the dashboard was taken when the truck was idling. As displayed, the idle engine speed is around 600 revolutions per minute (RPM).

The second truck is a PACCAR MX-13-powered 2015 Kenworth T660. We did not have physical access to this truck. Instead, we have access to a log that was collected from a 250 kbps CAN bus during 7 minutes of driving around an industrial block. During the drive, the driver performed some hard brake maneuvers. Figure 2.8b shows a snapshot of the truck's cabin from a video recorded during the drive. From the data log collected the following observations can be made



**(a)** 2014 Kenworth T270 Truck with a Snapshot of the DashBoard

**(b)** 2015 Kenworth T660 Truck Cabin

**Figure 2.8:** Experimental Trucks

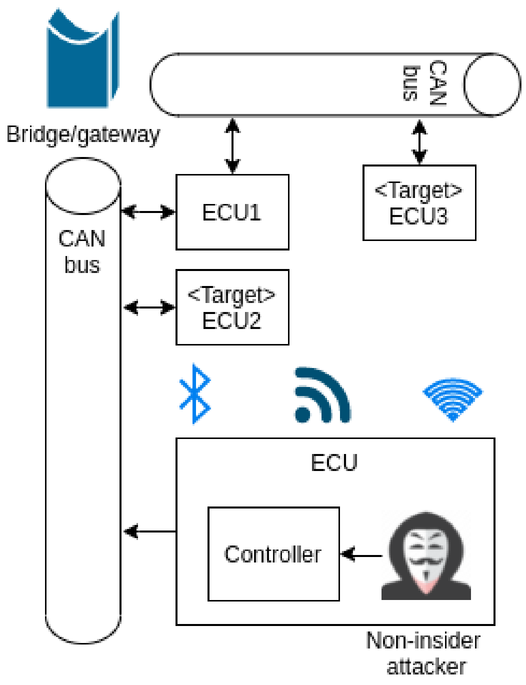


**Figure 2.9:** Attacker Access Model

- There are 137318 messages in the log.
- There are 5 unique transmitters: an engine controller, a brake controller, an engine retarder, a cab controller, and a diesel particulate filter controller.
- There are 41 unique parameter groups transmitted out of which 35 are periodic and 6 are ad hoc. The ad hoc transmitted parameter groups include request messages, responses to requests, broadcast announcement messages, and commands. The commands were sent to the engine controller when the hard brake maneuvers were performed.

## 2.3 Threat Model

This section of the background chapter presents our threat model. In particular, it discusses our assumptions about the attacker's capabilities and the attack strategies they can resort to. The intrusion detection and prevention techniques proposed in this dissertation try to defend against the attacks described in this section as well as those introduced in chapter 3 of this dissertation.

### 2.3.1 Attacker Capabilities

With physical access to the vehicle, the attacker can be considered as powerful as a trusted insider. They can attach custom-built hardware to the target network and can also remove a security device connected to it. To that end, this dissertation assumes that the attacker does not have physical access to the vehicle, and breaks in through a remote access point as shown in Figure 2.9. After breaking in, the attacker can send CAN frames through the onboard CAN controller or bypass it and send NRZ pulses (bits) directly to the transceiver. It has been shown that, in the latter case, the attacker can manipulate bits of a CAN frame, shut down another ECU on the bus and transmit on its behalf [28]. The success of this attack depends on multiple system pre-conditions (like physical access to the vehicle and/or existing vulnerabilities on the CAN controller) that may not be satisfied on the pivot device and, if launched successfully, such attacks can also be detected [29]. Given this, this dissertation assumes that the attacker cannot (or does not) manipulate CAN bits.

The attacker may be able to reprogram another ECU on the CAN bus and control the inputs to its transceiver. In this case, though, they will require elevated privileges and possibly even advanced social engineering skills. These types of elevated privileges are typically available with the "insiders" [5]: garage personnel, drivers, owners, manufacturers, etc. An insider can easily remove installed security devices or render them useless. It will be impossible to deploy any form of security on an in-vehicle network without having a complete understanding of the insider's capabilities. Therefore, this dissertation assumes that insiders are trusted and the attacker does not have insider capabilities.

As seen in Figure 2.9, this dissertation considers that the attacker's target can be on the network to which they have direct access or on a different network, in which case they can bypass gateway firewalls if required. They can interpret and create SAE J1939 messages as well as spoof the source address of another ECU while conducting an attack.

### 2.3.2 Attack Strategies

With the ability to send CAN frames onto the network, the attacker can launch in-vehicle cyberattacks. Attacks can be specific to MHD vehicles or can be extrapolated from the passenger vehicle domain. This dissertation considers the following types of attacks :

#### High Volume Denial-of-service (HVDoS)

This type of attack tries to consume the resources available to the ECUs through rapid injection of SAE J1939 messages.

#### Published Attacks

**Network Overload Attack** Miller et al. [30] have demonstrated that by rapidly injecting frames with CAN ID 0 network bandwidth available to the ECUs can be consumed completely. Recall from section 2.1.2 that in case of a collision, the CAN ID with the lowest value wins bus arbitration. 0 is the lowest-valued ID and can be used to clog the entire bus even if a collision occurs. Through experiments on the Kenworth T270 research truck, we observed that this attack halts a running truck and prevents it from moving forward. Miller et al. observed the same on a Ford Escape and Toyota Prius. This dissertation refers to this attack as the “network overload attack”.

#### Low Volume Denial-of-service (LVDoS)

This type of attack tries to disable ECU services by injecting messages at a normal rate.

#### Published Attacks

**Address Claim Attack** Recall from section 2.1.4 that an ECU relinquishes its address if a contending claim is received with a lower NAME value. Murvay et al. [13], leverage this behavior to demonstrate that an ECU can be rendered defunct on the network by repeatedly sending address claim messages with the target’s address in the SA field and all 0s in the data field until all addresses in the ECU’s pool are claimed. This dissertation refers to this attack as the “address claim attack". Murvay et al. [13] did not conduct this attack on an actual vehicle. We do so on the Kenworth T270 research truck and observe that it stops all communication to and from the target ECU. If this ECU is the engine controller, it halts automatic transmission and safety-critical functions like anti-lock braking and traction control. It also shows erroneous information on the dashboard.

### Command and Control (CnC)

This type of attack tries to control the cyber-physical functions of ECUs by sending one or more command messages defined in the SAE J1939 standards [31]. CnC attacks to open the doors of the vehicle, activate the windshield wipers, etc. have been demonstrated on passenger vehicles [32, 30] and MHD vehicle security researchers believe that the same can be executed with greater ease on MHD vehicles due to the open availability of message formatting specifications made in the SAE J1939 standards.

### Published Attacks

**Engine control Attack** Burakova et al. have demonstrated that continuous control over the engine can be established by sending SAE J1939 defined ad hoc messages with PGN $00000_{16}$. This dissertation refers to this attack as the "engine control attack".

**Retarder Jam Attack** Burakova et al. [12] have demonstrated that the truck’s ability to use engine braking (retardation) can be disabled by commanding 0% torque to the engine retarder in messages with PGN $00000_{16}$ at speeds below 30 mph. This dissertation refers to this attack as the "retarder jam attack".

**Throttle Jam Attack** In [12], Burakova et al. have also demonstrated that the driver’s input to the accelerator pedal can be effectively cut off by sending very low torque requests to the engine controller in messages with PGN $00000_{16}$. This dissertation refers to this attack as the "throttle jam attack".

**Fuzzing**

As a precursor to the above-mentioned attacks, the attacker may require random fuzzing to determine the capabilities of the target ECU. Fuzzing involves sending random CAN ID and data bytes sent at high rates [33]. In some cases fuzzing may even lead to unseen and possibly safety-critical scenarios [34] on MHD vehicles.

**Published Attacks** There are no known instances of fuzzing attacks published in MHD vehicle security literature. As such, we tried conducting such an attack and witnessed that the gauges on the instrument cluster moved in an erratic manner. No effect was noticed on the motion of the vehicle.

Fuzzing and HVDoS attacks on CAN networks have been reported on multiple occasions [15]. Albeit, to date, published LVDos and CnC attacks have been specific to SAE J1939. Replay attacks, with or without data modification, have also been shown to work on CAN networks [30, 32]. A replay of periodic traffic inevitably leads to a violation of periodicity due to the retransmission of the same PGN before the expiry of the period. This behavior has been shown to be easily detectable [35, 36, 37, 38, 39] and in some cases has been used to shut off the attacker's pivot point [40, 41]. This dissertation assumes that the attacker tries to stay stealthy by not conducting a replay attack.

## 2.4 Review of In-Vehicle Security Solutions

Researching security solutions for in-vehicle networks began when the first threats were perceived. In general, there have been four directions of research: cryptography-based, anomaly-based, specification-based, and rule-based. Although, the majority of it has focused on CAN networks in passenger vehicles [55]. Only a handful of papers have been written for SAE J1939 network security in MHD vehicles. To that end, this section reviews the four general directions of in-vehicle security research. A summary of this review is presented in Table 2.3.

| **Research Direction** | **References** | **Methodology** | **Pros** | **Cons** |
|---|---|---|---|---|
| Cryptography-based | [42, 43, 26, 13, 44] | Authenticate messages through digital signatures created using pre-shared keys | No false positives | • Resource intensive<br>• Unable to detect attacks from legitimate but compromised senders<br>•Introduces communication overhead<br>• Key management can be challenging |
| Anomaly-based | [45, 46, 35, 36, 37, 38, 39, 47, 16, 17] | Learn normal behavior from offline collected data and flag abnormal deviations from normal as attack | Can detect unknown (0-day) attacks if it causes significant deviations from normal behavior | Does not account for normal behavior that is not encountered during the training phase |
| Specification-based | [20, 48] | Build reference model for normal behavior using manufacturer specifications and flag deviations from normal as attack | Can detect unknown (0-day) attacks if they violate specifications | Unable to detect attacks that obey specifications |
| Rule-based | [49, 50, 51, 52, 21, 53, 41, 54] | Create a database of attack patterns based on CAN ID and data and flag frames as malicious if matching patterns are found in the database | Low false positives | Not all malicious CAN frames can be identified on the basis of their ID and data |

**Table 2.3:** Summary of Exiting In-Vehicle Security Solutions

**Cryptography-based In-Vehicle Security**

There has been extensive research on cryptographic authentication for CAN networks in passenger vehicles [42, 43], and lately, some papers have been published for SAE J1939 networks in MHD vehicles [44, 26, 13]. Albeit, prior work has pointed out that cryptographic solutions are difficult to deploy on in-vehicle networks due to their heavy computational requirement [15]. Even if such a deployment is made, it cannot defend against attacks from legitimate but compromised transmitters or if the victim ECU processes malicious messages irrespective of the sender's identity. For example, on the Kenworth T270 research truck, the engine controller accepts engine control commands from body controller units. Although there does not exist one in the Kenworth T270 research truck, modern body controller units come with wireless connectivity [56] that can be used to infiltrate into the in-vehicle network if remotely exploitable vulnerabilities are present on the unit and they are exploited.

Aside from this, there can be two major roadblocks. Firstly, cryptographic solutions will incur communication overheads if implemented on top of CAN. This is because CAN allows only 64 bits of data per frame and most of this data is used for parameter placement. Transmission of digital signatures will therefore require a change in the specifications or a separate message must be sent with the digital signature. In the latter case, the recipient of the message must wait for the digital signature to arrive. This will increase the processing time of a message and can be detrimental to the performance of time-critical systems like MHD vehicles. Key management in the interoperable SAE J1939 networking environment is a second issue. What happens when a trailer is attached to a tractor? How should keys be shared with ECUs in the trailer? Dynamic key distribution can be managed by key-granting servers which authenticate the requester before granting a key. However, this increases the number of unanswered questions. How should the requester's legitimacy be established? What if an attacker poses as a legitimate trailer ECU and asks for a key in order to communicate with their target ECU(s)? What if an ECU fails during a long haul? How should key revocation be handled? Cryptographic key management for in-vehicle networks is an ongoing field of research and may be a particularly hard problem for MHD vehicles [34].

### Anomaly-based In-Vehicle Security

Anomaly-based systems learn the normal behavior of the network and label unknown behavior as malicious. In this way, they can detect unknown attacks but can also observe false alarms [19]. Current research on in-vehicle anomaly detection assumes that the normal behavior of the network remains fairly static over the course of the vehicle's lifecycle [4]. As such, it learns normal behavior from data collected during a limited set of past drive cycles. This (training) data, however, may not reflect the different network configurations and behavioral variations that the vehicle may go through during its life-cycle [4, 20]. As such, current solutions for in-vehicle anomaly detection may need to be retrained frequently.

In some cases [45, 46] researcher use features related to the transmitting voltage on the CAN bus to fingerprint ECUs and compare the fingerprints with their runtime behavior. Albeit, voltage characteristics of ECUs can change depending on the ambient temperature and supply voltage [57]. Voltage-based fingerprints will also need to be reevaluated if a new ECU (possibly belonging to a trailer unit) is added or an existing one is replaced. In some other cases researchers rely on the fact that traffic on the CAN bus is periodic [35, 36, 37, 38, 39] and abnormal variations in periodicity can indicate attacks. Clearly, these techniques do not consider the transmission of ad hoc messages, some of which may not even be present in the training data. Finally, although low in number, some works [47, 16, 17] try to predict parameter values from historic data collected during past drive cycles and compare the predicted value with the actual to detect intrusions. Clearly, the prediction can be erroneous in some situations if the historic data is not available for those situations.

### Specification-based In-Vehicle Security

Specification-based systems use manufacturer specifications to build a reference model for the target system and, like anomaly-based systems, detect deviations from the built model. Studnia et al. [20] model ECU behavior through a set of logical expressions constructed from manufacturer specifications and propose implementing their solution on the ECUs. Larson et al. [48] model the behavior of the network as deterministic finite automata and suggest deploying their solution on a centralized intrusion detection system.

The major issue with specification-based approaches is that SAE J1939-specific attacks may abide by specifications and still succeed. For example, consider the engine control attack. Transmission of a message with PGN $00000_{16}$ at a pre-specified rate can be enough to execute the attack. This does not violate any specifications made in the SAE J1939 standards.

### Rule-based In-Vehicle Security

Rule-based systems compare information on the network with pre-defined attack signatures and raise alarms if a match is found. Rule-based intrusion detection and prevention systems for in-vehicle networks can be designed to identify malicious messages and prevent the target ECUs from processing them [21, 22]. Most rule-based techniques [49, 50, 51, 52, 21] suggest comparing the received CAN ID with a predefined whitelist or blacklist of IDs and filter them accordingly. This technique has also been adopted in commercial solutions like NXP's secure TJA115x CAN transceiver [53]. Some [41, 54] suggest inspecting the CAN data field bytes for specific patterns that indicate malicious behavior.

Unfortunately, the content of a single message (ID or data) may not be enough to detect cleverly crafted attacks on SAE J1939 networks. For example, a message to unlock doors is perfectly normal but may be a safety hazard if transmitted when the vehicle is in motion. Existing rule-based solutions will treat the later situation as normal and let the message pass.

# 3 Denial-of-Service Attacks on the SAE J1939 Transport Layer

Offensive research on SAE J1939 has discovered weaknesses in the application [12] and network management layers [13] of the standards but there are layers in the standards whose security has still not been analyzed. In this chapter, we investigate security weaknesses at the data-link layer of the standards. To that end, we present three denial-of-service attacks that exploit weaknesses in protocol specifications made in the SAE J1939 data-link layer document [23].

We begin by describing the testing setups used to conduct the experiments pertaining to the attacks. Next, we describe the three attacks. For each attack, we state our research hypothesis, the results of testing the hypothesis on the testbeds, and a discussion on the impact of the attacks and possible defense strategies.

## 3.1 The Testing Setup

To ensure the consistency of the experiment results, we conducted our experiments on multiple testbed configurations. The first set of experiments was conducted on a remote testbed setup at the University of Tulsa. To revalidate the results, we conducted a second set of experiments on a testbed set up locally. In this section, we describe the remote testbed and then the local setup.

### 3.1.1 Remote Testbed

The remote testbed is shown in Figure 3.1. It consisted of a single CAN backbone operating at a baud rate of 250 kbps. The CAN backbone connected a Caterpillar ADEM 2000 Engine Control Module (ECM)and a Bendix Electronic Brake Controller (EBC) (address = $0B_{16}$). The ECM hosted an engine controller (address = $00_{16}$) and a retarder controller (address = $0F_{16}$) applica-

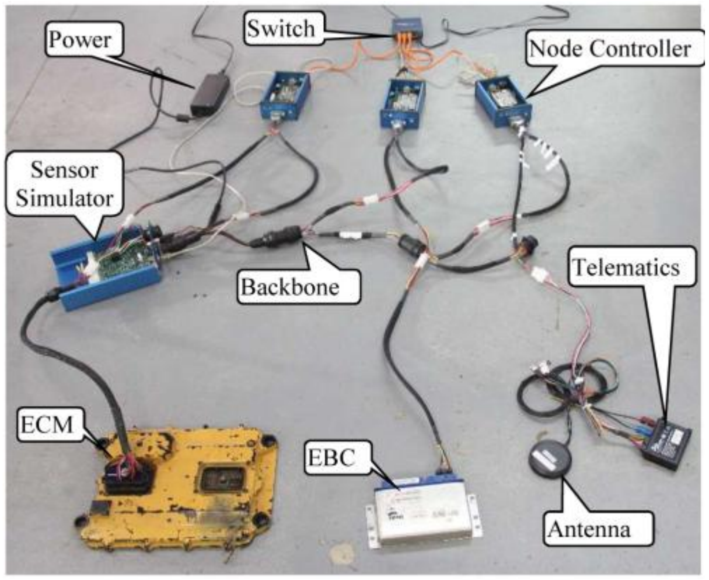
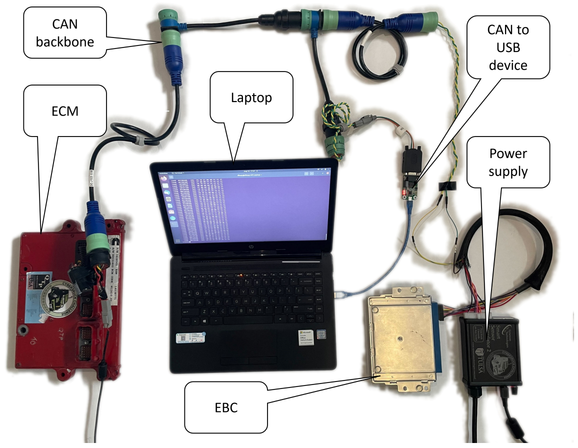


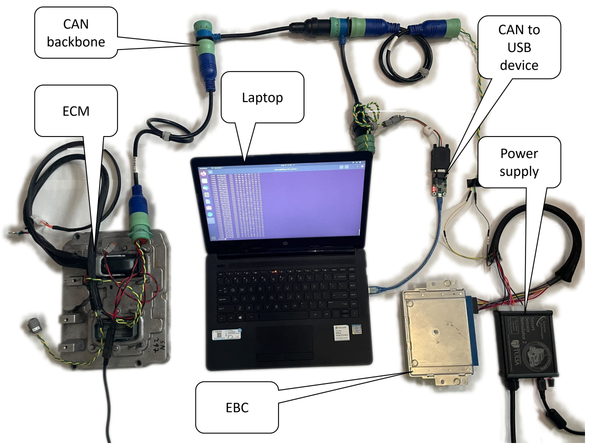
**Figure 3.1:** Remotely Accessible Experiment TestBed

tion. The testbed also included three custom-built BeagleBone Black[1] node controllers that were equipped with CAN controllers and provided access to the CAN network through the CAN-utils[2] software installed on a 32-bit Linux distribution. We only made use of two of these devices that we refer to as `BB1` and `BB2`. For some attacks `BB1` and `BB2` were used as pivot points, and for others, they were used as dummy targets. Along with these, there was a sensor simulator device that forwarded all CAN traffic to and from the ECM and a telematics control unit that we did not make use of.

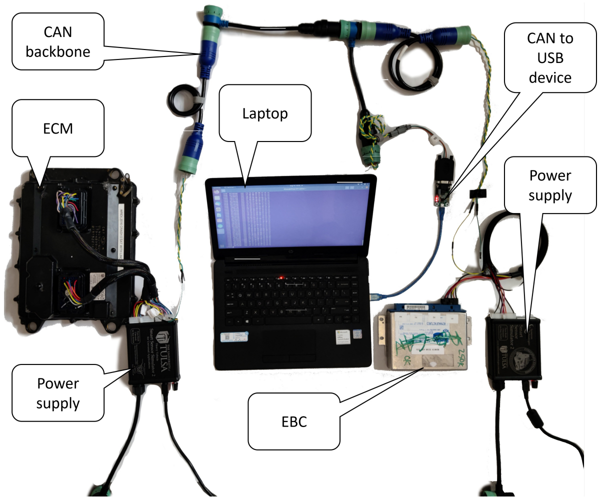
Most traffic on the CAN network was transmitted periodically. The ECM was responsible for the majority of the traffic volume (93.39%) in the testbed while the EBC was responsible for the rest. Also, 30% of the traffic was high priority ($3 \geq$ SAE J1939 message priority $\geq 0$), all of which was transmitted by the ECM.



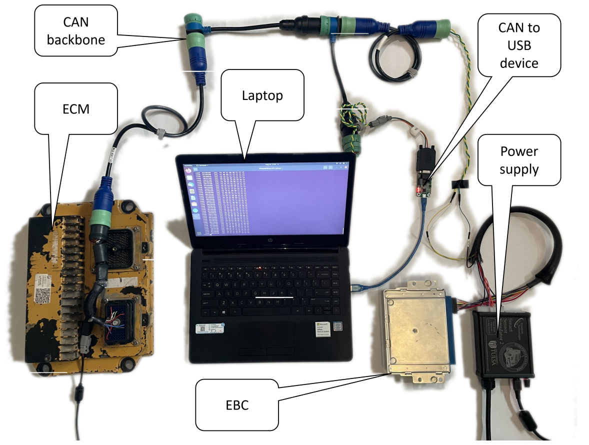
**(a)** Testbed 1



**(b)** Testbed 2



**(c)** Testbed 3



**(d)** Testbed 4

**Figure 3.2:** Four Different Testbed Configurations Setup Locally

### 3.1.2 Local Testbed

The local testbed consisted of four different configurations (*testbed 1*, *testbed 2*, *testbed 3*,and *testbed 4*) as shown in Figure 3.2. Each testbed configuration hosted a different ECM (engine control module) from a different manufacturer: *testbed 1* hosted a Cummins 870 ECM, *testbed 2* hosted a Cummins 2350 ECM, *testbed 3* hosted a Caterpillar ADEM 3 ECM, and *testbed 4* hosted a CaterPillar ADEM 4 ECM. Similar to the remote testbed, each testbed setup included a Bendix Brake Controller and each ECM hosted an engine controller and a retarder. A CAN backbone connected the ECM to an EBC (electronic brake controller) and a Linux laptop that was used to capture and transmit CAN messages through a CAN to USB device accessed via the CAN-utils software. The laptop was also used as the attacker's pivot point and, in some cases, as benign nodes that participated in the attack(s). One or more power supplies were also included.

Most traffic on the CAN network was transmitted periodically for each configuration. The ECM was responsible for the majority of the traffic volume in each of the four cases (65.11% in testbed 1, 66.31% in testbed 2, 62.56% in testbed 3, 70.76% in testbed 4) while the EBC was responsible for the rest. In testbed 1, 44.18% of the traffic was high priority ($3 \geq$ SAE J1939 message priority $\geq 0$) out of which 73.68% was transmitted by the ECM. In testbed 2, 31.28% of the traffic was high priority out of which 78.63% was transmitted by the ECM. In testbed 3, 54.68% of the traffic was high priority out of which 77.47% was transmitted by the ECM. In testbed 4, 43.46% of the traffic was high priority out of which 77.87% was transmitted by the ECM.

## 3.2 The Request Overload Attack

The first of the three attacks was coined the name "Request Overload" as it involves overloading the target ECU with requests for attacker-chosen parameter groups. This attack was conducted on

[1]https://beagleboard.org/black

[2]https://github.com/linux-can/can-utils

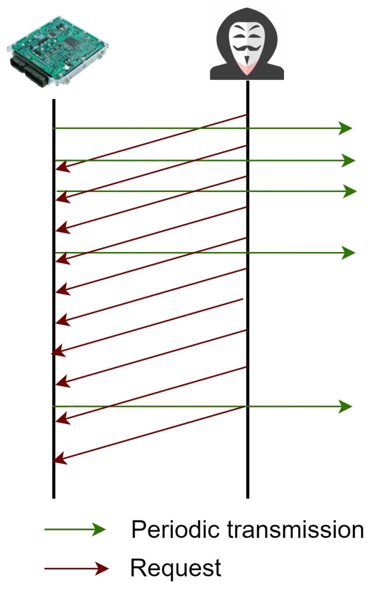


**Figure 3.3:** Request overload attack

each of the testbed setups and had a noticeable effect on each of them. We noticed that it also had a physical effect on the Kenworth T270 research truck.

### 3.2.1 Hypothesis

The SAE J1939/21 document specifies that all directed requests to an ECU must be processed. An attack can thus be constructed to send a high volume of SAE J1939 requests to the target ECU with the expectation that in an attempt to serve the sent requests, the ECU fails to perform regular, more critical tasks like transmission of periodic messages (Figure 3.3).

### 3.2.2 Testing

#### Testing on the Remote Testbed

To validate the hypotheses, an experiment was conducted on the remote testbed described at the beginning of this chapter. The ECM in the testbed was targeted with a high volume of requests for the component identifier. The component identifier was chosen as the requested parameter group because it is of a size greater than 64 bytes and will require the ECU to perform additional processing to initiate the multi-packet transfer. It was observed that, towards the beginning, the

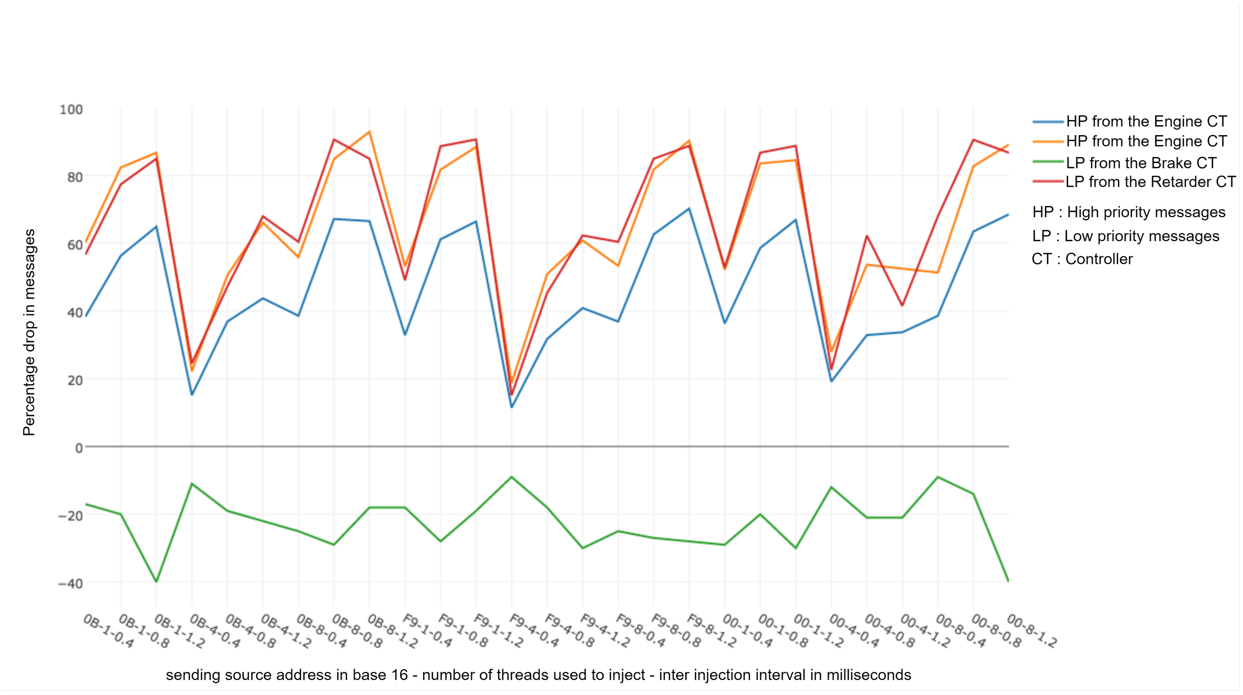


**Figure 3.4:** Request Overload Experiment Results on the Remote Testbed

ECM was able to handle the requests but gradually acknowledgment messages were being sent out to convey its inability to respond to the rapidly sent requests (refer back to section 2 for a review of acknowledgment messages in SAE J1939). There was also a significant drop in the count of normal messages transmitted by the ECM. This is shown in Figure 3.4. The y-axis of the plot shows the percentage reduction in high ($3 \geq$ priority $\leq 0$) and low ($7 \geq$ priority $\leq 4$) priority periodic messages transmitted by the controllers (engine controller, retarder, and brake) on the testbed network in one second. The percentage reduction is calculated as

$$\frac{(|messages_{s,pr} \text{ before the attack}| - |messages_{s,pr} \text{ after the attack}|) * 100}{messages_{s,pr} \text{ before the attack}}$$

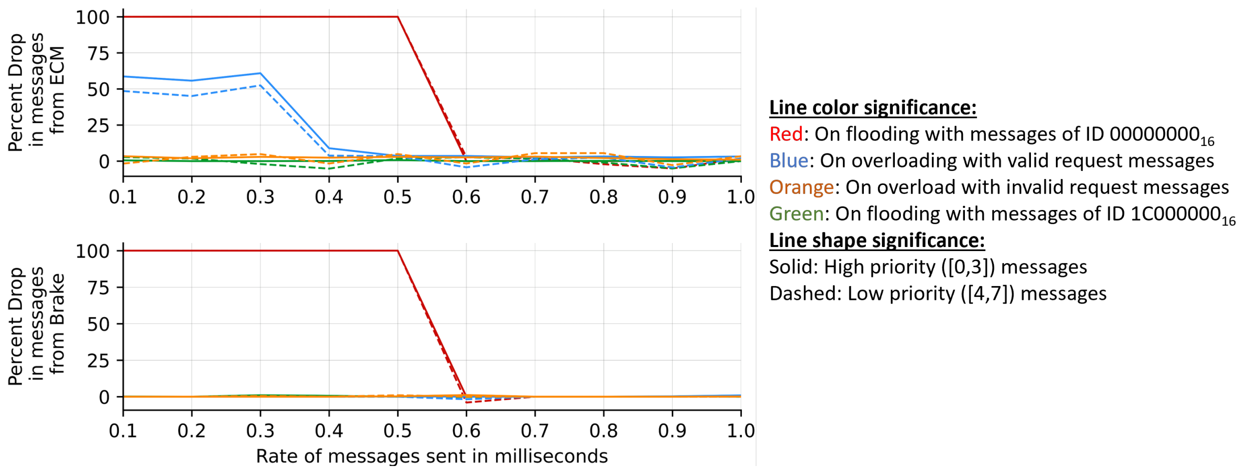
Here $messages_{s,pr}$ stands for the set of messages from sender $s$ with priority $pr$, $s \in \{$engine controller, retarder, EBC$\}$ and $pr \in \{$low,high$\}$. The x-axis ticks are instances of the triple: *injecting source address in base 16*, *number of threads used to inject* ($\in \{1, 4, 8\}$), *inter injection interval in millisecond* ($\in \{0.4, 0.8, 1.2\}$). These three independent factors, with three values for each, were chosen to see if the attack was successful in different settings. For all 27 combinations of factor values, a positive decrease was observed in ECM traffic. In contrast, additional messages were transmitted by the EBC.

