HELENIC MEDITERRANEAN UNIVERSITY
DEPARTMENT OF INFORMATICS ENGINEERING

SECURITY SOLUTIONS FOR DENIAL OF SERVICE ATTACKS IN SMART VEHICLES

Author: Kypraios Eleftherios
Major Professor: Grammatikakis Miltos

01/2020

Abstract

In this thesis, we have developed an open distributed embedded platform prototype that targets traffic monitoring across multiple CAN networks. This ecosystem interconnects multiple Raspberry Pi or AVR devices (e.g., RPI1, RPI2) to an Odroid XU3 device which serves as a gateway node. CAN interconnection is based a) for Raspberry Pi, on Industrial Berry's CANberry Dual V2.1 device, and b) for Odroid XU3, on two (incoming/outgoing) USB-to-CAN interfaces using Scantool OBD Development Kit.

Incoming and outgoing CAN interfaces at the gateway are controlled by different threads. Our embedded software toolchain uses a) for RPI, Linux CAN-utils tools, and b) for Odroid XU3, an extended serial terminal that uses multithreaded code to handle incoming/outgoing connections; configuration and CAN message send/receive functions use appropriate USB-to-serial STN2120's ELM327 AT, and ST commands. During normal operation, RPI2 (CAN2) carries actual engine traffic (based on an actual Korean car dataset [37]), while at the same time RPI1 packet requests related to dashboard display (e.g. engine speed, RPM, temperature etc) departing from RPI1 (CAN1), are received via the Gateway by RPI2 (CAN2), and answered back to RPI1 (making a round trip).

In our threat model, we consider a denial-of-service (DoS) from CAN1 and examine different metrics that can be used to detect the attack. At gateway-level, we can detect the DoS attack by using metrics and setting appropriate thresholds related to the Cortex-A15 energy consumption (available from integrated INA231 sensors), and four temperature gradients on the same chipset (available from integrated sensors). In addition, we are able to monitor variations of round-trip time (RTT) by monitoring the sequences of packets that originate from RPI, flow to RPI2 via Odroid XU3 and return back to RPI, in a ping-pong pattern. Our results show tradeoffs in the accuracy and effectiveness of the proposed metrics in detecting actual attacks. Accurate prediction of an attack results in shutting down, throttling down, or sleeping the appropriate outgoing interface, thus safeguarding the engine ECUs.

ΠΕΡΙΛΗΨΗ

Σε αυτή τη εργασία, αναπτύχθηκε μια ανοιχτή κατανεμημένη ενσωματωμένη πλατφόρμα που στοχεύει στην παρακολούθηση της κυκλοφορίας σε πολλαπλά δίκτυα CAN. Αυτό το οικοσύστημα διασύνδεει συσκευές Raspberry Pi ή AVR (π.χ. RPI1, RPI2) μέσω μια συσκευής Odroid XU3 που αναλαμβάνει τον ρολό της πύλης (Gateway). Η διασύνδεση CAN βασίζεται α) για το Raspberry Pi, στη συσκευή CANberry Dual V2.1 της Industrial Berry και β) για το Odroid XU3, σε δύο (εισερχόμενες / εξερχόμενες) διεπαφές USB-CAN χρησιμοποιώντας το Scantool Development Kit.

Οι εισερχόμενες και εξερχόμενες διεπαφές CAN στην πύλη ελέγχονται από διαφορετικά threads. Το ενσωματωμένο λογισμικό που αναπτύξαμε χρησιμοποιεί α) στο Raspberry PI, τα εργαλεία Linux CAN-utils και β) στο Odroid XU3, μια επέκταση μας ενός σειριακού τερματικού που χρησιμοποιεί κώδικα πολλών νημάτων (multithread) για να χειρίζεται τις εισερχόμενες/εξερχόμενες συνδέσεις. Η διαμόρφωση των ρυθμίσεων και οι λειτουργίες λήψης και αποστολής μηνυμάτων CAN χρησιμοποιούν κατάλληλες εντολές ELM327 (AT) και εντολές ST του STN2120.

Κατά τη διάρκεια της κανονικής λειτουργίας, το RPI2 (CAN2) μεταφέρει πραγματική κυκλοφορία μηνυμάτων του κινητήρα (βασισμένη σε ένα πραγματικό σύνολο δεδομένων κορεατικών αυτοκινήτων [37]), ενώ ταυτόχρονα τα αιτήματα πακέτων από RPI1 τα οποία σχετίζονται με την οθόνη του ταμπλό και εισάγονται στο CAN1, καταλήγουν μέσω της πύλης στο RPI2 (CAN2) και απαντώνται πίσω στο RPI1 (πραγματοποιώντας ένα ταξίδι μετ' επιστροφής). Στο threat model μας, θεωρούμε την περίπτωση που έχουμε επίθεση άρνησης εξυπηρέτησης (DoS) από το CAN1 και εξετάζουμε διαφορετικές μετρήσεις για την ανίχνευση της επίθεσης αυτής.

Σε επίπεδο Gateway, μπορούμε να ανιχνεύσουμε DoS χρησιμοποιώντας μετρήσεις και να καθορίσουμε κατάλληλα όρια σε σχέση με α) την κατανάλωση ενέργειας του Cortex-A15 (που είναι διαθέσιμη από τους ενσωματωμένους αισθητήρες INA231), β) περιοχές αυξημένης θερμοκρασίας (temperature zones) και συχνότητες εμφάνισης μηνυμάτων. Επιπλέον, μπορούμε να παρακολουθήσουμε τις διακυμάνσεις του round-trip time (RTT) παρακολουθώντας τις ακολουθίες πακέτων που προέρχονται από το RPI1, τα οποία στην συνέχεια μεταφέρονται στο RPI2 μέσω του Odroid XU3 και επιστρέφουν πίσω στο RPI1, σε μοτίβο πινγκ-πονγκ.

Τα αποτελέσματά μας δείχνουν ότι όλες οι παραπάνω μετρήσεις για την ανίχνευση πραγματικών επιθέσεων είναι ακριβείς και αποτελεσματικές και χρήζει περαιτέρω μελέτης. Η ακριβής πρόβλεψη μιας επίθεσης θα έχει ως αποτέλεσμα την απενεργοποίηση, τη μείωση της κίνησης, ή την προσωρινή αποδέσμευση της εξερχόμενης διεπαφής, προστατεύοντας έτσι το σύστημα του κινητήρα.

Table of Contents

1. Introduction 10
2. Safety and Security in Real Time Systems 12
    2.1 Real-Time Operating System 12
    2.2 Time-Triggered Communication Protocol 12
    2.3 Time-Triggered Ethernet 12
    2.4 The CAN Bus 13
    2.5 Bit Arbitration 16
    2.6 Frames 16
        2.6.1 Data Frames 16
        2.6.2 Remote Frame 19
        2.6.3 Error Frame 19
        2.6.4 Overload Frame 20
        2.6.5 Valid Frame 21
    2.7 Functional safety 22
        2.7.1 ISO 26262-1 standard 22
        2.7.2 Failure Modes Effects and Diagnostic Analysis (FMEDA) 24
    2.8 Mixed criticality 24
3. Threat model and State-of-the-Art 25
    3.1 Possible Attacks 25
        3.1.1 Replay (or Playback) Attack 25
        3.1.2 Masquerade (or Spoofing) Attack 25
        3.1.3 Denial of Service Attack 25

3.1.4 Distributed Denial of Service Attack 26

3.2 Threat Model and Our Solution 26

3.3 Related Work 26

4. Experimental Platform 30

4.1 Hardware Devices 30

4.1.1 Odroid XU3 30

4.1.2 OBD DEVELOPMENT BOARD 31

4.1.3 Gingko 33

4.1.4 Raspberry Pi3 34

4.1.5 CanBerryDual ISO 2.1 35

4.2 Drivers and Software 35

4.2.1 Integration Towards Final Platform 35

4.2.2 Concept Validation - Energy Monitor Tool 37

4.2.2.1 Code snippets - Receiver Thread on Odroid XU3 Gateway 38

4.2.2.2 Code Snippets - Sender Thread on Odroid XU3 Gateway 39

4.2.2.3 Gateway Software on Odroid XU3 41

4.2.2.4 RPI Setup - RPI Sender (RPI1) & Receiver (RPI2) 42

4.3 Troubleshooting Guide 42

4.4 Experimental Evaluation 43

4.4.1 Towards Deriving a DoS Metric: Example with Energy Metric 43

4.4.2 Detailed Results 44

4.4.2.1 RPI Sender (RPI1) 45

4.4.2.2 RPI Receiver (RPI2) 46

5. Conclusions 50

6. Future Work 51

References 52

Table of Figures

Figure 1: Example of a network connection without and with CAN [39] 13
Figure 2: ECU physical and Data link layer 14
Figure 3: In-vehicle network [40] 15
Figure 4: Standard Data Frame [41] 17
Figure 5: Extended Data Frame [41] 19
Figure 6: Remote Frame [41] 19
Figure 7: Error Frame [41] 20
Figure 8: Overload Frame [41] 20
Figure 9: Automotive networks 21
Figure 10: Odroid XU3 30
Figure 11: OBD development board 31
Figure 12: OBD Development Kit -Interface Diagram 32
Figure 13: Gingko 33
Figure 14: Rasberry Pi 3 device 34
Figure 15: CanBerryDual ISO 2.1 35
Figure 16: Initial prototype Gateway solution 36
Figure 17: Open distributed embedded platform prototype 36
Figure 18: Energy Monitor Tool window 37
Figure 19: Code for Energy Monitor Tool (I2C interface calls) 38
Figure 20: Gateway threads receiving and sending packets 40
Figure 21: Scenario with normal traffic 43
Figure 22: Scenario with network under attack 44
Figure 23: Experimental framework of our platform 44
Figure 24: Average instant energy on Cortex-A15 – diff. rates of malicious packets (buffer size 1) 47
Figure 25: Average instant energy on Cortex-A15 - diff. rates of malicious packets (buffer size 100) 48
Figure 26: Average temperature of four available thermal zones 48
Figure 27: Round-trip time (RPI1 to RPI2 and back in feedback loop) 49

Acknowledgments

Firstly, I would like to thank my family for supporting me all these years for my studies and life in general.

I would specifically like to express my gratitude to my thesis advisor Dr. Miltiadis Grammatikakis who was there to help me every time with his knowledge and wisdom and be patient to provide me answers for all my questions. I would like to thank Voula Piperaki for supporting me from the start. Moreover, I am also thankful to my colleagues of Hellenic Mediterranean University. Specifically, I would like to thank Nikos Mouzakitis and Nikos Papatheodorou that contributed to the project and helped make it possible with their experience and knowledge. Also, I would like to mention that the whole experience I had in the Artificial Intelligence and System Engineering Laboratory (AISE Lab) helped me improve, practice and learn crucial knowledge for my future.

1. Introduction

Over the years, evolution of the automotive industry has brought innovations that add to the conveniences and needs of a better driving experience. Modern cars that ensure these amenities are connected to several different electronic devices.

A modern car able to support smart services, including preliminary autonomous driving, has more than 80 ECUs (Engine Control Units) that are made responsible for different applications such as ABS, lights, windows etc. CAN (Controlled Area Network) is the most popular in-vehicle network. In fact, CAN bus is the preferred bus for all types of vehicles. ECUs, sensors and actuators are all connected with it.

CAN bus was proposed by Bosch in 1980s [38], a time that cyber security threats, attacks, and protection mechanisms were not at all in the scope. Therefore, CAN was designed without paying attention to security requirements. Since then different measures and solutions have been proposed that extend CAN for a variety of security threats, but they have not been incorporated into a standard, and hence, up to this day CAN is prone to different types of attacks.

In this context, we have developed an open distributed embedded platform prototype that targets traffic monitoring across multiple CAN networks. This ecosystem interconnects multiple Raspberry Pi or AVR devices (e.g., RPI1, RPI2) to an Odroid XU3 device which serves as a gateway node. CAN interconnection is based a) for Raspberry Pi, on Industrial Berry's CANberry Dual V2.1 device, and b) for Odroid XU3, on two (incoming/outgoing) USB-to-CAN interfaces using Scan tool OBD Development Kit.

Incoming and outgoing CAN interfaces at the gateway are controlled by different threads. Our embedded software toolchain uses a) for RPI, Linux CAN-utils tools, and b) for Odroid XU3, an extended serial terminal that uses multithreaded code to handle incoming/outgoing connections; configuration and CAN message send/receive functions use appropriate USB-to-serial STN2120's ELM327 AT, and ST commands. During normal operation, RPI2 (CAN2) carries actual engine traffic (based on an actual Korean car dataset), while at the same time RPI1 packet requests related to

dashboard display (e.g. engine speed, RPM, temperature etc.) departing from RPI1 (CAN1), are received via the Gateway by RPI2 (CAN2) and answered back to RPI1 (making a round trip).

