HOCHSCHULE BREMERHAVEN

MASTER THESIS

Automotive Hacking

Author:
Patel DHARMESH

Supervisor:
Prof. Dr. Werner UWE

A thesis submitted in fulfillment of the requirements
for the degree of Master of Embedded Systems Design

in the

Hochschule Bremerhaven
Department of embedded system design

July 17, 2022

Declaration of Authorship

I, Patel DHARMESH, declare that this thesis titled, "Automotive Hacking" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 17.07.2022

"Life is like riding a bicycle. To keep your balance, you must keep moving."

Albert Einstein

HOCHSCHULE BREMERHAVEN

Abstract

Embedded Systems Design
Department of embedded system design

Master of Embedded Systems Design

Automotive Hacking

by Patel DHARMESH

Modern cars include more and more features that first emerged from the consumer electronics industry. Technologies like Bluetooth and Internet-connected services found their way into the vehicle industry. The secure implementation of these functions presents a great challenge for the manufacturers, products originating from the consumer industry can often not be easily transferred to the safety-sensitive traffic environment due to security concerns. Car manufacturers are at a crossroads, as they're trying to deliver the features that customers want, while keeping safety and security. However, common automotive interfaces like the diagnostics port are now also used to implement new services into the car. Unified Diagnostic Services (UDS) specifies how diagnostic communication should be handled between a diagnostic tester and an on-vehicle ECU. This thesis wants to emphasize new threats that appear due to the firmware update services provided by the Unified Diagnostic Services (UDS). Potential attack vectors as well as proof-of-concept exploits will be shown and the implications of security breaches on the safe state of the vehicle will be investigated.

Acknowledgements

Firstly I want to thank Phalgun Upadhyaya for the idea to this project, which fit perfectly as a Master's thesis as the amount of work, could be customized to a large extent. Phalgun Upadhyaya has been the primary supervisor for this project; always answering every question I had and helped me structure the solution to this thesis by giving me relevant examples to follow and continuously discussing the structure. I also want to especially thank Neeraj Dharmadhikari and Ronak Ghodasara who provided relevant feedback continuously. The amount of support and advice I have received from the team at Matrickz meant to some degree I did not need as much advice from my university supervisor Prof. Dr. Werner Uwe but it felt like if I would have needed more support from him I would have received it. I would also like to thank my committee members for letting my defense be an enjoyable moment. Finally I would like to thanks to the Matrickz Gmbh for giving me the opportunity and platform to do the master thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

CAN	**C**ontroller **A**rea **N**etwork
LIN	**L**ocal **I**nterconnect **N**etwork
OBD2	**O**n **B**oard **D**Diagnostics 2
UDS	**U**nified **D**iagnostic **S**ervices
DoIP	**D**iagnostics **O**ver **I**nternet **P**rotocol
ECU	**E**lectronic **C**ontrol **U**nit
GPS	**G**lobal **P**ositioning **S**ystem
MP3	**M**oving **P**icture **E**xperts **G**roup **L**ayer-3
USB	**U**niversal **S**erial **B**us
ROM	**R**ead **O**nly **M**emory
CD	**C**ompact **D**isk
DVD	**D**igital **V**ideo **D**isk
ABS	**A**ntilock **B**raking **S**ystem
ADAS	**A**dvanced **D**river **A**ssist **S**ystem
ACC	**A**daptive **C**ruise **C**ontrol
EMS	**E**lectronics **M**anufacturing **S**ervices
OSI	**O**pen **S**ystem **I**nterconnection
CSMA	**C**arrier **S**ense **M**ultiple **A**ccess
MOST	**M**edia **O**riented **S**ystems **T**ransport
IEEE	**I**nstitute of **E**lectrical and **E**lectronic **E**ngineers
EEPROM	**E**lectronically **E**rasable **P**rogrammable **R**ead only memory
PBL	**P**rimary **B**ootloader
DTC	**D**iagnostic **T**rouble **C**odes
SAE	**S**ociety of **A**utomotive **E**ngineers
KWP2000	**K**eyword **P**Protocol 2000
DBC	**D**atabase of **C**onversion
GUI	**G**raphical **U**ser **I**nterface
OEM	**O**riginal **E**quipment **M**anufacturer
PoC	**P**roof **O**f **C**oncept

Chapter 1

Introduction

1.1 Motivation

The development of automobiles has been subject to constant change since its first day. None of these changes, however, were as striking as the incorporation of software. The introduction of software development processes in a domain formerly shaped by (physical) engineering forced the industry to adopt software engineering processes to that domain. Key principles of this shift revolve around reusability and saving costs. Since the turn of the millennium, scientific contributions to software security of cars have been published [1]. Earlier publications are practically non-existent due to the scarcity of software in vehicles back then. One of the first systematic analyses of attacks on automotive (software) security [7] describes the prevailing attacks in the automotive sector as either theft or modification of critical components: for example, an attacker would like to achieve financial gain by stealing the car or valuable components. Modification refers to the car owner that would like to change components (tuning), for example in order to increase the value of the car (reduced mileage) or decrease it for taxation reasons (increased mileage). The analysis also mentions that attackers want to steal competitors' expertise and intellectual property.