**Testing on the Local Testbed**

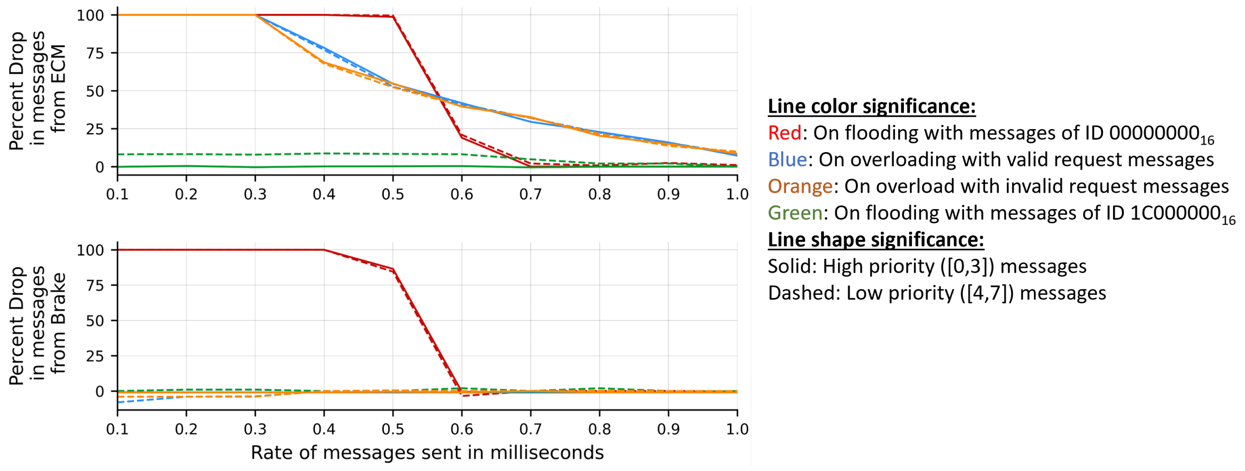
The experiment that was executed on the remote testbed had four shortcomings.

- Firstly, it was not investigated if the drop in count was because of a request overload or messages losing arbitration to higher priority request messages.
- Secondly, it was not investigated if the rate of injection of the request messages had any relation with the effect of the attack.
- Thirdly, it was not clear if the sensor simulator was dropping any messages while forwarding traffic to and from the engine controller.
- Fourthly, it was not investigated if requesting a Parameter Group Number (PGN) that is not present with the engine controller had any effect on the network traffic.

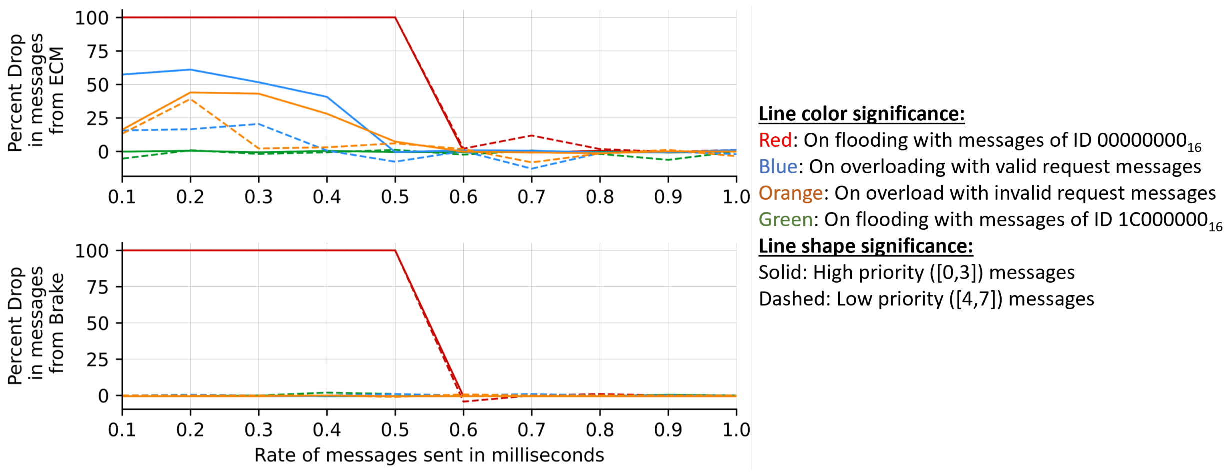
While conducting our experiments on the local testbed, we addressed these shortcomings. We sent four different types messages on the CAN network with varying intervals of transmission: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 1 milliseconds. The first type of message had a CAN ID of $00000000_{16}$. This is the highest priority CAN frame and is expected to flood the entire bus if sent at high rates (c.f. high-volume DoS attacks from section 2.3). The second type of message had the lowest SAE J1939 priority (7), but a 0 PGN (Parameter Group Number), 0 DA (destination address), and 0 SA (source address), i.e. the equivalent CAN ID was $1C000000_{16}$. The third type of message had the same SAE J1939 priority (7) but was a request (PGN $EA00_{16}$) from the engine controller to itself i.e. it had a CAN ID of $1CEA0000_{16}$. We did make sure that the engine controller responded to this request. Also, this request was for the component identifier that was present with all the ECMs used in the experiment. We refer to this as the *valid* request. Finally, the fourth type of message was a request for a PGN $FFFF_{16}$ that was not present with any engine controller. We refer to this as the *invalid* request.



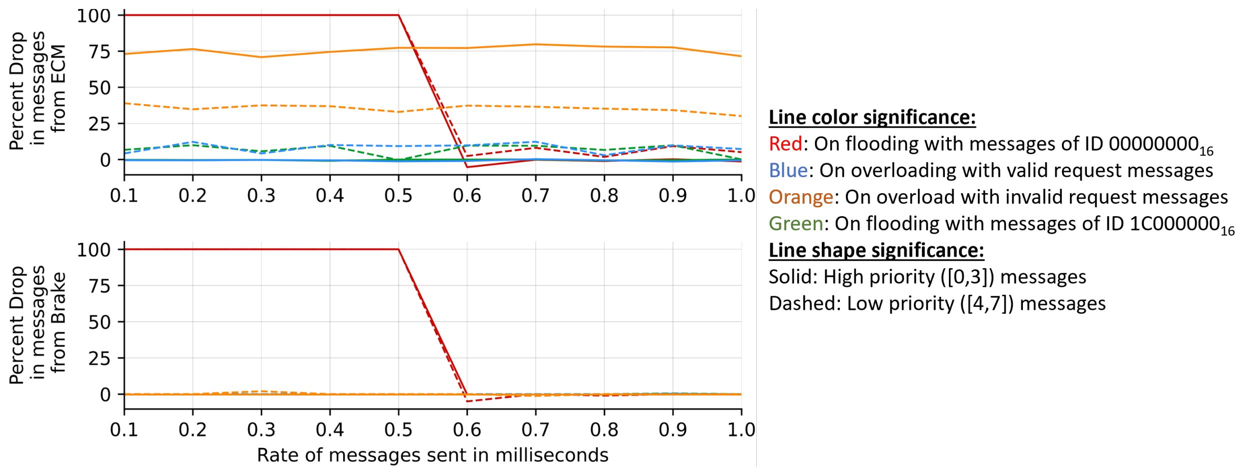
**(a)** Request Overload Results on Local Testbed 1



**(b)** Request Overload Experiment Results on Local Testbed 2



**(c)** Request Overload Experiment Results on Local Testbed 3



**(d)** Request Overload Experiment Results on Local Testbed 4

**Figure 3.5:** Request Overload Experiment Results on Different Configuration of the Local Testbed

The purpose of sending the invalid request was to address the final shortcoming. Because there was no forwarding device attached to the ECM in the local testbed the third shortcoming was addressed implicitly by the design. The different rates were chosen to address the second shortcoming and the different CAN IDs were chosen to address the first shortcoming. As such, our goal was to demonstrate that the while messages with CAN ID $00000000_{16}$ replace all normal messages on the bus when transmitted at high rates, messages with CAN ID $1C000000_{16}$ which only have a lower SAE J1939 priority, cannot replace any when transmitted at the same rates. If, on the other hand, requests with CAN ID $1CEA0000_{16}$ removed any normal messages transmitted by the target ECU, we could conclude that request overload had an effect on the processing capability of the target. This is simply because $1CEA0000_{16}$ is greater than $1C000000_{16}$ and according CAN 2.0 [6], frames with ID $1CEA0000_{16}$ have lower arbitration priority than frames with ID $1C000000_{16}$. Therefore, if transmitted at the same rate, messages with CAN ID $1CEA0000_{16}$ should not replace any normal messages on the network that messages with ID $1C000000_{16}$ cannot unless request overload is successful.

As it can be seen from Figure 3.5, for all the test cases a drop in message count was observed from all sources when the network was flooded with messages with CAN ID $00000000_{16}$. The results were in accordance with observations made in related works [30, 33]. Then again, in all the

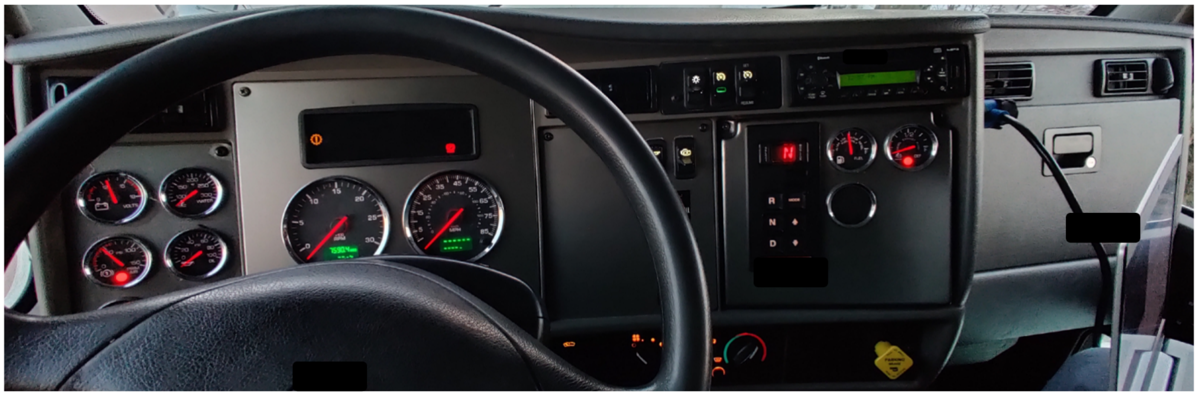


**Figure 3.6:** Effect of Request Overload on the Kenworth T270 Truck's Dashboard

cases of flooding with messages of CAN ID 1C000000$_{16}$ almost no normal traffic was removed. However, when the request overload attack was conducted, a certain percentage of normal messages transmitted by the ECM was removed. Albeit, the traffic volume from the brake controller remains constant during request overloads, thus indicating that it only affected the performance of the target. On testbed 1, about 50% of both high and low-priority traffic were removed by valid request overloads up to 0.3 milliseconds, but invalid request overloads did not have any effect. On testbed 2, all traffic transmitted by the ECM was removed by both valid and invalid request overloads up to 0.3 milliseconds. On testbed 3, greater than 20% of both high and low-priority traffic were removed by valid and invalid request overloads up to 0.2 milliseconds. On testbed 4, about 75% of high and 25 % of low-priority traffic was removed by invalid request overloads up to a millisecond but valid requests had no effect: it handled all valid requests promptly, initiating a multi-packet transfer in all cases. Overall, we noticed that, in general, a request overload attack launched at 0.3 milliseconds or below, had an effect on the target ECM, even when launched with the lowest priority.

### 3.2.3 Discussion

This attack can be classified in the high-volume DoS category. We conducted this attack on the Kenworth T270 research truck. This truck had the same ECM as used in the local testbed 2.

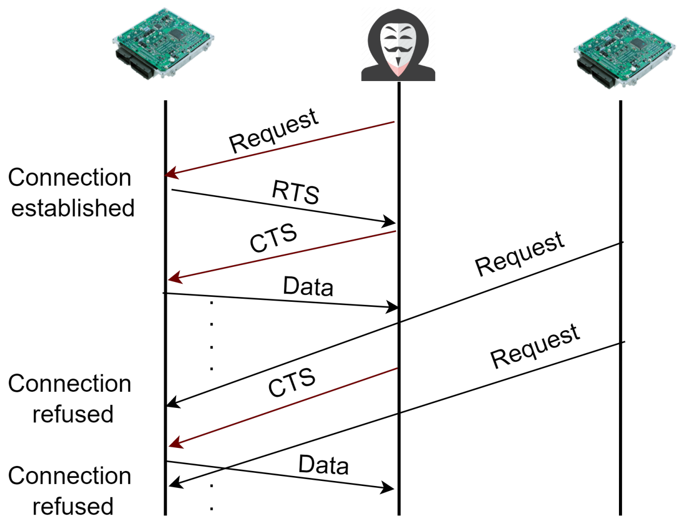


**Figure 3.7:** Connection Exhaustion

As such, when a request overload attack was conducted at 0.3 milliseconds, all transmissions from the ECM stopped. The physical effect could be seen in Figure 3.6 where the truck's dashboard displays erroneous information. Furthermore, the transmission did not shift gears and the engine speed remained high while moving forward.

The apparent defense against this attack is straightforward: not to process more than a certain number of requests in a millisecond. Albeit, this solution requires a change in the ECM firmware.

## 3.3 The Connection Exhaustion Attack

The second of the three attacks was coined the name "Connection Exhaustion" as it exhausts the ability of the ECU to establish legitimate connections for multi-packet data transfer. This attack was conducted on each of the testbed setups and had noticeable effect on most of them. We noticed that even though the attack works on the Kenworth T270 research truck, it did not have a physical impact on it.

### 3.3.1 Hypothesis

According to the J1939-21 standards, there can only be one established connection for multi-packet transfer between a source ECU and a destination ECU at a time. It also states that after data has been transmitted a connection can be kept open for a maximum of 1250 milliseconds by not sending the end of message acknowledgment. In addition, a CTS message can be sent to request one or more packets that may have been sent already. Using these three specifications, an attack can be crafted to deny legitimate connection attempts to an ECU by creating multiple spoofed connections and keeping them open by periodically (typically of less than a second) sending a CTS message but not the end of message acknowledgment (Figure 3.7).

### 3.3.2 Testing

#### Testing on the Remote Testbed

The attack was carried out on the remote testbed. The network trace obtained from the CAN bus during the course of the attack is labeled and shown in Figure 3.8. In the beginning, `BB1` establishes two connections by sending request and CTS packets from spoofed source addresses $11_{16}$ and $0B_{16}$. The data is then transferred to `BB1` by the engine controller. After some time, `BB2`, the honest party, attempts to connect to the engine controller but does not receive an RTS. `BB1` maintains its connection at the end of the trace by sending CTSs. As a result, any subsequent connection attempts from `BB2` are discarded, leaving `BB2` in need of the required PGN.

#### Testing on the Local Testbed

The attack was carried also out on the local testbed. The results are shown in Figure 3.9. Valid requests are transmitted periodically for 30 seconds. In response, valid RTSs are received for as long as the malicious connections are not established. For this case, we only show the CTS messages of the malicious connections that are sent periodically to keep connections alive. The attack is only effective on the first three testbed setups; on the last testbed setup valid RTSs are sent in response to the requests.

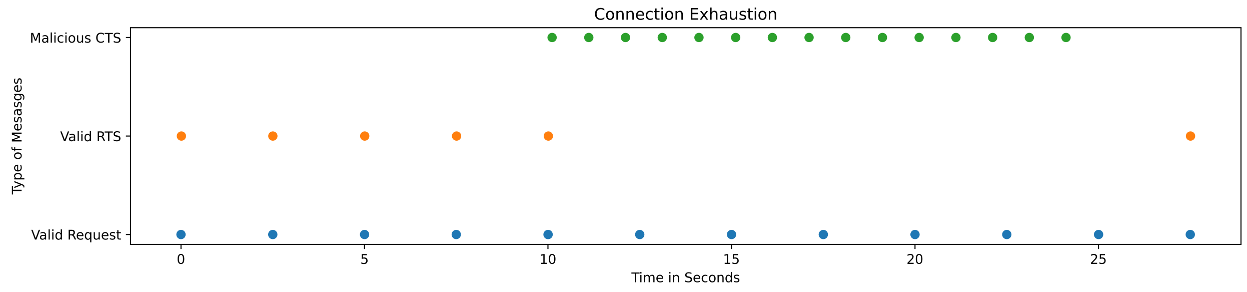
### 3.3.3 Discussion

We noticed that even though the attack works on the Kenworth T270 research truck, it did not have a physical impact on it. Albeit, a quick investigation of the SAE J1939 digital annex [25]

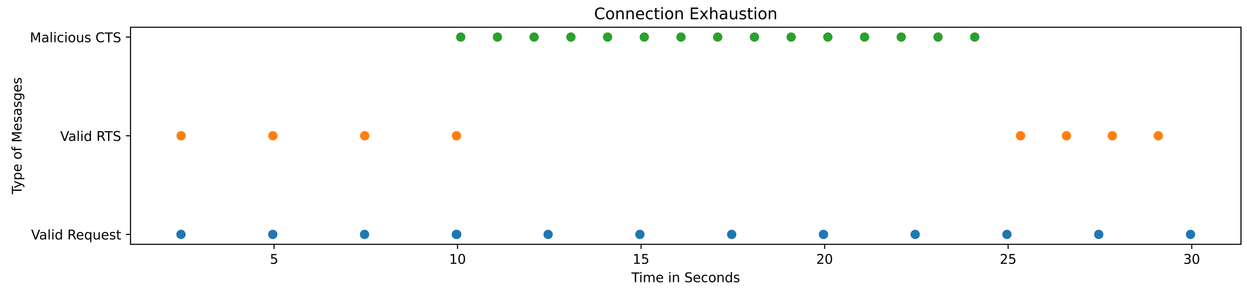
```
BB1->Engine-#1 request          00EA0011   EB FE 00 00 00 00 00 00
Engine-#1->BB1 RTS              18EC1100   10 2C 00 07 FF EB FE 00
BB1->Engine-#1 CTS              00EC0011   11 07 01 FF FF EB FE 00
BB1->Engine-#1 request          00EA000B   EB FE 00 00 00 00 00 00
Engine-#1->BB1 RTS              18EC0B00   10 2C 00 07 FF EB FE 00
BB1->Engine-#1 CTS              00EC000B   11 07 01 FF FF EB FE 00
Engine-#1->BB1 Data Transfer    18EB1100   01 43 4D 4D 4E 53 2A 36
Engine-#1->BB1 Data Transfer    18EB1100   02 43 20 75 30 37 44 30
Engine-#1->BB1 Data Transfer    18EB1100   03 38 33 30 30 30 30 30
Engine-#1->BB1 Data Transfer    18EB1100   04 30 30 2A 30 30 30 30
Engine-#1->BB1 Data Transfer    18EB1100   05 30 30 30 30 2A 78 30
Engine-#1->BB1 Data Transfer    18EB1100   06 36 42 42 42 42 42 42
Engine-#1->BB1 Data Transfer    18EB1100   07 42 2A FF FF FF FF FF
Engine-#1->BB1 Data Transfer    18EB0B00   01 43 4D 4D 4E 53 2A 36
Engine-#1->BB1 Data Transfer    18EB0B00   02 43 20 75 30 37 44 30
Engine-#1->BB1 Data Transfer    18EB0B00   03 38 33 30 30 30 30 30
Engine-#1->BB1 Data Transfer    18EB0B00   04 30 30 2A 30 30 30 30
Engine-#1->BB1 Data Transfer    18EB0B00   05 30 30 30 30 2A 78 30
Engine-#1->BB1 Data Transfer    18EB0B00   06 36 42 42 42 42 42 42
Engine-#1->BB1 Data Transfer    18EB0B00   07 42 2A FF FF FF FF FF
BB2->Engine-#1 request          00EA0011   EC FE 00 00 00 00 00 00
BB2->Engine-#1 request          00EA000B   EC FE 00 00 00 00 00 00
BB2->Engine-#1 request          00EA0011   EC FE 00 00 00 00 00 00
BB2->Engine-#1 request          00EA000B   EC FE 00 00 00 00 00 00
BB2->Engine-#1 request          00EA0011   EC FE 00 00 00 00 00 00
BB2->Engine-#1 request          00EA000B   EC FE 00 00 00 00 00 00
BB2->Engine-#1 request          00EA0011   EC FE 00 00 00 00 00 00
BB2->Engine-#1 request          00EA000B   EC FE 00 00 00 00 00 00
BB1->Engine-#1 CTS              00EC0011   11 07 01 FF FF EB FE 00
BB1->Engine-#1 CTS              00EC000B   11 07 01 FF FF EB FE 00
```

**Figure 3.8:** Connection Exhaustion Network Trace

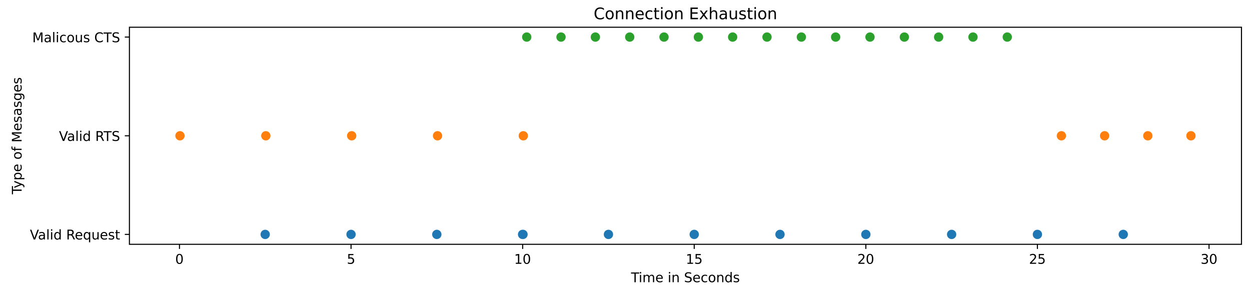
reveals that transport sessions are critical for diagnostic and proprietary communication over SAE J1939. This attack can significantly hamper those operations. An example of this is shown in Figure 3.10 where a Cummins proprietary diagnostic tool fails to connect to the ECM even though messages are transmitted at regular intervals from it (as seen in the “message delta" dialog for the “ECM1"). This can also be detrimental to the vehicle if data obtained from the session is used for control purposes. For example, PGN 65251 carries engine configuration information that may be required by more than one legitimate ECU. If this information is not received, those ECUs may malfunction. We categorize this attack in the low-volume DoS category (c.f. section 2.3).



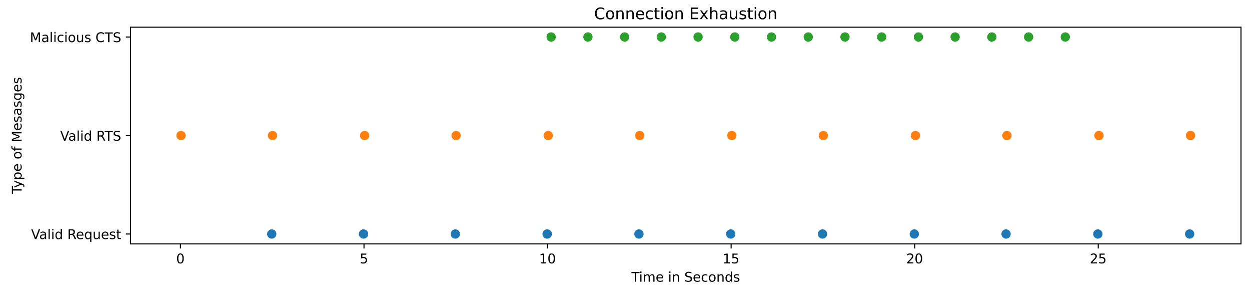
**(a)** Connection Exhaustion Experiment Results on Local Testbed 1



**(b)** Connection Exhaustion Experiment Results on Local Testbed 2

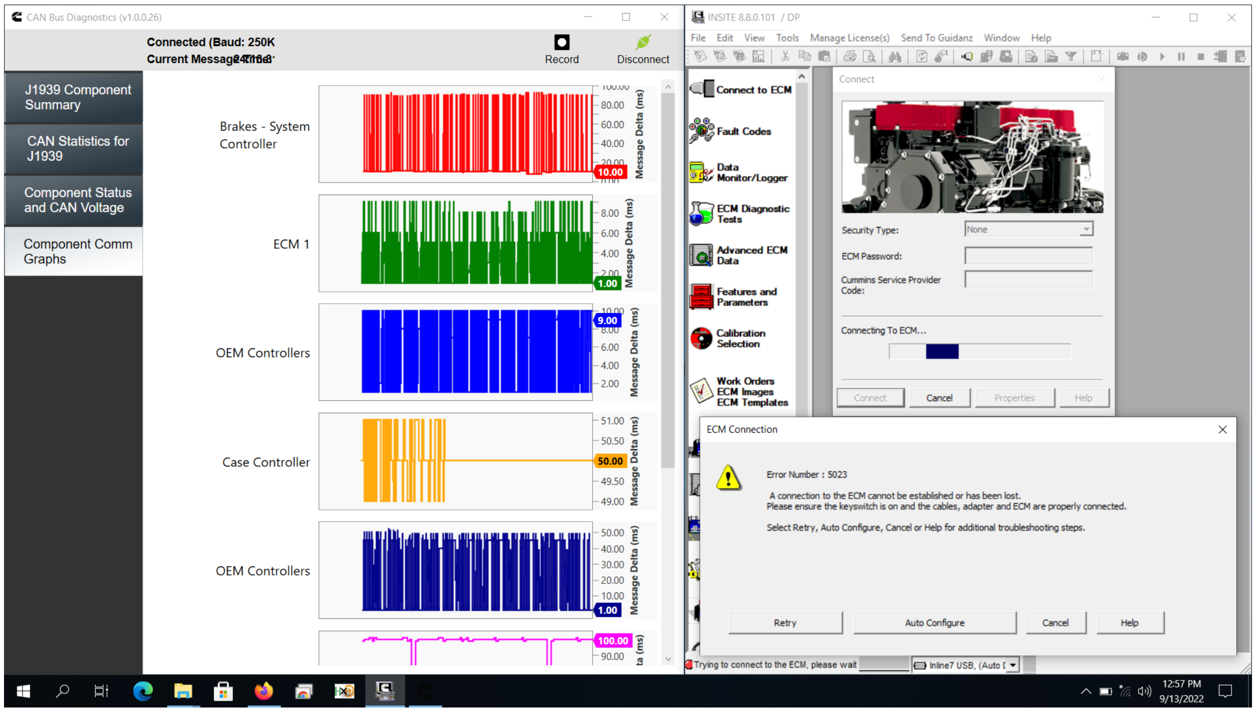


**(c)** Connection Exhaustion Experiment Results on Local Testbed 3



**(d)** Connection Exhaustion Experiment Results on Local Testbed 4

**Figure 3.9:** Connection Exhaustion Experiment Results on Different Configuration of the Local Testbed



Error Number : 5023

A connection to the ECM cannot be established or has been lost.
Please ensure the keyswitch is on and the cables, adapter and ECM are properly connected.

Select Retry, Auto Configure, Cancel or Help for additional troubleshooting steps.

Trying to connect to the ECM, please wait

**Figure 3.10:** Effect Of Connection Exhaustion on a Cummins Proprietary Diagnostic Tool

The apparent mitigation against this attack can be to not respond to more than a certain number of CTS retransmit requests. Albeit, this solution requires a change in the ECM firmware.