In our threat model, we consider a denial-of-service (DoS) from CAN1 and examine different metrics that can be used to detect the attack. At gateway-level, we can detect the DoS attack by using metrics and setting appropriate thresholds related to the Cortex-A15 energy consumption (available from integrated INA231 sensors), and four temperature gradients on the same chipset (available from integrated sensors). In addition, we are able to monitor variations of round-trip time (RTT) by monitoring the sequences of packets that originate from RPI1, flow to RPI2 via Odroid XU3 and return back to RPI1, in a Ping-Pong pattern. Our results show tradeoffs in the accuracy and effectiveness of the proposed metrics in detecting actual attacks. Accurate prediction of an attack results in shutting down, throttling down, or sleeping the appropriate outgoing interface, thus safeguarding the engine ECUs.

Next, in Chapter 2 we focus on CAN bus, as well as real-time, functional safety and security aspects. Chapter 3 examines the threat model and state of the art. Chapter 4 develops the experimental framework of our embedded platform and identifies key detection metrics for Denial-of-Service attacks. Finally, Chapters 5 and 6 draws conclusions and elaborates on future work.

2. Safety and Security in Real Time Systems

2.1 Real-Time Operating System

Real-time Operating Systems (RTOS) are all operating systems (OS) that can process with predictable delay [1]. Real-time applications are critical, and they are used to model safety-related tasks. Real time systems have time constraints and they have to be executed within specific time intervals (deadlines), otherwise the system will probably or certainly fail.

Most of the time RTOS processes a specific set of applications, for a real time OS is more important how fast it can respond to a specific event, rather than how many tasks it can perform at the same time. The two most common designs are the event-driven and the time-sharing the main difference between them is that event-driven works solely with events scheduling for higher priority, while time-sharing primarily switches tasks on a regular clocked interrupt, i.e., it follows a round-robin scheme.

2.2 Time-Triggered Communication Protocol

Time-Triggered protocol (TTP) is a computer network protocol designed in Vienna University of Technology as an industrial (e.g., transportation, space) application of control systems [2]. In 2011, it was standardized as SAE AS6003 Communication Protocol. TTP is also used in aircraft and aerospace applications, as well in railway signaling applications. TTP is a dual channel 4-25Mbit/s time-triggered field bus. TTP has an important feature that can inform all the connected nodes of a network at the same time if any of the nodes fails, and all the nodes update their status several times per second.

TTP in automotive is trying to replace mechanical and hydraulic parts with electronics. Such parts are the engine, steering, braking and transmission. This technology is called by-wire and is believed to be of low cost, higher reliability and improved performance.

2.3 Time-Triggered Ethernet

Time Triggered Ethernet (TTE) is a standard which defines a fault tolerant strategy for building and maintaining Ethernet networks. TTE establish robust synchronization, synchronous packet-switch, bandwidth partitioning and traffic scheduling (SAE AS6802 describes the above).

The TTEthernet network is used for OSI layer-2 applications and it supports IEEE 802.3 standards which are other Ethernet networks and services. Protocol control frames (PCF) are used for stabilization and maintenance of synchronization and have the highest priority.
Time triggered traffic: The Ethernet packets that are sent over the network are scheduled and they do not collide with other traffic because of the precedence that they have over rate constrained traffic.[3]

2.4 The CAN Bus

CAN (Controller Area Network) is a robust serial communication bus designed for industrial and automotive applications to perform in harsh environments [4]. It interconnects sensors, actuators and ECUs (electronic control units), and is very popular in most vehicle models, which sometime have 7-8 bus interconnects and 70-80 ECUs. The simple structure of the CAN based on two wires (CAN high, CAN low) reduces *cable wiring*.
Bosch created and introduced CAN Bus in the 1980s to provide a low-level networking solution to cover the needs for high-speed in-vehicle communication. Automotive industry began manufacturing it in 1990s, it was much later in 1993 that CAN Bus was standardized (e.g. ISO 11898-1, 11898-2, and 11898-3).

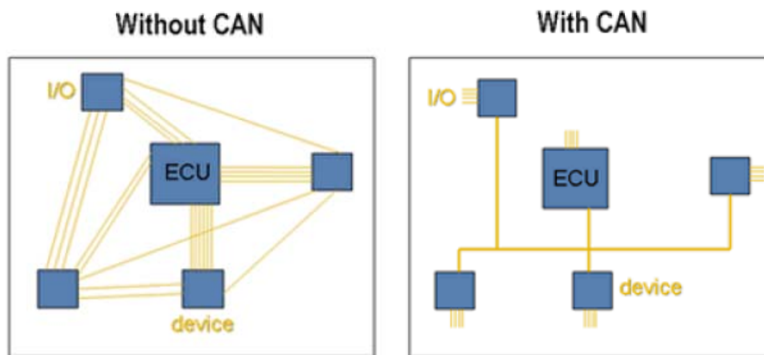


Figure 1: Example of a network connection without and with CAN [39]

As shown in *Figure 1*, shortly after the automotive industry began to adapt the new two-wire CAN bus, replacing the old point-to-point wired networks reduced dramatically the weight of the car.

After 1996, five higher-layer automotive-specific signaling protocols, such as ISO 15765 for global on-board diagnostics (OBD) in most US and European cars appeared. On-board diagnostics (OBD) can help you or the technician troubleshoot problems that occur with the diagnostic and report system making vehicle maintenance fast and simple. More recently, in 2012 CAN extensions for higher performance were presented by Bosch for higher data transmission rates, so-called flexible data-rates, or CAN FD.

CAN protocol is based on non-synchronous, communication on two different layers of OSI. The Physical Layer as in *Figure 2* with data rates from 125 Kbit/s to 1 Mbit/s, and the Data Link Layer. The bus consists of two wires: CAN Low & CAN High, while the maximum distance that the bus can operate is about 40 meters (or up to 120 meters with special high-performance transceivers). More specifically, there are two physical layer protocols: CAN high (defined in ISO 11898-2 protocol) capable of speeds up to 1 Mb/s, and CAN low (defined in ISO 11898-3) capable of speeds up to 125Kb/s and this also gives the possibility of higher fault tolerance.

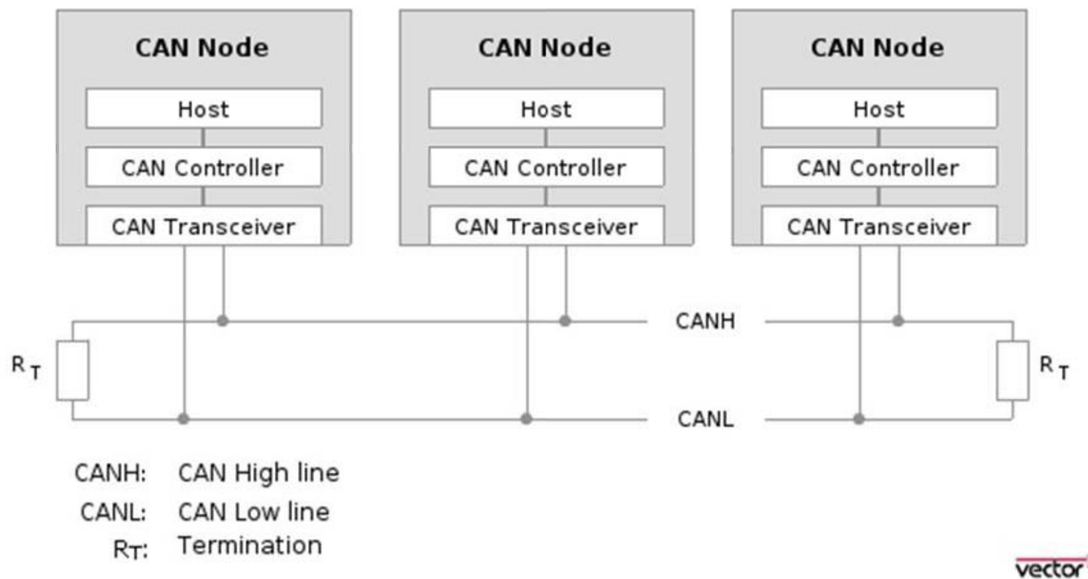


Figure 2: ECU physical and Data link layer

Safety and critical applications such as the engine or the ABS of the vehicle require higher data rates to operate at best. The policy the CAN protocol is Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). The CAN use san id priority which means even though all the nodes that are

connected to the bus have the right to access, only those with the higher priority (the one with the lowest ID) are authorized by the interface (CAN controller and transceiver) to broadcast on the bus. Thus, CAN bus gives priority on the critical (high priority CAN messages) and the non-critical (lower priority CAN messages) are delayed. When a node with higher priority starts transmitting to the bus all other node with lower priorities switch to the receiving state in order to listen to the broadcast message.

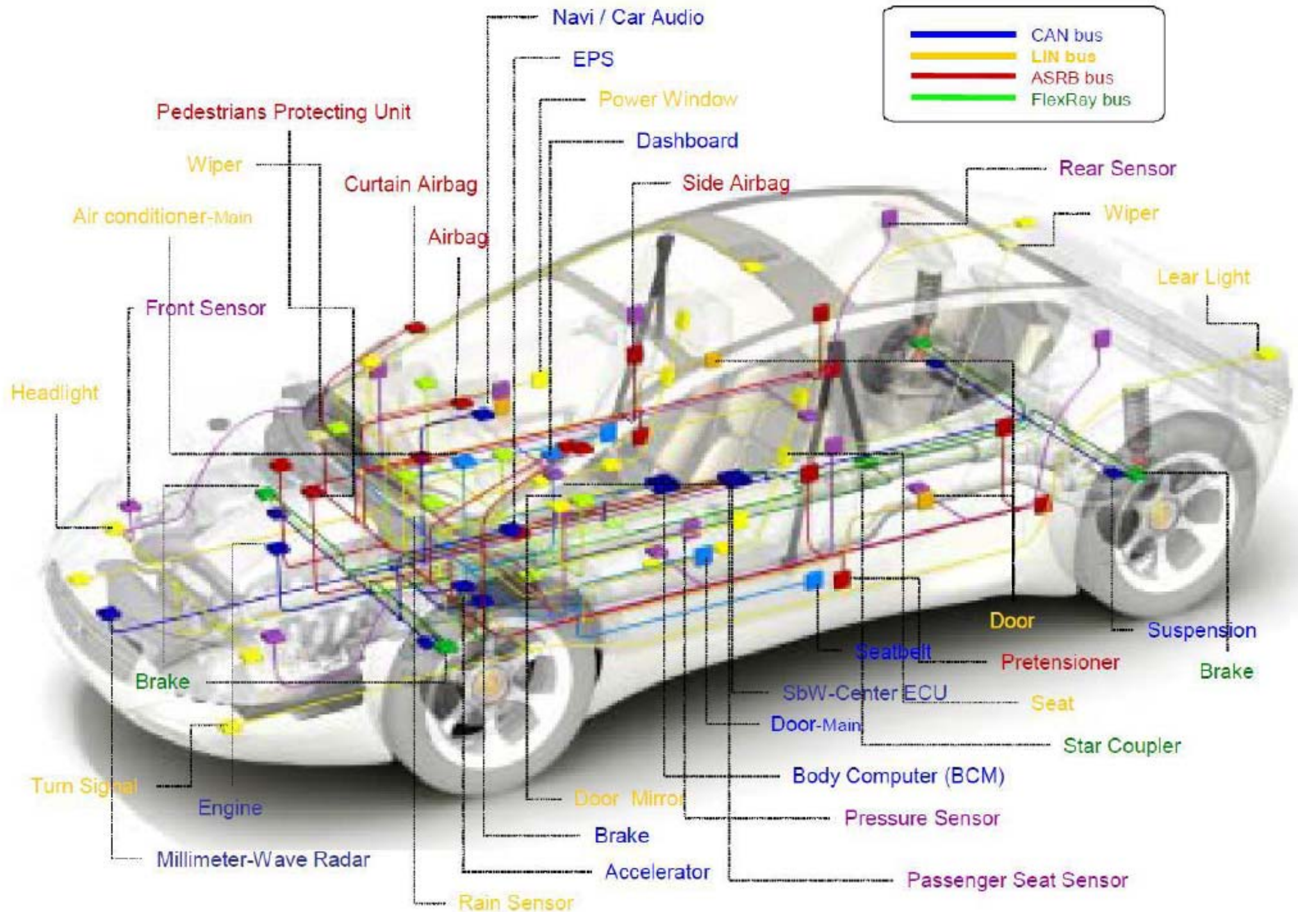


Figure 3: In-vehicle network [40]

Functions of the car are controlled and monitored by the ECUs. ECUs send messages and exchange information among them, as shown in *Figure 3*. ECUs have different roles the can control non critical functions like the windows roll up/down or they can operate functions such as ABS system that are more critical for the safety of passengers. Nodes interact with the physical layer of the CAN via a transceiver that all ECUs provide, enabling a two-wire

analog data connection, one for the low CAN and the other wire for the high CAN. The data link layer (ISO 11898-1) is implemented by the CAN controller, which is responsible to send packets (known as frames) to CAN network or receive from it.