As far as security experts are concerned, it should be noted that car attackers do not target cars the same way as they attack desktop computer systems because cars use different networks (CAN, LIN, K-Line, FlexRay, etc.), protocols (OBD2, ISO-TP, UDS, DoIP, etc.) and architectures (e.g. AUTOSAR) [10]. In addition, vehicles often contain obsolete legacy mechanisms with unsecured and unencrypted protocols (e.g., Controller Area Network (CAN)) in their system design, because they were originally not designed in accordance with today's security principles [3]. Secure automotive network architectures were not prioritized in the past due to the general prejudice that cars are secure due to their technical complexity (security by obscurity). Sluggish development processes, lack of standard guidelines, and low societal pressure, due to little attack experience in practice, lead to a rather slow transformation of automotive development processes, which systematically implement security by design. Most existing countermeasures against cyber-attacks, e.g., the use of message cryptography for encrypting, authenticating, or randomizing vehicle-level network messages, focus on concrete attacks and do not consider the complexity of the access options offered by modern vehicles, as shown by our study in 2019 [14].

1.2 Previous Work

To avoid different attacks which possibly happen on vehicles, some evaluation process is defined for security access services [9]. At first, the regulator region was identified so that the network ID can answer each analytic transmission. At that point network ID identifies the data compared to a specific CAN ID. For recognized CANID, checked if there is any security access administration that exists or not, and the relating security access level is stamped. After the completion of these steps, they calculated the entire length of the seed value used for each detected security access service. A Controller Area Network (CAN) transport is a standard bus network for cars and the plan of this transport standard is made so that, it permits microcontrollers and different gadgets to speak with one another's applications without the presence of the host PC. The primary focus was to know the attacker's intention if he/she had the option to vindictively impact the vehicle's inner matrix. In the automotive field CAN bus security is the major challenge because all the ECUs are connected to the common CAN bus and these CAN packets are broadcast to all nodes both physically and logically, it creates several problems when there is hostile components are present in the network [6]. This revealed that the remarkable assaults don't need a total comprehension of a solitary segment of the vehicle. The scope of legitimate CAN bundles is little. Hence simple fuzzing of packets can make significant damage.

1.3 Delimitations

This thesis does not cover the whole ECU software update process. It focuses on the software interface of ECUs for performing diagnostic communication and UDS over CAN communication with an external node. Matters concerning cryptographic key management, as well as the distribution, production, and signing of software, are not examined. Due to a confidentiality agreement, some additional details have been left out from the report of experiment results. Namely, it is the calculated estimations on how much time it would take for particular attacks to succeed, as well as measured timespans of experimental attacks.

To perform a thorough security analysis, the experiments performed in this thesis build internally and following the standard UDS protocol guideline. These would not necessarily be available to a real adversary, who would have to spend a considerable amount of time extracting the details of communicating to an ECU. Also, for the sake of convenience, the research uses some software code developed internally to perform accurate software updates. The focus of this research is mainly on whether an unauthorized person can break the security using the UDS standard services and update malfunctioning software to the ECU. Several attacks are performed as a part of this thesis. However, none of them attempt to attain the most valuable data. This is a very interesting and important question in terms of the purpose of this research. Building upon the results of this thesis, it could be explored as the next step. The goal of the experimental attacks described in Chapter 3 is to program the ECU with arbitrary software using the UDS services. Also, the ECU hardware used in this thesis is not the automotive standard. However, it can demonstrate the same mechanism and functionality used in real-time ECUs.

1.4 Outline

Chapter 2, introduces the background for this thesis. It explains the concept of an different communication protocols used in modern car. Also, it explains the concept of the ECU, as well as how its software is updated. The overview of the UDS protocol and its security related services is explained in this chapter.

Chapter 3, explains the purpose and goal of the thesis, and how the implementation has been done to achieve the goal.

Chapter 4, describes the testing and validation of the implementation. Also, it captures all the results.

Finally, Chapter 5 concludes the findings of the research, evaluates them, and outlines topics for future work.

Chapter 2

Background

2.1 What Is Car Hacking?

Car hacking refers to all of the ways hackers can exploit weaknesses in an automobile's software, hardware, and communication systems in order to gain unauthorized access.

Modern cars contain a number of on board computerized equipment, including an electronic control unit (ECU), a controller area network (CAN), Bluetooth connections, key fob entry, and more. Many also connect to central servers through the internet. And each of these different computerized technologies can be attacked in a variety of ways.