## 3.4 The False Request To Send (RTS) Attack

The third of the three attacks was coined the name "False RTS" as it involves the attacker falsifying an RTS message (first described in section 2) with an aim to crash the ECU firmware when it receives data packets in a multi-packet transfer session. This attack could not be conducted on any of the testbed setups since none of the ECUs accepted data packets without authentication for which we did not have access to the proper credentials. Understandably, the same situation was encountered on the Kenworth T270 research truck and, as such, we could not test the attack on it. For proof-of-concept, we experimented on a simulation ECU firmware that implemented the vulnerability.

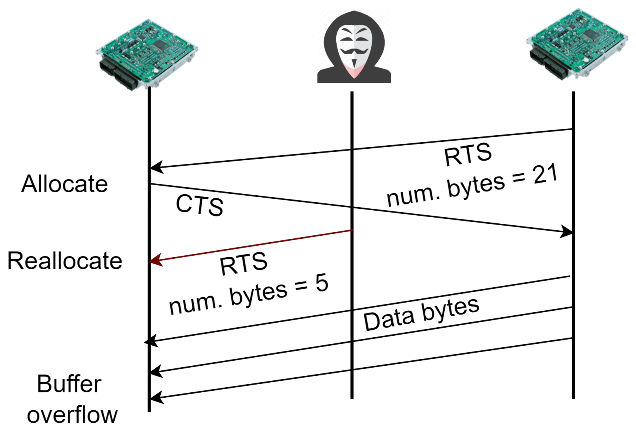


**Figure 3.11:** False RTS

### 3.4.1 Hypothesis

According to the SAE J1939/21 specifications, if multiple RTS messages are received from the same source address, the most recently received shall be considered without notifying the sender of the first RTS. Consider an ongoing connection in which a requester receives an RTS and allocates a buffer to store multipacket data of the same size as that received in RTS data bytes 2 and 3. After that, the requester sends a CTS. An attacker can spoof the original sender's source address and send a second RTS with a smaller data size to the requester. If the spoofed RTS's receiver reallocates the buffer and continues to receive data packets from the original sender, the allocated buffer may overflow, causing the ECU firmware to crash.

### 3.4.2 Testing

For this attack to work on the ECUs in our testbed, they were required to accept data packets. However, we noticed that upon receiving an RTS neither the ECM nor the EBC responded back with a CTS. We suspected that this may be because of the requirement to be authenticated before being able to write to ECU memory. At that time, we did not have the necessary tools and/or secrets to authenticate with the ECUs. As such, given the existence of the specification flaw and the severity of the eventual impact, we developed proof-of-concept software to demonstrate the effect of the attack. The software binary is made available at https://github.com/j4l-Shvn/J1939_21_Transport_Vuln_POC. On `BB1`, the faulty software was run by the heap profiler Valgrind[3] with the option `--leak-check` set to `yes`. A summarized workflow of the program is shown below

```
Wait for RTS;
On receiving RTS allocate/reallocate buffer space
   with size as obtained from bytes 2-3 of the RTS's data;
Record the number of packets to be received;
Send CTS;
Store data until all packets have been received;
```

Memory allocation and reallocation were performed on the heap. On `BB2` we ran the attack script as shown below

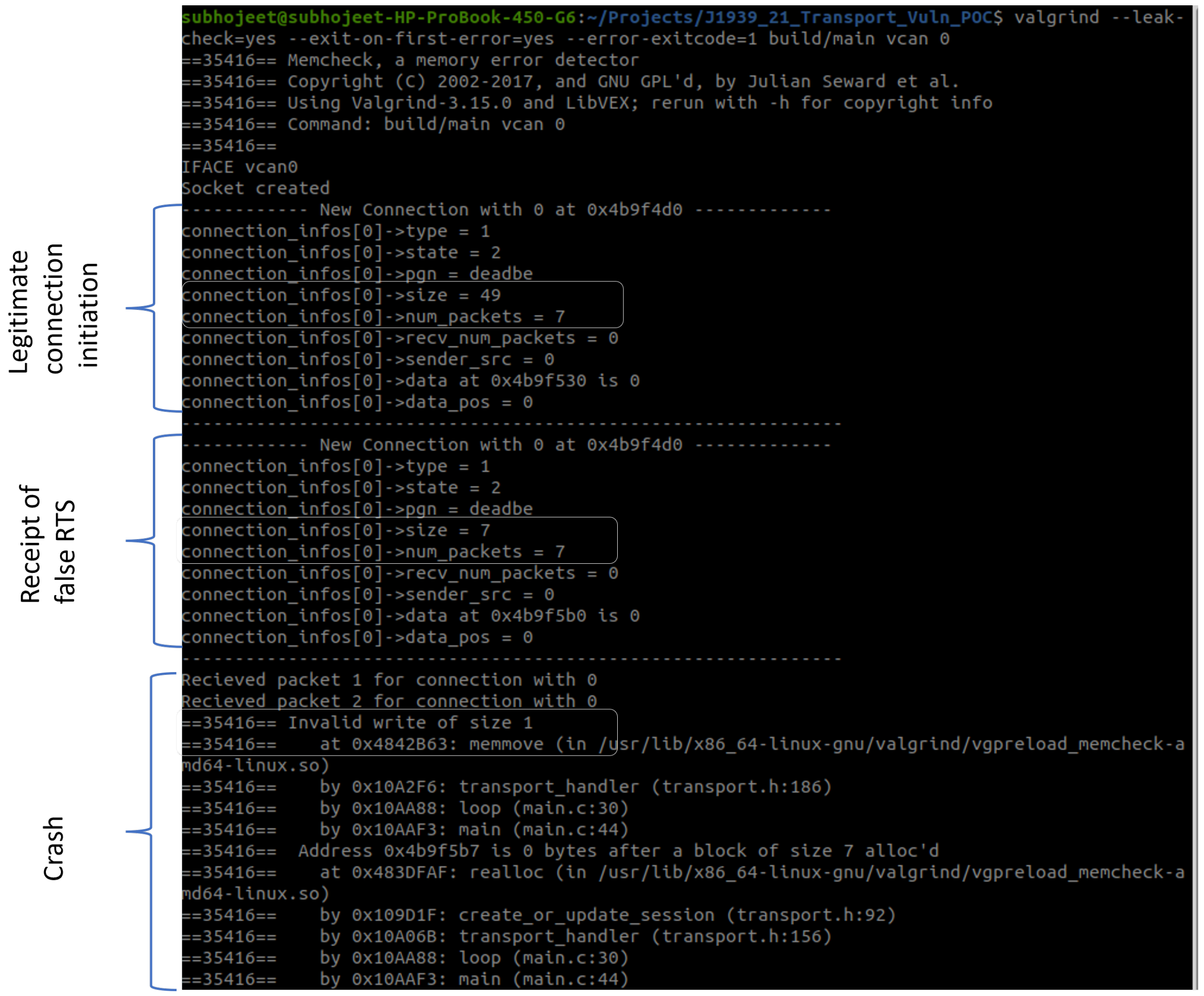
```
Sniff bus for CTS from attack target;
Send crafted RTS with lesser data size;
```

To simulate the attack an RTS was sent from a separate terminal on `BB1` when both programs were running. After a delay of 10 milliseconds, data packets were sent to overflow the allocated buffer on the vulnerable recipient. It was observed that Valgrind reports a heap overflow while data packets are received. This is shown in Figure 3.12.

### 3.4.3 Discussion

We categorize this attack in the low-volume DoS category (c.f. section 2.3). In theory, the entire attack can be carried out without the need to wait for the first RTS. For this, the attacker needs to send an RTS, and if the ECU responds with a CTS, send the malicious RTS, followed by the data packets.

[3]https://valgrind.org/



```
subhojeet@subhojeet-HP-ProBook-450-G6:~/Projects/J1939_21_Transport_Vuln_POC$ valgrind --leak-
check=yes --exit-on-first-error=yes --error-exitcode=1 build/main vcan 0
==35416== Memcheck, a memory error detector
==35416== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==35416== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==35416== Command: build/main vcan 0
==35416==
IFACE vcan0
Socket created
------------ New Connection with 0 at 0x4b9f4d0 -------------
connection_infos[0]->type = 1
connection_infos[0]->state = 2
connection_infos[0]->pgn = deadbe
connection_infos[0]->size = 49
connection_infos[0]->num_packets = 7
connection_infos[0]->recv_num_packets = 0
connection_infos[0]->sender_src = 0
connection_infos[0]->data at 0x4b9f530 is 0
connection_infos[0]->data_pos = 0
-------------------------------------------------------------
------------ New Connection with 0 at 0x4b9f4d0 -------------
connection_infos[0]->type = 1
connection_infos[0]->state = 2
connection_infos[0]->pgn = deadbe
connection_infos[0]->size = 7
connection_infos[0]->num_packets = 7
connection_infos[0]->recv_num_packets = 0
connection_infos[0]->sender_src = 0
connection_infos[0]->data at 0x4b9f5b0 is 0
connection_infos[0]->data_pos = 0
-------------------------------------------------------------
Recieved packet 1 for connection with 0
Recieved packet 2 for connection with 0
==35416== Invalid write of size 1
==35416==    at 0x4842B63: memmove (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-a
md64-linux.so)
==35416==    by 0x10A2F6: transport_handler (transport.h:186)
==35416==    by 0x10AA88: loop (main.c:30)
==35416==    by 0x10AAF3: main (main.c:44)
==35416==  Address 0x4b9f5b7 is 0 bytes after a block of size 7 alloc'd
==35416==    at 0x483DFAF: realloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-a
md64-linux.so)
==35416==    by 0x109D1F: create_or_update_session (transport.h:92)
==35416==    by 0x10A06B: transport_handler (transport.h:156)
==35416==    by 0x10AA88: loop (main.c:30)
==35416==    by 0x10AAF3: main (main.c:44)
```

**Figure 3.12:** False RTS Results on Vulnerable Firmware Simulation

Because we did not find any evidence of this attack working on the real-world ECUs, we project this attack as a warning. We tried executing the attack on the Kenworth T270 research truck but faced the same authentication roadblock as on the testbeds.

To defend against the attack, we emphasize that memory safety must be ensured by checking for mismatching entries in the fields specifying the number of bytes and packets in the RTS message. Additionally, the second (falsifiable) RTS can also be ignored.

## 3.5 Summary

In this chapter we answer the research question *can weaknesses in the data-link layer specifications of SAE J1939 be exploited to attack in-MHD vehicle ECUs?*. To that end, we demonstrate three denial-of-service attacks that utilize protocol specifications made in the SAE J1939 standards. For each of these attacks, we observed noticeable impacts on network communication from the target ECUs. We also, observed noticeable impact on a research truck upon execution of the first two of these attacks: in the first case transmission shifts were impeded, and in the second case a diagnostic tool malfunctioned.

# 4 Behavioral Anomaly-based Detection

Research on behavioral intrusion detection for in-vehicle networks has so far focussed on CAN networks in passenger vehicles and has largely been based on machine learning of network logs collected offline. As Stachowski et al. [4] have pointed out, such techniques are not fit to be operated on networks within medium and heavy-duty (MHD) vehicles that exhibit dynamically changing behavior. As such, in this section, we describe our research towards developing an online anomaly detection system that models network behavior using SAE J1939 concepts and flags abnormal deviations from normal behavior as security infringements. We begin by describing two features that capture the network's behavior. Next, we use some observations from a case study on manually crafted attack traffic to describe how these feature values can be used to model the behavior of the network. From these observations, we lay down our attack detection hypothesis. We then present an algorithm that implements our hypothesis and tries to detect attacks as messages are received on the network. Following this, we analyze the performance of our algorithm on real-world attack data collected from the Kenworth T270 research truck and present a summary of our accomplishments.

## 4.1 Behavioral Feature Engineering

We model network behavior through a *Report Precedence Graph* (*RPG*). An RPG captures the temporal relationships between the transmission of vehicular parameters on the CAN bus. Each node of an RPG is a *report*.

**Definition 1 (Report)** *Let $m_t$ be a message on the bus at time $t$ and let $pgn_{m_t}$ be its PGN and $sa_{m_t}$ be its sending source address. A report $r$ is a collection of discretized parameter values created from message $m_t$, such that $r = \{(spn, d(v_{spn}), pgn_{m_t}, sa_{m_t})\}$, every $spn$ is assigned to*

*$pgn_{m_t}$ in the SAE J1939 digital annex and $d(v_{spn})$ represents the discretization of value $v_{spn}$ of the parameter identified by $spn$ transported in message $m_t$.*

Discretization is required to control the number the reports generated from continuous-valued parameters like engine speed, vehicle speed, etc. For discrete-valued parameters (e.g. state of a door lock) that can only assume a limited set of values, we set $d(v_{spn}) = v_{spn}$. For continuous-valued parameters, discretization can be achieved through various means as described in [58] but in this dissertation we set

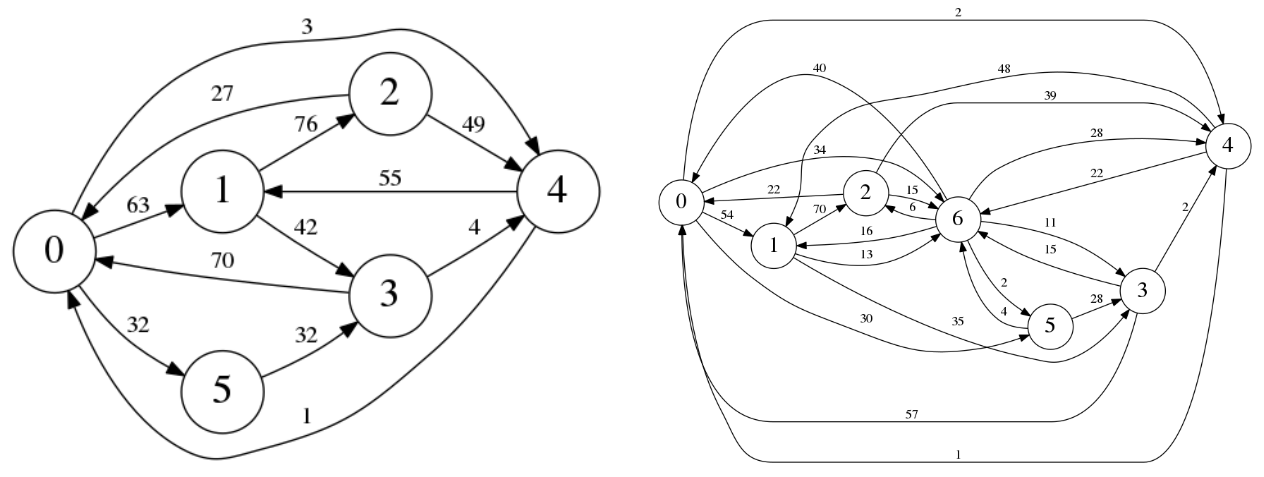
$$d(v_{spn}) = \lfloor length_{spn} * \frac{(v_{spn} - offset_{spn})}{(resolution_{spn} * (2^{length_{spn}} - 1))} \rceil$$

Here $\lfloor \rceil$ denotes the round operation and $length_{spn}$, $offset_{spn}$ and $resolution_{spn}$ are the length, offset and resolution of the parameter identified by Suspect Parameter Number (SPN) $spn$. This sets the domain of $d(v_{spn})$ to ${}_{\geq 0}\mathbb{Z}_{\leq length_{spn}}$ for continuous-valued parameters. As an example, wheel speed is a continuous-valued parameter that is assigned a length of 16, offset of 0 and resolution of 1/256. After discretization, it assumes values integral between 0 and 16, inclusive of both. Speeds 95 and 100 km/h are both discretized to 6 while speeds 105 km/h and 110 km/h are discretized to 7.

An example of a report is $\{(597,00_2,65265,00_2),(598,01_2,65265,00_2)\}$. This report is generated from a message associated with the PGN 65265 and transmitted by source $00_{16}$ i.e. the engine control unit. It captures the state information `(Brake pedal released, Clutch pedal depressed)`. Using reports, we generate an RPG.

**Definition 2 (Report Precedence Graph)** *The Report Precedence Graph (RPG) is a labeled directed graph $G = (R, T, L)$ where each node $r \in R$ is a report and each edge $\langle r_i, r_j \rangle \in T$ denotes report $r_j$ is transmitted after report $r_i$ ($r_i \prec r_j$ with no other $r_k$ such that $r_i \prec r_k$ and $r_k \prec r_j$), a total of $l_{i,j} \in L$ number of times.*

Figures 4.1a and 4.1b show simple RPGs built from a 15-second snapshot taken from the log collected on the 2015 Kenworth T660 research truck (refer back to section 2.2 for a description



**(a)** 15-second RPG Constructed from Normal Driving Data

**(b)** Same 15-second RPG with Manually Injected messages

**Figure 4.1:** Example Report Precedence Graphs (RPG)

| Report | Interpretation | SA | PGN |
|---|---|---|---|
| 0 | Accelerator pedal 1 in low idle condition | $00_{16}$ | 61443 |
| 1 | Brake pedal released | $00_{16}$ | 65265 |
| 2 | Proprietary retarder control mode | $0F_{16}$ | 61440 |
| 3 | Low idle governor retarder control mode | $0F_{16}$ | 61440 |
| 4 | Accelerator pedal 1 not in low idle condition | $00_{16}$ | 61443 |
| 5 | Brake pedal depressed | $00_{16}$ | 65265 |
| 6 | Request for high engine speed | $0B_{16}$ | 0 |

**Table 4.1:** Report Interpretations from RPG in Figure 4.1a and Figure 4.1b; SA = Source Address, PGN = Parameter Group Number

of this log). Except, in Figure 4.1b we manually insert messages into the recorded traffic at a 50% probability for a period of 15 seconds. There are 4 distinct PGNs in the RPGs. These are transmitted by the engine with source address (SA) = $00_{16}$, the retarder with SA = $0F_{16}$, and the brake controller with SA = $0B_{16}$. Table 4.1 shows the interpretation of the reports (numbered 1 through 6) in the RPGs. The edge labels in the graph denote the number of times $report_i$ precedes $report_j$. We now describe two features to evaluate the shape of the RPG.

### Normalized Graph Flux Capacity

Flux capacity was introduced by Martinez et al. [59] to quantify the amount of information flow that passes through gene regulators. It is defined as the product of the in-degree and out-degree of a node. In this work, we use it to denote the number of two-edge paths passing through a report $r$ in an RPG.

$$
\begin{aligned}
fc_{r_i} &= |(r_j, r_i)| * |(r_i, r_k)| \; \forall (r_j, r_i), (r_i, r_k) \in T \\
&= d^{in}(r_i) * d^{out}(r_i)
\end{aligned}
$$

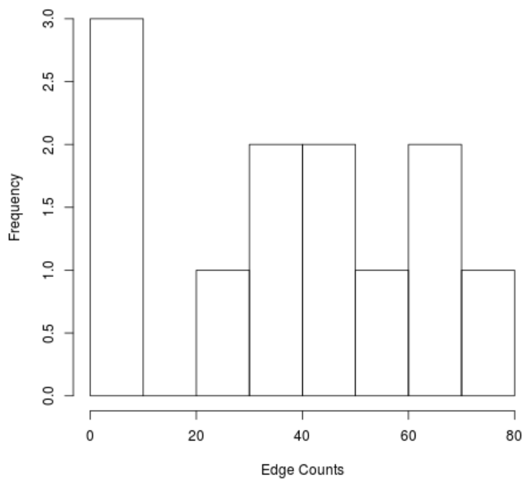
For example, the flux capacity of report 1 in Figure 4.1a is 2*2 = 4 and that of report 6 in Figure 4.1b is 6*6 = 36. This is essentially the number of two edge paths passing through these reports. Using the concept of flux capacity we now define the concept of *Normalized Graph Flux Capacity* (NGFC) as the average flux capacity of an RPG and express it as

$$NGFC = \frac{1}{|R|^3} \sum_{i=1}^{|R|} fc_{r_i}$$

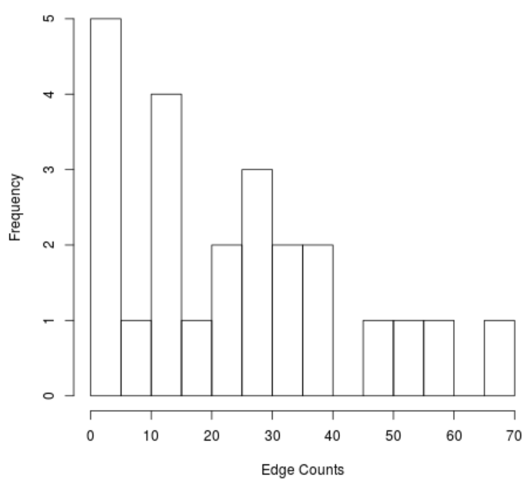
Essentially, this is the cumulative flux capacity of all reports in the RPG, normalized to the cubic power of the number of reports $R$. The choice of the cubic power in the denominator is motivated by the extreme case, where, if an RPG is a complete graph having n nodes, NGFC will be of the order of $n^3$ ($\sum_{i=1}^{n}(n-1)*(n-1)$). Thus, as chaotic precedences begin to be seen in the RPG and every report becomes equally likely to be preceded by every other report, the NGFC value starts approaching 1 ($\frac{n^3}{n^3}$). This phenomenon can be observed in Figure 4.1b where report 6 preceded every other report and the NGFC is calculated to be 0.27, almost double of what is calculated from Figure 4.1a i.e 0.12. For an RPG with no edges, we set its NGFC to 1, assuming all self-loops.

**Edge Weight Skewness (EWS)**

Another important feature to observe from the RPG in Figure 4.1a and Figure 4.1b is the skewness of the distribution of the edge weights (*EWS*). As observed from Figure 4.2, the distribution of the edge weights under the attack becomes increasingly more right-skewed. In other words, the level of positive skewness increases in Figure 4.2b as compared to Figure 4.2a. For this work, we use Python Scipy's skewness measure [60] which is calculated as $\frac{m_3}{m_2^{3/2}}$, where $m_3$ and $m_2$ are the



**(a)** EWS for RPG Figure 4.1a



**(b)** EWS for RPG Figure 4.1b

**Figure 4.2:** RPG Edge Weight Distributions (EWS)

third and second moments about the mean of the distribution. Using this formula, the skewness of the distribution in Figure 4.2a is obtained as -0.113, and that for the distribution in Figure 4.2b is obtained as 0.745. Due to the chaotic precedences observed after the introduction of report 6 in Figure 4.1b, the injected report interleaves with normal reports, and the original edge weights in Figure 4.1a reduce, making room for new, less-weighted edges. For example, the weight for edge $\langle r_0, r_1 \rangle$ reduces from 63 to 54, and newer lower weight edges such as $\langle r_6, r_5 \rangle$ and $\langle r_3, r_6 \rangle$ are introduced. This causes an increase in EWS. For an RPG with no edges, we set its EWS to 20 i.e. a very high right skew.

## 4.2 Network Behavior Modelling Through Feature Value Time Series

So far we have established that RPGs are useful in observing the temporal relationships between the transmission of vehicular parameters and the shape of an RPG can be evaluated through the features NGFC and EWS. We now use a case study to describe how the values of these two features can characterize in-vehicle network behavior. From this description, we lay out our hypothesis for attack detection.

**Sampling Window** NGFC and EWS are calculated on pre-built RPGs. In order to build an RPG one needs to observe a set of SAE J1939 messages on the CAN bus for some period of time. We refer to this period as *sampling period* after which NGFC and EWS values are sampled. Calculating graph-based and statistical features can be time-consuming. If deployed in a resource-constrained in-vehicle environment, processing large graphs in real time can lead to undesired performance bottlenecks. It is therefore desirable to calculate NGFC and EWS in a small sampling period (in the range of seconds), possibly while a new RPG is being built in parallel. Then again, the sampling period should not be too small, as in that case, the RPG may not capture all the parameter groups being transmitted on the network. Keeping these factors in mind, we decided to choose a sampling period of 1 sec. As such, most SAE J1939 parameter groups are transmitted at rates of 1 sec or lower. A sampling period of 1 sec would thus allow most capture of most parameters on the bus in a single RPG, at least once.

NGFC and EWS values measured after the sampling period of 1 second are shown in Figure 4.3 as a pair of time series. The RPGs for this case were generated from the log collected on the 2015 Kenworth T660 research truck (refer back to section 2.2 for a description of this log). Attack traffic was simulated by injecting messages for 7 seconds with a probability of 0.5 from the 197-second mark. This portion is marked with a red box. While building the RPG from which these NGFC and EWS values were calculated we considered every parameter that denotes the state of a component of the vehicle. For example, the parameter "brake depressed" can assume the values 0 and 1 to denote the state of the brakes. With these time series, we now describe our observations on the behavior of the network during normal driving, as well as at the time of malicious message injection.

- **Under normal driving** During the periods of normal driving both time series appear to be stationary i.e. they fluctuate around some base value. In some cases, they exhibit occasional

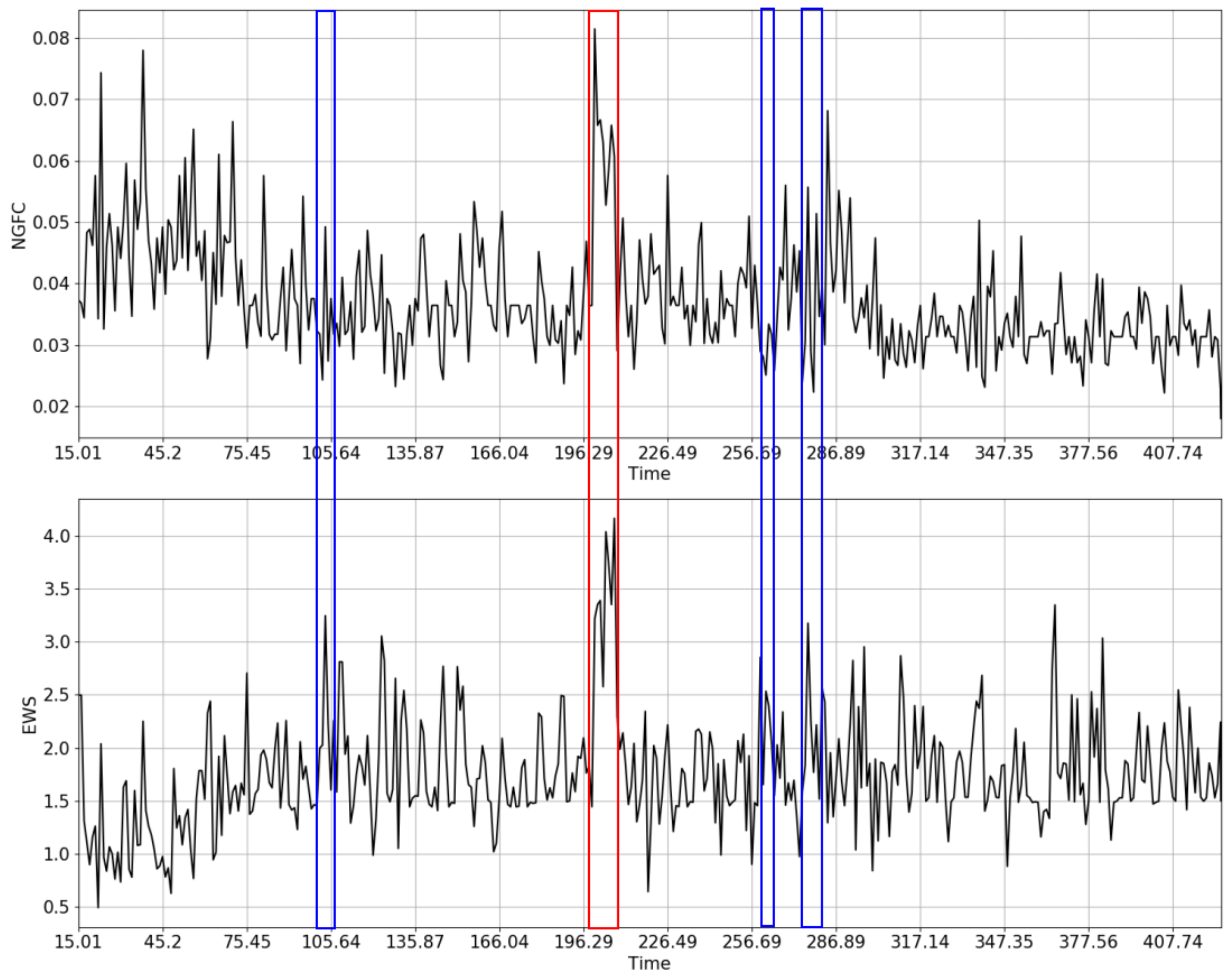


**Figure 4.3:** Sampled NGFC and EWS Time Series

short trends (rising and falling), especially towards the beginning. The possible reason for this could be the advent of new reports and low-weighted edges in the generated RPG. The new additions begin to dissipate soon leading to an almost stationary time series. There is also no evidence of seasonal patterns. The capture from which these time series are generated includes legitimate command messages transmitted at the time of hard-braking events. These events happened at around the 100-second and 300-second marks. Corresponding timelines are labeled with blue boxes in Figure 4.3. No significant deviations are spotted in the time series within the blue boxes.

- **Under attack** The portion of the NGFC and EWS time series that show simultaneous significant abrupt peaks denote some form of unnatural behavior. This section is marked by a red box. This is the portion where malicious messages were injected into the recorded bus traffic.

Given these observations, we hypothesize that *under normal driving conditions NGFC and EWS time-series are usually stationary with the possibility of short trends but upon malicious message injections, they exhibit significant abrupt changes that can be detected.* Although the aforementioned example shows abrupt increments only, we speculate that there may be abrupt decrements as well.