2.5 Bit Arbitration

CAN nodes that want to send messages to another node do not use direct point-to-point communication. Instead, they send the message over the bus that all the connected nodes have access. If a node then reacts to the message, it means is interested. CAN bus is based on a bitwise arbitration protocol so that no collisions may occur from concurrent access to the CAN bus. The IDs CAN uses for communication purpose are specific, but some automotive companies may have added extra ones for specific tasks. The CAN priority defined as the node with the lowest ID number is the highest priority on the bus (ID 0 has the highest), so if multiple nodes transmit repeatedly at the same time only the nodes with the highest priority will send the message, and only one node can send at the time. Lower priority nodes will send their messages as soon as the network is not busy. Thus, all nodes will send their messages eventually after some delay, but it is ensured that the highest priority messages will be transmitted on time for real time communication.

For instance, we have three nodes, node 1 with ID 0001, node 2 with ID 0010, and a node with ID 0100. In this way, if they try to broadcast all together at the same time the node 1 will send first and the other two will wait and listen until the network is not busy, then node 2 will send second and, finally node 3.

2.6 Frames

CAN bus messages, called CAN frames, are of four different types [6].

2.6.1 Data Frames

Data frames have two different types, the standard frame or base frame with 11 identifier bits, and the extended frame version with 29 identifier bits.

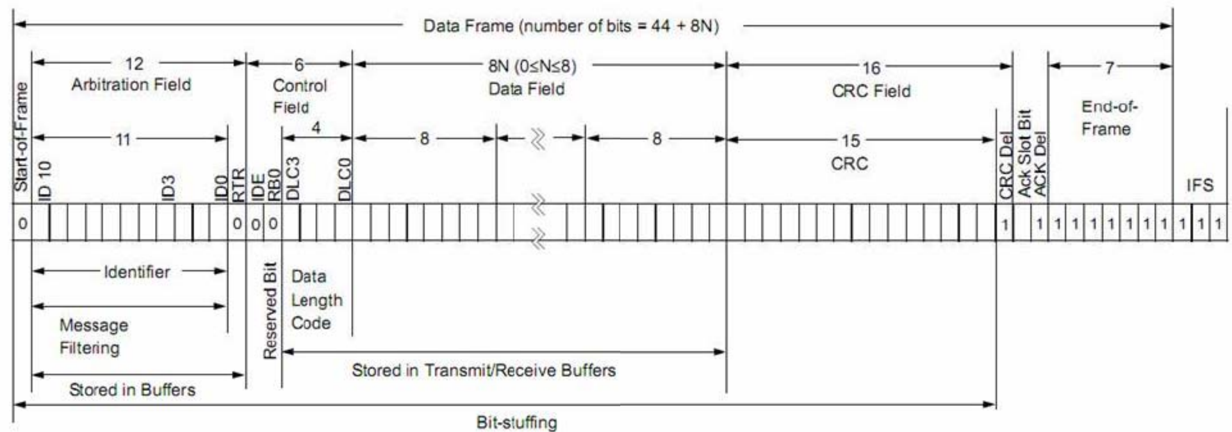


Figure 4: Standard Data Frame [41]

A CAN data frame is split in two versions the layout differences are small, but the size is different. The Standard Data Frame (see in *Figure 4*) (CAN 2.0A), and the Extended Data Frame (see *Figure 5*) (CAN 2.0B).

• The first bit of CAN 2.0A and 2.0B is zero to indicate that a transmission began, as the inter-frame state of the CAN bus is logical one.

• The next 11 bits form the identifier.

• The Remote Transmission Request (RTR) is a bit whose value is zero in case of a Data Frame and one in case of a *Remote Frame*.

• The next part of this is the Identifier Extension (IDE) bit that is zero in case of a standard frame.

• The following bit Reserved Bit Zero (RB0) is also zero.

• After that comes a four-bit representation of the size of the data field named Data Length

Code (DLC).

• Data Length that can be up to 8 bytes and are the data transmitted.

• Cyclic Redundancy Check (CRC) of 15 bits used to identify errors that may occur during broadcast phase.

• CRC Delimiter that always has the value of one.

• Acknowledgment Slot (ACK Slot) bit that has the value one when a node receives a Data Frame without mistakes. The ACK Slot takes the value zero if there are any errors.

• At the end of the frame there are 7 logical one bits.

➢Extended vs Standard Data Frame

The main difference between the Extended (Figure 4) and the Standard Data Frame (Figure 5) version is their size. The Extended version has 20 bits more than the other and most of them are in CAN ID part as it is 29 instead of 11 bits. The idea for this is the need of the automotive companies to have a vast set of universally unique identifiers for their products, even if there are not very unique at their function. So, in the Extended Frame the arbitration field has 32 bits compared with 12 bits of the Standard Frame. The 11-bit (basic) Identifier exists in both versions. The Extended version uses the Substitute Remote Request (SRR) bit that has the value of one.
IDE comes next as logical one. At the end, we have the 18-bit part of the Extended Identifier.

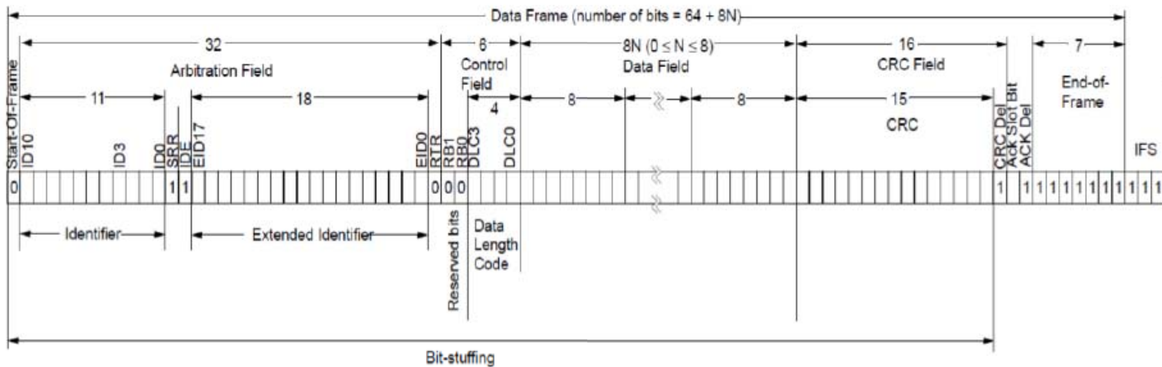


Figure 5: Extended Data Frame [41]

2.6.2 Remote Frame

The purpose of the remote frame shown in *Figure 6* is to seek permission for the transmission of data from another node. The remote frame and date frame are similar. The two differences they have is that this type of message is marked as remote frame in the arbitration field, and it doesn't have any data.

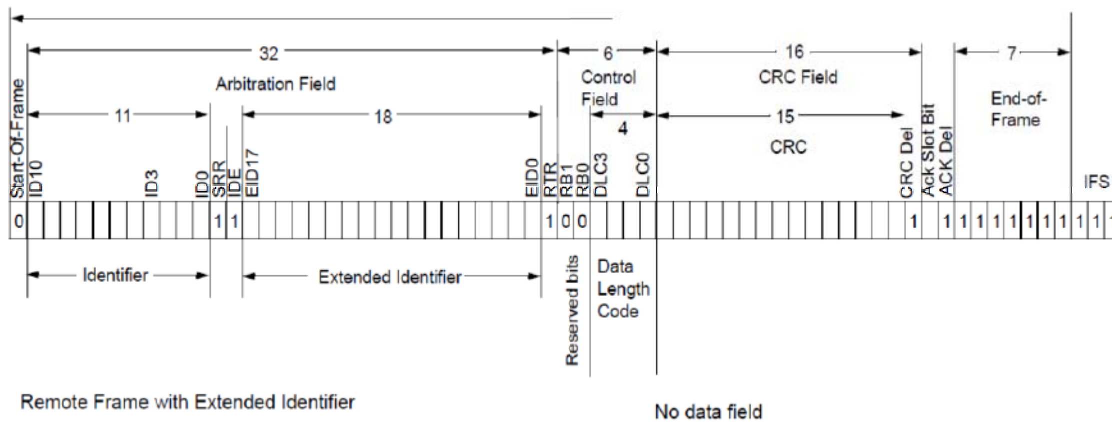


Figure 6: Remote Frame [41]

2.6.3 Error Frame

As shown in Figure 7, when an error is spotted on the bus all the other nodes send an Error Frame. If there are 5 bits or more set in a row (that do return to zero), then the other nodes begin to send an Error Frame with 8 zeros in the field of Error Flag and after that bus transmission stops. Multiple consecutive

errors result to error passive mode (slower transmission rate), and eventually after an 8-bit counter expires, bus off.

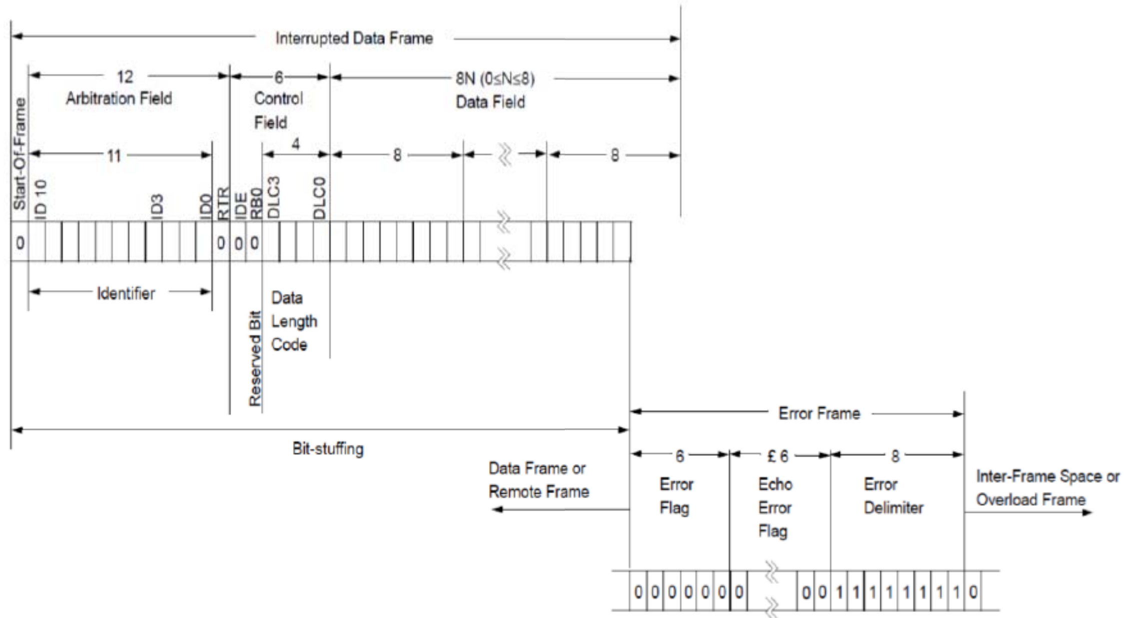


Figure 7: Error Frame [41]

2.6.4 Overload Frame

Overload Frames are like Error Frames. They are transmitted during the Inter-Frame phase to introduce additional delay between Data and Remote Frames, but unlike Error Frames they do not stop transmission (see *Figure 8*).