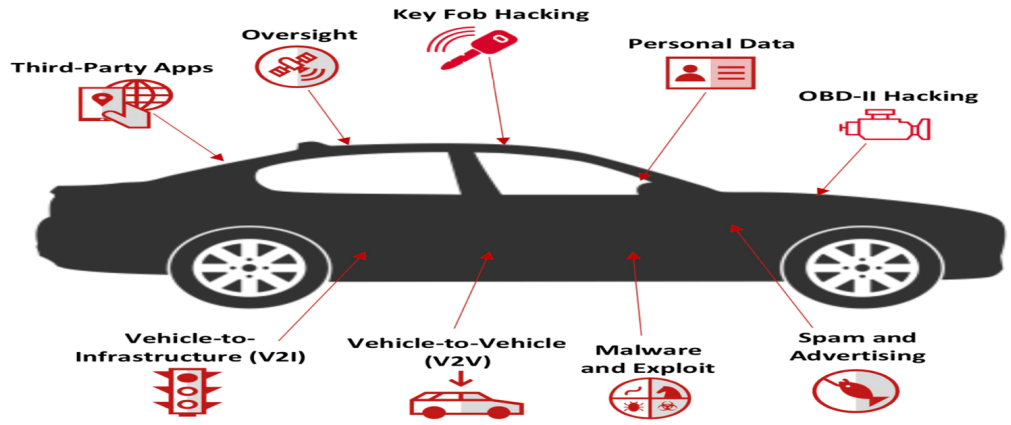

FIGURE 2.1: Modern Cars features which are most vulnerable to hacking [13]

2.1.1 Different ways Hackers can control your car?

- Tire pressure monitoring systems
- Disabling brakes
- Manipulating vehicle diagnostics
- Changing the time, a song on the radio, or GPS destination
- MP3 malware
- Forced acceleration

- Extended key fob range
- Driving data downloads
- Smartphone access
- Turning on heat in the summer or air conditioning in the winter
- Windshield wiper control

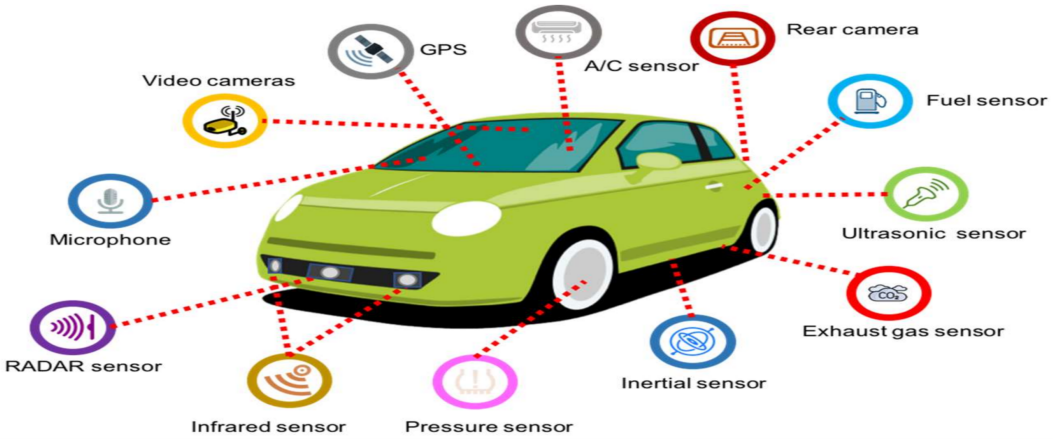

FIGURE 2.2: Most used sensors in the modern cars [8]

2.1.2 Common attack surfaces linked to the infotainment console

- Bluetooth
- Wi-Fi
- USB Ports
- SD card ports
- CD-ROM/DVD-ROM
- Touch Screen and other inputs that allows you to control the console
- Audio jack
- Cellular Connection, GPS, etc.

2.2 Communication Buses Used In Automotive Applications

2.2.1 CAN Bus

The Controller area network or CAN protocol is a method of communication between electronic devices embedded in a vehicle, such as the engine-management systems, active suspension, central locking, air conditioning, airbags, etc. The idea was initiated by Robert Bosch GmbH in 1983 to improve the quality and safety of automobiles, enhancing automobile reliability and fuel efficiency.

The protocol set rules by which electronic devices can exchange information with one another over a common serial bus. It reduced the wiring connections and the overall complexity of the system.The standard technology of the time asynchronous transmitter/receiver was unable to support multi-domain communication. A domain is a group of electronic devices that have similar requirements to work properly in the system. For example, a CD/DVD player, GPS, monitors, and displays form a single domain. Similarly, the dashboard, air-conditioning system (or climate control), wipers, lights, and door locks form another domain. The electronic devices in a vehicle can be classified under different domains and CAN facilitates multi-domain communication, which is a great help to auto engineers.

Domain	**Application area**	**Examples**
Power train	Power generation in engine and transmission through gear box	Engine control
Chassis	Active safety, driving mechanisum and assistance	ABS, ASC
Body	Implements body comfort functions	Dashboard, Climate control, Wipers
Telemetric	Implements entertainment units	CD/DVD Player, GPS system, Multimedia
Passive safety	Safety mechanism	Rollover sensors, Air bag system and Belt pretensions
ADAS	Advanced driver assist systems	Adaptive cruise control (ACC), Automatic parking, Proximity monitors, Pedestrian monitors

TABLE 2.1: A vehicle's multi-domain communication

The CAN protocol is a set of rules for transmitting and receiving messages in a network of electronic devices. It defines how data is transferred from one device to another in a network. Every electronic device (or node) that communicates via the CAN protocol is connected with one another through a common serial bus, which allows for the transfer of messages. For this data exchange to occur, the nodes first require the necessary hardware and the software.