## 4.3 Detecting Anomalous Behavior at Runtime

Algorithm 1 shows the methodology to detect abrupt changes in the NGFC and EWS time series at runtime. Every time a message is received, an RPG is updated with reports generated from it. Lines 5-11 show this process. In the *getReport* method the PGN of the message is obtained and used to extract the parameter values following SAE J1939 specifications described in section 2. In case a PGN is not found in the SAE J1939 digital annex or if any of the position, length, offset, and resolution of a parameter is not defined, a default position of 0, length of 64, offset of 0, and resolution of 1 are assumed. Extracted parameter values are grouped to form a report. In line 6, the newly formed report is added to the RPG, and a new precedence edge $\langle r_i, r_j \rangle$ is added if it is not a self-loop. $l_{i,j}$ is also updated to maintain an up-to-date list of edge weights.

Every time the *samplingWindow* of 1 second expires, i.e. after receiving messages for a second, the algorithm calculates NGFC and EWS values from the current RPG (line 13). Calculated values are pushed into two circular queues *NGQ* and *EWQ* of size *trainingSetSize* (in lines 21-23) only if no attack is detected. The RPG is reinitialized after that in lines 25-26.

Attack detection is performed as long as the queues are full, i.e. after the first *trainingSetSize* number of windows have expired and every time after that. Following the hypothesis established in the previous section, the algorithm aims to detect an attack by checking if both feature values

**Algorithm 1:** Runtime Behavioral Anomaly Detection

```
   /* Global variables                                              */
1  r$_{current}$ $\leftarrow$ null
2  startTime $\leftarrow$ 0, samplingWindow $\leftarrow$ 1
3  NGQ, EWQ= $\emptyset$                         $\triangleright$ Circular queues
4  RPG = (R,T,L), R,T,L $\leftarrow$ $\emptyset$
   /* Beginning of algorithm                                        */
   Input: SAE J1939 Message, trainingSetSize, ADFConfidence, HoltsConfidence
5  r$_i$ $\leftarrow$ getReport(J1939 Message)
6  R $\leftarrow$ R $\cup$ r$_i$
7  if $r_i \neq r_{current} \wedge r_{current} \neq null$ then
8      T $\leftarrow$ T $\cup$ $\langle$ r$_{current}$, r$_i$ $\rangle$
9      Update L, set l$_{i,j}$ $\leftarrow$ l$_{i,j}$ + 1
10 end
11 r$_{current}$ $\leftarrow$ r$_i$
12 if time $\geq$ startTime + samplingWindow then
13     NGFC $\leftarrow$ NFGC(R,T), EWS $\leftarrow$ EWS($L$)
14     is_attack $\leftarrow$ False
15     if number of items queued = trainingSetSize then
16         is_attack $\leftarrow$ sig-inc(NGQ, NGFC) $\wedge$ sig-inc(EWQ, EWS)
17         if is_attack then
18             RAISE ALARM
19         end
20     end
21     if is_attack = False then
22         NGQ.push(NFGC), EWQ.push(EWS)
23     end
24     startTime $\leftarrow$ time
25     R,T,L $\leftarrow$ $\emptyset$
26     r$_{current}$ $\leftarrow$ null
27 end
1  Procedure sig-inc(buf, current)
2      has_trend $\leftarrow$ Augmented_Dickey–Fuller_test(buf).p-value $\leq$ (100 -
         ADFConfidence)/100
3      forecast_interval $\leftarrow$ HoltWinters(buf,HoltsConfidence,has_trend)
4      return value $\notin$ forecast_interval
```

are significantly deviant from their most recent observations (lines 16-19). Brutlag et al. [61] demonstrate a method that can be used to achieve this for a single feature. In their method, they use a window of the most recent (at time $t$) feature values in a time series $(v_i)_{t-window-1 \leq i \leq t-1}$ to predict an interval $i_t$ from it. They then detect if the newly calculated value $v_t$ falls within the boundaries of $i_t$. If not, they label it as significantly deviant.

Brutlag et al. [61] esimate the interval using the HoltWinters method. An understanding of the HoltWinters method is not core to understanding the contributions made in this work. As such, we do not describe it in detail but present a brief overview of the specific implementation provided by the statistical platform R [62, 63] that we make use of. R's implementation of HoltWinters method predicts an interval $i_t$ using four components namely, level ($l_t$), trend ($t_t$), seasonality ($s_t$), and confidence ($cf$). If the time series is graphed in an x-y plane and a line is passed through its points, the level can be thought of as the intercept on the y-axis, while the trend is the slope and the seasonality is any repeating pattern observed in the layout of the points. The size of the interval is directly proportional to $cf$, i.e. $(i_t[1] - i_t[0]) \propto cf$. In Algorithm 1, we refer to $cf$ as *HoltsConfidence*. R's implementation allows us to exclude the trend and seasonality components from the estimation if there are no distinctive signs of such activity in the input time series. As observed from the previous section, our feature value time series do not observe seasonality but can observe short trends. Therefore, we estimate the interval based on level and trend, if the time series shows a trend. Otherwise, we use only the level. To determine if the time series exhibits a trend we use the Augmented Dickey-Fuller (ADF) unit-root test [64]. The unit-root test returns a *p-value* statistic that, if greater than (100 - *ADFConfidence*)/100, indicates that there is no trend. This process is shown in procedure *sig-inc* in Algorithm 1 which is called twice in line 17, once for each feature. If it returns true for both calls, an alarm is raised.

## 4.4 Performance Analysis

In this section, we analyze our solution with respect to real-world attack data collected from a real vehicle under different circumstances. We begin by presenting a description of our experiment methodology. We then present our performance evaluations in two cases: under normal driving and when attacks are being conducted.

### 4.4.1 Experiment methods

The experiments we ran for our analysis were designed with two objectives namely, to demonstrate the efficacy of our solution and to find the optimum operating characteristics of the three configuration factors of Algorithm 1: *trainingSetSize*, *ADFConfidence* and *HoltsConfidence*. To that end, we configured a Python implementation of Algorithm 1 with different values of the three factors and executed it on 30 seconds of network logs captured in different circumstances encompassing both normal driving and attack scenarios. Only 30 seconds of logs were captured to ensure the safety of the driver in attack scenarios. Five out of the six attacks were conducted by periodically injecting messages for 7-10 seconds in accordance with the observations made in [12], while one, the address claim attack, was conducted using a single message. All attacks were started at around 15 seconds to allow enough time for filling of the feature values queues from Algorithm 1. To ensure consistency of the results, Algorithm 1 was executed on two captures per circumstance.

Because the attacks were started at around 15 seconds, two values for *trainingSetSize* were chosen: 5 and 10. *ADFConfidence* and *HoltsConfidence* were varied in intervals of 20, starting from 10, i.e. 10%, 30%, 50%, 70%, and 90%. This was done to evaluate the performance of Algorithm 1 across varied values of these parameters.

### 4.4.2 Evaluation Under Normal Driving Scenarios

The goal of this evaluation is to estimate the false-positive rate of our solution, i.e. the percentage of windows flagged as malicious in normal driving circumstances.

#### Data Collection

We captured normal driving logs from the 2014 Kenworth T270 research truck in two different circumstances: first, while driving on an airstrip, and second, during a cross-country trip.

**Evaluation Metrics**

Our metric for evaluation of performance on the normal driving logs was simple. We used the metric (*100*Number of flagged windows*)/*Total number of windows* to determine the false-positive rate.

**Observations**

The results are shown in Figure 4.4. Both time series do not demonstrate significant abrupt changes. This supports our initial hypothesis. False positive rates are minimum for both logs when the *trainingSetSize* is 10 and *HoltsConfidence* is 90%. This shows that higher *trainingSetSize* and *HoltsConfidence* leads to reduced false positives. The actual minimum false positive rate obtained on the log collected at the airstrip is 3.4% and that collected during the cross-country trip is 0%.

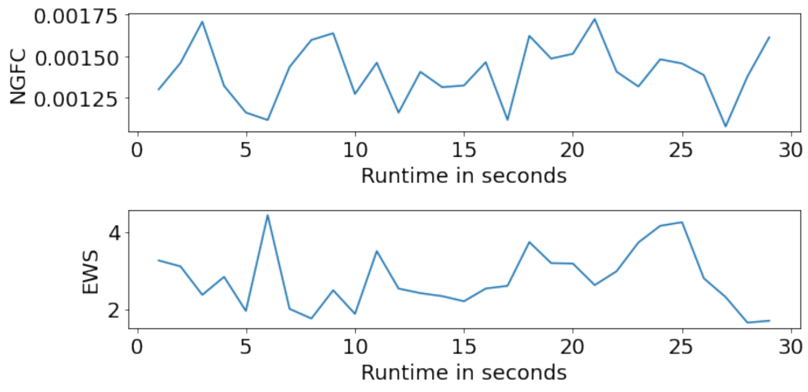
### 4.4.3 Evaluation Under Attacks

The goal of this evaluation is to estimate the efficacy of our solution in detecting cyber attacks and discerning them from false alarms.

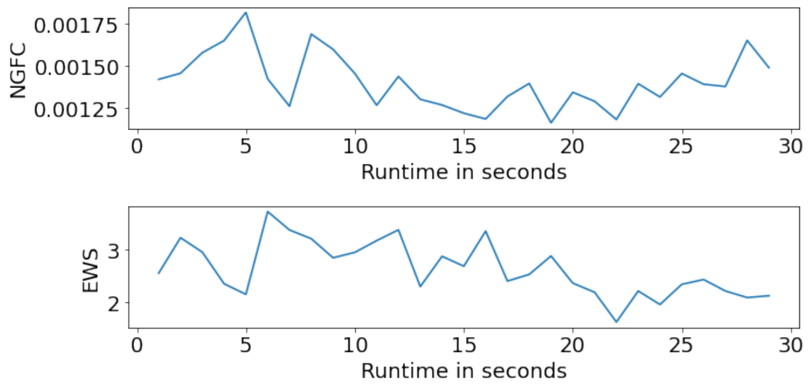
**Data Collection**

We captured attack logs by driving the Kenworth T270 Research truck on an airstrip and conducting two instances of six different attacks on it. We chose these six attacks as they demonstrated some effects on the truck. The six attacks and the means to conduct them are described in the following bullet points. Background knowledge about these attacks were already provided in section 2.3.

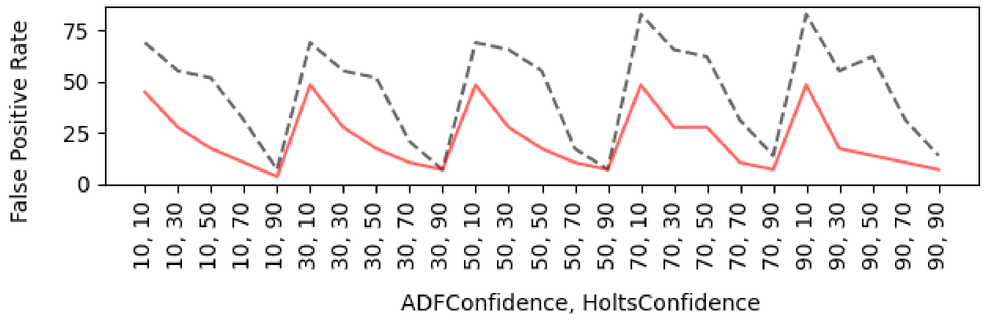
- We conducted an address claim attack by sending a message with PGN $0EA00_{16}$ and claiming the address of the engine controller.
- We conducted a network overload attack by sending messages with CAN ID 0 at intervals of 0.3 milliseconds. This flooded the bus and all normal traffic was removed for the duration of the attack.



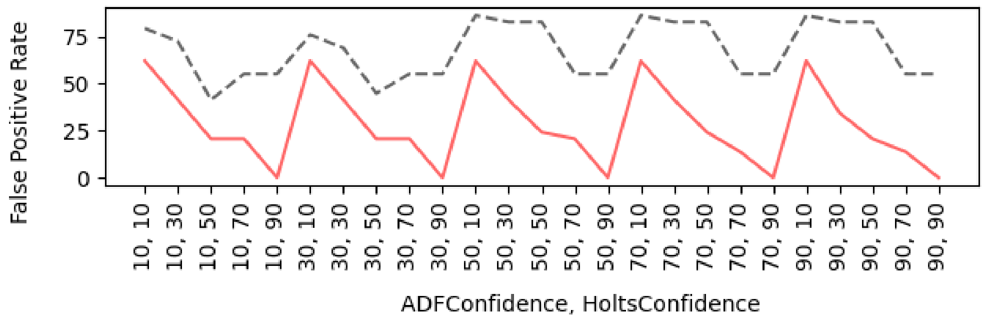
**(a)** Feature Value Time Series For Normal Driving On an Airstrip



**(b)** Feature Value Time Series For Normal Driving On a Cross Country Trip



**(c)** False Positive Rates For Normal Driving On an Airstrip



**(d)** False Positive Rates For Normal Driving On a Cross Country Trip

**Figure 4.4:** Behavioral Intrusion Detection Results For Normal Driving

- We conducted two variations of the fuzzing attack by sending messages with random CAN ID. In the first variation, the data bytes were kept fixed and in the second variation, they were varied randomly at every injection. In both variations, an inter-message interval of 0.5 milliseconds was chosen in accordance with prior literature [65].

- We conducted a request overload attack by sending requests for the engine controller's component ID at 0.5 milliseconds.

- We conducted an engine control attack by sending the SAE J1939 specified torque and speed control message with a fixed request engine speed at intervals of 10 milliseconds.

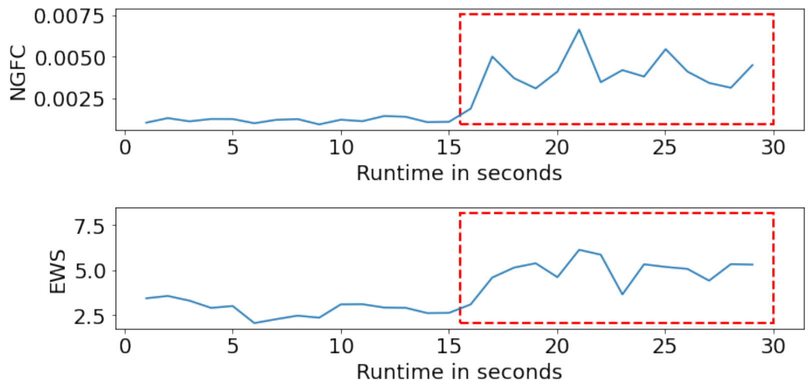
Videos of the attacks and links to the corresponding log files are provided at https://projects-web.engr.colostate.edu/cybersystems/j1939-attacks/.

**Evaluation Metrics**

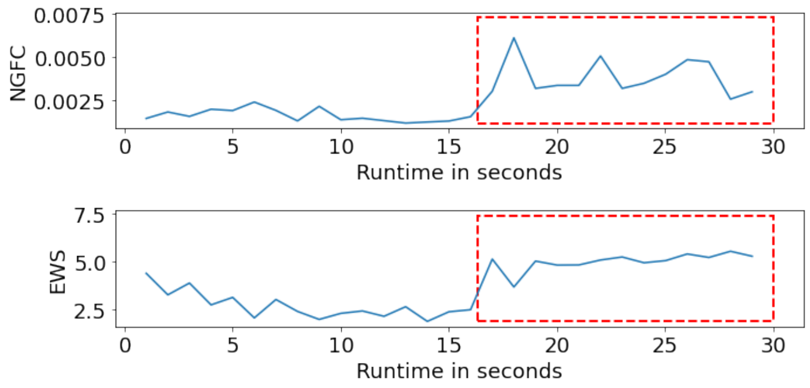
Because the goal of this evaluation was to demonstrate the ability of our solution to detect cyber attacks and differentiate them from false alarms, we chose to use *precision* as our evaluation metric. We calculated precision as (*100*Number of attack windows flagged*)/*Total number of windows flagged*. The start and end times of the attacks were noted in the collected logs and if a window was flagged within this period, it was labeled as an attack window.

**Observations**

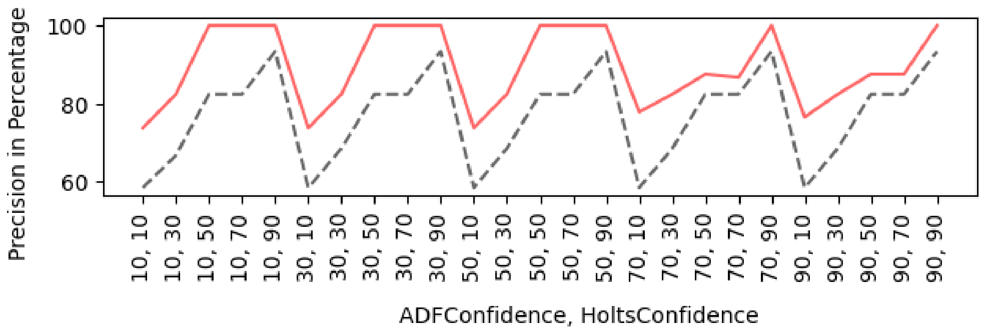
The results of executing Algorithm 1 on logs collected during the address claim attack, network overload attack and the two variations of the fuzzing attack validate our hypothesis in a real-world scenario. As seen in figures 4.5, 4.6, 4.7 and 4.8, the NGFC and EWS values show noticeable changes during the timeframe of the attack. The HoltsWinter-based anomaly detection technique detects the changes and in all cases, the best precision is obtained at 90% *HoltsConfidence* and training window size of 10. That is to say that increasing the training window size and keeping the *HoltsConfidence* at a high value increases the efficacy of our approach. In most of the cases, we achieve 100% precision at those configuration values.



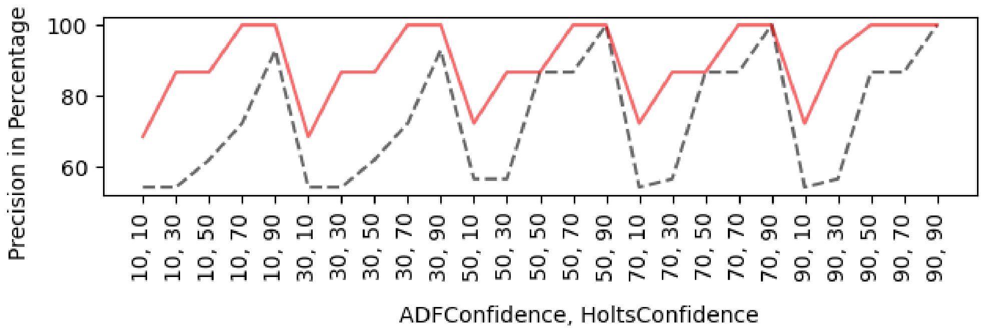
**(a)** Feature Value Time Series for First Address Claim Attack (Attack Timeframe in Red Box)



**(b)** Feature Value Time Series for Second Address Claim Attack (Attack Timeframe in Red Box)

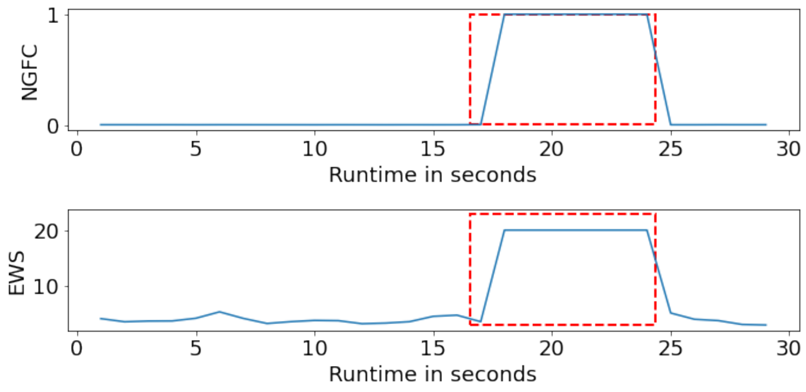


**(c)** Precision of Attack Detection for First Address Claim Attack

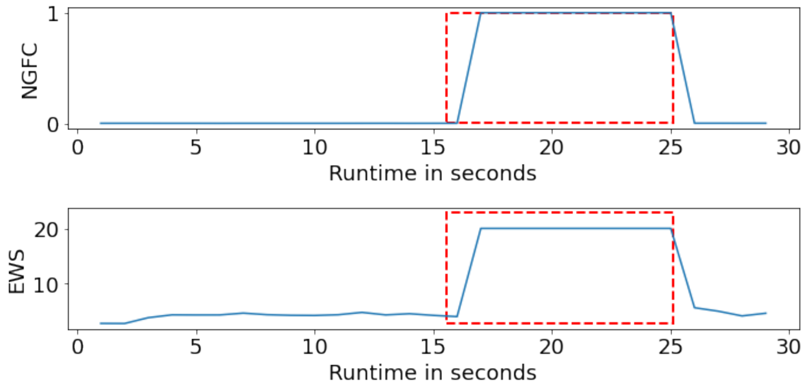


**(d)** Precision of Attack Detection for Second Address Claim Attack

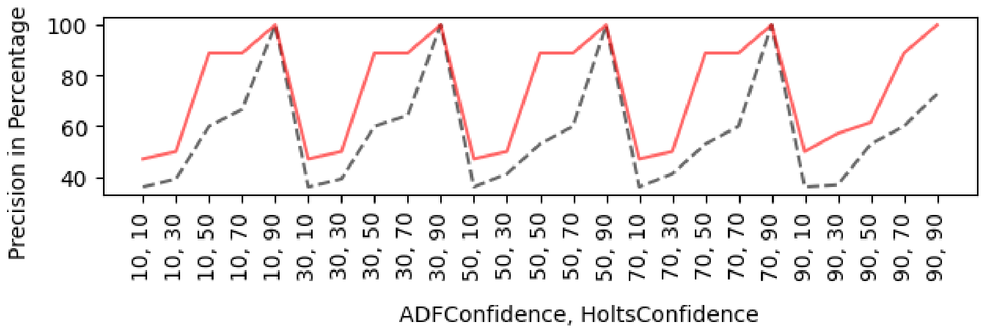
**Figure 4.5:** Address Claim Attack Detection Results



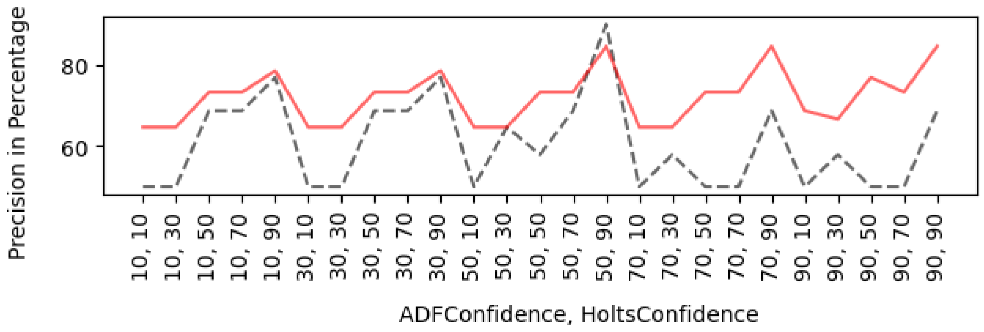
**(a)** Feature Value Time Series for First Network Overload Attack (Attack Timeframe in Red Box)



**(b)** Feature Value Time Series for Second Network Overload Attack (Attack Timeframe in Red Box)

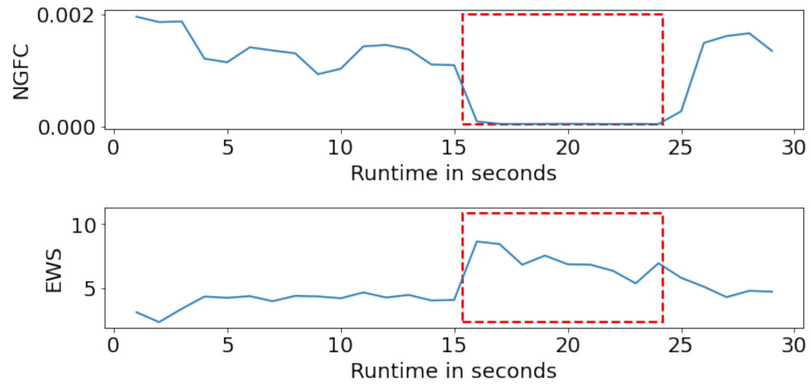


**(c)** Precision of Attack Detection for First Network Overload Attack

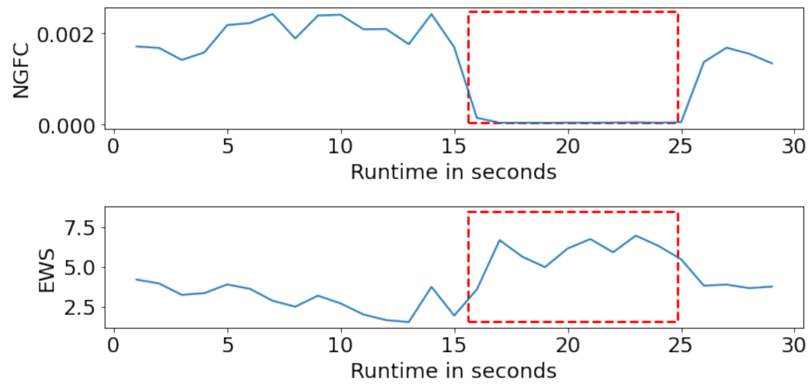


**(d)** Precision of Attack Detection for Second Network Overload Attack

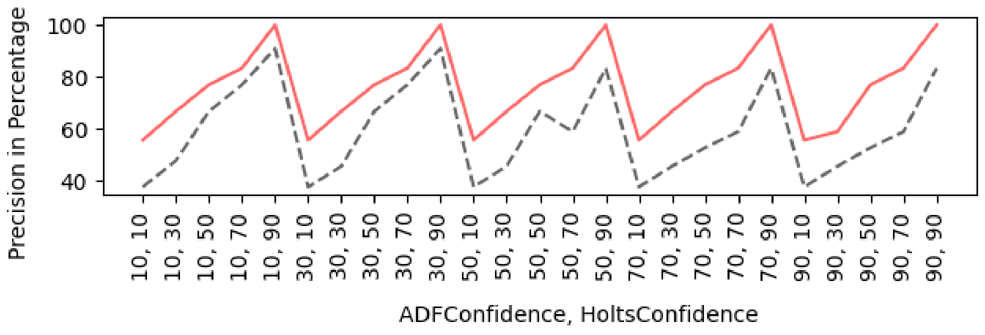
**Figure 4.6:** Network Overload Attack Detection Results



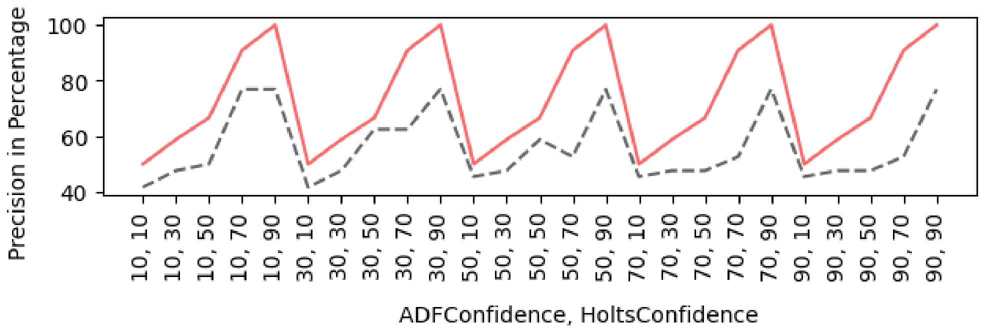
**(a)** Feature Value Time Series for First Attack Without Data Fuzzing (Attack Timeframe in Red Box)



**(b)** Feature Value Time Series for Second Attack Without Data Fuzzing (Attack Timeframe in Red Box)

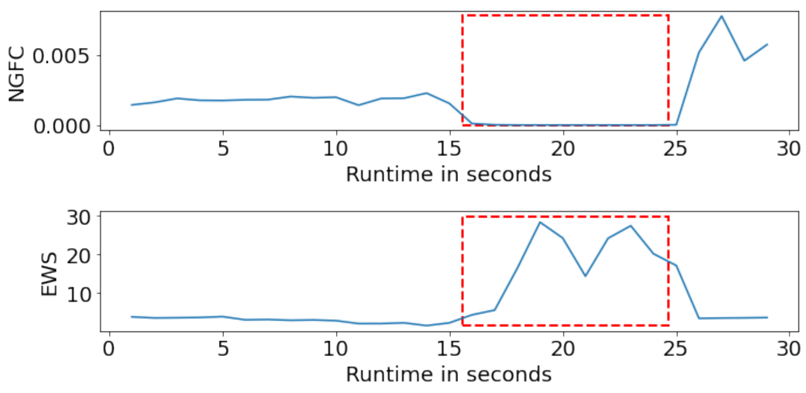


**(c)** Precision of Attack Detection for First Attack Without Data Fuzzing

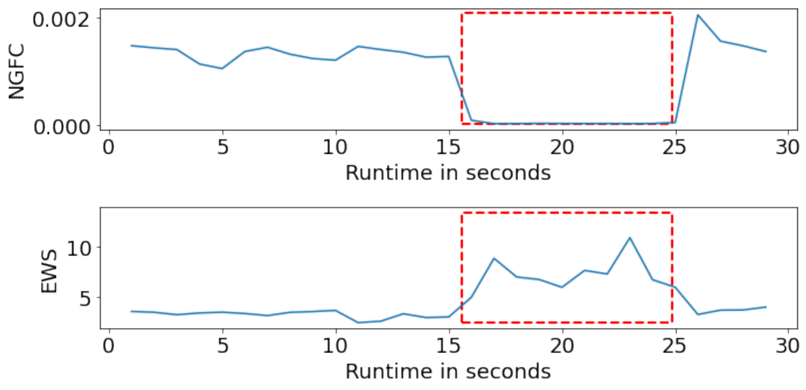


**(d)** Precision of Attack Detection for Second Attack Without Data Fuzzing

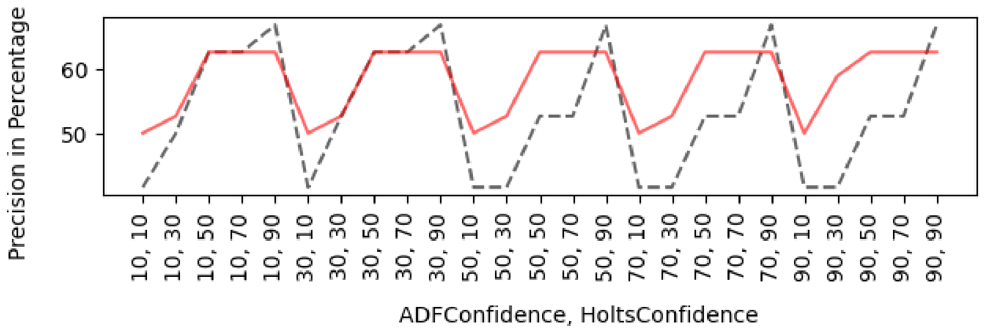
**Figure 4.7:** Attack Without Data Fuzzing Detection Results