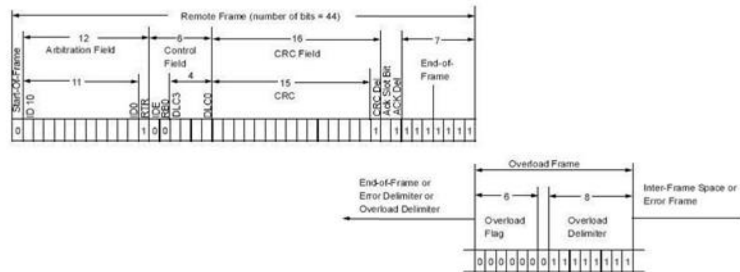


Figure 8: Overload Frame [41]

2.6.5 Valid Frame

If the last bit of the EOF field is received in an error free state, then this message is considered as error-free. The transmission will begin again by the transmitter if in the EOF field there exist a dominant bit. In automotive, there are three other important in-vehicle network technologies differing in the protocol and baud rate. These are used in different configurations as identified in *Figure 9.*

BUS	LIN	CAN	FlexRay	MOST
Application	Low level communication Systems	Soft Real Time Systems	Hard Real Time Systems (X - by - Wire)	Multimedia
Example	Body	Engine, Powertrain, Chassis	Powertrain, Chassis	Multimedia, Telematics
Architecture	Single Master typ 2...10 slaves	Multi Master typ 10...40 nodes	Multi Master up to 64 nodes	Multi Master up to 64 nodes
Bus Access	polling	CSMA/CA	TDMA/FTDMA	TDM CSMA/CS
Topology	Bus	Bus	Bus/Star	Ring/Star
Message Transmission	Synchronous	Asynchronous	Synchronous & Asynchronous	Synchronous & Asynchronous
Msg Identification	Identifier	Identifier	Time Slot	
Data Rate	20 Kbps	1 Mbps	10 Mbps	24 Mbps
Data bytes per frame	0 to 8	0 to 8	0 to 254	0 to 60
Physical layer	Electrical - Single wire	Electrical - Dual wire	Dual wire – Optical/Electrical	Optical fiber

Figure 9: Automotive networks

• Local Interconnect Network (LIN) [7] is a single-wire bus that is low cost, good for interconnecting low bandwidth sensors and actuators. LIN is a single master – slave bus and each master is responsible for up to 16 slaves. The speed of LIN is limited to 19.2 Kb/s.

• Flexray is designed to cover fast networks [8]. It is a dual-wire bus with high priority. It offers speeds up to 10Mbp/s, and implements a multi-master with up to 64 nodes, but due to the high cost is only available on few high end models.

• Media Oriented Systems Transport (MOST) [5] mainly used for multimedia transmission such as audio, video and navigation is an optical fiber bus that offers bandwidth up to

24Mbp/s. Automotive companies that use MOST bus are Audi, BMW, General Motors, Honda, Hyundai, Jaguar, Land Rover, Porsche, Toyota, Volkswagen, SKODA, and Volvo.
Even though there are options, in this thesis, we focus on CAN bus, by far the most common in-vehicle network in the automotive industry.

2.7 Functional safety

Functional safety is an important subject in all areas of automation industry. With the passing years challenges for safety have increased. Nowadays we have more autonomous machines that are able to perform tasks without supervision. Examples of autonomous machines are new smart cars aid the driver, aiming towards safer self-driven vehicles. In this direction, new requirements for safety measures in transportation systems take into account automated control in a framework that ensures correctness by design and functional safety. [9]

2.7.1 ISO 26262-1 standard

The ISO 26262-1 is a standard for "Road Vehicles Functional Safety" in automotive industry. This scheme is typically used to integrate necessary functional safety concepts in the design for different circumstances. ISO 26262-1 sets functional safety standards that are caused by non-reasonable risks due to errors that are produced by malfunctioning behavior of electrical and electronic (E/E) systems. Fault tolerance is not explicit defined in 26262. Instead the protocol handles four different Automotive Safety Integrity Levels (ASIL) that concentrate on defining safety requirements based on any non-reasonable risk of an item or component. Non reasonable risk is a risk that is caused by an unacceptable reason according the moral concepts of the society. An item is described as a system or a set of systems for implementing functions at vehicle level, compared to a component that is described as a system or part of it which includes software, hardware part and software units.
For every misadventure event that occurs an ASIL is appraised for the identification of an item. It is defined by combining misadventure analysis and risk appraisal. With the ratio of misadventure events that may occur, estimation of severity, probability and control-ability can be calculated. Based on the above estimations the ASIL type is selected, the most strict one is ASIL D while ASIL A is the least strict. For ISO 26262-1 to be applicable, the design must obey to the following criteria.

- All safety related systems with one or more E/E systems are passenger vehicles with no more than 3.500kg of weight.
- Possible hazards on E/E safety-related systems are caused by malfunction or interaction of the.

The ISO 26262-1 cannot support the following.

- E/E system that have specific purpose such as vehicles that are designed for passengers with disabilities.
- Systems, software units, hardware or components in production before the publication date of ISO 26262.
- The nominal performance of E/E systems, even if standards for functional performance exist.
- Risk factors that are related with the natural elements such as fire, water, radiation, exposure to sun, etc.

The ISO 26262-1 standard strictly requires all product life-cycle phases, such as idea, concept phase, development phase, validation phase, etc to be tested and approved. The ISO 26262 is the ideal standard that is globally accepted as the main standard to cover functional safety in the automotive industry. [10]

2.7.2 Cybersecurity standards

Cybersecurity standards are used for cyber protection of materials and include techniques that protect environments of a user or organizations. They reduce risks and prevent cyber-attacks.

They consist of several tools, policies, safeguards etc. The cybersecurity standards have existed for several years and have been considered in many domestic and international forums to provide the best results.

The IEC-62443 cyber security standards are known to be used for cyber security protection in different industries, including automotive. Cyber security certification programs for the IEC-62443 is globally offered by exida, TUV Nord, TUV Sud etc [11]. Standards such as ISA/IEC 62443 or even ISO 27001 form a starting point for controlling cyber security threats.

2.7.2 Failure Modes Effects and Diagnostic Analysis (FMEDA)

Failure Modes Effects and Diagnostic Analysis (FMEDA) is a systematic analysis technique that was developed between late 1980s and early 1990s [12]. FMEDA is an upgraded FMEA. It involves two extra concepts: probability of failure-detection and quantity of failure data.

For FMEDA to be adapted to electronic device level we need certain prerequisites. Few of them are Schematics, Standards, Data Libraries, Layouts, PCB reports, (component types, ratings, etc.).

FMEDA is commonly used in functional-safety tests of under-development products in order to verify if they meet the conditions and requirements. Even though FMEDA is a great way to test the integrity of safety measures, it is insufficient for a full evaluation.

2.8 Mixed criticality

Mixed criticality systems were designed to offer a solution for balancing the priority between real-time or safety critical with non-real-time or non-safety critical situations [13]. The cost of a mixed criticality system depends on the assurance and safety it provides.

A mixed criticality system must be tested and certified before it is used, this way it is ensured that it is safe, and it will not fail if a problem occurs. Mixed criticality varies depending on the scenario or application domain. In some cases, critical safety is fundamental, e.g. in engine control, while in other cases, such as in infotainment, safety is not critical. Even though most mixed criticality applications can be isolated in space by mapping them, separately from non-critical components, to predictable systems, e.g. a real-time network or operating system, there are a lot of challenges involving the use of mixed critical multicore systems that at the same time can reduce costs.

3. Threat model and State-of-the-Art

3.1 Possible Attacks

There are several types attacks on in-vehicle automotive networks, such as eavesdropping, fabrication, modification, replay, masquerade, and denial of service. While the former three types simply refer to listening to packets, and inserting a new or modifying an existing message, the remaining types are defined below.

3.1.1 Replay (or Playback) Attack

A replay attack occurs when a device connected to the network duplicates prior data though re-transmission [14]. The data are obtained by a third-party node which is assumed trustworthy. In this way older valid messages can be retransmitted at random times and afflict the network with unwanted data. With this threat model, messages seem to originate from a reliably node, and thus affect critical network functionalities, including causing serious system malfunction.

3.1.2 Masquerade (or Spoofing) Attack

Masquerade attacks occur when the attacker uses a fake identity to impersonate a legitimate node of the network. The purpose of that act is to gain access to the system and obtain information. There are several types of spoofing attacks in different settings, such as ID spoofing, GPS spoofing, referer spoofing etc [15]. The masquerade/spoofing attack on CAN bus can cause different issues, from malfunctions of a system UI to vehicle safety.

3.1.3 Denial of Service Attack

A Denial of Service (DoS) attack refers to making a computer or a service unavailable to accept other connections, this way any future client may not be accepted [16]. The DoS term is common for web services but is not limited to this field only. Within CAN bus network, the attacker sends a lot of high priority messages, e.g. with ID 0, flooding the bus so that the system cannot accept legitimate traffic. The DoS attack in an automotive system is critical, since if a component goes out of service, a serious accident may result.

Although our embedded automotive platform creates an ecosystem on which different threats can be examined, such as fabrication, modification, replay or masquerade, within this thesis, we only focus on detecting and countering DoS attack.

3.1.4 Distributed Denial of Service Attack

Distributed Denial of Service (DDoS) attack is similar to the DoS attack that we have focused, with the only difference being that the attack vector does not originate from a single source, but from multiple sources transmitting simultaneously [17]. This kind of attack is more powerful and has a better chance to flood the CAN network. In fact, multiple components can be attacked at once.

3.2 Threat Model and Our Solution

A DoS attack on CAN can make critical automotive subsystems, e.g. engine, throttle, ABS, unavailable by sending a lot of high priority messages. The simplicity and importance of DoS attacks are important reasons measures for detection and countering.

Within this scope, assuming as our threat model a DoS or DDoS attack on CAN, we examine and evaluate methods to secure CAN bus. For this reason, we develop an embedded platform targeting real automotive systems. The platform is equipped with a gateway component that connects to two CAN buses (CAN1, CAN2). Using this platform, we can develop, test and evaluate different DoS detection methods on our embedded platform.

3.3 Related Work

In this part we examine related work to *gateway/firewall protection in-vehicle communication* and i*ntrusion detection on CAN Bus*. To the best of our knowledge, previous work has focused on a different threat model, do not present any quantitative results, or propose the use of a new detection technique (usually frequency-based or derivatives, e.g. FFT) without comparisons with other existing techniques.

More specifically, K. Han and K.G. Shin proposed an external gateway-based monitoring mechanism **Error! Reference source not found.**. Their work focuses on providing additional security for data that is transferred among internal ECUs (via CAN, LIN, or FlexRay) and external networks (e.g. 3G/4G,

Bluetooth) in an automotive system. Their protocols provide data protection, assuming a bogus gateway problem. For instance, they examine if a message is genuine when it arrives to the receiver and if a message finally arrives to the receiver.

K. Daimi et al. focus on the design and implementation of secure group communications on in-vehicle bus-based networking systems using public key cryptography **Error! Reference source not found.**. The proposed scheme consists of 4 ECU groups, where each group has a MECU (Master ECU) while all MECUs are controlled from a Super MECU. Moreover, they examined and compared the use of Symmetric, Elliptic Curve Cryptography and Stream ciphers. The SMECU is the only component connected to the outsight world. Notice, that no actual implementation exists.

M. Wolf et al. have analyzed different bus protocols (e.g. LIN, CAN, FlexRay, MOST, Bluetooth etc.) examining possible malicious actions on the in-vehicle system **Error! Reference source not found.**. Theoretical practices proposed for secure Bus commination include sender authentication, cryptographic mechanisms and a gateway firewall mechanism that works as an authorization party. Again, no quantitative results are presented.

X. Guoqi et al. analyze the delay (upper bound delay) in in-vehicle networks when a CAN gateway is implemented for secure communication**Error! Reference source not found.**. Their scheme implements the discrete channel gateway architecture for real-time communication to analyze upper bound delay when higher priority messages be in the CAN Network. Notice, their scheme shows upper bound delay in reasonable time when high network traffic.

In **Error! Reference source not found.** a dedicated CAN router based on TTNoC (Time Triggered NoC) is implemented on CAN bus to protect the CAN system from malicious actions. The system implements a trust model which analyzes and prevents specific attacks is happening in CAN Bus such as: removal, DoS, eavesdropping, fabrications (fake message) and man-in-the-middle attacks. Notice, that respectively tests take place for each attack. However, the communication between two nodes isn't secure if the CAN messages not transmitting via the router.

Hoppe et al. **Error! Reference source not found.** gives an overview of intrusion detection techniques and attacks that are happening in CAN Bus and propose an IDs to alarm the CAN system when attacks occurred. Their work focuses on two things. In the first, they show a prototypical way to detect

anomalies in the automotive network and their technical aspects. The second thing, focus to inform the driver about the attack through the already installed technologies system like multimedia, screen, etc. in the best way possible.

OBD_Secure Alert is an alert anomaly CAN detection system based on Hidden Markov Model **Error! Reference source not found.**. While the automotive is in operation, the designed system examines different ECUs states (normal or abnormal) and it uses alert messages if needed. Notice that OBD Secure Alert is only alert system and non-preventive.

In addition, Weber et al, proposed an efficient hybrid intrusion detection system, that improves security in various ECU vehicle networks. Their method is based on experiments based on existing anomaly detection and machine learning algorithms. For instance, their scheme examines LODA algorithm, which reduces anomalies in vehicle **Error! Reference source not found.**. They showed a different way of anomaly detection with LODA in time series and this particular protocol is not based in previous threat patterns but in run time threats.

Moreover, an anomaly detection algorithm presented in **Error! Reference source not found.** analyzes the sequences of messages without knowledge of syntax and semantic of CAN messages. Different attacks are examined as replay attacks, bad injections, mixed injections; where corresponding experiments take place. Their results show that performance is improved in all scenarios while injection rates were different from each attack. Proposed an algorithm that can detect anomalies in the CAN system by analyzing the sequence of packets. This is very efficient because it uses the pre-installed hardware like ECUs to be noticed that they had a really good and evaluated results.