As shown in the above figure 2.1, a typical CAN network consists of several nodes. Each device has a host controller (ECU/MCU), which is responsible for the function of a specific node, and the CAN controller and transceiver. The CAN controller converts messages from the nodes per the CAN protocols, which are then transmitted via the CAN transceiver over the serial bus and vice versa. The controller is a chip that's embedded inside the host controller of the node or added separately.The CAN protocol does not follow the master-slave architecture, which means every nodes has access to read and write data on the CAN bus. When the node is ready to send data, it checks the availability of the bus and writes a CAN frame onto the network. A frame is a structure that carries a meaningful sequence of bit or bytes of data within the network.

The CAN transmitted frame is typically a message-based protocol. A message is

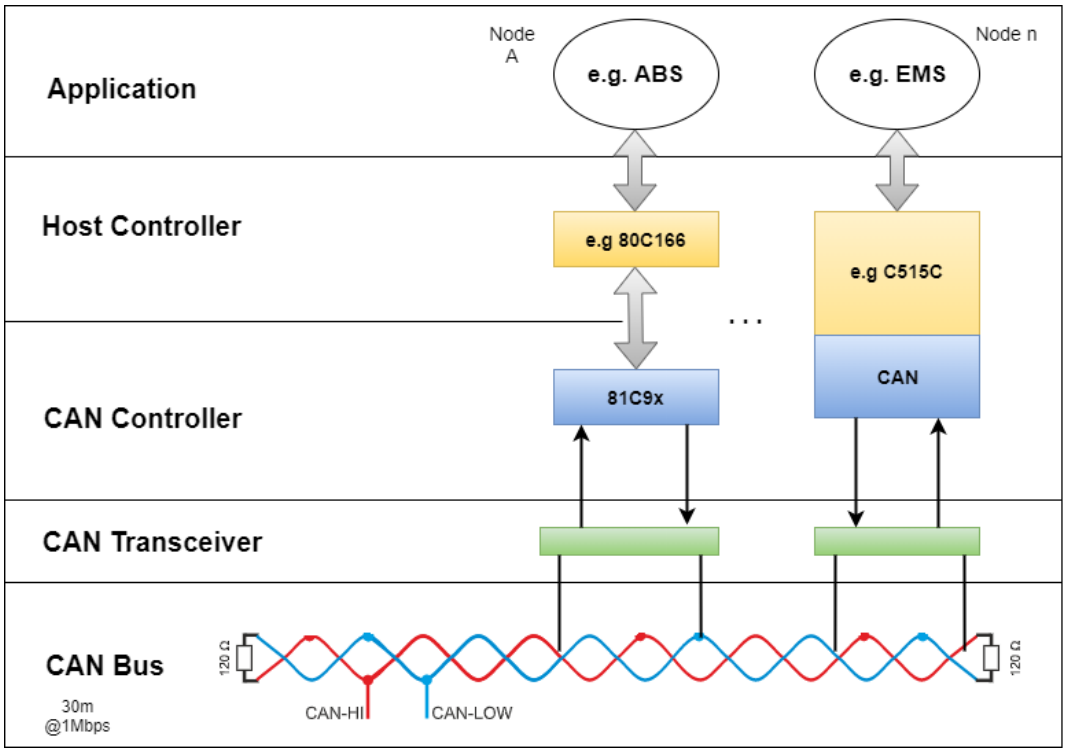

FIGURE 2.3: Various nodes present in a CAN network

a packet of data that carries information. A CAN message is made up of 10 bytes of data, which are organized in a specific structure (called a frame). The data carried in each byte is defined in the CAN protocol. All nodes using the CAN protocol receive a frame and depending on the node's ID, the CAN "decides" whether or not to accept it. If multiple nodes send the message at the same time, the node with the highest priority (so, the lowest arbitration ID) receives the bus access. Lower priority nodes must wait until the bus is available.

The CAN protocol uses the existing OSI reference model to transfer data between the nodes connected in a network.CAN protocol uses lower two layers of OSI i.e. physical layer and data link layer. The remaining five layers that are communication layers are left out by BOSCH CAN specification for system designers to optimize and adapt according to their needs.

Bus Values

Binary values in CAN protocol are termed as dominant and recessive bits.

- CAN define the logic "0" as dominant bit.
- CAN define the logic "1" as recessive bit.

In the CAN system dominant bit always overwrites the recessive bit.

CAN Message Framing

Messages in CAN are sent in a format called frames. A frame is defined structure, carrying meaningful sequence of bit or bytes of data within the network. Framing of