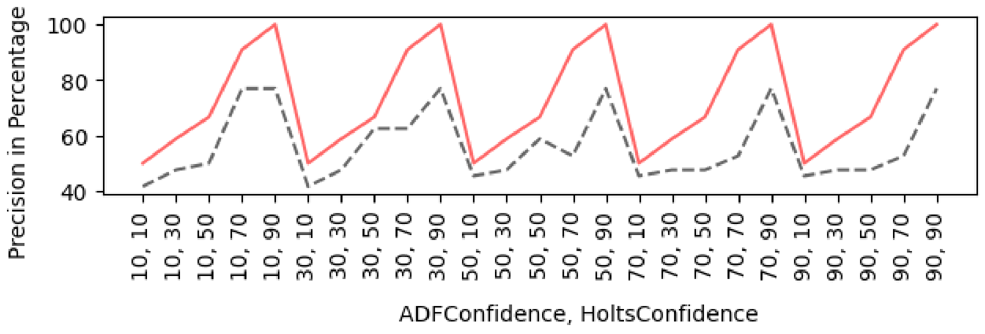
**(a)** Feature Value Time Series for First Attack With Data Fuzzing (Attack Timeframe in Red Box)



**(b)** Feature Value Time Series for Second Attack With Data Fuzzing (Attack Timeframe in Red Box)

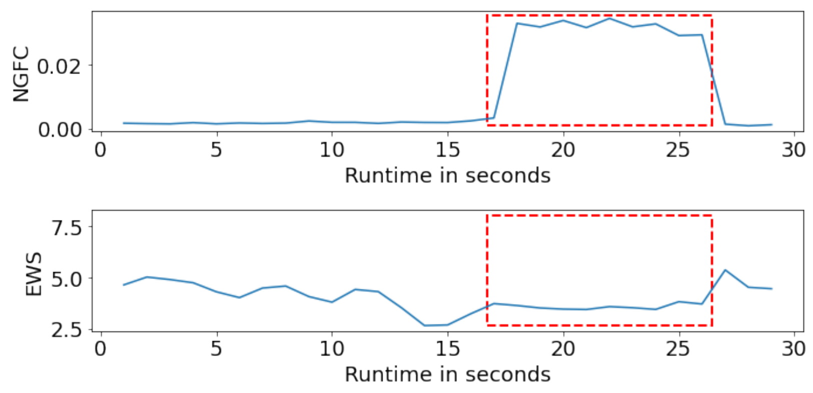


**(c)** Precision of Attack Detection for First Attack With Data Fuzzing

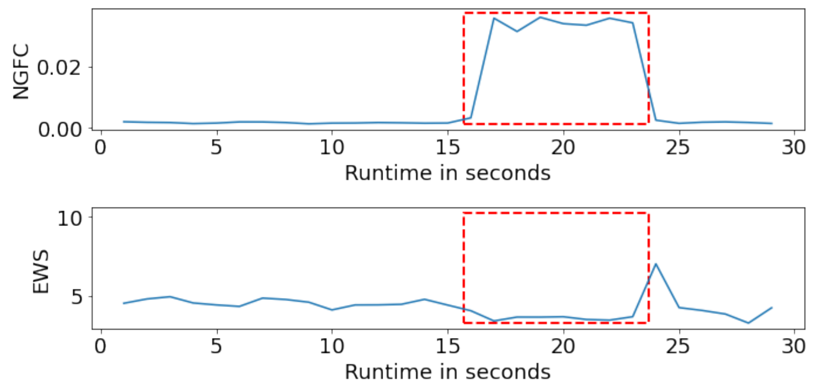


**(d)** Precision of Attack Detection for Second Attack With Data Fuzzing

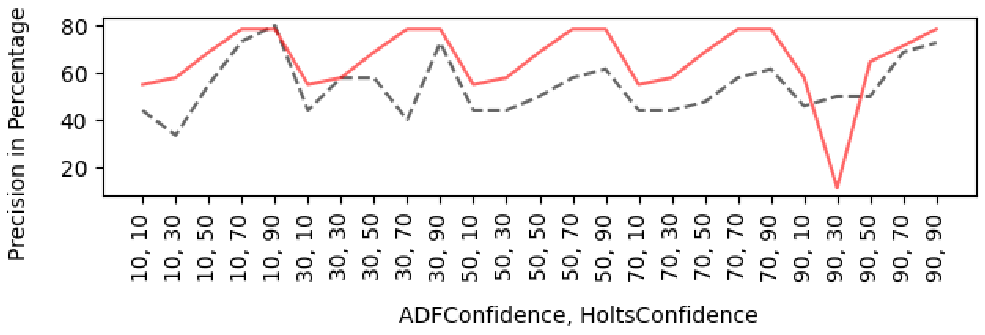
**Figure 4.8:** Attack With Data Fuzzing Detection Results



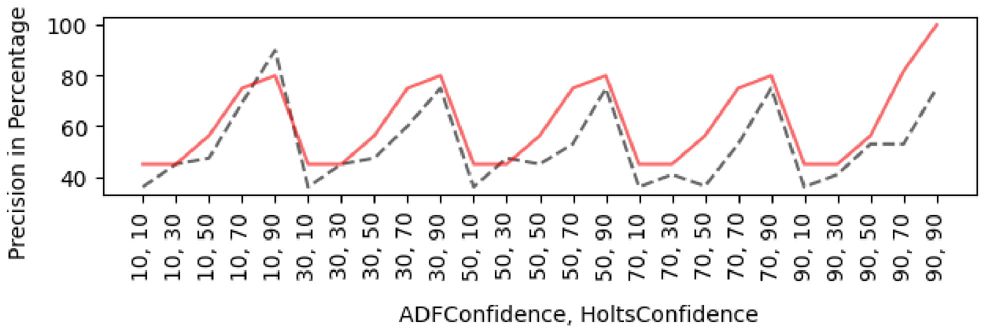
**(a)** Feature Value Time Series for First Request Overload Attack (Attack Timeframe in Red Box)



**(b)** Feature Value Time Series for Second Request Overload Attack (Attack Timeframe in Red Box)

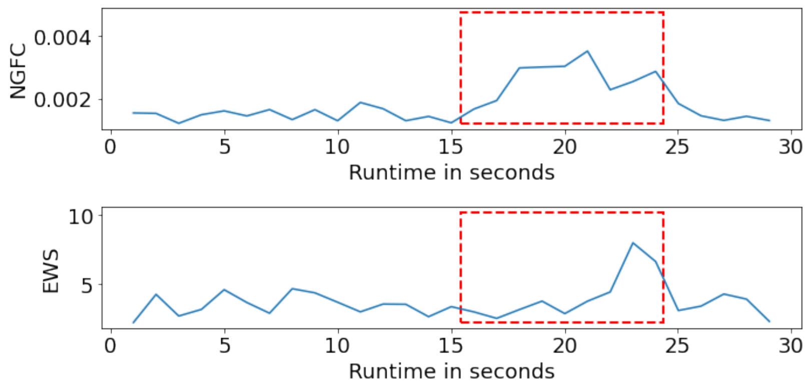


**(c)** Precision of Attack Detection for First Request Overload Attack

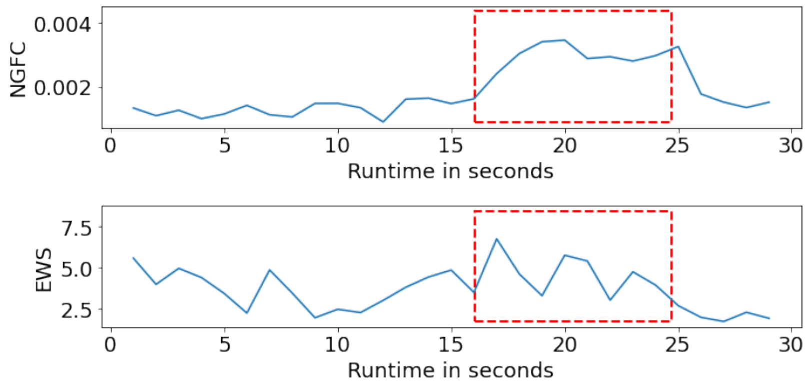


**(d)** Precision of Attack Detection for Second Request Overload Attack

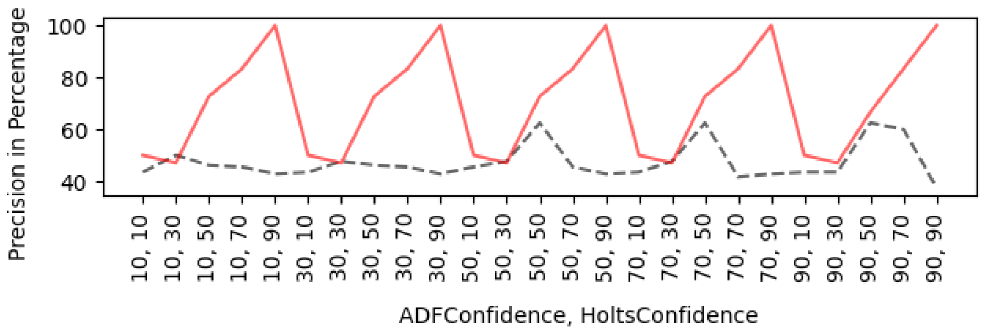
**Figure 4.9:** Request Overload Attack Detection Results



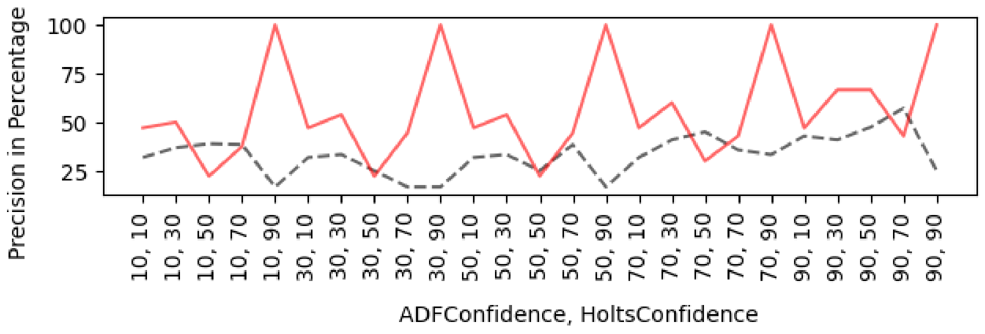
**(a)** Feature Value Time Series for First Engine Control Attack (Attack Timeframe in Red Box)



**(b)** Feature Value Time Series for Second Engine Control Attack (Attack Timeframe in Red Box)



**(c)** Precision of Attack Detection for First Engine Control Attack



**(d)** Precision of Attack Detection for Second Engine Control Attack

**Figure 4.10:** Engine Control Attack Detection Results

NGFC values show noticeable changes when the request overload and engine control attack are conducted, even though the same cannot be said about EWS values for the same. For the request overload attacks the EWS values during the timeframe of the attacks barely deviate from normal. For the engine control attacks, some deviations are seen but, because there are no common patterns, it cannot be inferred if the deviations are related to the attack or caused by normal behavior from the ECUs not under attack. Even so, our solution detects attacks. This is because the HoltsWinter-based anomaly detection technique detects the abnormal changes in NGFC values and, any deviations in EWS values causes it to raise alarms. Similar to the first four attacks, the best results are obtained at 90% *HoltsConfidence* and a training window size of 10. This substantiates that increasing the training window size and keeping the *HoltsConfidence* at a high value increases the efficacy of our approach. At those configuration values, we achieve 100% precision for engine control attack detection, but about 80% precision for request overload attack detection.

## 4.5 Summary

In this chapter, we answer the research question *can a system be designed to detect network anomalies on an SAE J1939 network in an online manner?* To that end we present a real-time SAE J1939-based intrusion detection system that does not require offline training. Instead, it uses time series forecasting using minimal historical data to predict an interval of expected behavioral feature values and compares them with the latest values to flag anomalies. We evaluate our system on 30 seconds of real-world network traffic collected from the Kenworth T270 research truck in normal driving circumstances, as well as under six different types of attack. The results indicate that, in most cases, our attack detection hypothesis is true and can be used to flag behavioral anomalies in SAE J1939 networks. According to the results, our system shows close to 100% precision in detecting most cyber attacks when configured appropriately. It also, very low false positive rates when tested on normal driving logs.

# 5 Rule-based Detection and Prevention

Research on rule-based intrusion detection and prevention for in-vehicle networks has so far relied on attack signatures defined on message content only and, as already discussed before, this may not be enough to detect cleverly crafted attacks on SAE J1939 networks. In this chapter, we present a rule-based intrusion detection and prevention system that presents users with ways to flag messages in transit that can and cannot be identified solely on the basis of their content. In the first section, we demonstrate how our system lets users define rules to capture SAE J1939-specific attacks. In the process, we describe the structure of the rules and attack detection features it captures. We also present a set of example rules in the same section. Next, in the second section, we describe a method to enforce user-specified rules in real-time, i.e. as messages are being transmitted. Following this, we analyze the performance of our rule enforcement procedure in the worst case. In there, we provide size estimates for sets of rules that can be processed with guarantee by our real-time enforcement mechanism. Eventually, we present real-world demonstrations of Controller Area Network (CAN) security for medium and heavy-duty (MHD) vehicles through the enforcement of example rules composed on our system. We conclude this chapter with a summary of our accomplishments.

## 5.1 Towards Rule-based Identification of Cyber Attacks on SAE J1939 Networks

In this section, we demonstrate how our system lets users define rules to capture SAE J1939-specific attacks. We first describe a set of features pertaining to SAE J1939 network messages that can be leveraged to detect attacks. We then describe how rules can be defined through our system to capture these features.

### 5.1.1 Attack Detection Features

#### Invalid Identifier

As already described in the background section, an attacker can spoof the sender's source addresses while conducting any attack. Also, for fuzzing attacks they can inject messages with random Parameter Group Numbers (PGN), some of which may not be valid for the network. These types of attacks can be detected by detecting messages with invalid (unsupported) PGNs, invalid (not present on the network) source addresses, or a combination of both. As an example, on the Kenworth T270 research truck, the engine controller accepts commands for torque and speed control (through messages with PGN = 0) from a body controller (with source address 33) but the truck does not include a body controller. This is a case of using an invalid source address that can be detected and acted upon. Abbot et al. [21] have also highlighted the importance of detecting invalid CAN IDs on passenger in-vehicle CAN networks.

#### Hazardous Parameter Values

Certain parameter values can be hazardous if processed by the recipient Electronic Control Unit (ECU). As an example, Burakova et al. [12] demonstrate an attack where, by sending a very low (e.g. less than 0%) torque to the engine controller, the vehicle's ability to accelerate can be hampered. A message carrying such a parameter value can be detected and acted upon in a security-critical manner. Lenard et al. [54] have also highlighted the importance of detecting invalid data bytes in CAN frames within passenger vehicles.

#### Hazardous Transmission Interval

It has already been discussed in section 2.3, that messages inserted at very small intervals can lead to denial-of-service. Network overload and request overload described before are examples of such attacks. Clearly, these attacks cannot be successful if exploit messages are sent at intervals higher than a certain value.

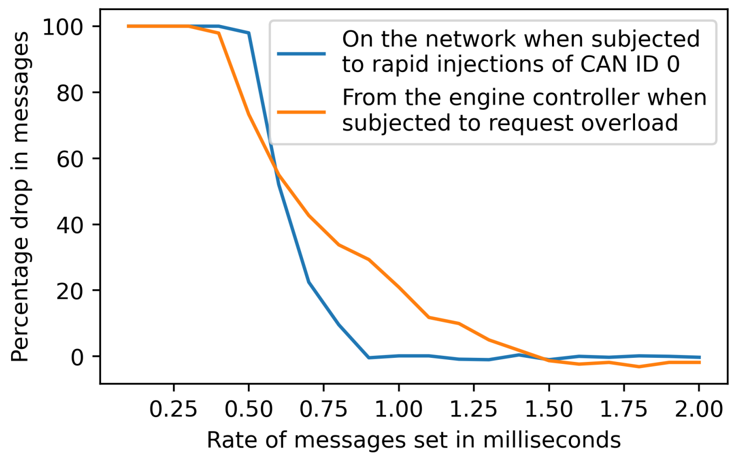


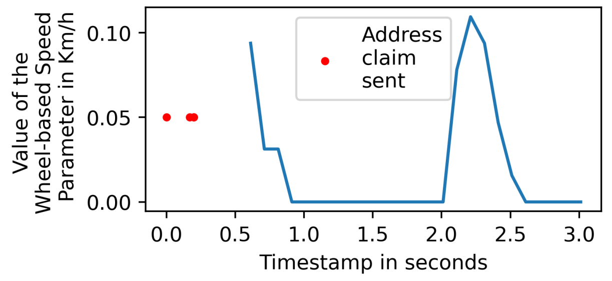
**Figure 5.1:** Effect of Network and Request Overload Attacks on the Kenworth T270 Research Truck

We conducted network and request overload attacks on the Kenworth T270 research truck and the results are shown in Figure 5.1. It is evident that when messages with CAN ID 0 are sent at rates higher than (approximately) 1 millisecond there is no significant drop in network traffic. Similarly, when request messages are sent at rates higher than (approximately) 1.5 milliseconds there is no significant drop in traffic from the engine controller on the network. As such, messages with CAN ID 0 that are sent at intervals lower than (approximately) 1 millisecond and request messages sent at intervals lower than 1.5 milliseconds to a specific device (like the engine controller) can be detected and acted upon in a security-critical manner.

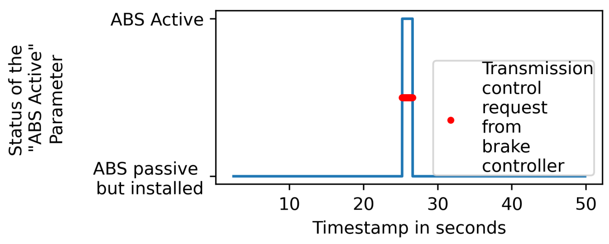
### Invalid or Hazardous Transmission Context

On an SAE J1939 network, ad hoc messages are transmitted occasionally. If an ad hoc message is always transmitted in a specific context and information describing the context is available on the network, it can be used to flag out-of-context injections as malicious. To better illustrate the use of this feature, we consider four real-world scenarios (demonstrated through Figure 5.2) that show the relationship between the transmission of ad hoc messages and contextual information.

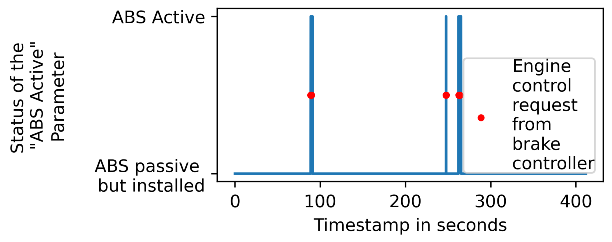
The first scenario is related to address claiming. As already mentioned in section 2.1.4, address claiming is performed before other communications are made on the network. Figure 5.2a validates this by showing the transmission of the address claim messages before the vehicle speed parameter



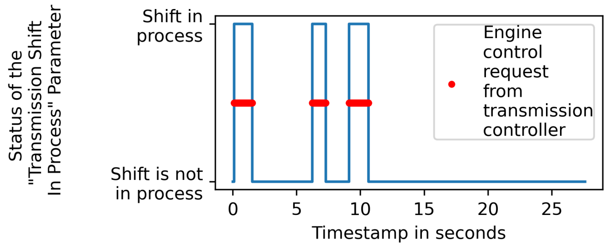
**(a)** Address Claim in Context of Vehicle Speed on the Kenworth T270



**(b)** Transmission Control in Context of ABS Activity on the Kenworth T270



**(c)** Torque and Speed Control Requests in Context of ABS Activity on The Kenworth T660



**(d)** Torque and Speed Control Requests in Context of Shift Activity on the Kenworth T270

**Figure 5.2:** Example Circumstances Depicting The Correspondence Between The Transmission of ad hoc Messages and Vehicle Context

is available on the network. However, address claim messages can be used to disable network activity from the target ECU as described in the address claim attack. If executed when the vehicle is in motion, this attack can lead to safety-critical scenarios (section 2.3). Given this, address claim messages can be flagged as malicious if they are transmitted when the vehicle speed parameter is at or above a certain non-zero value, e.g. 5 km/h.

Prior work on passenger vehicle security [32] has also identified messages that can be threatening if transmitted when the vehicle is driven at speed, a command to unlock the doors being an example. Messages similar are supported on SAE J1939 networks. These messages can be flagged if transmitted when the vehicle is above certain speeds.

In Figures 5.2c and 5.2d, we show two cases where legitimate torque and speed control messages are sent to the engine controller by the anti-lock braking system (ABS) and the transmission controller on two different trucks. In the first case, the messages are transmitted only when ABS is active and in the second case the same happens when transmission shift is in process, i.e. they are transmitted in certain context. For both cases, the context information is gathered from parameter values transmitted on the network. Figure 5.2b shows a similar case where legitimate transmission control messages are transmitted by the anti-lock braking system (ABS) only when the value of the parameter “ABS active" denotes that ABS is active. These observations can be leveraged to flag out-of-context (e.g. when ABS is not active or transmission shift is not in process) torque and speed control attempts and mitigate the attacks demonstrated by Burakova et al. [12].

In some cases, attacks may themselves be context-dependent. An example of such an attack is the retarder jam attack where a safety-critical scenario can be created by commanding 0% torque from the engine retarder only when the vehicle is being driven at speeds below 30 miles/h (48 km/h). Clearly, such a malicious activity can be detected (and possibly prevented from succeeding) by identifying the message to the engine retarder carrying a 0% torque request in the context of the vehicle speed being below 30 miles/h.

Given these observations, we propose that context can be used as an attack detection feature. Formally, we define context as a collection of a range of values for distinct parameters. That is, if $C$ is a context then it is a set where $\forall c \in C$, $c$ is a two tuple $< p, V >$, where V is a range of values of parameter $p$ and no parameter is repeated in $C$. We say that a parameter is *active* in a context $C$ if $\exists c \in C$ such that the first element of $c$ is the parameter whose latest transmitted value lies within the second element $c$. We say that a context is *active* if all the constituent parameters are active. An example context is: vehicle speed within [0,30] (km per hour) and electronic brakes pressed i.e. set to 1. This context is active when parameters vehicle speed and the status of the electronic brake switch are active i.e. when the vehicle speed is between 0-30 (km per hour) and the status of the electronic brake switch is set to 1.

### Hazardous Detection Count

While all of the above-mentioned features can be used to detect malicious messages, one may need to observe multiple suspicious messages before acting in a security-critical manner. Consider, the connection exhaustion attack described in section 3.3 of this dissertation. In this attack, a connection can be kept alive by periodically sending Clear to Send (CTS) messages to a specific

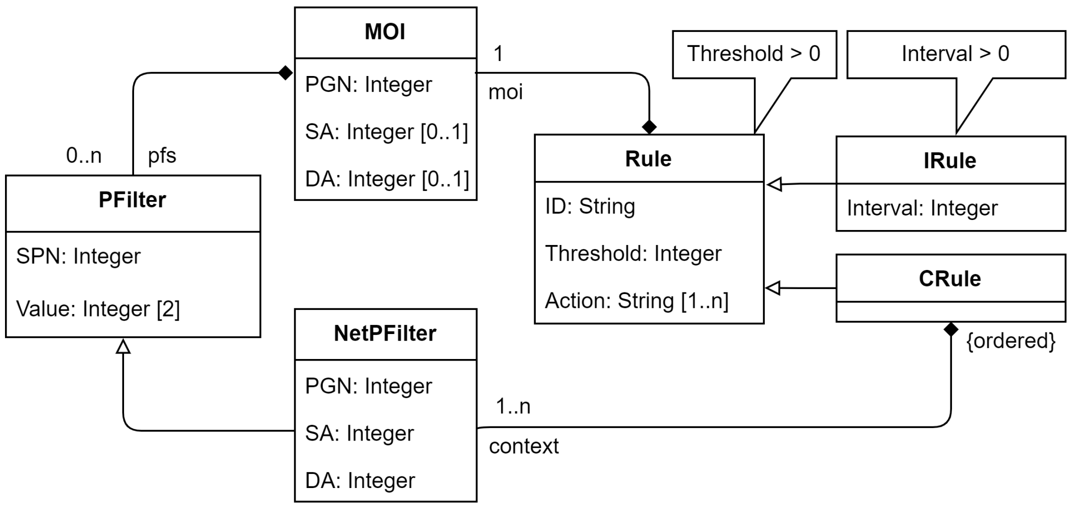


**Figure 5.3:** Rule Structure

target from a spoofed sender at intervals of less than 1250 ms. Given this, a user may determine CTS transmissions to an ECU to be malicious if more than a certain number of them have been transmitted from a specific sender in intervals of less than 1250 milliseconds.

### 5.1.2 Rule Description

Figure 5.3 shows the structure of rules using standard UML 2.5 notations [27]. In there, each block is a class that has attributes, can be extended using arrows with empty arrowheads and can contain other classes connected via lines that start with a black diamond and end with the cardinality and names of the relationship.

As per Figure 5.3, we support three types of rule classes. The first is `Rule` and the second and third are its extensions `IRule` and `CRule`. Each class is described next in this subsection.

#### `Rule` Objects

A `Rule` object can be used to detect messages with invalid PGN, DA, SA, and hazardous parameter values. It identifies a message of interest using an `MOI` object. A message is considered of interest to a rule object `o` if its PGN is equal to `o.moi.PGN`, its source address (SA) is equal to `o.moi.SA` if `o.moi.SA` is defined, its destination address (DA) is equal to `o.moi.DA` if `o.moi.DA` is defined and $\forall$ `p` $\in$ `o.moi.pfs` the value of the parameter (carried in the message)

| Type | Description | Threshold | Interval | MOI/NetPFilter |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  | Relation | PGN | DA | SA | PFilter |  |
|  |  |  |  |  |  |  |  | SPN | Value |
| Rule | Engine control request from body controller | 1 | N/A | moi | 0 | 0 | 33 |  |  |
| Rule | Very low torque request | 1 | N/A | moi | 0 | 0 |  | 518 | [-125,-125] |
| IRule | Request overload on engine controller | 1 | 5 | moi | 59904 | 0 |  |  |  |
| IRule | Connection exhaustion from a diagnostic tool on engine controller | 5 | 1250 | moi | 60416 | 0 | 249 | 2556 | [17,17] |
| IRule | Network overload | 1 | 5 | moi | 0 | 0 | 0 |  |  |
| CRule | Address claim after vehicle speed has reached 5 km/h | 1 | N/A | moi | 60928 |  |  |  |  |
|  |  |  |  | context | 65265 | 255 | 0 | 84 | [5,300] |
| CRule | Engine control request from ABS when it is not active | 1 | N/A | moi | 0 | 0 | 11 |  |  |
|  |  |  |  | context | 61441 | 255 | 11 | 563 | [0,0] |
| CRule | 0% torque request to engine retarder when a low vehicle speed | 1 | N/A | moi | 0 | 15 |  | 518 | [0,0] |
|  |  |  |  | context | 65265 | 255 | 0 | 84 | [0,30] |

**Table 5.1:** Example Rules

having Suspect Parameter Number (SPN) = `p.SPN` is $\leq$ `p.Value[1]` and $\geq$ `p.Value[0]`. We say that the message of interest triggers $o$. As such, a message of interest for `Rule` object `r` is acted upon if it triggers `r` and the previous $k$ messages of interest for `r` have all triggered it, $k$ being $\geq$ `r.threshold`-1.

The first two rows of Table 5.1 show examples of `Rule` objects.

- In the first example, a torque and speed control message (PGN = 0) to the engine controller (DA = 0) transmitted by the body controller (SA = 33) is specified to be acted upon from its first (Threshold = 1) transmission.
- In the second example, a torque and speed control message (PGN = 0) to the engine controller (DA = 0) requesting a -125% (Value = [-125,-125]) torque (SPN = 518) is specified to be acted upon from its first (Threshold = 1) transmission. This can be a possible defense strategy against the throttle jam attack described in section 2.3.2.