In 0 observe how CAN version 2.0 & CANopen application layer draft standard 3.01 detects anomalies in the system. With this technique they figured how ECUs and protocols communicate and evaluate a set of attacks.

The 0 is a survey on connected cars with emphasis on the security of in-vehicle network. They conclude that even though there are a lot of suggestions almost all of them aim to detect the attack, but do not propose a full solution.

Zhou and Fei examined different attacks that happen on CAN bus and proposed a three-layer anomaly detection system in order to protect it, according to the severity of the attack **Error! Reference source**

not found.. ECU traffic passes via gateway, where a detection component is placed. An alert message corresponds to three main levels based on automotive severity: 1) non-critical level (like window up and down, infotainment system), 2) critical level (actions to the wheel, acceleration of speed etc.), and 3) severe danger level (on-road vehicle – in high velocity).

Taylor et al. presented a frequency-based anomaly detection system which identifies overt and subtle attacks, that occur in wireless or wired network automotive CAN bus **Error! Reference source not found.**. The protocol examines the rate of data packets and especially the time sequence of CAN IDs, in order to understand the anomalies that happen in CAN traffic (high-rate corresponds to malicious action). To arrange data statistics (called flows), they compare the same information to an OCSVM (one class support vector machine). However, they have analyzed only non-periodic CAN messages and have insufficient data to validate their algorithm.

4. Experimental Platform

4.1 Hardware Devices

Focusing on the experimental, several different devices have been used. In fact, different configurations of the experimental platform have involved Arduino, Raspberry PI, Odroid XU3/4, Gingko nodes.

4.1.1 Odroid XU3

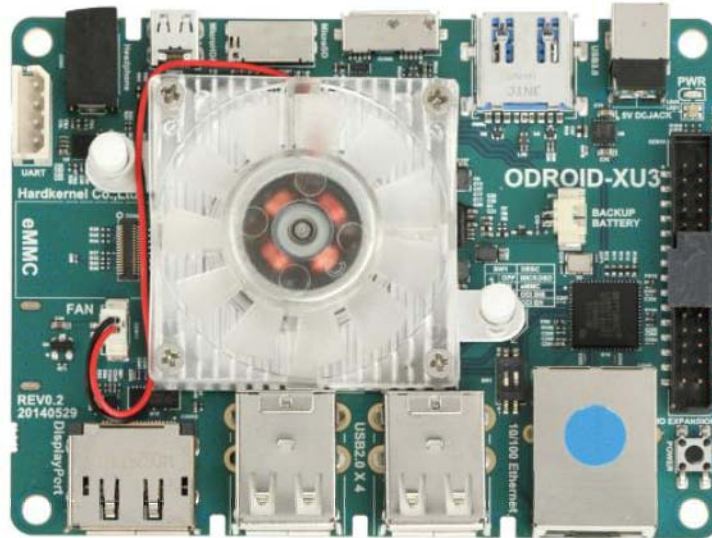


Figure 10: Odroid XU3

We chose ODROID-XU3 **Error! Reference source not found.** as our CAN gateway. This device, shown in *Figure 10*, is a relatively new computing device with nice open-source Linux-based capabilities. The most important factor in our choice is that this device has four integrated TI INA231 current-shunt and power sensors for A15, A7, memory and GPU (with a common I2C interface). The INA231 Linux driver exposes new power data on the sysfs file system. Using file operations, we can simply integrate power data analysis within our CAN gateway controller. Moreover, we are able to view power data using the power analysis tool (energy monitor).

The board provides:

- A Samsung Exynos 5422 chipset with Big-Little CPU architecture (Cortex A15 quadcore and second smaller Cortex A7 quadcore)
- 2Gbyte DDR3 RAM 933MHZ with bandwidth of 14.9GB/s

- eMMC5.0 HS400 Flash Storage
- 4 USB 2.0 ports, 1 USB 3.0 port and 1 USB 3.0 OTG port
- A HDMI 1.4 and a DisplayPort1.1 for display
- Integrated power consumption monitoring tool

4.1.2 OBD DEVELOPMENT BOARD

Figure 11: OBD development board

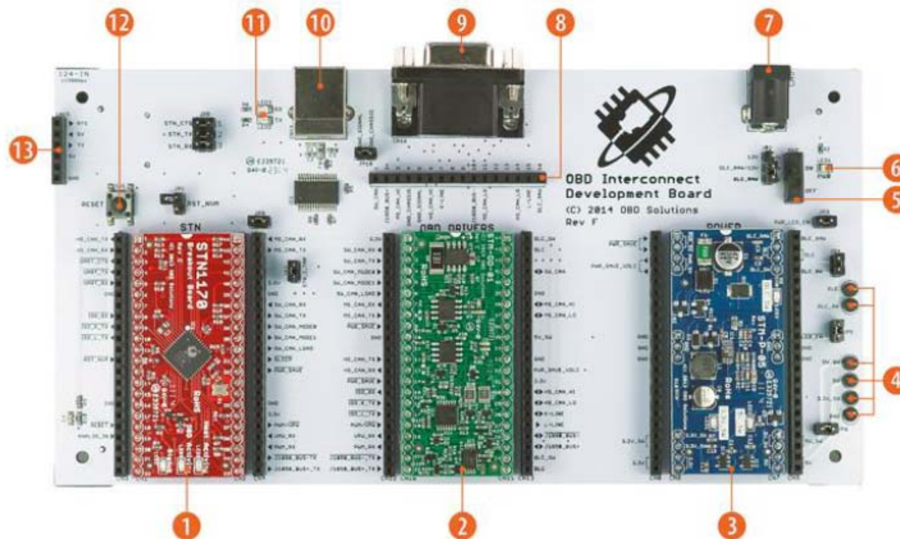
The OBD development kit, shown in *Figure 11 and Figure 12,* supports a USB-to-CAN interface with ELM327 and STN2120 software capabilities related to sending and receiving CAN messages [32]. CAN interfaces at different layers (DB15, RX/TX, and two High/Low) are featured, this is beneficial for evaluation, verification and prototyping of new automotive products. Another key factor that made OBD development kit important for our platform is the multiple sleep/wake-up mechanisms that are important for suspending attacks.

Features:

- UART and USB interfaces
- Power Module
- 16- pin OBD breakout header
- Compatible with ELM327 AT command set for configuration and message transport
- Additional ST commands for configuration and message transport

- Sleep and Wakeup mode

User Interface Diagram



1 - OBD interpreter module (STN1170)
2 - OBD transceiver module
3 - Power module
4 - Test points
5 - Power switch
6 - "Power" LED
7 - Power jack (12VDC)
8 - OBD breakout
9 - OBD port
10 - USB port
11 - UART activity LEDs
12 - Reset button
13 - UART breakout

Figure 12: OBD Development Kit -Interface Diagram

For our platform needs, we used the ELM 327 commands. For the configuration of the OBD development kit we used the following commands.

- **STP 31 and STP 33**

  sending RAW or 15765 (OBD-based) CAN protocol

- **STPRS**

  checking the protocol

- **STSBR 2000000**

  setting the baud rate, In this case 2Mb/s (required for a 500Kbits/sec CAN speed)

- **STI**

  For checking communication with a new baud rate.

- **STWBR**

  For saving the new baud rate.

- **ATSTFF**

  set receive timeout max

- **ATMA**

  receive a message (set flow control, pass and block filters, the old filters are restored when the command terminates)

- **ATSH**

  change the header for a CAN message to be transmitted

- **BD**

  empty the buffer

- **STSLU**

  UART inactivity trigger (on/of switch)

4.1.3 Gingko

Figure 13: Gingko

The Ginkgo USB-to-CAN (see *Figure 13*) is used as an interface adapter for bidirectional data transmission. It is a small device that supports a USB 2.0 interface and a 2-channel CAN

interface. Ginkgo is a powerful tool for validation in laboratory tests or in industrial automotive applications [33].

It can used for data collection or data prepossessing which can later be analyzed.

We chose Ginkgo USB CAN for our platform to send/sniff test packets, analyze and evaluate results.

Main features of this board:

- Support Linux (x86_64), Linux (ARMv7/8, e.g. RPi), MAC OS, Windows, Android
- USB bus power supply 5V and 3.3V output
- SPI Host/Master and slave mode
- Supports USB-to-UART

4.1.4 Raspberry Pi3

Figure 14: Rasberry Pi 3 device

Raspberry Pi 3 (see *Figure 14*) is a single board computer, that supports wireless LAN and Bluetooth connectivity. Due of its small size and low-cost, Raspberry offers a good solution for an embedded Linux platform [34]. We chose to include Raspberry for our platform, as it can support up to four CAN channels connection using the CanberryDual Shield (see *Figure 15*).

Specifications

- quadcore 1.2GHz 64-bit processor
- 1 GB RAM
- 4 USB-2.0 ports
- Supports HDMI
- 40-pin extended GPIO
- Micro SD port for customized operating system and data storage

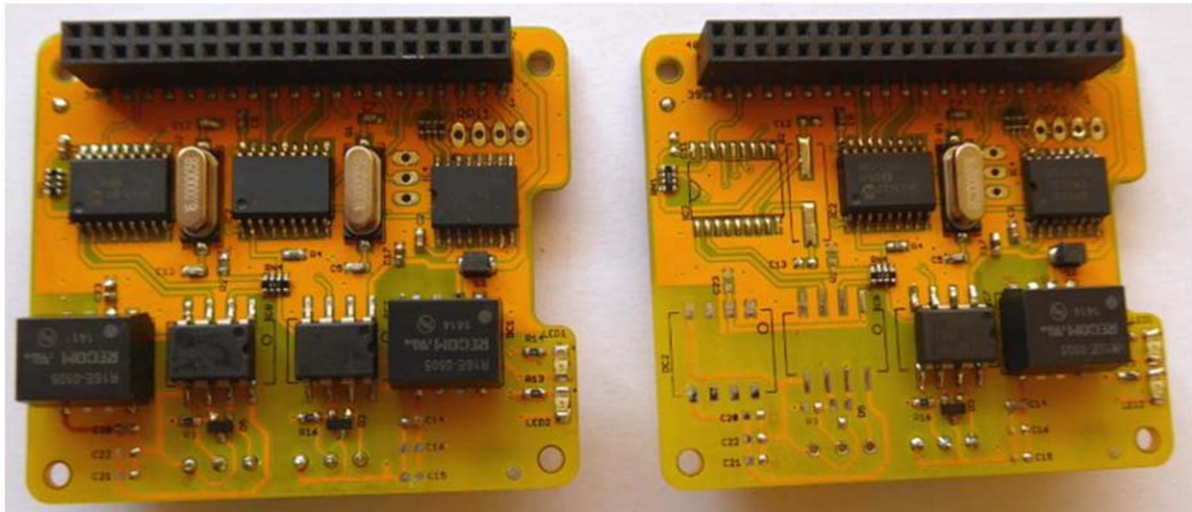


Figure 15: CanBerryDual ISO 2.1

4.1.5 CanBerryDual ISO 2.1

To make Raspberry Pi compatible with our platform we require a CAN BUS shield. In our case we took advantage of CanBerryDual ISO 2.1 which includes an onboard real time clock. Another important benefit is that it provides more than four CAN channels, so it is useful for implementing cross-connections of large CAN networks[35].

4.2 Drivers and Software

4.2.1 Integration Towards Final Platform

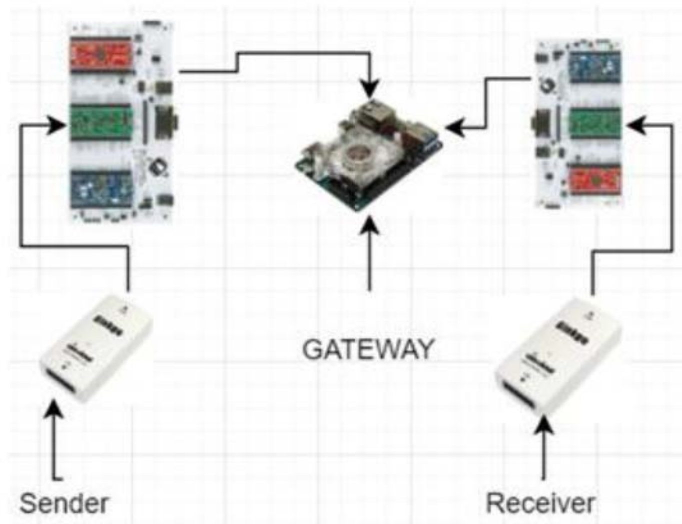


Figure 16: Initial prototype Gateway solution

The platform went through several phases of integration and testing, replacing certain CAN nodes with others, before reaching its final form. For example, in our first tests, we used Gingko as shown in Figure 16 and Arduino, but later replaced them with Raspberry PI 3. *Figure 17* below shows our final platform.