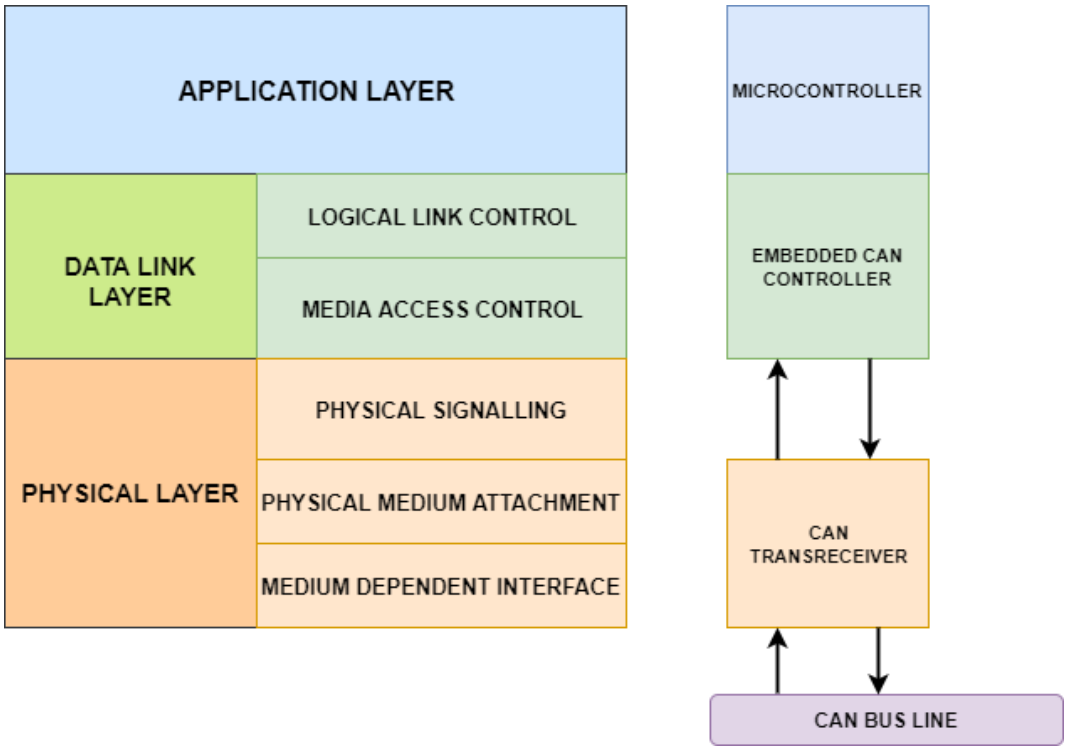

FIGURE 2.4: Various Layers Of a Network Using CAN Protocol

message is done by MAC sub layer of Data Link Layer .There are two type of frames standard or extended .These frames can be differentiated on the basis of identifier fields. A CAN frame with 11 bit identifier fields called Standard CAN and with 29 bit identifier field is called extended frame.

Standard frame

Figure 2.3 depicts standard CAN frame structure. Following table 2.2 describes fields used in standard CAN frame format. It uses 11 bit identifier.

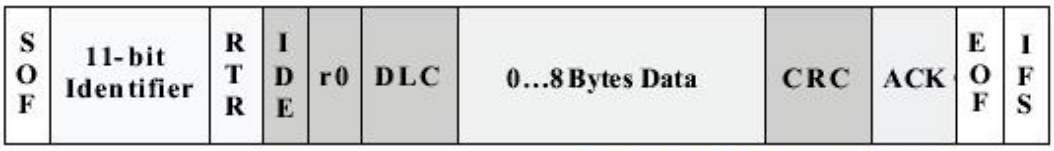

FIGURE 2.5: Standard CAN Frame [12]

Fields	Description
SOF	Start of Frame bit. It marks start of message. It is used to synchronize nodes on the CAN bus.
Identifier	It is 11 bit (binary) in size. It establishes priority of message. Lower the value, higher is the priority.
RTR	It stands for Remote Transmission Request bit. This field is dominant when node requires information from another remote node. All the nodes receive request and all the nodes receive reply. Specific node processes the request based on identifier and transmits the reply.
IDE	Stands for Identifier Extension bit. It indicates standard CAN frame is being transmitted with no extension.
r0	It is reserved for future use.
DLC	Stands for Data length code. It is 4 bits in size. It indicates number of bytes to be transmitted over the CAN bus.
Data	It contains up to 64 bits of application data.
CRC	It is used for error detection. It is 16 bits in size. It holds checksum for application data preceding to it.
ACK	It is 2 bits in size. It contains first bit as ACK bit and second bit as delimiter. Each node uses this to show integrity of its data. Node receiving correct message overwrites this bit in original received message with dominate bit as mentioned above to indicate error free message has been transmitted. The node receiving erroneous message leaves this bit as recessive. Moreover it discards the message and hence prompts the sending node to re-transmit the message after re-arbitration process.
EOF	Stands for End of Frame. It is 7 bits in size. It marks end of CAN frame or message.
IFS	stands for Inter frame space. It is 7 bits in size. It contains time required by controller to move correctly received frame to its proper position in message buffer area.

TABLE 2.2: Fields used in standard CAN frame format

Extended CAN frame

Figure 2.4 depicts extended CAN frame structure. Following table 2.3 describes fields used in extended CAN frame format. It uses 29 bit identifier.

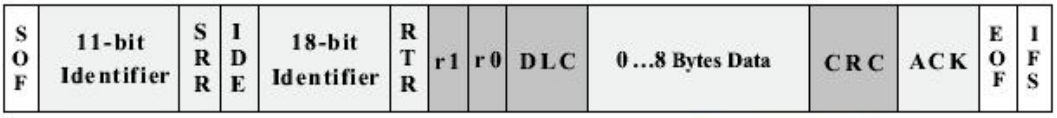

FIGURE 2.6: Extended CAN Frame [12]

Fields	Description
SRR	It stands for Substitute Remote Request. This bit replaces RTR bit of standard CAN message location as placeholder in this extended CAN format.
IDE	It functions as recessive bit in identifier extension. It indicates that more identifier bits are followed. 18 bit extension follows IDE.
r1	It is additional reserved bit for future use.