### `IRule` Objects

An `IRule` object can be used to detect messages transmitted at hazardous intervals. Being an extension of the `Rule` class, it identifies messages of interest in the same manner, i.e. using the included `MOI` object. As seen in figure 5.3, `IRules` allows specifying an interval greater than 0 in the `interval` attribute. We say that a message of interest triggers an `IRule` object `r` if it is transmitted at an interval of less than or equal to `r.interval` milliseconds from the previous message of interest. A message of interest for `IRule` object `r` is acted upon if it triggers rule `r` and the previous $k$ message of interest for $r$ have all triggered $r$, $k$ being $\geq$ `r.threshold` -1.

The rows three, four, and five of Table 5.1 show examples of `IRule` objects.

- In the first example, request messages (PGN = 59904) to the engine controller (DA = 0) are specified to be acted upon if they are transmitted within 5 milliseconds. This can be a possible defense strategy against the request overload attack described in section 3.

- In the second example, back-to-back CTS messages (PGN = 60416 and first data byte occupied by the parameter with SPN 2556 = 17) sent to the engine controller (DA = 0) from the onboard data logger (SA = 249) within 1250 milliseconds are specified to be acted upon after 5 such transmissions. This can be a possible defense strategy against the connection exhaustion attack described in section 3.

- In the third example, messages with PGN = 0, DA = 0, and SA = 0 are specified to be acted upon if they are transmitted within 5 milliseconds. This can be a possible defense strategy against the network overload attack described in section 2.3.2.

### `CRule` Objects

A `Crule` object is used to detect out-of-context message injections. Being an extension of the `Rule` class, it identifies messages of interest in the same manner, i.e. using the included `MOI` object. As seen in figure 5.3, `CRules` allow specifying the context as a collection of `NetPFilter` objects. Being an extension of the `PFilter` class, `NetPFilter` identifies a parameter using its SPN and denotes the value range as a pair of integers. Parameters on the SAE J1939 network can be transmitted in messages having different PGNs, DA, and SA. As such, a `NetPFilter` object includes a `PGN`, `DA`, and `SA` attribute that helps in identifying the message from which the parameter value must be read to verify if the context is active. We say that a message of interest triggers an `CRule` object `r` if it is transmitted when the context is active. A message of interest for `CRule` object `r` is acted upon if it triggers rule `r` and the previous $k$ message of interest for $r$ have all triggered $r$, $k$ being $\geq$ `r.threshold`-1.

The rows six, seven, and eight of Table 5.1 show examples of `CRule` objects.

- In the first example, address claim messages (PGN = 60928) to the engine controller (DA = 0) are specified to be acted upon if they are transmitted when the vehicle speed (SPN = 84) transmitted by the engine controller (SA = 0) in a message with PGN 65265 is greater than 5 km/h (Value = [5, 300]). This can be a possible defense strategy against the address claim attack described in section 2.3.2.
- In the second example, torque and speed control messages (PGN = 0) to the engine controller (DA = 0) from the ABS controller (SA = 11) are specified to be acted upon if they are transmitted when the status of the ABS (SPN = 563), as transmitted by the ABS controller (SA = 11) in messages with PGN 61441, is not active (Value = [0,0]). This can be a possible defense strategy against the engine control attack described in section 2.3.2 launched by spoofing the ABS controller's source address.
- In the third example, torque and speed control messages (PGN = 0) carrying 0% torque requests (SPN = 518) to the engine retarder (DA = 15) are specified to be acted upon if they are transmitted when the vehicle speed (SPN = 84) transmitted by the engine controller (SA = 0) in a message with PGN 65265 is less than 30 km/h (Value = [0, 30]). This can be a possible defense strategy against the retarder jam attack described in section 2.3.2.

### Rule Actions

Every rule in our system has a unique `i`dentifier and can be associated with one or more action strategies by specifying the `action` strings. Although we do not explicitly specify the exact actions to perform, we discuss some action strategies here. Each `action` string can be used as an identifier for an action strategy. At runtime, we pass these strings to an abstract function that should implement one or more action strategies and executes them based on the passed `action` strings.

**Alarm** Alarms can be raised to alert the driver or external monitors. Other devices on an SAE J1939 network can also be alerted by sending proprietary alert messages.

**Message disruption** Prior work [21, 22, 40] has highlighted that it may be possible to disrupt the transmission of a frame if attacks are detected in real-time i.e. as bits of the frame are transmitted on the CAN network. We enforce rules in real-time and as such, this method can be adopted as a possible action strategy. Recall from section 2.1.2 that CAN frames are allowed to collide during the transmission of the ID but after that, the bus is allocated exclusively to a specific transmitter. A frame is disrupted if a collision occurs after the bus is allocated, i.e. if a recessive bit is overwritten with a dominant one after the transmission of the ID is complete. If a rule specifies an `action` string that requires frame disruption, a sequence of six dominant bits can be sent after the ID of the message of interest has been transmitted. Because of bit stuffing, this will ensure that at least one recessive bit after the CAN ID of the message of interest is overwritten by one of the six dominant bits, leading to frame disruption.

**Event logging** Lenard et al. [54] suggest logging security events locally if an attack is detected on the CAN network. The event log can be used later for digital forensics. Albeit, an attacker can generate several security events and overflow the on-device storage. If event logging is chosen, adequate on device space must be allocated and the data must be exfiltrated periodically to ensure the space can be reused. The amount of on-device space required will depend on how often the logs are exfiltrated and must be estimated apriori.

**Vehicle restart** An action strategy to mitigate the attack can be to restart the vehicle. This will allow the driver to take defensive actions as the injection of malicious messages is suspended. Albeit, this should only be performed if the issuer of the rule that includes this action strategy is confident that the action will be enforced only if an attack is detected, not in a benign scenario.

**Rule Constraints**

Our system enforces three constraints on the defined rules.

1. The first constraint states that, if a PGN is used in an `MOI` object, it cannot be used in a `NetPFilter` object and vice-versa. A complicated situation may arise if a message that carries parameters used in a context definition is also malicious. It raises the question, should the parameter values be used to check if context definitions that use them are active? If the message is disrupted, we may not be able to obtain all parameter values to compare with context definitions. This work does not address this situation at this time. As such, we enforce the aforementioned constraint.

2. The second constraint states that, for any `MOI` object if *SA* is specified, then so should *DA*. This constraint is enforced to ensure the correct functionality of the rule enforcement procedure described in the next section.

3. The third constraint states that no two rules of the same type can exist without `PFilters` but with the same PGN, DA and SA. Since `CRules` always include `PFilters`, this constraint only applies to objects of type `Rule` and `IRule`. Essentially, it prevents the same message of interest to be acted upon by two different rules of the same type having no `PFilters`.

## 5.2 Rule Enforcement

In this section, we describe the method to enforce user-specified rules in real time, i.e. as messages are being transmitted. One way to approach this problem can be to buffer message bits until the CAN data field is received fully and then search exhaustively through all the rules comparing the PGN, DA, SA, and parameter values with the information specified in the rules. However, in a real-world scenario, it is highly unlikely that the same PGN, DA, and SA combination will be used in all rules. As such, we create an index from the PGN, DA, and SA used in rules and look

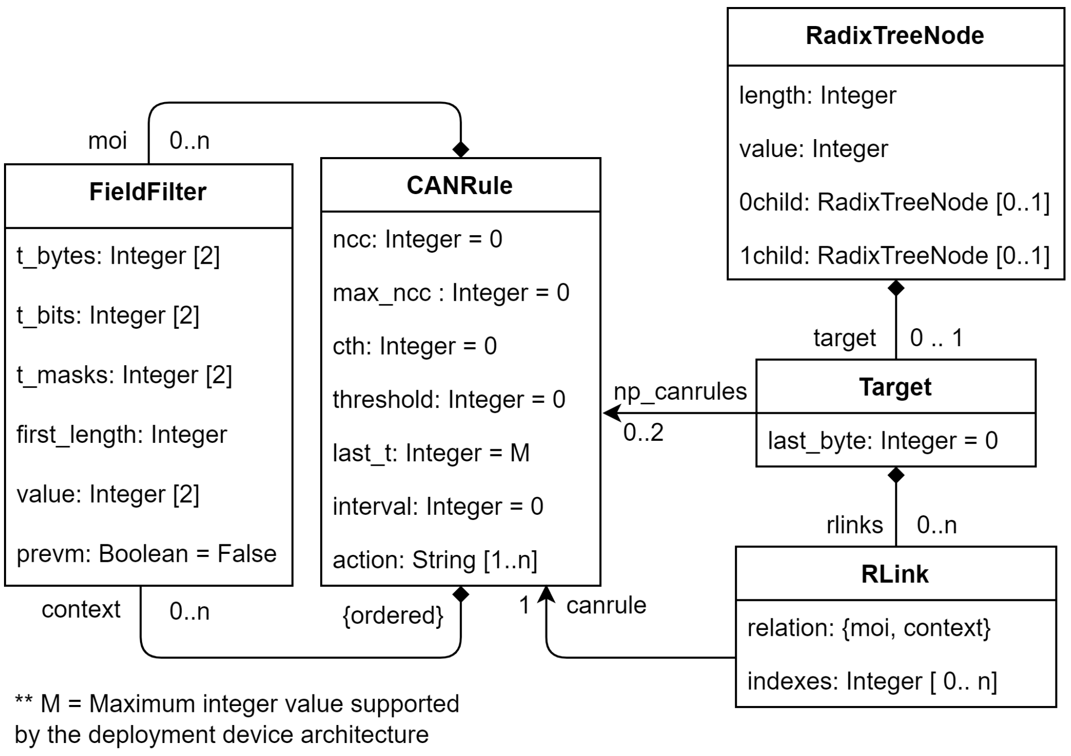


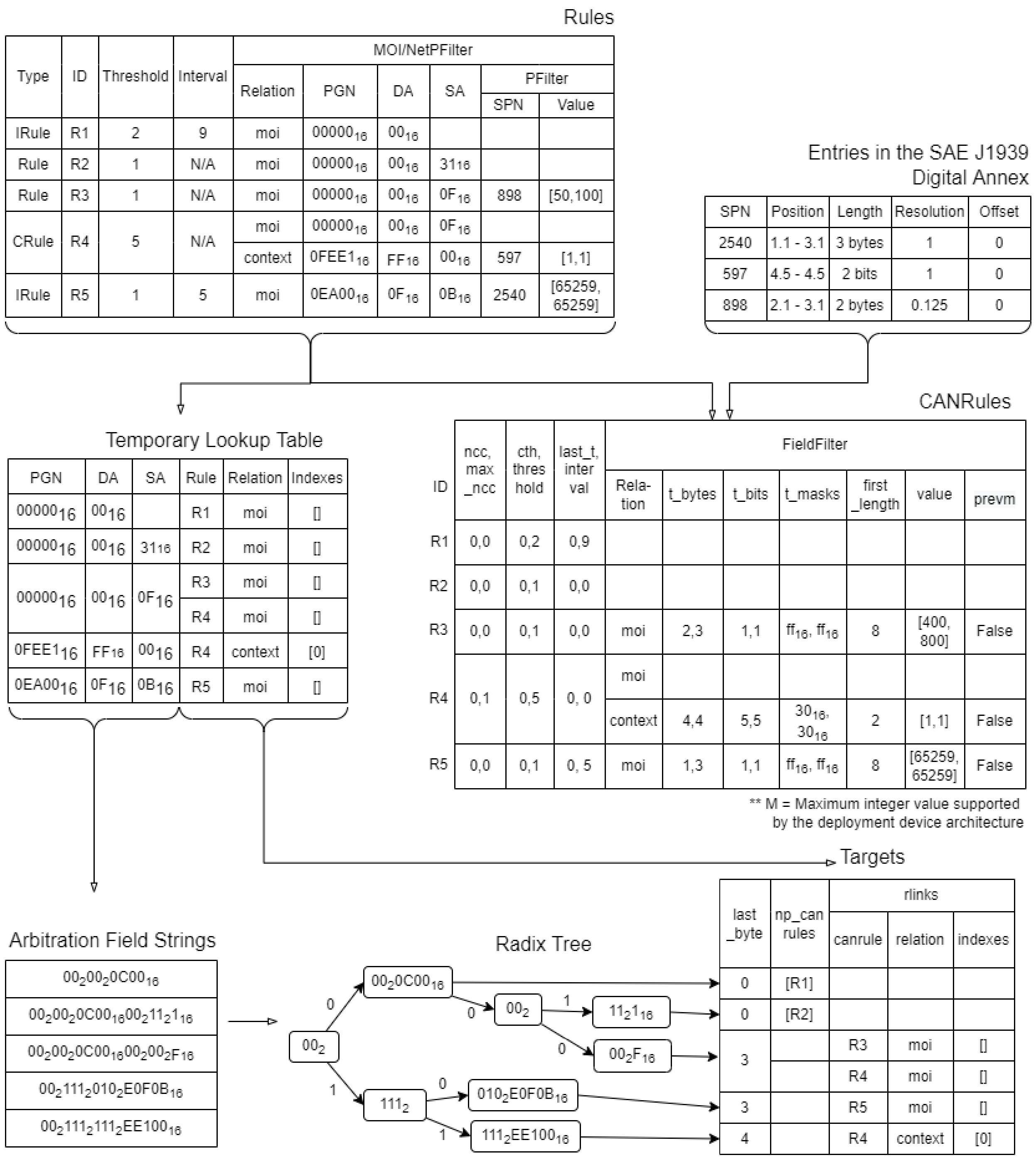
**Figure 5.4:** Low-Level Data Structure Used for Rule Enforcement

up the index as the bits of the CAN ID arrive. The lookup yields information related to the set of rules that use the PGN, DA, and SA of the incoming message. This is expected to reduce the search space significantly. Moreover, we do not wait till the end of the CAN data field. Instead, we wait until we have received the last byte of the CAN data field in which a parameter, used in a rule from the retrieved set, is placed into. This allows us increased time to search through the already reduced rule space.

In the first subsection of this section, we describe the preprocessing steps required to generate the index and associated information from the user-provided rules. In the next subsection, we describe how we look up the index in real-time and use the retrieved information to enforce the rules.

### 5.2.1 Preprocessing

During the preprocessing phases, we convert user-provided rules into low-level objects whose structure is shown in Figure 5.4 using standard UML 2.5 notations [27]. In there, each block is



Rules

| Type | ID | Threshold | Interval | MOI/NetPFilter | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Relation | PGN | DA | SA | PFilter | |
| | | | | | | | | SPN | Value |
| IRule | R1 | 2 | 9 | moi | $00000_{16}$ | $00_{16}$ | | | |
| Rule | R2 | 1 | N/A | moi | $00000_{16}$ | $00_{16}$ | $31_{16}$ | | |
| Rule | R3 | 1 | N/A | moi | $00000_{16}$ | $00_{16}$ | $0F_{16}$ | 898 | [50,100] |
| CRule | R4 | 5 | N/A | moi | $00000_{16}$ | $00_{16}$ | $0F_{16}$ | | |
| | | | | context | $0FEE1_{16}$ | $FF_{16}$ | $00_{16}$ | 597 | [1,1] |
| IRule | R5 | 1 | 5 | moi | $0EA00_{16}$ | $0F_{16}$ | $0B_{16}$ | 2540 | [65259, 65259] |

Entries in the SAE J1939 Digital Annex

| SPN | Position | Length | Resolution | Offset |
|---|---|---|---|---|
| 2540 | 1.1 - 3.1 | 3 bytes | 1 | 0 |
| 597 | 4.5 - 4.5 | 2 bits | 1 | 0 |
| 898 | 2.1 - 3.1 | 2 bytes | 0.125 | 0 |

Temporary Lookup Table

| PGN | DA | SA | Rule | Relation | Indexes |
|---|---|---|---|---|---|
| $00000_{16}$ | $00_{16}$ | | R1 | moi | [] |
| $00000_{16}$ | $00_{16}$ | $31_{16}$ | R2 | moi | [] |
| $00000_{16}$ | $00_{16}$ | $0F_{16}$ | R3 | moi | [] |
| | | | R4 | moi | [] |
| $0FEE1_{16}$ | $FF_{16}$ | $00_{16}$ | R4 | context | [0] |
| $0EA00_{16}$ | $0F_{16}$ | $0B_{16}$ | R5 | moi | [] |

CANRules

| ID | ncc, max _ncc | cth, thres hold | last_t, inter val | FieldFilter | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Rela-tion | t_bytes | t_bits | t_masks | first _length | value | prevm |
| R1 | 0,0 | 0,2 | 0,9 | | | | | | | |
| R2 | 0,0 | 0,1 | 0,0 | | | | | | | |
| R3 | 0,0 | 0,1 | 0,0 | moi | 2,3 | 1,1 | $ff_{16}$, $ff_{16}$ | 8 | [400, 800] | False |
| R4 | 0,1 | 0,5 | 0, 0 | moi | | | | | | |
| | | | | context | 4,4 | 5,5 | $30_{16}$, $30_{16}$ | 2 | [1,1] | False |
| R5 | 0,0 | 0,1 | 0, 5 | moi | 1,3 | 1,1 | $ff_{16}$, $ff_{16}$ | 8 | [65259, 65259] | False |

** M = Maximum integer value supported by the deployment device architecture

Arbitration Field Strings

| |
|---|
| $00_2 00_2 0C00_{16}$ |
| $00_2 00_2 0C00_{16} 00_2 11_2 1_{16}$ |
| $00_2 00_2 0C00_{16} 00_2 00_2 F_{16}$ |
| $00_2 111_2 010_2 E0F0B_{16}$ |
| $00_2 111_2 111_2 EE100_{16}$ |

Targets

| last _byte | np_can rules | rlinks | | |
|---|---|---|---|---|
| | | canrule | relation | indexes |
| 0 | [R1] | | | |
| 0 | [R2] | | | |
| 3 | | R3 | moi | [] |
| | | R4 | moi | [] |
| 3 | | R5 | moi | [] |
| 4 | | R4 | context | [0] |

**Figure 5.5:** Example Preprocessing of Rules

a class that has attributes, can be extended using arrows with empty arrowheads and can contain other classes connected via lines that start with a black diamond and end with the cardinality and names of the relationship. Essentially, these objects represent the index and associated information required to enforce user-provided rules in real time. An example process of generating these objects is also elucidated in Figure 5.5.

Preprocessing is done in three steps, which are detailed next.

### Generating a Temporary Lookup Table from Rules

In the temporary lookup table, we map PGN, DA, and SA combinations to the rules and their parts (`MOI` or `NetPFilter`) in which they are used in. To identify the parts, we use the name (`Relation` in the temporary lookup table) of their relationship with the containing rule. This can be *moi* or *context*. Additionally, to identify `NetPFilter` parts we use their indexes (`Indexes` in the temporary lookup table) in the *context* associations. For `MOI` parts `Indexes` is set to an empty list.

### Generating CANRules from Rules

`CANRule` objects contain information that is updated from the network and compared to specifications made in the rules. We create one `CANRule` for each rule. Throughout the rest of this chapter, we refer to a rule as the *parent* of an `CANRule` object if the latter is created using information from the former. For the purpose of clarity, Figure 5.5 shows an `ID` that is assigned to each `CANRule` to identify it and its parent rule. The ID is not used in the implementation.

Each `CANRule` object includes eight fields namely `ncc`, `max_ncc`, `cth`, `threshold`, `last_t`, `interval`, `moi` and `context`. Fields `ncc`, `cth`, `last_t` and `prevm` are temporary variables that are set to default values and updated at runtime to keep track of decisions made during the rule enforcement procedure. At runtime, values of `ncc`, `cth` and `last_t` are compared against values of `max_ncc`, `threshold` and `interval` respectively. `max_ncc` denotes the total number of the parameters used in the context definition provided by the parent rule and is equal to the cardinality of the `context` association of a rule. By default, it is set to 0 if the parent rule is not an instance of `CRule`. `threshold` and `interval` is directly copied from the parent rule. `interval` is set to 0 if the parent rule is not of type `IRule`. One `FieldFilter` is created for every `PFilter` used in the parent rule. The $ith$ `FieldFilter` in the `moi` association is created from the i$th$ PFilter associated with the MOI object of the parent rule. Similarly, the i$th$ FieldFilter in the context association is created from the i$th$ NetPFilter in the context association of the parent rule.

Because the SPN in a PFilter does not provide the neccessary information required to parse parameter values out of messages, in the fields t_bytes, t_bits, t_masks and first_length a FieldFilter object carries that information. t_bytes, t_bits, t_masks and first_length are assigned values by querying the SAE J1939 digital annex with the SPN used in the original PFilter object. Fields t_bytes and t_bits hold the byte (R and S) and bit (x and w) component of the SAE J1939 placement notation (position) “R.x - S.w". The t_masks hold two-bit masks of length 8 each and first_length is the number of bits allocated in byte R. If R = S (i.e. the parameter is placed entirely in byte R), bits x to x + $l$ -1 of t_mask are set and first_length is set to $l$, where $l$ is the length of the parameter as obtained from the SAE J1939 digital annex. If R is not equal to S, in the first-bit mask, bits x to 8 are set, and in the second mask, bits w to the last bit used by the parameter are set. For this case, first_length is set to 8 - x + 1. We suggest the reader refer back to section 2.1.3 for a review of J1939 parameter placement if required.

The value of a FieldFilter object is produced by subtracting the offset from each endpoint of the original PFilter's Value and dividing the results by the resolution. An example of this transformation is shown in the third record of the CANRules table in Figure 5.5. The value in that record is [400, 800] which is obtained by subtracting 0 from each of 50 and 100 and dividing the results by 0.125, which, as seen in the snippet of the digital annex just above, is the resolution for the parameter with SPN 898 used in rule R3. This transformation is done to avoid storing the resolution and offset factors for use in parameter parsing at runtime.

### Converting The Lookup Table into a Search Tree

The temporary lookup table can be used to look up information required for rule enforcement. However, this means that the entire CAN ID needs to be available before PGN, DA, and SA can be extracted from it and used for lookup. Furthermore, the lookup will be linear with respect to the size of the table and can consume a significant amount of time that could instead be used for processing the looked-up information. Given this, we convert the PGN, DA, and SA combinations in the temporary lookup table into *arbitration field bit strings* and create a radix search tree from them. At runtime, we search in this tree as CAN ID bits become available. Prior work on IP packet classification [66, 67, 68] has used prefix search trees for purposes similar, although, in those cases, the search has been performed after the packet has been received completely. A radix tree is a space-optimized version of the prefix tree.

An arbitration field bit string is the base 2 representations of the CAN arbitration field for a given PGN, DA, and SA (refer back to section 2.1.2 for a review of the CAN frame's format). Each arbitration field bit string can be at most 28 bits long. It does not include the first three ID bits that correspond to the priority field in the SAE J1939 protocol data unit (PDU) and the last bit that corresponds to the CAN RTR field. These fields are not used explicitly in rule definition. For the purpose of brevity and to showcase the common prefixes used in the radix tree, the arbitration field bit strings in Figure 5.5 are shown as concatenations of base 2 and base 16 strings. In Figure 5.5, each node of the radix tree is labeled with a substring of an arbitration field bit string and can be attached to a `Target` object. If a `Target` object is attached to a node $n$, the concatenation of the node labels from the root to $n$ is an arbitration field bit string. We say that a record in the temporary lookup table *corresponds* to the target object if the PGN, DA, and SA in that record is used to generate the arbitration field bit string. A node in our radix tree can have zero to two children. The link between a parent and a child node is labeled with the first character of the child's label. A node is implemented using a `RadixtrieNode` object described in Figure 5.4. The `RadixtrieNode` object has a `length` and `value` that respectively are equal to the length and decimal equivalent of the label in the node. Links to the children are stored in the `0child` and `1child` attributes, 0 and 1 in the names denoting the label of the link. The process of creating a radix tree from a set of strings is considered to be common knowledge and, hence, is not described herein.

A `Target` object represents the association between the `PGN`, `DA`, `SA` triple and the `Rule`, `Relation`, `Indexes` triples in its corresponding record of the temporary lookup table. It splits the set of `Rule`, `Relation`, `Indexes` triples into two sets and associates them with two attributes. The first attribute `np_canrules` points to those `CANrules` whose parents appear in the corresponding records but do not include any `PFilters`. The second attribute `rlinks` points to the parts of those `CANrules` whose parents appear in the corresponding records but include `PFilters`. This partitioning is done so rules with no `PFilters` can be evaluated before CAN data is received. The `Target` object also includes a `last_byte` that denotes the last CAN data byte that needs to be received before the `CANRules` pointed by `rlinks` can be processed. As such, this is set to the maximum of the second element of `t_bytes` of all the `FieldFilters` used in those `CANRules`. If the cardinality of the `rlinks` relation is 0, `last_byte` is set to 0 by default.

### 5.2.2 Runtime Approach

To enforce the rules in real-time i.e. as messages are transmitted on the network, we need to read in the bits of the message and use this information to make security decisions. In [69] Campo et al. have shown that the Non-Return-To-Zero (NRZ) encoded output of a CAN transceiver can be decoded using an interrupt-driven technique to obtain three pieces of information namely,

- Number of bits in the pulse that we refer to as pulse width or `pw`.
- Signal level of the pulse (0 or 1) that we refer to as signal level or `sl`.
- Index/position of the starting bit of the pulse in the CAN frame. We refer to this as `pos`. As such, `pos` is 1 at start of frame (SOF).

In this work, we assume that such a method exists and outputs the above-mentioned three pieces of information every time an NRZ pulse is received from the CAN transceiver. We also assume the existence of a wrapper that executes after this method and enqueues these three pieces of information in a queue. Our rule enforcement procedure runs in a loop and dequeues one item

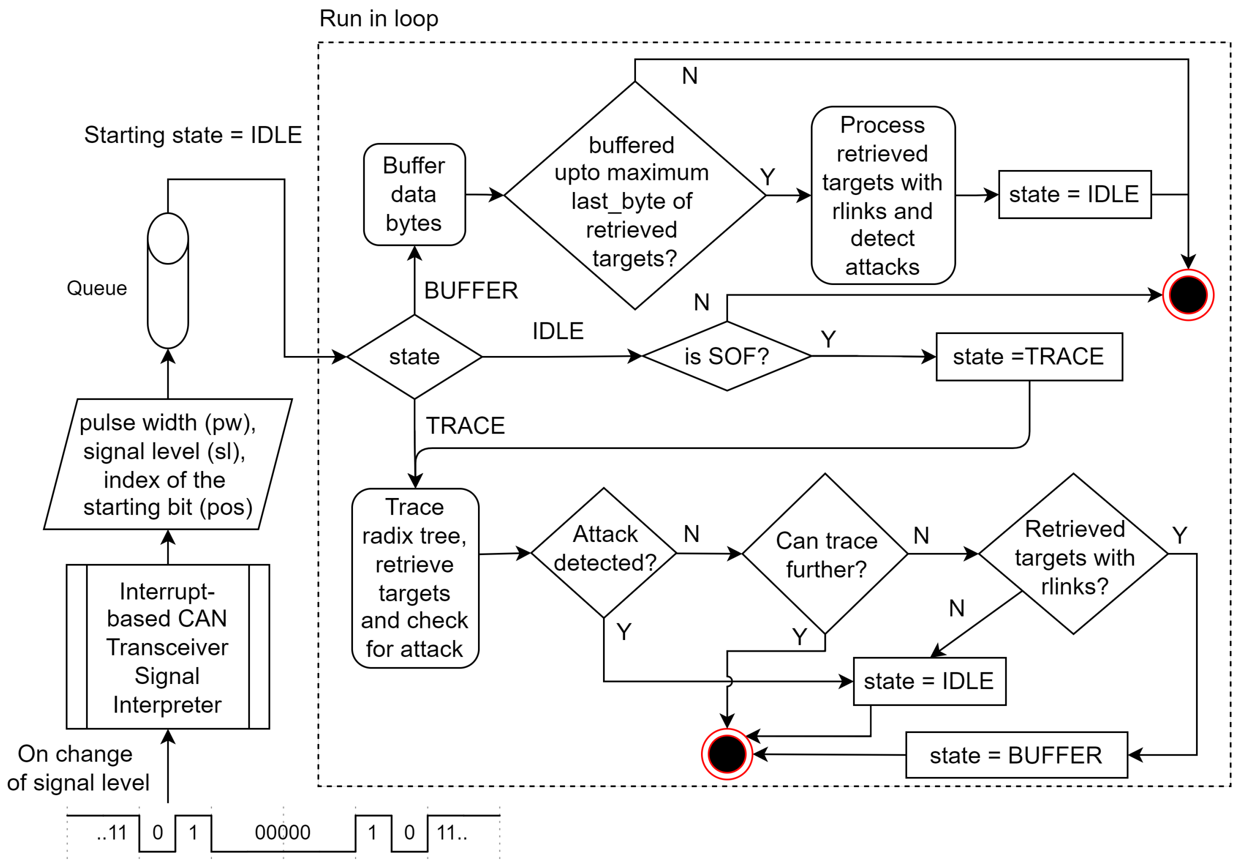

Run in loop
Starting state = IDLE
Queue
Buffer data bytes
buffered upto maximum last_byte of retrieved targets?
N
Y
Process retrieved targets with rlinks and detect attacks
state = IDLE
BUFFER
state
IDLE
N
is SOF?
Y
state =TRACE
TRACE
pulse width (pw), signal level (sl), index of the starting bit (pos)
Trace radix tree, retrieve targets and check for attack
Attack detected?
N
Y
Can trace further?
N
Y
Retrieved targets with rlinks?
Y
N
state = IDLE
state = BUFFER
Interrupt-based CAN Transceiver Signal Interpreter
On change of signal level
..11 0 1 00000 1 0 11..

**Figure 5.6:** Rule Enforcement in Real-time

at a time from the input queue. Using a queue ensures atomicity of the ongoing attack detection process i.e. it finishes execution before acting on the next item in the queue.