Figure 17: Open distributed embedded platform prototype

For better understanding of Linux CAN utilities (`can-utils`), we have used Raspberry with Canberry dual shield to import traffic in our platform. The Canberry shield also integrates a real time clock that enables correct timing operation in cases of power failure.

Our software code on the Odroid XU3 gateway transmits messages from one CAN to the other, while also enabling the measurement of three different metrics, more specifically energy consumption,

temperature, and frequency of messages on the gateway. Another, fourth metric, round trip-time, is also measured on the packet's return path.

By averaging values for these metrics within fixed sliding windows, appropriate thresholds can be established for detecting DoS. Based on these metrics, we hope to detect when the system is under normal circumstances and when it is under DoS attack.

4.2.2 Concept Validation - Energy Monitor Tool

Concentrating on energy, we are able to demonstrate normal system behavior, as well as behavior during a DoS attack. Energy Monitor tool (and driver code) provides a great way for viewing (or registering) power consumption values. In our platform, we made power consumption tests to evaluate thresholds related to our threat model.

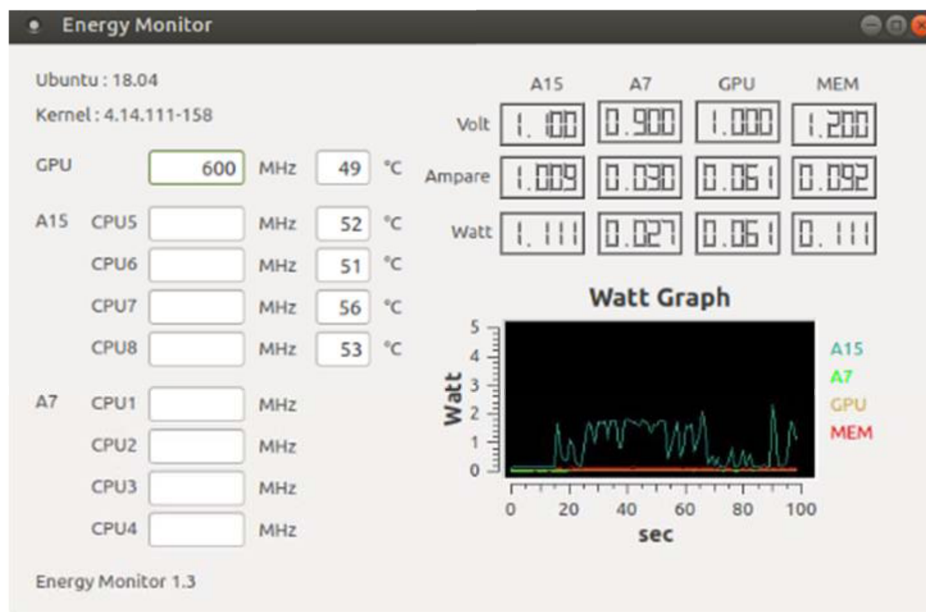


Figure 18: Energy Monitor Tool window

Figure 18 shows the UI of the Energy monitor tool. A15 is the big high-performance CPU, while A7 is the smaller, low-power one.

The energy consumption is measured in three different ways Volts, Ampere and Watts. The code that updates the outputs is listed in *Figure 19*. We have used similar code to register values in our gateway data structures. The situation is similar for temperature zones, which are also read via I2C driver.

```
void DisplaySysInfo::float2string()
{
    a15Volt.sprintf("%.3f", getNode->armuV);
    a15Ampere.sprintf("%.3f", getNode->armuA);
    a15Watt.sprintf("%.3f", getNode->armuW);

    a7Volt.sprintf("%.3f", getNode->kfcuV);
    a7Ampere.sprintf("%.3f", getNode->kfcuA);
    a7Watt.sprintf("%.3f", getNode->kfcuW);

    gpuVolt.sprintf("%.3f", getNode->g3duV);
    gpuAmpere.sprintf("%.3f", getNode->g3duA);
    gpuWatt.sprintf("%.3f", getNode->g3duW);

    memVolt.sprintf("%.3f", getNode->memuV);
    memAmpere.sprintf("%.3f", getNode->memuA);
    memWatt.sprintf("%.3f", getNode->memuW);
}
```

Figure 19: Code for Energy Monitor Tool (I2C interface calls)

4.2.2.1 Code snippets - Receiver Thread on Odroid XU3 Gateway

- ***data0[candata_index0].id = check_rid;***

  The id is stored as integer

- **if ((indx0 > 0) && (c == '\r'))**

  checks indx0 when CAN msg is received through UART

- **if((indx0 == 19) || (indx0 == 30))**

  indx0 is 19 when CAN message received successfully (3 hex ID, and 16 hex body)

- **if(data0[candata_index0].id == 0x123)**

  receiving counters are incremented

- **data0[candata_index0].chid[0] = tmpbuf[0];**

  data0[candata_index0].chid[1] = tmpbuf[1];

  data0[candata_index0].chid[2] = tmpbuf[2];

  ID is stored in a character form

- **candata_index0 = (candata_index0 + 1) % BUFF_SIZE_RECV0;**

  candata_index0_all++;

  index is increased for storing the next CAN message in the circular queue

```
for(i = 0; i < 16; i++) {
    data0[candata_index0].payload[i] = tmpbuf[3+i];
    input[i] = tmpbuf[3+i];
}
hex_binary(input, res);
```

storing the Data payload.

4.2.2.2 Code Snippets - Sender Thread on Odroid XU3 Gateway

- **char oldid[3], newid[3];**

  stores new CAN ID to send to second bus

- **char instb[50];**

  used for sending AT and ST configuration commands, e.g. ATMA

- **char rawp[17];**

  16 hex (4bits) plus '\r' plus '\n'.

- **char atsh[9] = {'A','T','S','H','','\0','\0','\0','\r'};**

  change header of transmitted CAN message

- **sem_wait(&go_send1);**

  waiting for sender thread to receive signal (producer/consumer)

- **strcpy(instb, "ATMA\r");**
  do_write(3, sfd, instb, 5);

  configuration command for receiving new messages

- **i = candata_index1;**

  The candata_index1 shows where the sender pointer is inserted in the circular queue

- **for(j = 0; j < 3; j++)**
  oldid[j] = data0[i].chid[j]-48;

  This loop gives the id for the sender, the number 48 is the offset for character '0' (conversion)

- **atsh[5] = newid[0];**

  atsh[6] = newid[1];

  atsh[7] = newid[2];

  ATSH is used to change the header of the packet that will be sent via OBD development kit to Raspberry2

- **get_raw_data(data0[i].payload, rawp);**

  rawp gets the payload of the message.

- **sem_post(&go_recv0);**

Figure 20 shows the gateway when the gateway code runs and messages received from CAN1 and sent to CAN2. Notice that

- **A15::** energy consumption of the processor A15
- A7:: energy consumption of the processor A7
- **T0, T1, T2, T3::** thermal zone 0, thermal zone 1, thermal zone 2, thermal zone 3.

In addition, `candata_index_all` prints the number of messages that have been sent.

Figure 20: Gateway threads receiving and sending packets

4.2.2.3 Gateway Software on Odroid XU3

The gateway embedded software (approximately 3K lines of C code) initiates two POSIX threads: a) receiver acting as producer of CAN messages, and b) sender acting as consumer in the classical producer-consumer Operating Systems problem. These threads receive (or send CAN messages) from CAN1 (resp. to CAN2) and require two POSIX semaphores (initialized in the main) in order to manage a shared circular queue (bounded buffer) that stores messages. The receiver fills and the sender empties this queue concurrently.

In addition, the main program must configure the two serial USB-to-CAN interfaces implemented with the two OBD development kits. These interfaces point towards the RPI1 sender on CAN1 and RPI2 receiver on CAN2.

Serial programming is performed using TTY commands, e.g. termios `cf` and `tc` commands, such as `tcgetattr` and `tcsetatt`; this code builds upon a simplified version of an open source serial terminal, cf.[36]

After configuring the serial port, including baud rate, the STN2120 device on each OBD Development Kit (USB0, USB1 corresponding to file descriptors `sfd0`, `sfd1`) must be programmed using appropriate AT (ELM 327) and ST commands in the right order; Notice that the RPI2 receiver device, which receives messages from CAN2 and sends towards the return path via CAN0, must be configured first; interface programming is indicated by blinking LEDs that identify the CAN messages packets arriving.

```
static void init_config_L(int sfd)
{
      char * instb = (char *) malloc (50 * sizeof(char));
      //strcpy(instb, "STP 33\r"); // CAN std 15765 (predefined IDs, e.g. 7DF)
      strcpy(instb, "STP 31\r"); // raw CAN protocol
      do_write(4, sfd, instb, 7);
      strcpy(instb, "STPRS\r"); // checking protocol
      do_write(4, sfd, instb, 6);
      strcpy(instb, "STSBR 2000000\r"); // setting serial baud rate
      do_write(5, sfd, instb, 14);
```

```
    strcpy(instb, "ATS0\r"); // spaces off
    do_write(4, sfd, instb, 5);
    strcpy(instb, "ATH1\r"); // print headers
    do_write(4, sfd, instb, 5);
    strcpy(instb, "ATAL 00\r"); // no long messages (above 8 byte)
    do_write(4, sfd, instb, 8);
    strcpy(instb, "STCMM1\r"); // continuous monitoring
    do_write(4, sfd, instb, 7);
    strcpy(instb, "ATMA\r"); // waiting to receive
    do_write(4, sfd, instb, 5);
}
```

4.2.2.4 RPI Setup - RPI Sender (RPI1) & Receiver (RPI2)

The following commands configure the Raspberry Pi Can0 and Can1 interfaces on both RPI1 and RPI2.

```
sudo ip link set can0 up type can bitrate 500000
sudo ip link set can1 up type can bitrate 500000
sudo ifconfig can0 txqueuelen 1000000
sudo ifconfig can1 txqueuelen 100000
```

4.3 Troubleshooting Guide

The following troubleshooting guide is a short, very incomplete list of useful signs and symptoms that help us indicate what is wrong and why. (A full list is also available from the author.)

Setting up Devices (OBD Dev Kit, Odroid XU3/4, RPI)

1. Power OBD Dev Kit - Switch button to get power externally.
2. Odroid XU3 using Ubuntu 18.04.1 (20181203), kernel 4.14.1.
3. RPI using 2019-04-08-raspbian, kernel 4.9, update kernel support for SPI.
4. No changes to bit timing, all default values for CNF1, CNF2, CNF3 registers
5. For baud rate
    - set baud rate on OBD dev kit via Minicom to 2M (STP33, STPRS, STSBR 2000000)

- set same rate for serial (Minicom)
- check by pressing enter
- exit via CTRL-A SHIFT-Z Q (quit no reset)

6. Issues with our Serial Code `(sc.c)`. Interfaces must be configured the right way, check leds, first receiver must blink, then transmitter

4.4 Experimental Evaluation

4.4.1 Towards Deriving a DoS Metric: Example with Energy Metric

Initially, for concept validation, we operate the Odroid XU3 gateway with increasing traffic. We have observed that with increasing injection rate, power consumption and energy levels rise (compare *Figure 21*, *Figure 22)*. This convinces us that deriving an energy metric for detecting DoS attack could be beneficial.