TABLE 2.3: Fields used in Extended CAN frame format

Message frame

There are four different frames which can be used on the bus.

Data frames: These are most commonly used frame and used when a node transmits information to any or all other nodes in the system. Data Frames consist of fields that provide additional information about the message as defined by the CAN specification. Embedded in the Data Frames are Arbitration Fields, Control Fields, Data Fields, CRC Fields, a 2-bit Acknowledge Field and an End of Frame.

Remote frames: The purpose of the remote frame is to seek permission for the transmission of data from another node. This is similar to data frame without data field and RTR bit is recessive. For example, the microprocessor controlling the central locking on your car may need to know the state of the transmission gear selector from the power train controller.

Error frames: If transmitting or receiving node detects an error, it will immediately abort transmission and send error frame consisting of an error flag made up of six dominant bits and error flag delimiter made up of eight recessive bits. The CAN controller ensures that a node cannot tie up a bus by repeatedly transmitting error frame.

Overload frame: It is similar to error frame but used for providing extra delay between the messages. An Overload frame is generated by a node when it becomes too busy and is not ready to receive.

Arbitration

It is a mechanism which resolves the conflict when two or more nodes try to send the message at the same time. In this technique whenever the bus is free any unit can transmit a message. If two or more units starts transmitting at the same time access to the bus is conflicted, but this problem can be solved by arbitration using identifier. During arbitration every transmitter compares the value of transmitted bit with bit value on the bus. If the bit value is same, the node continues to send the bits. But at any time if transmitted bit value is different from bus value the dominant bit overwrites the recessive bits. The arbitration field of the CAN message consists

of an 11- or 29-bit identifier and a remote transmission (RTR) bit. The identifier having lowest numerical value has the highest priority. RTR simply distinguishes between remote frame for which RTR is recessive and data frame for which RTR is dominant. If both data frame and remote frame with the same identifier is initiated at the same time data frame will prevail over remote frame. With the concept of arbitration neither information nor time is lost.

CAN as a CSMA protocol

CSMA is a carrier sense, multiple-access protocol in which node verifies the absence of traffic before transmitting on a shared medium such as electrical bus. In CSMA each node on a bus waits for a specific time before sending the message. Once this wait period is over every node has equal opportunity to send the message. Based on pre-programmed priority of each message in identifier field i.e. highest priority identifier wins the bus access. It is implemented on the physical layer of OSI model. Let us understand CSMA with an example. In a discussion every person gets an equal opportunity to voice their thoughts however when a person is talking others keep quiet and listens and waits for their chance to speak (carrier sense). But if two or more people start speaking at the same time then they detect the fact and quit speaking (collision detection).

Error Control

Error Checking and Fault Confinement

This is one of the attributes of CAN that makes it robust. CAN protocol has five methods of error checking, out of which three are at message level while other two are at bit level. Every frame is simultaneously accepted or rejected by every node in the network. If a node detects an error it transmits an error flag to every node and destroys the transmitted frame and the transmitting node resends the frame.

Message level

CRC check: In this stage a 15-bit cyclic redundancy check value is calculated by transmitting node and is transmitted in the CRC field. This value is received by all nodes. Then all the nodes calculate CRC value and matches the results with the transmitted value. If values differ than an Error Frame is generated. Since one of the nodes did not receive the message properly it is resent.

ACK slots: When transmitting node sends a message, a recessive bit is sent in acknowledgement slot. After message is received acknowledge slot is replaced by dominant bit which would acknowledge that at least one node correctly received the message. If this bit is recessive, then none of the node has received the message properly.

Form Error: End of frame, Inter-frame space, Acknowledge Delimiter are fields that are always recessive, if any node detects dominant bit in one of these fields than CAN protocol calls it a violation and a Form Frame is generated and original message is resent after certain period.

Bit level

Stuff error: Bit stuffing – It is a very common technique used in telecommunication and data transmission to insert non -informative bits to have same bit rates or to fill the frames .These extra bits are removed by data link layer to retrieve the original message. This same technique is used in bit error. CAN bus is never idle because it uses NRZ method. After five consecutive bits of the same value, a bit with a complement or opposite value is stuffed into the bit stream. If six bits of the same value are detected between SOF and CRC delimiter, error frame is generated. Upon detection of errors, the transmission is aborted and frame is repeated. If errors continue, then the station or node may switch itself off to prevent the bus from being tied up.

Bit error: A node that is sending the bit always monitors the bus. If the bit sent by transmitter differs from the bit value on the bus then error frame is generated. But there is an exception in case of arbitration field or Acknowledge slot where a recessive bit is sent and a dominant bit is received. Then no Bit Error is occurs when dominant bit is monitored.

2.2.2 FlexRay

FlexRay is a relatively new communications bus, which is expected to replace the CAN-bus for high-end applications in future automobiles. The main advantages of the FlexRay-bus over the CAN-bus are higher bandwidth and support for deterministic communication. A consortium of automobile- and semiconductor manufacturers developed FlexRay that became an ISO-standard in 2010.