Our system maintains the program state in a variable called `state`, which is initialized to `IDLE`. In the `IDLE` state, the system waits for a new message by checking if the received `pos` is 1. If so, it sets the `state` of the system to `TRACE` and starts tracing the radix tree.

The tracing procedure is implemented using the `trace_radix_tree` function described in Algorithm 2. The function takes as input three variables namely, `pw`, `sl`, and `pos` which have already been described in the previous paragraph. It accesses a global variable `node` that points to the current node in the radix tree being processed. At SOF, `node` is set to the root node of the tree. It also computes two local variables `endpos` and `node_endpos` using `pos` and a global variable `nodepos`. `pos` and `endpos` denote the start and end positions of the received bits (pulse) in the CAN frame. `nodepos` and `node_endpos` denote the start and end positions of the bits (in the

**Algorithm 2:** trace_radix_tree(pw, sl, pos)

```
Global variable: nodepos, node, val, targets, attack_detected
if pos = 1 then
    node ← root of the radix tree
    nodepos ← 5
    targets ← ∅
    attack_detected ← False
end
endpos ← pos + pw -1
node_endpos ← nodepos + node.length -1
overlap ← min(endpos, node_endpos) - max(pos,nodepos) + 1
if overlap > 0 then
    val ← val << overlap | ((2^overlap - 1)*sl)
end
if endpos ≤ node_endpos then
    return
end
if node.value = val then
    node ← NULL
    return
end
if node.target ≠ NULL then
    if check_for_attack_during_ID(node.target) then // refer to Alg.  3
        attack_detected ← True
    end
    if |node.target.rlinks| > 0 then
        targets ← targets ∪ node.target
    end
end
if sl = 0 then
    node ← node.0child
end
else
    node ← node.1child
end
nodepos ← nodepos + node.length
val ← 0
if node ≠ NULL then
    trace_radix_tree(pw,sl,pos);
end
return
```

CAN frame) that must be used for comparison with the `value` in `node`. Because the first four bits of the CAN ID are not included in generating the arbitration bits string (refer to the documentation of the preprocessing phase), at SOF we set `nodepos` to 5. `overlap`, calculated as one more than the difference between the minimum of `node_endpos` and `endpos` and the maximum of `nodepos` and `pos`, denotes the number of bits in the pulse that can be used for comparison with the value in `node`. If `overlap` is greater than 0, we accumulate the overlapping bits in a global variable `val` that is set to 0 every time `node` changes. For accumulation, we left shift `val` and bitwise OR the bits in the pulse with it. If `endpos` is greater than `nodepos`, it indicates that the incoming pulse carries bits that must be compared with the next node if there exists one. To that end, we check if the accumulated bits match with the `value` in the current node. If not we set `node` to NULL and return. Otherwise, we proceed to check for an attack. We check if the node has a `Target` attached to it and if so, we check for an attack using `np_canrules` of the `Target`. If the `Target` has at least one `RLink` associated with it, we accumulate it in a buffer `targets` that is reset to empty at every SOF. At this point, we perform activities necessary for node changeover in the radix tree. We select the next node depending on the signal level `sl`, reset `val` to 0 and increment `nodepos` by the current node's `length` such that it points to the starting position of the bits (in the CAN frame) that must be used for comparison with the `value` in the schanged `node`.

In Algorithm 3, for every `CANRule` pointed to by `target.np_canrules` we do the following. We first increment the `cth` counter of the `CANRule` object with the assumption that the parent rule is triggered. If the `interval` of the `CANRule` object is a non-zero value we obtain the current time in milliseconds and verify if the difference between the current time and the last time of transmission (stored in `last_t`) is greater than `interval` specified in the `CANRule` object. This indicates that the parent `IRule` is not triggered and, as such, we reset the `cth` counter of the `CANRule` object. In this way, we keep a count of the consecutive number of triggers of the parent rule. In line 8, we set the `last_t` attribute of the `CANRule` object to the latest obtained time. If the `cth` equals or exceeds the `threshold` of the `CANRule` object, it implies that this and the

**Algorithm 3:** check_for_attack_during_ID(target)

```
forall $canrule \in target.np\_canrules$ do
    canrule.cth ++
    if $canrule.interval > 0$ then
        time $\leftarrow$ get milliseconds from start
        if $time - canrule.last\_t > canrule.interval$ then
            canrule.cth $\leftarrow$ 0
        end
        canrule.last_t $\leftarrow$ time
    end
    if $canrule.cth \geq canrule.threshold$ then
        act(canrule.action)
        return True
    end
end
return False
```

previous threshold - 1 messages of interest have all triggered the parent rule. Therefore, we act upon the message by calling the act function. We do not implement act in this work but keep it abstract. We pass the action strings to it, so it can perform the necessary actions based on them.

If an attack is detected after the call to the trace_radix_tree function, the program goes back to the IDLE state and waits for a new message. Otherwise, the tracing procedure continues until no more nodes are available to trace, i.e. the variable node is NULL. In this case, targets has a non-zero size. This implies that there is at least one element in targets whose corresponding record contains rules that include parameters. As such, the system moves into the BUFFER state where it buffers data bytes in a byte array until the k$^{th}$ byte has been received, where $k = max(\{t.\text{last\_byte} | t \in \text{targets}\})$ At that point, it executes Algorithm 4 and processes the retrieved targets with at least one rlinks.

In Algorithm 4, for every rlink in target.rlinks for every target in targets we do the following. If the relation to be processed is *context* we iterate through the FieldFilter objects using rlink.indexes. For each FieldFilter object f, we check if the parameter value obtained from function get_value lies within f.value. We compare the result of the check with the f.prevm and if they are different we increment or decrement the value

**Algorithm 4:** process_retrieved_targets(targets, buffered data bytes)

```
forall target ∈ targets do
    forall rlink ∈ target.rlinks do
        if rlink.relation = moi then
            m ← True
            for ff ∈ rlink.canrule.moi do
                m ← m AND (ff.value[0] ≤
                get_value(buffered data bytes, ff) ≤ ff.value[1]);
            end
            if m = True then
                rlink.canrule.cth ++
                if rlink.canrule.interval > 0 then
                    time ← get milliseconds from start
                    if time - rlink.canrule.last_t >
                    rlink.canrule.interval then
                        rlink.canrule.cth ← 0;
                    end
                    rlink.canrule.last_t ← time
                end
                if rlink.canrule.ncc < rlink.canrule.max_ncc then
                    rlink.canrule.cth ← 0;
                end
                if rlink.canrule.cth ≥ rlink.canrule.threshold then
                    act(rlink.canrule.action)
                    return
                end
            end
        end
        else
            for i in rlink.indexes do
                m ← (rlink.canrule.context[i].value[0] ≤
                get_value(buffered data bytes,
                ff) ≤ rlink.canrule.context[i].value[1])
                if m ≠ rlink.canrule.context[i].prevm then
                    if m = True then
                        rlink.canrule.ncc ++;
                    end
                    else
                        rlink.canrule.ncc –;
                    end
                    rlink.canrule.context[i].prevm ← m
                end
            end
        end
    end
end
```

of rlink.canrule.ncc depending on the truth value of the check. The comparison with f.prevm ensures that rlink.canrule.ncc is updated only when the parameter value goes outside or comes inside the range specified in f.value. In this way, rlink.canrule.ncc keeps count of the active parameters in the context specified by the parent rule of rlink.canrule. If the relation to be processed is *moi*, we first check if the message being transmitted is of interest to the parent of rlink.canrule. This is done by extracting the parameter values from the get_value function and checking if they lie within the value of the FieldFilters in rlink.canrule.moi. If they do, we first increment the cth counter of the CANRule object with the assumption that the parent rule is triggered. Then, we proceed to check the parent of rlink.canrule is not triggered. This is done in lines 10 through 19. In lines 10 to 15, we check if the parent of rlink.canrule is of type IRule (by checking if rlink.canrule.interval is greater than 0) and if so, whether the message of interest is transmitted at an interval greater than rlink.canrule.interval. In lines 17 to 19, we check if the parent of rlink.canrule is of type CRule and if so, whether the context specified by it is not active. In either case, the rule is not triggered and hence we reset target.cth to 0, thereby nullifying the effect of line 9. In this way, we keep a count of the consecutive number of triggers of the parent rule. If the cth equals or exceeds the threshold of the CANRule object, it implies that this and the previous threshold - 1 messages of interest have all triggered the parent rule. Therefore, we act upon the message by calling the act function. Recall from before that act is abstract and needs to be implemented using the passed rlink.canrule.action strings.

In get_value() we first extract the parameter values from the CAN data field using t_bytes, t_bits, t_masks and first_length from the supplied instance of FieldFilter. If the elements of t_bytes are same, i.e. R = S in the SAE J1939 parameter placement, we apply t_masks[0] to the t_bytes[0]$^{th}$ data byte and return it after right shifting by t_bits[0] -1. If the elements of t_bytes are not the same, i.e. R $\neq$ S in the SAE J1939 parameter placement, we first apply t_masks[1] to the t_bytes[1]$^{th}$ (i.e. S$^{th}$) data byte and assign it to a temporary variable after right shifting by t_bits[1] -1. We then append (using left shift and bitwise OR like line 12 of Algorithm 2) the bits of bytes S -1 through R+1 to the temporary variable in that order. Finally, we apply `t_masks`[0] to the `t_bytes`[0]$^{th}$ (i.e. R$^{th}$) data byte and append it to the temporary variable after right shifting it by `t_bits`[0] -1 and left shifting the temporary variable by `first_length`. This approach is followed to regard the reverse transmission byte order obeyed by SAE J1939 parameters.

## 5.3 Performance Analysis

The goal of this section is to analyze the performance of the rule enforcement procedure (described in Figure 5.6) in terms of the number of rules that can be enforced in real-time i.e. as messages are being transmitted on the CAN bus. Our system enforces rules in two phases. Rules without `PFilters` are enforced in Algorithm 3 when the program is in the `TRACE` state. Rules with `PFilters` are enforced during the processing of the retrieved targets in Algorithm 4. All of the other steps in the rule enforcement procedure are executed in constant time, independent of the characteristics of the rule database.

In this section, we first describe the setup in which we conduct the experiments for our analysis. Next, we describe the performance of the enforcement procedure in the aforementioned two phases.

### 5.3.1 Experiment Platform

For the purpose of analysis, we choose two embedded development boards designed for automotive application development: Teensy 3.6 and Teensy 4.1. Teensy 3.6 hosts a 180MHz ARM Cortex-M4 processor and 256 KB of RAM while Teensy 4.1 hosts a 600 MHz ARM Cortex-M7 processor and 1024 KB of RAM. Both boards support Arduino-based programming and can be interfaced with CAN transceivers. For the purpose of testing, we implement the preprocessing phase in Python on a desktop computer and the rule enforcement procedure (described in Figure 5.6) in C on the development boards. Low-level data structures generated by the Python pre-processing code are transferred to the development boards using a secure digital (SD) card.

All timing measurements provided in this section are related to the rule enforcement procedure and are made using the `elapsedMicros` [70] library function provided by the Teensy software development kit. `elapsedMicros` measures the elapsed microseconds between two points of execution in code.

### 5.3.2 Enforcing Rules Without `PFilters`

In Algorithm 3, we scan the `CANRules` linked through `np_canrules` of the retrieved `Target` objects. The cardinality of `np_canrules` is restricted to 2 because of the third rule constraint. As such, Algorithm 3 is effectively constant in time. Algorithm 2, that calls Algorithm 3, is also constant in time as it does not involve any loops.

On both the experimental development boards we noticed that all steps in the `TRACE` state of the program finish execution in less than 2 microseconds when the longest execution path of Algorithm 2 is executed i.e. the one that ends at line 39. This included the execution of Algorithm 3 with a `Target` input having 2 `np_rules` and four recursive calls to Algorithm 2. Note that four is the worst case number of times Algorithm 2 can call itself at line 38. This is because the minimum `length` of a node can be one and the maximum number of bits in a CAN pulse can be five due to bit stuffing. The execution time is less than the width of a single bit (2 microseconds) for the highest CAN baud rate (500 kbps) supported by SAE J1939. To that end, we posit that `trace_tradix_tree` is executed in real-time on our experimental boards, irrespective of the number of rules without `PFilters`. In other words, if executed on platforms similar to the experimental development boards, our system can support any number of rules without `PFilters` as long as they obey the rule constraints enumerated at the end of section 5.1.2.

### 5.3.3 Enforcing Rules With `PFilters`

Rules with `PFilters` are enforced in Algorithm 4: each `rlink` in line 2 pointing to an `CANRule`, one of which is created per rule in the system. Therefore, the number of rules that can be enforced by Algorithm 4 decides the number of executions of line 2. That is to say that the runtime of the algorithm is directly proportional to the number of rules that can be enforced by it. As such, we can estimate the maximum number of rules by executing the algorithm with increasing counts of `target.rlinks` and stopping when it finishes after the end of the CAN frame. The execution of the algorithm, however, also depends on other attributes of the input `Target` objects like the value of `rlink.relation` in line 3 and the cardinality of `rlink.canrule.moi` or `rlink.indexes` in lines 5 and 27. The values for these attributes are dependent on the characteristics of the input rule database, but, because we do not have access to real-world databases, we estimate them in the worst case and conduct the experiments. To that end, our experiments estimate the number of rules with `PFilter`s that can be processed in the worst case as a message is being transmitted.

**Experiment Configuration**

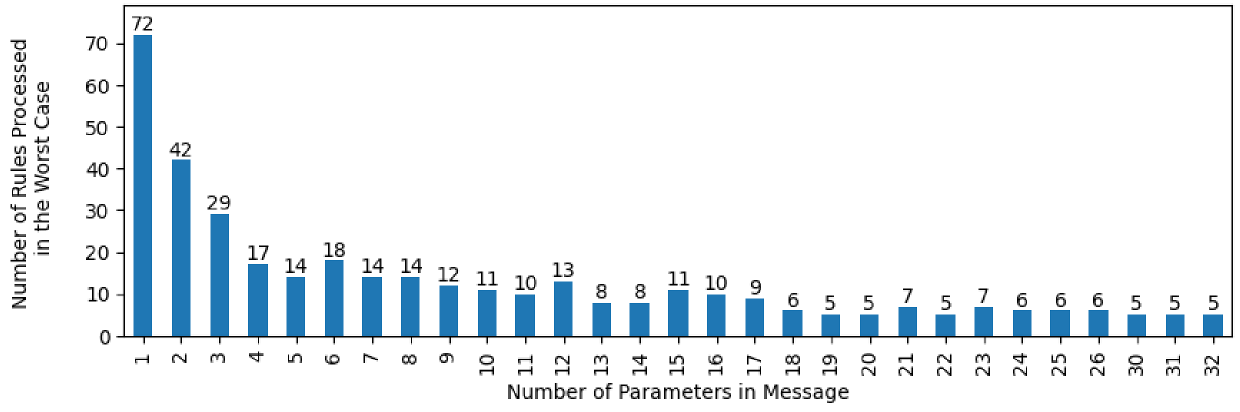
We create two `Target` objects `t1` and `t2`, each containing $n$ `rlinks`. In the first target object `t1`, we set `rlink.relation` = *moi* and assign an `CANRule` object to `rlink.canrule`, $\forall$ `rlink` $\in$ `t1.rlinks`. In the second target object `t2`, we set `rlink.relation` = *context* and assign an `CANRule` object to `rlink.canrule`, $\forall$ `rlink` $\in$ `t2.rlinks`. Next, we create a set of $p$ `FieldFilter` objects and assign them to `rlink.canrule.moi`. Similarly, we create a set of $p$ `FieldFilter` objects and assign them to `rlink.canrule.context`. $p$, here, is the number of parameters that can be carried by a message and is, therefore, the worst-case number of executions of lines 5-7 and 27-38 per execution of line 2. Also, in `t2` we set `rlink.indexes` = [0 .. p-1] to ensure that all elements of `rlink.canrule.context` are processed in lines 28-37. For each `FieldFilter` created above, we set `value` to 0, and at runtime, we pass all 0 data bytes. This ensures $m$ is true at lines 8 and 29 and, as such, lines 9-23 and 30-36 are executed at experiment time. For each `CANRule` object `rlink.canrule`, $\forall$ `rlink` $\in$ `t1.rlinks` and $\in$ `t2.rlinks`, we set `rlink.canrule.max_ncc` to 0, `rlink.canrule.threshold` to 1 and `rlink.canrule.interval` to 1. Also, at runtime, we enforce a delay of 2 milliseconds before making a call to Algorithm 4. This ensures that the process always executes line 13 but not lines 18 and 21 i.e. it executes the path in lines 3-25 with the most instructions and does not return at line 22. For each `FieldFilter` used in the experimentation, we set `t_bits`, `t_masks`, and first_length to 0 as these attributes do not affect the runtime of the algorithm. However, we set t_bytes to [0,$l$], where $l = max(\{S - R|(R, S$ are the starting and ending message data bytes occupied by parameter $pm) \wedge (pm$ belongs to a group with $p$ parameters $)\})$. We obtain the values for $R$ and $S$ from the SAE-J1939 digital annex. Because get_values processes each byte from t_bytes[0] to t_bytes[1], this ensures that it consumes the worst-case execution time when experimenting with a specific value of $p$.

**Experiment Execution**

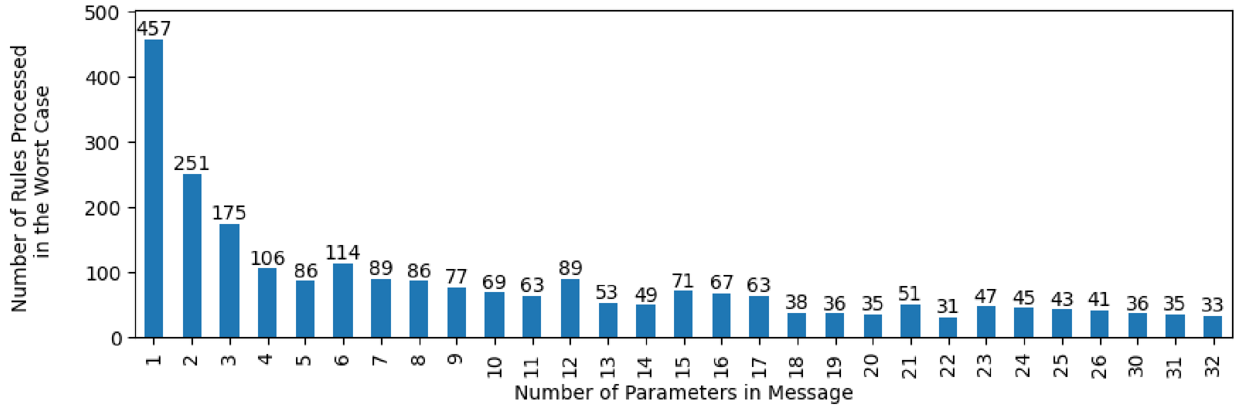
As per the specifications, there can be a maximum of 32 parameters carried in an SAE J1939 message. As such, we run 32 experiments on each development board by incrementing $p$ from 1 to 32. For each run, we increment $n$ and call Algorithm 4 two times: once with the input targets set to [t1] and once with the input targets set to [t2]. In the worst case Algorithm 4 needs to be executed after all message data bytes have been received. This allows a maximum space of 25 CAN bits that can be utilized for the completion of Algorithm 4. As such, we stop an experiment when the runtime of the algorithm exceeds 96 microseconds, i.e. the time taken for the transmission of the first 24 bits of the CAN frame after the data field on a 250 kbps bus. We do not consider the 25th bits as it signifies the end of the frame and no action can be taken after that. We record the value of $n$ and plot it against $p$ in Figure 5.7.

**Observations**

Figure 5.7 captures the number of rules with PFilters $n$ that can be processed in the worst case during the transmission of a message that carries $p$ parameters. Because the maximum number of parameters carried in a message can be 32, theoretically we can process 5 rules with PFilters on Teensy 3.6 and 33 rules with PFilters on Teensy 4.1 in the worst case. Albeit, on real-world networks the maximum number of parameters carried in a message can be much less than 32. On the Kenworth T270 research truck, we found a maximum of 17 parameters transmitted in a message, and on the Kenworth T660 research truck this number was 12. In such a case, our system can support more than 60 rules in the worst case on platforms similar to the Teensy 4.1



**(a)** Results on Teensy 3.6



**(b)** Results on Teensy 4.1

**Figure 5.7:** Number of Rules Processed in the Worst Case Before Message Transmission Ends

development board. Finally, even though messages can carry more than one parameter, a quick look at the example rules in Table 5.1 indicate that, in a real-world setting, all parameters in a message may not be used in rule processing. As a matter of fact, for the rules defined in Table 5.1 only one parameter of a message will be used by Algorithm 4. In this case, our system can support a significant number of rules with `PFilters` in the worst case: more than 70 on a Teensy 3.6 and more than 450 on a Teensy 4.1

## 5.4 Real-World Demonstration and Discussion

In this section, we demonstrate the ability of our system to enforce user-provided rules on real-world network data. For this, we utilize the SAE J1939 attack detection rules specified in Table 5.1 and network data captured from CAN networks within the research trucks.

### 5.4.1 Implementation

To test the rule enforcement procedure, we designed a wrapper program that reads a CAN frame from a log file, records the timestamp to be returned to lines 4 and 11 of algorithms 3 and 4, generates the corresponding NRZ pulse stream, calls the rule enforcement procedure for each pulse in the stream and records if the `act` function is called from either algorithm. At the end of execution, it outputs a log file in which each line contains ID and data of the CAN frame as well as a boolean value indicating if the `act` function was called. Contents of this file are used to generate the plots in Figure 5.8.

The wrapper program and the rule enforcement procedure are written in C, compiled into x86, and executed on a 32-bit Windows platform. This is done because the number of rules in Table 5.1 is less than the minimum number Teensy 4.1 can support (as shown in the previous section) and timing is not a concern as such. The rules are provided as Python objects and compiled into low-level objects that are stored on disk as a text file. The simulated rule enforcement procedure reads from this file at startup, loads the low-level objects into memory, and processes the input pulses thereafter.

### 5.4.2 Data Collection

We captured normal driving logs from the Kenworth T270 research truck while driving on an airstrip. We also used the normal driving logs captured from the Kenworth T660 research truck for our experimentation. For the attack data, we performed 8 different attacks on the Kenworth T270 research truck. Each attack was conducted so it can be detected by a rule from Table 5.1. The attacks are described below.

**Attack 1** We conducted an engine control attack spoofing the source address of the body controller unit that was actually not present on the CAN network. The PGN, DA, and SA of the attack messages were set to 0, 0, and 33 respectively. This attack should be detected by the first rule in Table 5.1.

**Attack 2** We conducted the throttle jam attack by sending a -125 % torque request in torque and speed control messages to the engine controller. The PGN, DA, and SA of the attack messages were set to 0, 0, and 13 respectively. This attack should be detected by the second rule in Table 5.1.

**Attack 3** We conducted the request overload attack by sending request messages at 3-millisecond intervals to the engine controller. The PGN, DA, and SA of the attack messages were set to 59904, 0, and 249 respectively. This attack should be detected by the third rule in Table 5.1.

**Attack 4** We conducted the connection exhaustion attack (as described in section 3.3) by sending CTS messages within 1250 ms to keep a connection alive. The PGN, DA, and SA of the attack messages were set to 60416, 0, and 249 respectively. This attack should be detected by the fourth rule in Table 5.1.

**Attack 5** We conducted the network overload attack by sending the highest priority messages at intervals of 3 milliseconds. The PGN, DA, and SA of the attack messages were set to 0, 0, and 0 respectively. This attack should be detected by the fifth rule in Table 5.1.

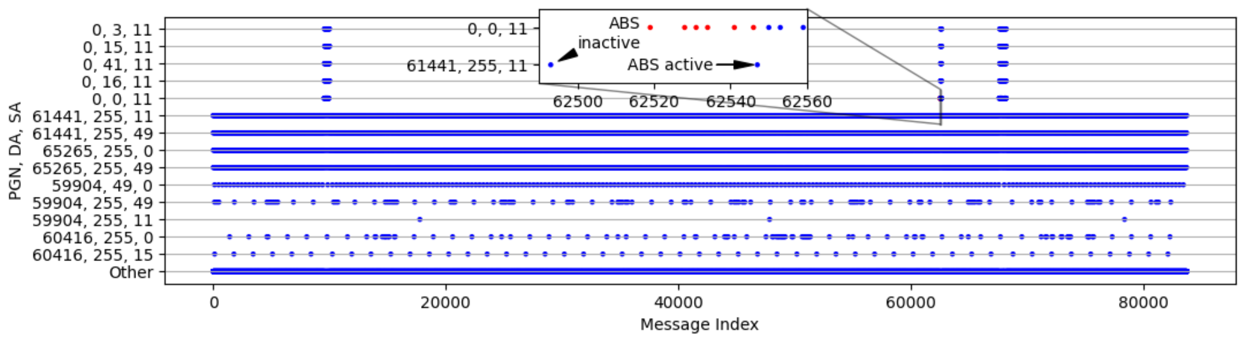
**Attack 6** We conducted an address claim attack by sending a message with PGN $0EA00_{16}$ and claiming the address of the engine controller when the vehicle was at speeds greater than 5 km/h. The PGN, DA, and SA of the attack message were set to 60928, 255, and 0 respectively. This attack should be detected by the sixth rule in Table 5.1.

**Attack 7** We conducted an engine control attack by spoofing the source address of the ABS unit which was present on the CAN network. The PGN, DA, and SA of the attack messages were set to 0, 0, and 11 respectively. This attack should be detected by the seventh rule in Table 5.1.

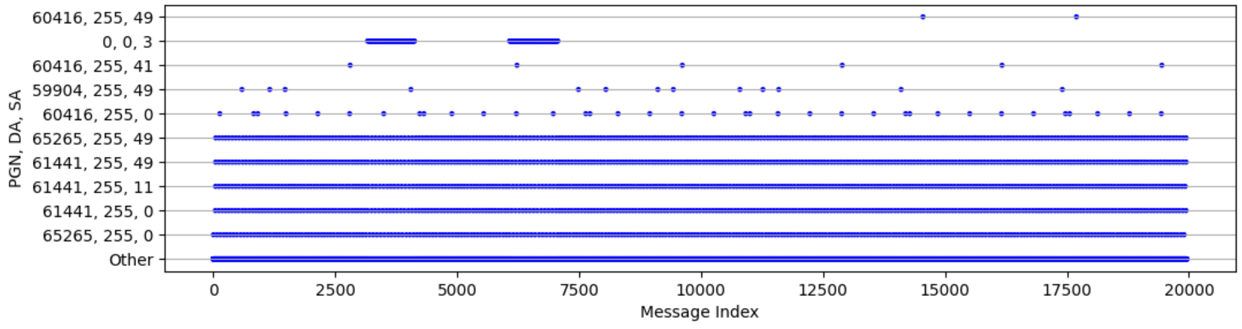
**Attack 8** We conducted the retarder jam attack by sending a 0 % torque request in a message with PGN 0 to the engine retarder when the vehicle speed was less than 30km/h. The PGN, DA, and SA of the attack messages were set to 0, 15, and 3 respectively. This attack should be detected by the eighth rule in Table 5.1.