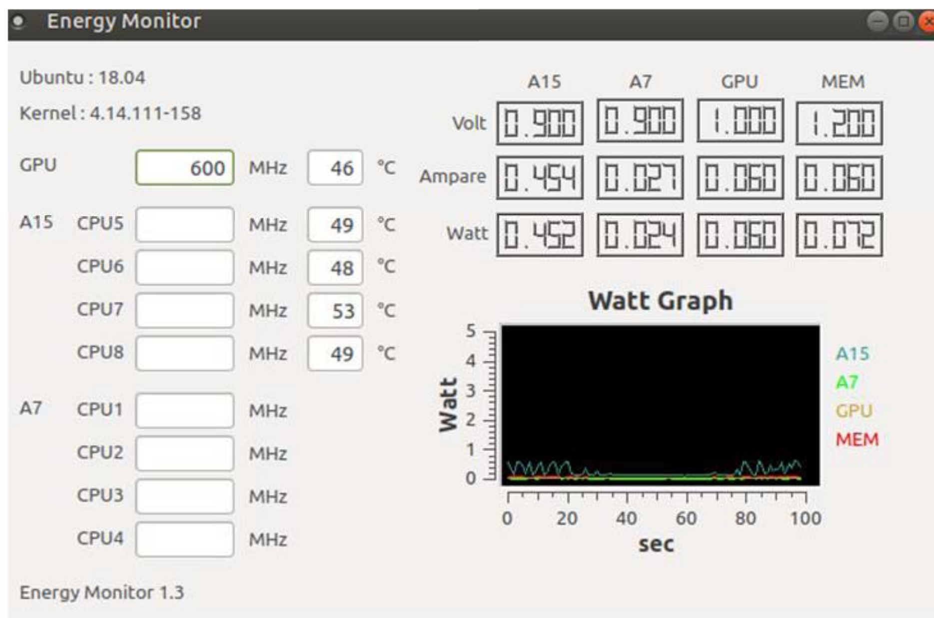


Figure 21: Scenario with normal traffic

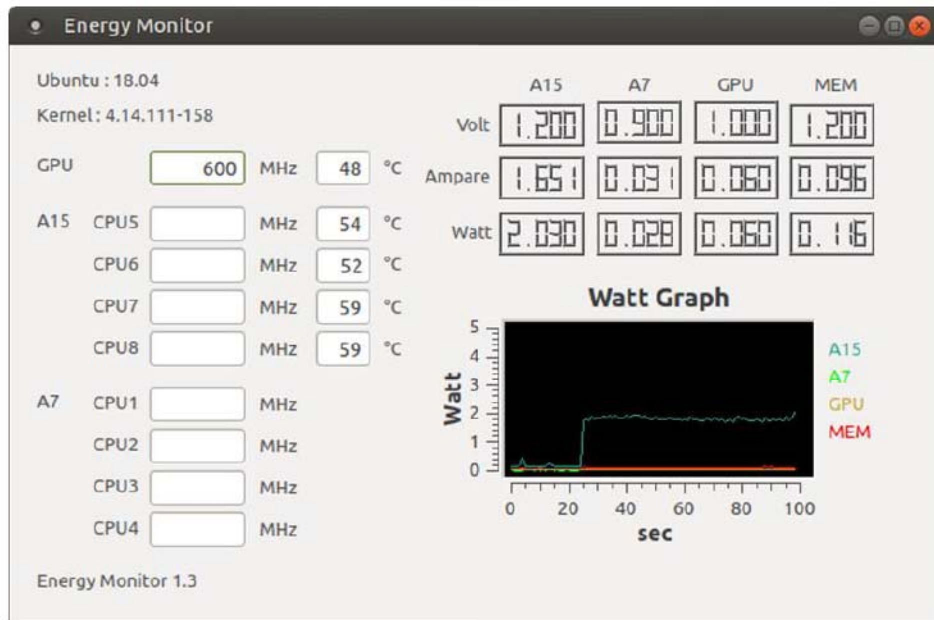


Figure 22: Scenario with network under attack

4.4.2 Detailed Results

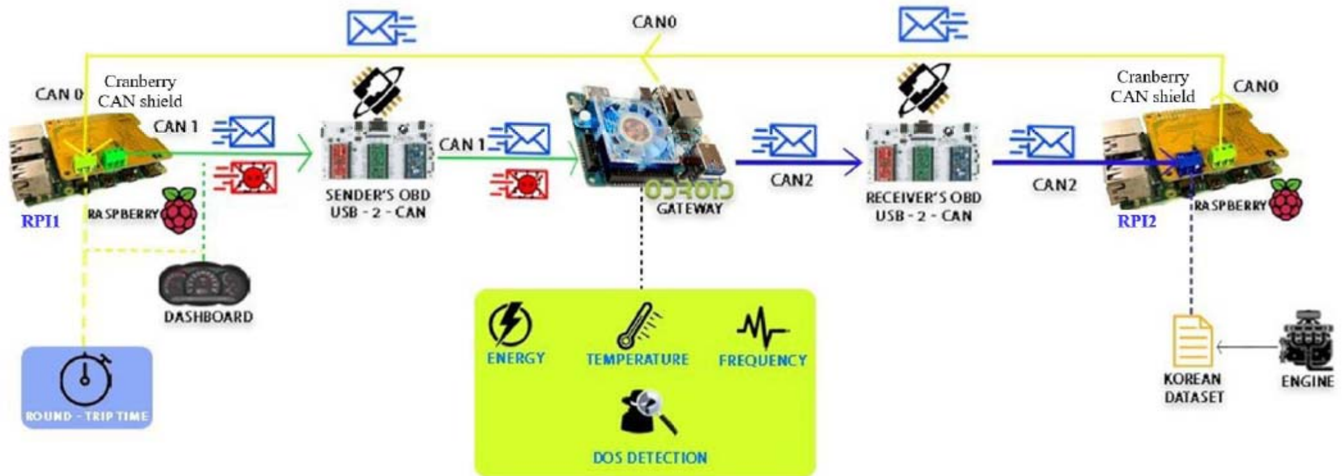


Figure 23: Experimental framework of our platform

Figure 23 describes our experimental framework consisting of a distributed embedded platform prototype that targets traffic monitoring across multiple CAN networks. This ecosystem interconnects multiple Raspberry Pi devices (e.g., RPI1, RPI2) to an Odroid XU3 device which serves as a gateway node between two CAN networks (CAN1 and CAN2).

CAN interconnection is based a) for Raspberry Pi, on IndustrialBerry's CANberry Dual V2.1 device, and b) for Odroid XU3, on two (incoming/outgoing) USB-to-CAN interfaces using Scantool OBD Development Kit.

Incoming and outgoing CAN interfaces at the gateway are controlled by different threads. Our embedded software toolchain uses a) for Odroid XU3, an extended serial terminal that uses multithreaded code to handle incoming/outgoing connections; configuration and CAN message send/receive functions use appropriate USB-to-serial STN2120's ELM327 AT, and ST commands, and b) for RPI, code based on Linux CAN-utils tools that is described below, separately for the *RPI sender and RPI receiver*.

4.4.2.1 RPI Sender (RPI1)

During normal operation, RPI2 (CAN2) carries actual engine traffic (based on an actual Korean car dataset), while at the same time RPI1 packet requests related to dashboard (e.g. engine speed, RPM, temperature etc.) departing from RPI1 (CAN1), are received via the Gateway by RPI2 (CAN2), and answered back to RPI1 (making a round trip). In accordance with our threat model, we consider a denial-of-service (DoS) from CAN1 (by modifying the parameter `no_malicious` in our script) and examine different metrics that can be used to detect the attack.

The following bash script injects malicious and legitimate traffic from RPI1 on CAN1 (towards the gateway).

```
#123 is legitimate -> becomes 7df at gateway
#007 is malicious -> stays same at gateway
echo "test_all_engine_and_maliciou"
no_malicious=128
sleeptime=`bc <<< "scale=6; 0.5/$no_malicious"`
echo $sleeptime
c=1; while [ $c -le 60 ] ; do
      let c=$c+1;
      #send malicious
      iteration=1
      while [ $iteration -le $no_malicious ]; do
```

```
        cansend can1 007#08.08.08.08.08.08.08.08;
        sleep $sleeptime;
        let iteration=$iteration+1
    done
    #send legitimate
    cansend can1 123#08.01.01.01.01.01.01.01;
    sleep 0.25;
    cansend can1 123#08.01.01.01.01.01.01.01;
    sleep 0.25;
done
```

At the same time, RPI1 listens for response messages on CAN0, with timing enabled via the command, This enables computing the round-trip time of messages sent to (and received) from the engine.

```
candump can0 -t
```

4.4.2.2 RPI Receiver (RPI2)

RPI2 listens for request messages on CAN0, with specified ID and sends responses on another CAN ID (we use raw CAN ID 123).

```
echo "Start"
c=1;
i=0;
while [ $c -le 100000000 ] ;
do
    let c=$c+1;
    candump can1,7DF:FFF -n 1 -t a
    cansend can0 123#08010$i010$i01010$i
    let i=$i+1
    if [ $i -eq 10 ] ; then
        let i=0
    fi
done
exit
```

The above script creates a ping-pong between RPI1 and RPI2, however packets making the round-trip from (RPI1 to RPI2 and back to RPI1) are delayed not only due to malicious packets arriving on the

gateway, but also due to actual engine traffic (Korean attack-free automotive dataset from Hyundai's YF Sonata, c.f. emulated on CAN2). This traffic is injected to CAN2 via an independent interface of the OBD Development Kit. The exact timing logs of 988,987 packets make this emulation very realistic.

At gateway-level, we can detect the DoS attack by using metrics and setting appropriate thresholds related to the Cortex-A15 energy consumption (available from integrated INA231 sensors), and four temperature gradients on the same chipset (available from integrated sensors). Notice that our gateway code (process and threads) runs solely on Cortex-A15. The corresponding energy for A7 remains steady between 27-30 μW. *Figure 24* and *Figure 25* show results (averaged over multiple runs) for two different buffer sizes of the circular queue (buffer 1 and 100). These results show that for both buffer sizes, average instant energy on Cortex-A15 sharply increases between the rates of 1-10 malicious messages/sec. Hence, this metric can be used as a metric for detecting denial of service, provided that the correct threshold within this range of malicious message rates is set. This threshold must be before the energy saturation point which defines our system limits before messages arrive too fast and cannot further be accepted by the UART-to-CAN interface.

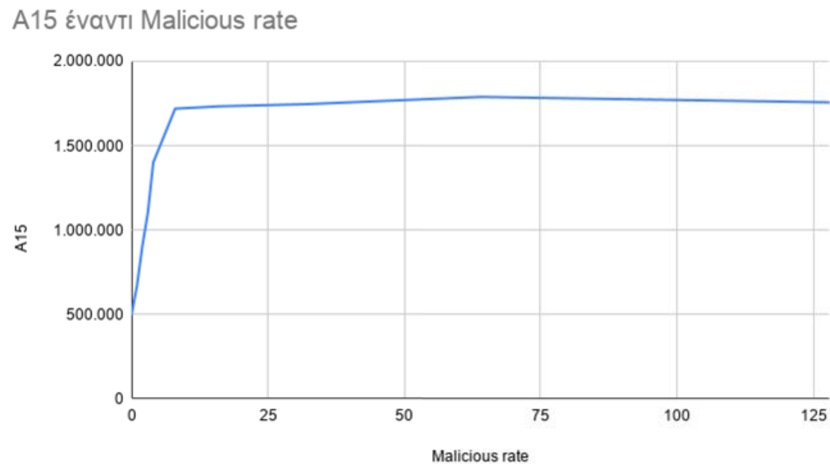


Figure 24: Average instant energy on Cortex-A15 – diff. rates of malicious packets (buffer size 1)

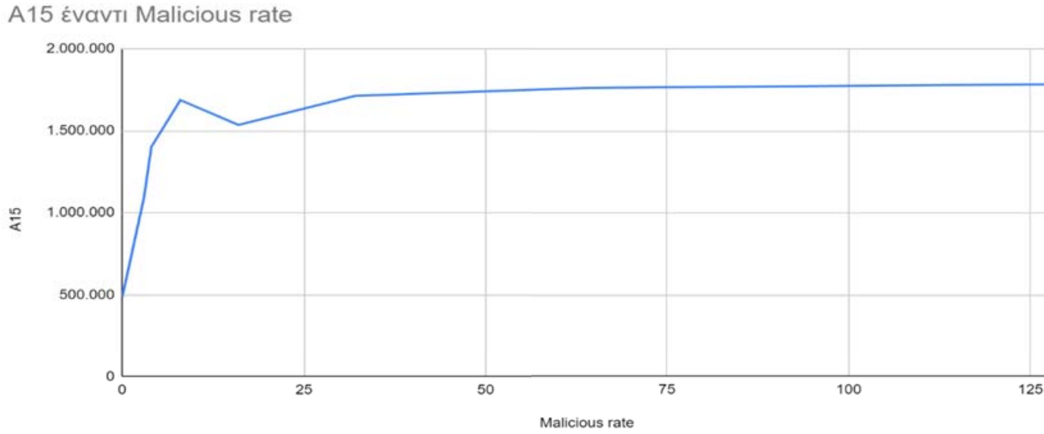


Figure 25: Average instant energy on Cortex-A15 - diff. rates of malicious packets (buffer size 100)