Network topology of a FlexRay-network

The FlexRay standard adds flexibility to the design process of the network topology compared to the CAN-standard. In addition to supporting multi-drop bus topology, which is used in a CAN-network, the FlexRay protocol also supports a star network topology. In a star network topology there is a central node which handles the communication between the different ECUs.

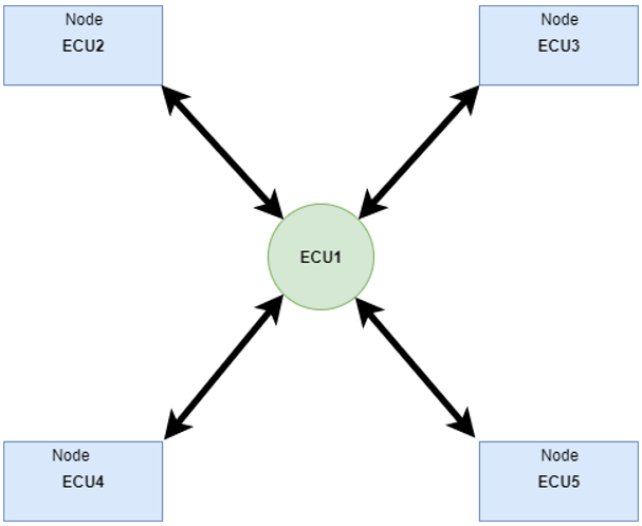

FIGURE 2.7: network topology of flexray

Communication cycle and time division multiple access

A FlexRay network can accommodate both efficient data transfer and deterministic behavior when required. The FlexRay standard manages communication on a single bus with multiple nodes by using a Time Division Multiple Access (TDMA) scheme. ECUs have a predetermined timeslot where they are permitted to transfer data in a communication cycle. The communication cycle in the FlexRay protocol is divided into different parts, there are static and dynamic segments. There is also a symbol window and idle timeslot. The structure of a communication cycle can be seen in Figure 2.6. The communication cycle contains different parts: a static segment (blue), a dynamic segment (yellow), a symbol window (green) and an included idle time (white).

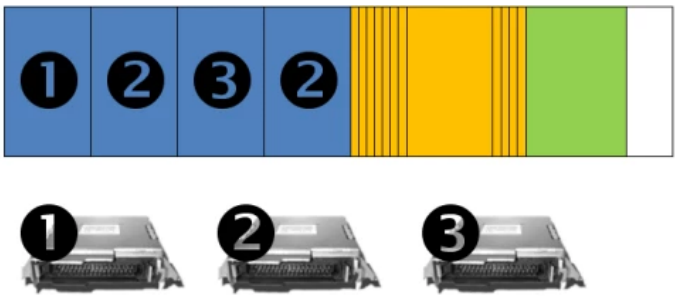

FIGURE 2.8: Communication cycle for a FlexRay network [5]

In a static segment a timeslot is reserved for one ECU, in which only the specified ECU may transmit data as seen in Figure 2.6 where ECU 1 and 3 have one determined timeslot in the static segment (blue) and ECU 2 has access to 2 timeslots to transmit data. The static ensures deterministic communication, which is important in various applications. It is for example important when calculating control loops where equally spaced measurements are advantageous. In the dynamic segment the communication is divided into micro-slots where each ECU has the ability to signal that it has data to transmit to the network. The ECUs with the highest priority will have a micro-slot earlier in the dynamic segment in order to ensure that the ECUs, which have higher priority will get access to the FlexRay bus before lower priority ECUs. Once an ECU has received permission to send it will occupy the bus. The dynamic segment in the FlexRay bus has similar real-time characteristics to the communication in a CAN-network. The mix of static and dynamic segments makes it possible to achieve real deterministic behavior and at the same time not sacrificing the performance of the network by permitting all ECUs to occupy a timeslot in the dynamic segment when the ECU has data to send.

The symbol window is mainly used when performing a startup of a FlexRay network and the idle timeslot is used to synchronize all ECU-nodes to the communication cycle so that all nodes will communicate at the correct timeslot in the communication cycle [5].

2.2.3 MOST and LIN

Local Interconnect Network or LIN was developed to support applications where the features of the CAN-bus were unnecessary and therefore more expensive than what they had to be. The LIN standard is today used in applications where the

relatively high data transfer rate and robust characteristics of the CAN-bus are not required. Examples of these applications are seat, door and mirror control as well as climate control in the vehicle.

Media Oriented Systems Transport or MOST is another communications standard used in automotive applications. It is optimized for multimedia and infotainment applications that require high data transfer rates.

2.2.4 Automotive Ethernet

Automotive electronics in today's vehicles are increasingly complex. With more sensors, controls, and interfaces all using higher bandwidth, faster data throughput and more reliable networks are required. The weight of the cables and harnesses in the vehicle is also of concern to manufacturers.

Ethernet has already proven itself as a secure transfer medium that can handle large amounts of data and can reduce weight by 30% over the traditional CAN/LIN harnesses. The advantages of Ethernet—multi-point connections, higher bandwidth, and low latency—are attractive to automotive manufacturers. However, traditional Ethernet is too noisy and interference-sensitive to be used in automobiles.