### 5.4.3 Observations and Discussion

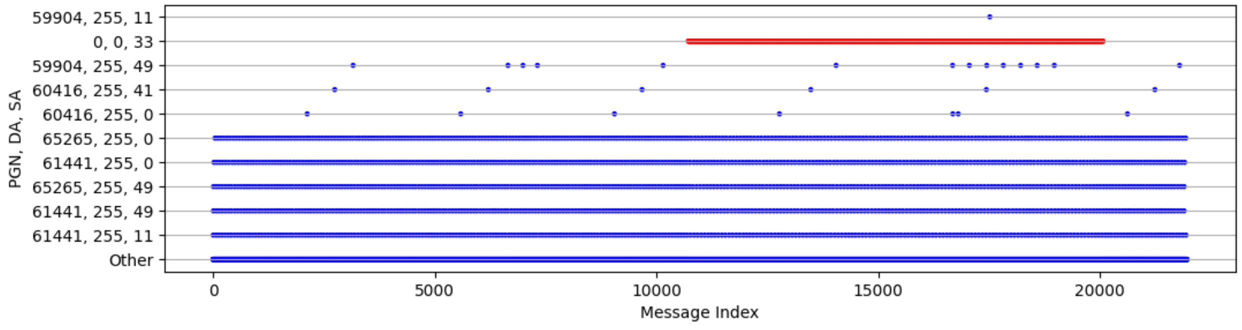
The results of our experiments are shown in Figure 5.8. In all of the cases depicted therein, attack messages are flagged. However, false alarms are observed when conducting the experiment on normal traffic collected from the Kenworth T660 research truck, the results for which are shown in Figure 5.8c. Ideally, red dots should not be present in this plot, however, they are present in one case that is zoomed in on the plot. This is because ABS is not signaled to be active when the first few torque and speed and control messages (PGN = 0) to the engine controller (DA = 0) are transmitted by the ABS unit (SA = 11). If such a situation occurs and is discovered before deploying a rule like the one in the seventh row of Table 5.1, we recommend a two-step solution. First, the rule should be modified to increase the `threshold`. This will wait out a certain number of torque and speed control messages before flagging them as malicious. As an example, on the Kenworth T660 research truck the status of the "ABS active" parameter is transmitted periodically at 100 ms intervals and torque and speed control messages are transmitted at intervals of 10 milliseconds. In this case, the rule deployer may wish to wait out 10 torque and speed control messages before verifying if ABS has been activated and acting accordingly on the next set of torque and speed control messages. This is simply because, in the worst case, 100/10 = 10 torque and speed control messages will be transmitted before ABS is signaled as active. Albeit, suppose that an attacker conducts the engine control attack. Clearly, the first 10 messages will be passed before the system begins acting on them. This will provide the attacker with a maximum of 100 milliseconds of engine control. This is a very low time of control to cause any effect on the behavior of the



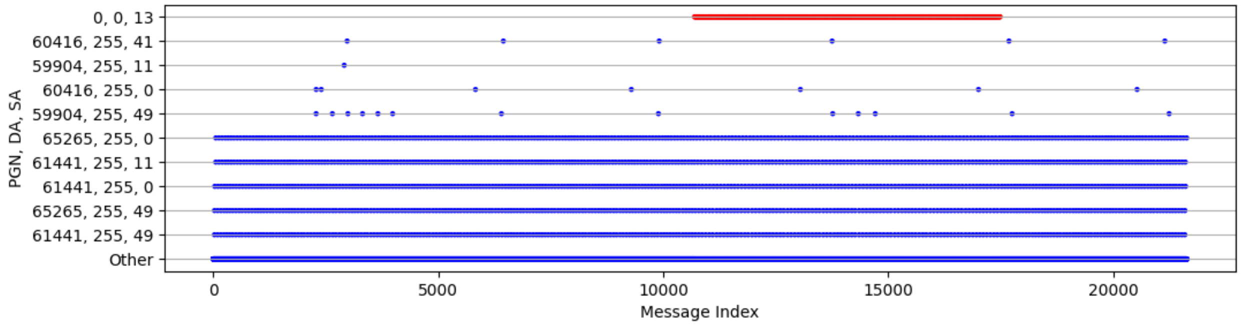
**(a)** Detection Results on Normal Data Collected from the Kenworth T600 Research Truck



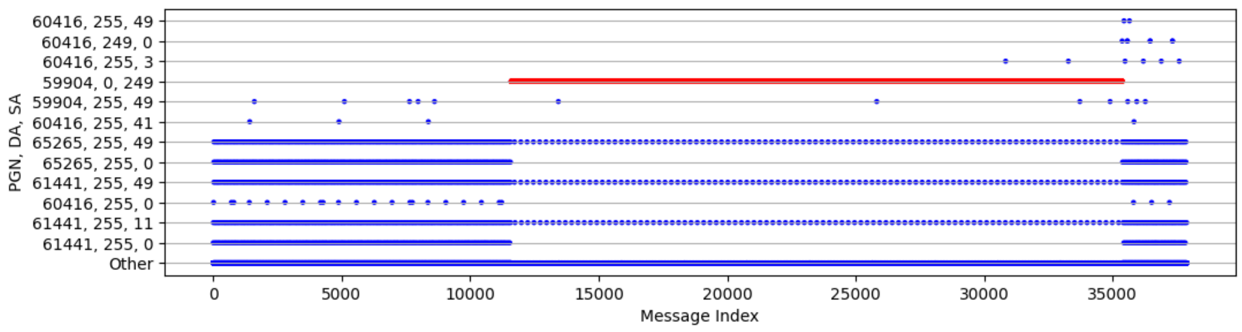
**(b)** Detection Results on Normal Data Collected from the Kenworth T270 Research Truck



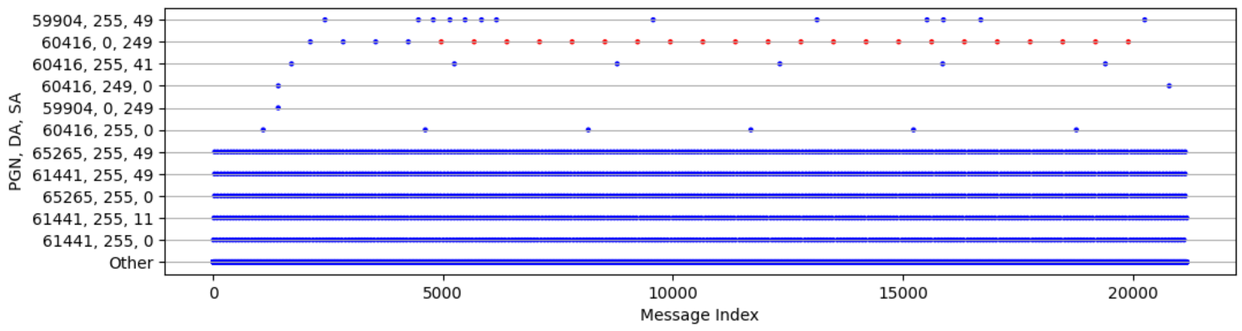
**(c)** Detection Results for Attack 1



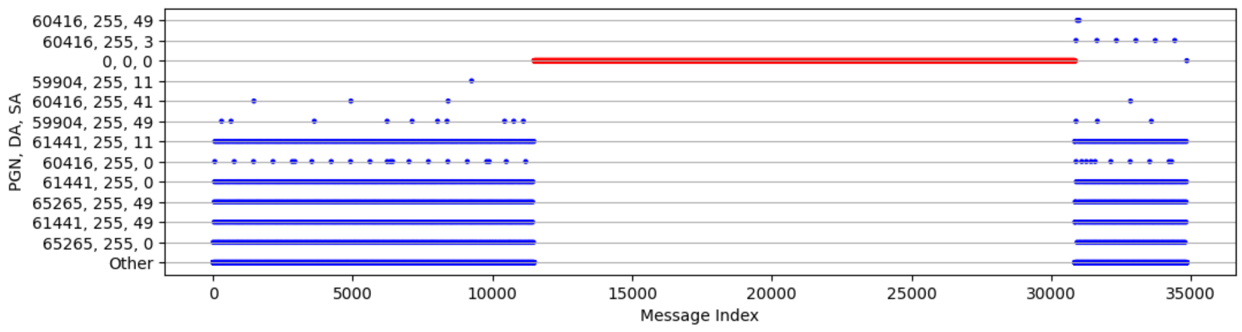
**(d)** Detection Results for Attack 2



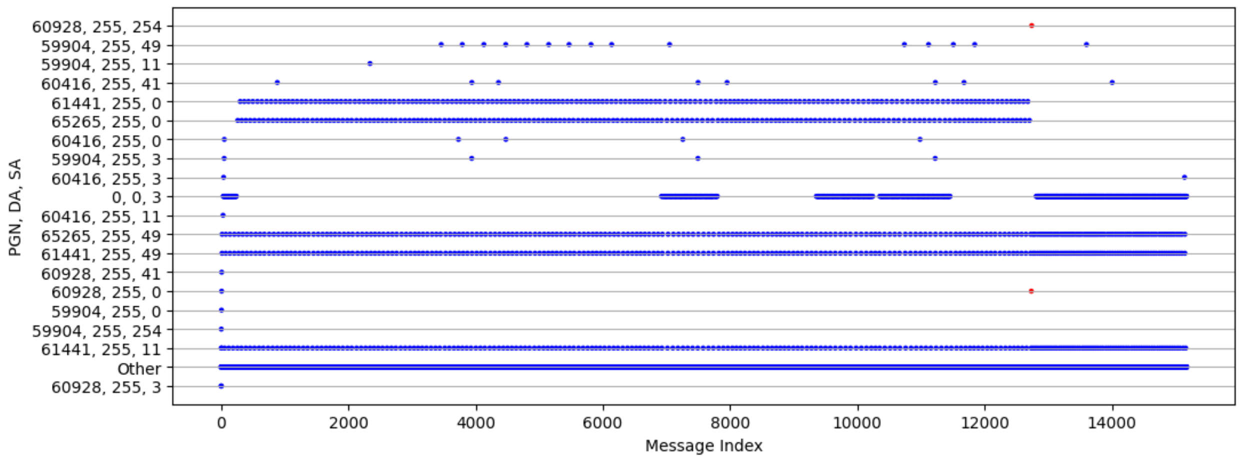
**(e)** Detection Results for Attack 3



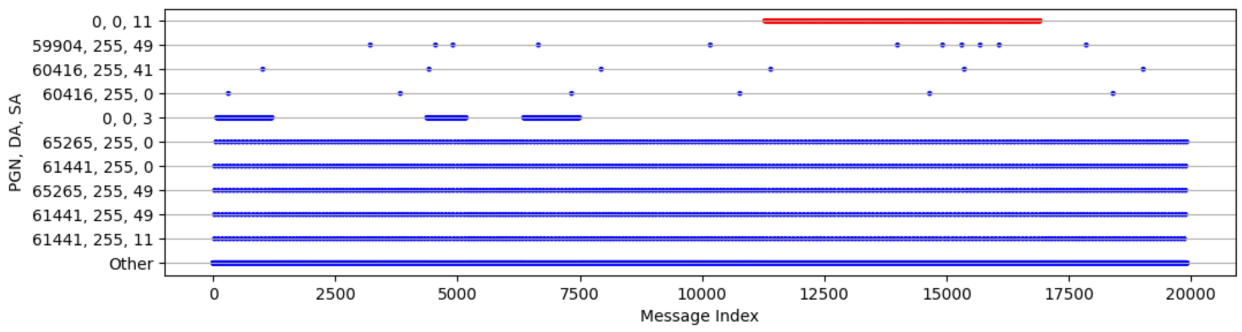
**(f)** Detection Results for Attack 4



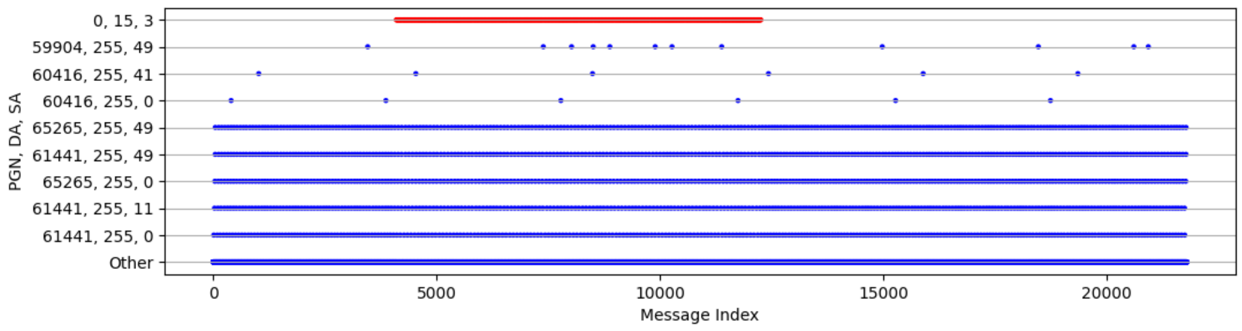
**(g)** Detection Results for Attack 5



**(h)** Detection Results for Attack 6



**(i)** Detection Results for Attack 7



**(j)** Detection results for Attack 8

**Figure 5.8:** Detection Results on Real World Data; Legend: Red dots are flagged messages, Blue dots are passed messages

vehicle. However, this will increase the `cth` counter of the corresponding `CANRule` to above its `threshold` and it will not be reset to 0 until the context is deactivated i.e. ABS is signaled to be active. Any legitimate engine control attempt by the ABS during this period will be flagged, even if the `threshold` is set to 10. To counter this scenario, we recommend that, in addition to increasing the `threshold`, the rule action be set to vehicle restart such that the driver is alerted and the situation is handled gracefully upon detecting the first illegitimate engine control attempt.

Another aspect of context-based detection is the assurance of the legitimacy of the context information available on the network. Assume that the attacker transmits a message that indicates that ABS is active immediately before sending a torque and speed control message to the engine controller. In this case, they will successfully falsify the context and conduct the attack even if rule 7 in Table 5.1 is deployed. To counter this scenario, we recommend using our solution with a periodicity-based intrusion detection and prevention system as described in section 2.4. In this type of system, a message can be flagged and even disrupted [40] if it is transmitted at an interval that is different from its usual transmission period. If context information is obtained from periodically transmitted messages, any attempt to spoof them will lead to a violation of periodicity, and a periodicity-based intrusion detection and prevention system will immediately act on it. Albeit, a context carrier CAN frame can still fail CRC check and or be negatively acknowledged by the network. Our solution does not account for such cases but assumes that an attacker-injected message does not occur between a failed transmission of a context carrier and its correct retransmission.

## 5.5 Summary

In this chapter, we answer the research question: *can a rule-based system be designed to detect threatening SAE J1939 messages as they are being transmitted and mitigate their effect based on features other than message content only?* Our research reveals that there can be three features, other than message content that can be leveraged to create a rule-based system to detect malicious messages on an SAE J1939 network: inter-message transmission interval, the context of transmission, and count of suspicious transmission. Our research also shows such a system can theoretically enforce an unlimited number of rules in real time if they are based on the PGN, DA, and SA of the message only. Otherwise, if the rules utilize information carried in parameters, then, at least 30 of them can be enforced in real time in the worst case if deployed on commodity hardware such as a Teensy 4.1 that hosts a 600 MHz ARM-Cortex M7 processor and 1024 Kb of RAM.

# 6 Conclusion and Future Work

## 6.1 Conclusion

In this dissertation, we investigated the cyber security of SAE J1939 networks in medium and heavy duty. Our preliminary research showed that there are gaps in both offensive and defensive security research in our domain of concentration. As such, we investigated both areas of research and tried to fill the gaps. For the offensive side, we investigated the security aspects of the SAE J1939 data-link layer specifications. Our research revealed three denial-of-service attacks on the SAE J1939 data-link layer specifications. We observed noticeable impacts on network communication for all of these attacks. We also observed a noticeable impact on a research truck upon execution of two of these attacks. From the defensive side of research, we devised two solutions: a behavioral intrusion detection system that does not require offline training and a rule-based intrusion detection prevention system that can detect malicious messages in real time based on features other than message content. The first solution demonstrated that network behavior can be modeled through a directed graph and features of the graph can be used by a time series forecasting technique to detect significant deviations and flag them as intrusions. The second solution demonstrated that rules can be created from features other than the content of the message namely, inter-message transmission interval, the context of transmission, and the number of suspicious transmissions. The solution also demonstrated that, depending on the platform of deployment, a certain number of rules can be processed in real-time.

## 6.2 Future Work

In this section, we mention some future directions of work that can emanate from the research done on this dissertation topic.

**Extending to Other Areas of Interest** Although this dissertation utilizes specifics of SAE J1939, there may be a possibility to extend its contributions to other areas of interest. The offensive research presented in chapter 3 exploits protocol specifications made in the SAE J1939 data-link layer document SAE J1939/21. This raises the question, can similar exploits be developed for other documents in the SAE J1939 standards as well as other standards followed by medium and heavy-duty vehicles? The defensive research presented in chapters 4 and 5 utilizes high-level information presented in messages. As such, if messages are transmitted unencrypted on other types of networks, it can be investigated if the intrusion detection techniques established in this dissertation can be applied to those types of networks.

**Remote Testbenches and Generation of Research Data** Throughout this dissertation, several experiments have been demonstrated: some on homegrown test benches while others on actual trucks. The laboratory environment in which these experiments were carried out included the necessary facilities. Such facilities may not be available to many research groups. As such, one area of future research can be to outsource the equipment utilized in this dissertation and allow remote access to them so that further experimentation can be carried out. A vital requirement for this testbench will be reconfigurability so that network settings within different trucks can be emulated. Research data generated from such a testbench can be disseminated for enhancement of research, albeit while abiding by intellectual property restrictions. As a matter of fact, newer attacks can be discovered on this testbench and their signatures can be used to form inputs to the rule-based intrusion detection and prevention system presented in chapter 5 of this dissertation.

**Hybrid Detection and Prevention Systems** This dissertation presents two security systems, the behavioral anomaly-based, and rule-based detection systems. In the future, a hybrid security system can be designed from these two and this system can harvest the benefits of both worlds. Using behavioral anomaly detection, the hybrid security system can detect unknown attacks while using rule-based intrusion detection it can detect known attacks. In addition, the concept of context (established in chapter 5) can be leveraged to reduce false alarms raised by the behavioral anomaly detection system. As an example, if false alarms are raised at the time of hard braking events, it can be detected if the anti-lock braking is active and alarms can be suppressed.

# Bibliography

[1] Bureau of Transportation Statistics. Number of U.S. Aircraft, Vehicles, Vessels, and Other Conveyances.

[2] Society of Automotive Engineers. SAE J1939 Standards Collection.

[3] International Organization for Standardization. Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. Standard ISO/IEC 7498-1:1994.

[4] Stephen Stachowski, Russ Bielawski, and André Weimerskirch. Cybersecurity Research Considerations for Heavy Vehicles. Technical report, University of Michigan, Ann Arbor, Transportation Research Institute, 2019.

[5] Marko Wolf and Robert Lambert. Hacking Trucks – Cybersecurity Risks and Effective Cybersecurity Protection for Heavy-Duty Vehicles. In *Automotive - Safety & Security 2017 - Sicherheit und Zuverlässigkeit für automobile Informationstechnik*, pages 45–60, Stuttgart, Germany, 2017. Gesellschaft für Informatik, Bonn.

[6] Robert Bosch GmbH. CAN Specification. Standard 2.0, Robert Bosch GmbH, 1991.

[7] International Organization for Standardization. ISO - 43.040.15 - Car informatics. On board computer systems.

[8] Cesar Bernardini, Muhammad Rizwan Asghar, and Bruno Crispo. Security and privacy in vehicular communications: Challenges and opportunities. *Vehicular Communications*, 10:13–28, 2017.

[9] Marko Wolf, André Weimerskirch, and Christof Paar. Security in Automotive Bus Systems. In *Proceedings of the Workshop on Embedded Security in Cars*, pages 1–13, Bochum, Germany, 2004. Springer-Verlag.

[10] Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. In *Blackhat USA*, Las Vegas, NV, USA, 2015. Blackhat Press.

[11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *USENIX Security Symposium*, volume 4, pages 447–462, San Francisco, CA, USA, 2011. USENIX Association.

[12] Yelizaveta Burakova, Bill Hass, Leif Millar, and Andre Weimerskirch. Truck Hacking: An Experimental Analysis of the SAE J1939 Standard. In *Proceedings of the 10th USENIX Conference on Offensive Technologies*, pages 211–220, Austin, TX, USA, 2016. USENIX Association.

[13] P. Murvay and B. Groza. Security Shortcomings and Countermeasures for the SAE J1939 Commercial Vehicle Bus Protocol. *IEEE Transactions on Vehicular Technology*, 67(5):4325–4339, 2018.

[14] Upstream. AutoThreat® Intelligence Cyber Incident Repository.

[15] Emad Aliwa, Omer Rana, Charith Perera, and Peter Burnap. Cyberattacks and Countermeasures for In-Vehicle Networks. *ACM Computing Surveys*, 54(1):1–37, 2021.

[16] Hossein Shirazi, Indrakshi Ray, and Charles Anderson. Using Machine Learning to Detect Anomalies in Embedded Networks in Heavy Vehicles. In *Foundations and Practice of Security*, volume 12056, pages 39–55. Springer International Publishing, Cham, 2020.

[17] Hossein Shirazi, William Pickard, Indrakshi Ray, and Haonan Wang. Towards Resiliency of Heavy Vehicles through Compromised Sensor Data Reconstruction. In *Proceedings of the*

*Twelveth ACM Conference on Data and Application Security and Privacy*, pages 276–287, Baltimore MD USA, 2022. ACM.

[18] Karen Scarfone and Peter Mell. Guide to Intrusion Detection and Prevention Systems (IDPS). NIST Special Publication 800-94, Naitonal Institute of Science and Technology, 2007.

[19] C. Young, J. Zambreno, H. Olufowobi, and G. Bloom. Survey of Automotive Controller Area Network Intrusion Detection Systems. *IEEE Design Test*, 36(6):48–55, December 2019.

[20] Ivan Studnia, Eric Alata, Vincent Nicomette, Mohamed Kaâniche, and Youssef Laarouchi. A language-based intrusion detection approach for automotive embedded networks. In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing ( PRDC )*, page 11, Zhangjiajie, Chin, 2015. IEEE.

[21] S. Abbott-McCune and L. A. Shay. Intrusion prevention system of automotive network CAN bus. In *2016 IEEE International Carnahan Conference on Security Technology (ICCST)*, pages 1–8, Orlando, FL, USA, 2016. IEEE.

[22] H. Giannopoulos, A. M. Wyglinski, and J. Chapman. Securing Vehicular Controller Area Networks: An Approach to Active Bus-Level Countermeasures. *IEEE Vehicular Technology Magazine*, 12(4):60–68, 2017.

[23] SAE International. Data Link Layer. Standard J1939-21, SAE International, 2015.

[24] SAE International. Recommended Practice for a Serial Control and Communications Vehicle Network. Standard, 2009.

[25] Society of Automotitve Engineers. J1939DA_202208: J1939 Digital Annex - SAE International.

[26] 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*, pages 1–15, 2022.

[27] Object Management Group. Unified Modeling Language 2.5.1, December 2017.

[28] Sibylle Fröschle and Alexander Stühring. Analyzing the Capabilities of the CAN Attacker. In *Computer Security – ESORICS 2017*, volume 10492, pages 464–482, Oslo, Norway, 2017. Springer International Publishing.

[29] Abdulmalik Humayed, Fengjun Li, Jingqiang Lin, and Bo Luo. CANSentry: Securing CAN-Based Cyber-Physical Systems against Denial and Spoofing Attacks. In *Computer Security – ESORICS 2020*, volume 12308, pages 153–173, Guildford, United Kingdom, 2020. Springer International Publishing.

[30] Charlie Miller and Chris Valasek. Adventures in automotive networks and control units. In *Def Con 21*, pages 15–31. Def Con, 2013.

[31] SAE International. Vehicle Application Layer. Standard J1939-71, SAE International, 2015.

[32] 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 *IEEE Symposium on Security and Privacy*, pages 447–462, Oakland, CA, USA, 2010. IEEE.

[33] Miki E. Verma, Michael D. Iannacone, Robert A. Bridges, Samuel C. Hollifield, Pablo Moriano, Bill Kay, and Frank L. Combs. Addressing the Lack of Comparability & Testing in CAN Intrusion Detection Research: A Comprehensive Guide to CAN IDS Data & Introduction of the ROAD Dataset. *arXiv:2012.14600 [cs]*, 2022.

[34] André Weimerskirch, Steffen Becker, and Bill Hass. Commercial Vehicle vs. Automotive Cybersecurity – Commonalities & Differences. In *Cybersecurity for Commercial Vehicles*, pages 19 – 64. SAE International, August 2018.

[35] A. Taylor and N. Japkowicz and S. Leblanc. Frequency-Based anomaly detection for the automotive CAN bus. In *Proc. of WCICSS*, pages 45–49, 2015.

[36] Hyun Min Song, Ha Rang Kim, and Huy Kang Kim. Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network. In *2016 International Conference on Information Networking (ICOIN)*, pages 63–68, 2016.

[37] Michael R. Moore, Robert A. Bridges, Frank L. Combs, Michael S. Starr, and Stacy J. Prowell. Modeling inter-signal arrival times for accurate detection of CAN bus signal injection attacks: a data-driven approach to in-vehicle intrusion detection. In *Proceedings of the 12th Annual Conference on Cyber and Information Security Research*, pages 1–4, Oak Ridge Tennessee USA, 2017. ACM.

[38] Charlie Miller and Chris Valasek. A Survey of Remote Automotive Attack Surfaces. In *Black hat USA*, page 94, Las Vegas, NV, USA, 2014. Blackhat Press.

[39] Kyong-Tak Cho and Kang G Shin. Fingerprinting Electronic Control Units for Vehicle Intrusion Detection. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 911–927, Austin, TX, USA, 2016. USENIX Association.

[40] Habeeb Olufowobi, Sena Hounsinou, and Gedare Bloom. Controller Area Network Intrusion Prevention System Leveraging Fault Recovery. In *Proceedings of the ACM Workshop on Cyber-Physical Systems Security & Privacy - CPS-SPC'19*, pages 63–73, London, United Kingdom, 2019. ACM Press.

[41] Yoshihiro Ujiie, Takeshi Kishikawa, Tomoyuki Haga, Hideki Matsushima, Tohru Wakabayashi, Masato Tanabe, Yoshihiko Kitamura, and Jun Anzai. A Method for Disabling Malicious CAN Messages by Using a CMI-ECU. In *SAE 2016 World Congress and Exhibition*, Detriot, Michigan, USA, 2016.

[42] Bogdan Groza, Stefan Murvay, Anthony Van Herrewege, and Ingrid Verbauwhede. LiBrA-CAN: Lightweight Broadcast Authentication for Controller Area Networks. *ACM Transactions on Embedded Computing Systems*, 16(3):1–28, 2017.

[43] Ryo Kurachi, Yutaka Matsubara, Hiroaki Takada, Naoki Adachi, Yukihiro Miyashita, and Satoshi Horihata. CaCAN - Centralized Authentication System in CAN (Controller Area Network). In *14th Int. Conf. on Embedded Security in Cars (ESCAR 2014).*, page 10, Munich, Germany, 2014. ESCAR.

[44] Jeremy S. Daily and Prakash Kulkarni. Cyber_0920_secure Heavy Vehicle Diagnostics_paper1.pdf. In *2020 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM*, NOVI, MICHIGAN, 2020.

[45] Kyong-Tak Cho and Kang G. Shin. Viden: Attacker Identification on In-Vehicle Networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1109–1123, Dallas Texas USA, 2017. ACM.

[46] Wonsuk Choi, Hyo Jin Jo, Samuel Woo, Ji Young Chun, Jooyoung Park, and Dong Hoon Lee. Identifying ECUs Using Inimitable Characteristics of Signals in Controller Area Networks. *IEEE Transactions on Vehicular Technology*, 67(6):4757–4770, 2018.

[47] Sandeep Nair Narayanan, Sudip Mittal, and Anupam Joshi. OBD_securealert: An Anomaly Detection System for Vehicles. In *Proceedings of the 2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6, St Louis, MO, USA, 2016. IEEE.

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

[49] Aymen Boudguiga, Witold Klaudel, Antoine Boulanger, and Pascal Chiron. A simple intrusion detection method for controller area network. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–7, Kuala Lumpur, Malaysia, 2016. IEEE.

[50] Tsvika Dagan and Avishai Wool. Parrot, a software-only anti-spoofing defense system for the CAN bus. In *Embedded Security in Cars, EUROPE*, page 10, Munich, Germany,, 2016. ESCAR EUROPE.

[51] Mohammad Raashid Ansari, W. Thomas Miller, Chenghua She, and Qiaoyan Yu. A low-cost masquerade and replay attack detection method for CAN in automobiles. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, Baltimore, MD, USA, 2017. IEEE.

[52] T. Matsumoto, M. Hata, M. Tanabe, K. Yoshioka, and K. Oishi. A Method of Preventing Unauthorized Data Transmission in Controller Area Network. In *2012 IEEE 75th Vehicular Technology Conference (VTC Spring)*, pages 1–5, Yokohama, Japan, 2012. IEEE.

[53] NXP. Secure TJA115x CAN Transceivers | NXP Semiconductors, 2021.

[54] Teri Lenard and Roland Bolboaca. A Statefull Firewall and Intrusion Detection System Enforced with Secure Logging for Controller Area Network. In *European Interdisciplinary Cybersecurity Conference*, pages 39–45, Virtual Event Romania, 2021. ACM.

[55] George Loukas, Eirini Karapistoli, Emmanouil Panaousis, Panagiotis Sarigiannidis, Anatolij Bezemskij, and Tuan Vuong. A taxonomy and survey of cyber-physical intrusion detection approaches for vehicles. *Ad Hoc Networks*, 84:124–147, 2019.

[56] Body Control Module (BCM) - STMicroelectronics.

[57] Hyo Jin Jo and Wonsuk Choi. A Survey of Attacks on Controller Area Networks and Corresponding Countermeasures. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–19, 2021. Conference Name: IEEE Transactions on Intelligent Transportation Systems.

[58] Salvador Garcia, Julian Luengo, José Antonio Sáez, Victoria Lopez, and Francisco Herrera. A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE transactions on Knowledge and Data Engineering*, 25(4):734–750, 2012.

[59] N. J. Martinez and M. C Ow and M. I. Barrasa and M. Hammell and R. Sequerra and L. Doucette-Stamm and F. P. Roth and V. R. Ambros and A. JM Walhout. A C. elegans genome-scale microRNA network contains composite feedback motifs with high flux capacity. *Genes & development*, 22(18):2535–2549, 2008.

[60] Scipy. scipy.stats.skew.

[61] J. D. Brutlag. Aberrant Behavior Detection in Time Series for Network Monitoring. In *Proc. of LISA*, pages 139–146, 2000.

[62] RDocumentation. forecast.HoltWinters: Forecasting using Holt-Winters objects.

[63] RDocumentation. HoltWinters: Holt-Winters Filtering.

[64] C. Goh and R. Law. Modeling and forecasting tourism demand for arrivals with stochastic nonstationary seasonality and intervention. *Tourism management*, 23(5):499–510, 2002.

[65] 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, 2017.

[66] Florin Baboescu and George Varghese. Scalable Packet Classification. *ACM SIGCOMM Computer Communication Review*, 31(4):12, 2001.

[67] T.V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *ACM SIGCOMM Computer Communication Review*, 28(4):203–214, 1998.

[68] Lili Qiu, G. Varghese, and S. Suri. Fast firewall implementations for software and hardware-based routers. In *Proceedings Ninth International Conference on Network Protocols. ICNP 2001*, pages 241–250, Riverside, CA, USA, 2001. IEEE Comput. Soc.

[69] Matthew Timothy Campo, Subhojeet Mukherjee, and Jeremy Daily. Real-Time Network Defense of SAE J1939 Address Claim Attacks. *SAE International Journal of Commercial Vehicles*, 14(3), August 2021.

[70] PJRC. Delay and Timing Functions.