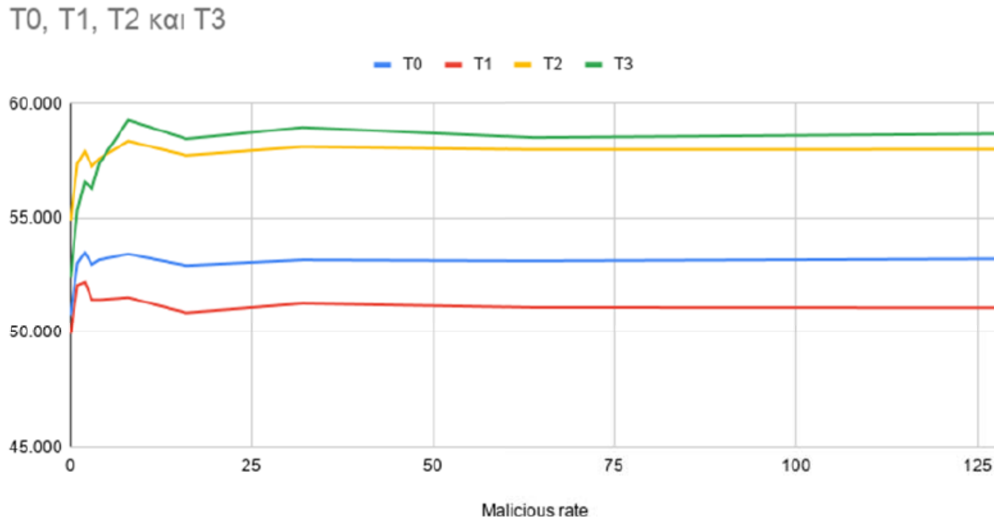


Figure 26: Average temperature of four available thermal zones

In *Figure 26,* we inspect the average temperature of four available thermal zones versus increasing rate of receiving messages. We monitor an abruptly increase of temperature in thermal zones 2 and 3, in the range of 1-10 malicious messages/sec. Hence, this metric can be used as a metric for detecting denial of service, provided that the correct threshold within this range of malicious message rates is set. In fact, this metric should be used in conjunction with the energy metric to provide more reliable results

Finally, we examine variations of round-trip time (RTT) by monitoring delays of sequences of packets that originate from RPI1 (a.k.a. dashboard), flow to RPI2 (a.k.a. engine) via Odroid XU3 (a.k.a.

gateway) and return back to RPI1 in a feedback loop. Hence, in *Figure 27*, we show the round-trip time (RTT) of messages arriving at RPI1 versus the injection rate of malicious messages. We observe an increasing latency proportional to the malicious messages for both buffer sizes. This indicates that RTT metric can be used with a sliding window approach to detect DoS attack at the CAN0 injection point, hence possibly throttling down traffic, and thus alleviating the gateway from handling enormous traffic.

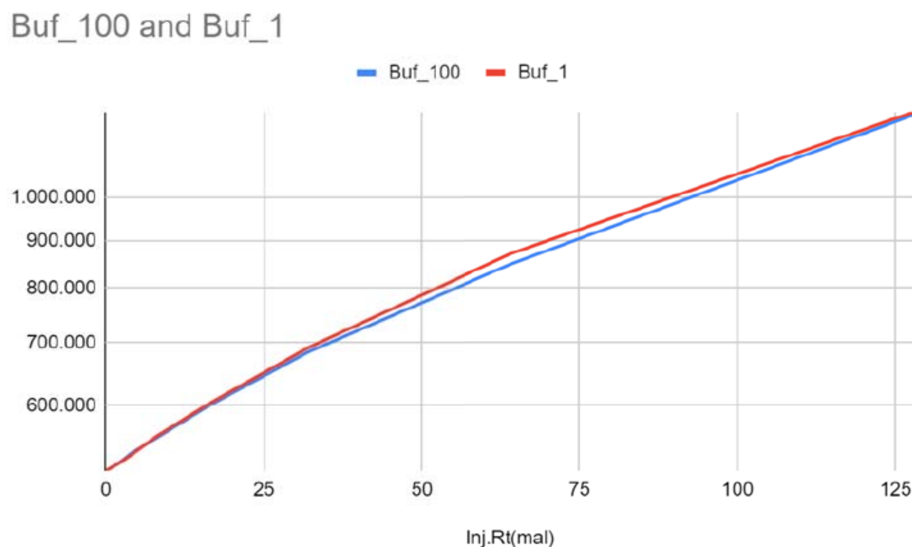


Figure 27: Round-trip time (RPI1 to RPI2 and back in feedback loop)

5. Conclusions

Vehicles have come a long way and evolving with fast pace since they were discovered. However, even advanced modern vehicles are still prone to several attacks and the CAN philosophy does not offer a security solution. The attacker can easily manipulate and gain control over a system or can make the system unavailable by simply adding enormous high priority traffic to the real-time network.

Within this context, we design, implement and validate an open, distributed embedded platform prototype which consists of a Gateway that can isolate one or more critical in-vehicle networks (e.g. CAN bus) in case of a Denial-of-Service (DoS) attack.

In detail, the platform includes: an ODROID XU3 used as the Gateway (with two UART-to-CAN OBD Development Kits), and two Raspberry Pi (RPI) devices (with CANberry CAN interfaces). In prior platform incarnations we have used Gingko USB-to-CAN devices and multiple Arduino devices (with DFRobot CAN shield), however these initial platforms were used for concept validation, since they do not support multiple CAN interfaces.

In our experimental framework, we consider DoS attack scenarios towards a system that emulates actual engine traffic (RPI2) and define different metrics for detecting a DoS attack. The attack originates from another node (RPI1) which emulates a dashboard device requesting for engine data. Our metrics are based on evaluating energy, temperature, or frequency profiles on the gateway, or alternatively measuring round-trip times on the receiver node (in a feedback loop). Extensive experimental results indicate that it is possible to use our metrics: a) gateway profiles for energy A15, temperature zones, frequency by counting messages, and b) round-trip time at RPI1 to shut down, throttle down, or sleep a CAN interface under attack.

6. Future Work

Further work relates to evaluating tradeoffs in the accuracy and effectiveness of the proposed metrics in detecting actual attacks. Accurate prediction of an attack results in shutting down, throttling down, or sleeping the appropriate outgoing interface, thus safeguarding the engine ECUs. In addition, we hope to secure our experimental platform from other types of threats, such as spoofing and replay.

It is also interesting to experiment with multiple CAN interfaces (beyond two) and provide faster implementations of our gateway using concurrent lock-free queues. These implementations extend our producer consumer philosophy used by the two threads that control reception and transmission of messages in different CAN buses

References

[1]. *Real-time operating system*, Wikipedia. (Sept 26, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Real-time_operating_system

[2]. *Time trigger protocol*, Wikipedia. (Feb 18, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Time-Triggered_Protocol

[3]. *Time trigger Ethernet*, Wikipedia. (Feb 16, 2018) [Online]. Available: https://en.wikipedia.org/wiki/TTEthernet

[4]. *CAN_bus*, Wikipedia. (Jan 5, 2019) [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus

[5]. *Most,* Wikipedia. (Dec 3, 2018) [Online]. Available: https://en.wikipedia.org/wiki/MOST_Bus

[6]. *CAN_bus#Frames*, Wikipedia. (Feb 3, 2018) [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus#Frames

[7]. *Local_Interconnect_Network*, Wikipedia. (Aug 15, 2018) [Online]. Available: https://en.wikipedia.org/wiki/Local_Interconnect_Network

[8]. *Flex Ray*, Wikipedia. (Oct 27, 2028) [Online]. Available: https://en.wikipedia.org/wiki/FlexRay

[9]. *Functional_safety*, Wikipedia. (Jan 18, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Functional_safety

[10]. *ISO_26262*, Wikipedia. (Oct 22, 2018) [Online]. Available: https://en.wikipedia.org/wiki/ISO_26262

[11]. *Cyber_security_standards*, Wikipedia. (Aug 5, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Cyber_security_standards

[12]. *Failure_modes_effects*,_and_diagnostic_analysis, Wikipedia. (Feb 12, 2018) [Online]. Available: https://en.wikipedia.org/wiki/Failure_modes,_effects,_and_diagnostic_analysis

[13]. *Mixed_criticality*, Wikipedia. (Dec 15, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Mixed_criticality

[14]. *Replay_attack*, Wikipedia. (Aug 9, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Replay_attack

[15]. *Spoofing_attack*, Wikipedia. (Feb 11, 2018) [Online]. Available:

https://en.wikipedia.org/wiki/Spoofing_attack

[16]. *Denial-of-service_attack*, Wikipedia. (Jan 2, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Denial-of-service_attack

[17]. *Denial-of-service_attack#Distributed_DoS*, Wikipedia. (Jan 2, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Denial-of-service_attack#Distributed_DoS

[18]. Rola, N. Japkowicz, and S. Leblanc, "Prevention of Information Mis-translation by a Malicious Gateway in Connected Vehicles," in Proc. 14th Int. Conference on Privacy, Security and Trust (PST 2016), Dec. 2016.

[19]. K. Daimi, M. Saed, S. Bone, and J. Robb, "Securing Vehicle's Electronic Control Units", in 12th International Conference on Networking and Services (ICNS, 2016), June 26-30, Lisbon, Portugal.

[20]. M. Wolf, A. Weimerskirch, and C. Paar, "Security in automotive bus systems," 2004.

[21]. G. Xie, G. Zeng, R. Kurachi, H. Takada, R. Li, "Gateway modeling and Response time analysis on CAN clusters of automobiles", in 17th EEE International Conference on High Performance Computing and Communications, Aug 2015, New York, USA.

[22]. R. Kammerer, B. Frömel, and A. Wasicek, "Enhancing security in can systems using a star coupling router," in 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), June 2012

[23]. T. Hoppe, S. Kiltz, J. Dittmann, "Applying Intrusion Detection to Automotive IT – Early Insights and Remaining Challenges" (JIAS) Journal of Information Assurance and Security 4, Jan 2009

[24]. S.N Narayanan, S. Mittal and A. J. Taylor, "OBD SecureAlert: An Anomaly Detection System for Vehicles", in Proc. of the 1th IEEE on Smart Service Systems (SmartSys 2016). IEEE, May. 2016

[25]. M. Weber, S. Klug, E. Sax, and B. Zimmer, "Embedded Hybrid Anomaly Detection for Automotive CAN Communication", in Proc. of the 9th European congress on Embedded Real Time software and systems (ERTS 2018). Jan 2018, Toulouse, France

[26]. M. Marchetti and D. Stabili, "Anomaly detection of CAN bus messages through analysis of ID sequences" in IEEE Intelligent Vehicles Symposium (IV). IEEE, June. 2017, Los Angeles, CA.

[27]. Ulf E. Larson, Dennis K. Nilsson, E. Jonsson, "An Approach to Specification-based Attack Detection for In-Vehicle Networks", in IEEE Symposium on Intelligent vehicle, June 2008

[28]. P. Kleberger, T. Olovsson, and E. Jonsson, "Security Aspects of the In-Vehicle Network in the Connected Car" in IEEE International Conference on Intelligent vehicle, June 2011

[29]. Zhou Qin and Fei LiAn "An intrusion defense approach for vehicle electronic control system" Conference: International Conference on Communication and Electronic Information Engineering, 2016

[30]. Taylor, N. Japkowicz, and S. Leblanc, “Frequency-based anomaly detection for the automotive CAN bus,” in Proc. 2015 World Congress on Industrial Control

[31]. *Odroid*, Wikipedia. (Feb 14, 2018) [Online]. Available: https://en.wikipedia.org/wiki/ODROID

[32]. *OBD-development-kit*, Scantool. Available: https://www.scantool.net/obd-development-kit/

[33]. *Ginkgo-usb-can*, *Viewtool*. Available: http://www.viewtool.com/index.php/en/27-2016-07-29-02-13-53/44-ginkgo-usb-can-9

[34]. *Raspberry_Pi,* Wikipedia. (Aug 1, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Raspberry_Pi

[35]. CanberryDual_isolated, Industrialberry. Available: http://www.industrialberry.com/canberry-v-2-1-isolated/

[36]. *Serial Console,* Sourceforce. [Online]. Available: https://sourceforge.net/projects/serialconsole/files/serialconsole.

[37]. *Car-Hacking Dataset*, HCRL. [Online]. Available: https://sites.google.com/a/hksecurity.net/ocslab/Datasets/CAN-intrusion-dataset

[38]. R Buu R. Buttigieg, M. Farrugia, and C. Meli, " Security Issues in Controller Area Networks in Automobiles," in Proc. 18th International Conference on Sciences and techniques of Automatic Control and Computer engineering (STA). IEE 2017. Conf., Monastir, Tunisia, 2017.,

[39]. *Controller Area Network (CAN) Overview ,* National Instruments.(Mar, 5, 2019), [Online]. Available: https://www.ni.com/en-us/innovations/white-papers/06/controller-area-network--can--overview.html

[40]. CAN bus (Controller Area Network), EMC-EV.(Sept, 8, 2015), [Online]. Available: http://www.flexautomotive.net/EMCFLEXBLOG/post/2015/09/08/can-bus-for-controller-area-network

[41]. Jiannis Vougioukalos,"An embedded platform for developing data protection protocols on smart vehicles” , Thesis, Dept. of Informatics Engineering , HMU, Heraklion, 2018. [Online]. Available: https://apothesis.lib.teicrete.gr/bitstream/handle/11713/9024/VougioukalosIoannis2018.pdf?sequence=1&isAllowed=y