Automotive Ethernet is a physical layer standard designed for use in automotive connectivity applications. IEEE standardized the technology with 802.3bw (100BASE-T1) expanded to add 802.3bp (1000BASE-T1). In the table 2.4 below we compare automotive Ethernet to the more familiar version of Ethernet (100BASE-TX). The 10BASE-T1S specification was developed as part of the IEEE 802.3cg standard, which was published in February 2020.

Cables

The automotive Ethernet cable is a single unshielded copper twisted pair making it low weight and low-cost to manufacture. The single twisted pair is a fundamental difference between standard Ethernet and automotive Ethernet. Unlike standard ethernet, with a dedicated transmit-and-receive path, automotive Ethernet, has a single twisted pair being used for both transmit and receive operations at the same time. This implies that both ends of the link employ a hybrid transceiver that's capable of distinguishing what it is sent from what's it is receiving.

Signal encoding

PAM-3 is employed in the automotive standard for both the 100 Mb and 1 Gb version. The different data encodings, or data modulation schemes, meaning the map of raw data bits to symbols as they're transmitted on the link. A combination of the signal encoding, and the modulation type, serve to achieve target spectral efficiencies, and that is something automotive Ethernet does well. In automotive Ethernet particularly, we want to transmit as many symbols for the least amount of bandwidth possible.

Length

Automotive Ethernet has a shorter cable length due to the harsh environment of the automobile. Impedance tolerances and loss are a big deal in the vehicle environment and they are controlled very tightly.

Connector

RJ45 is the classic, standard interface for home and office Ethernet. The connector type for automotive Ethernet is not defined – meaning that you are free to use whatever connector you would like. This can be an issue in form fitting standard test requirements and does require some custom cabling and fixtures.

	Ethernet 100Base-TX (IEEE 802.3)	**Automotive Ethernet 100Base-T1 (IEEE 802.3bw)**	**Automotive Ethernet 1000Base-TX (IEEE 802.3bp)**
Data Rate	100 Mbps	100 Mbps	1000 Mbps
Signal	MLT3	PAM3 66.667 Mb/s	PAM3 750 Mb/s
Modulation	4B/5B	4B/3B	80B/81B
Length	100 m	15m	15m
Connector	RJ45	Not Defined.Varies by OEMs and models.	Not Defined.Varies by OEMs and models.
Cable	Two twisted pair single direction	One twisted pair bi-directional	One twisted pair bi-directional

TABLE 2.4: automotive Ethernet Vs Standard Ethernet

2.3 Electronic Control Unit (ECU)

An Electronic Control Unit or ECU is an embedded computer system, which controls parts of the electrical system in a motor vehicle. By gathering and processing information from several sensors (for example temperature sensors, accelerometers, and gyroscopes) placed in different parts of the vehicle it can control various automated processes in the vehicle. ECUs are also used to check the performance of key components in the car and to monitor changes over time [4].

2.3.1 ECU architecture

A reprogrammable ECU has to have a hardware and software architecture, which supports the software loading process. An ECU, which is reprogrammable after it leaves the manufacturing plant, has the benefit that fixes and new features can be added later during the lifetime of the vehicle. There are also downsides to having a reprogrammable ECU by enabling more people to alter the software on the ECU, some control of how the software loading is done is lost. Because of this, the ECU must implement a structure, which prevents the software on the ECU from becoming unusable. To prevent the ECU from becoming unusable there is usually a protected flash memory or EEPROM sector (see Chapter 2.3) where a primary bootloader or PBL is placed. The primary bootloader should theoretically be impossible to remove without special access, and should not be altered during a normal software loading sequence, see the memory structure of a typical reprogrammable ECU in Figure 2.7. The flash memory or EEPROM contains a protected sector where the

primary bootloader (PBL) is stored.

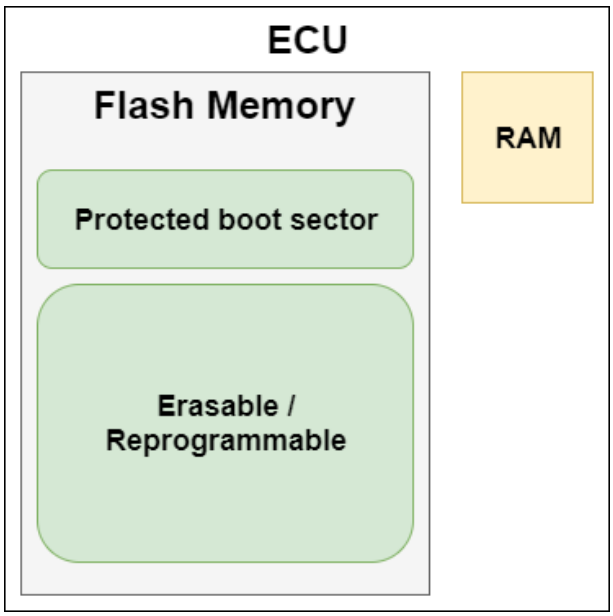

FIGURE 2.9: Memory structure of a typical programmable ECU

When the ECU is powered up the primary bootloader will the first code that will be run. The primary bootloader will then start the application on the ECU if there is a valid application installed (see Figure 2.8). The principle of using a bootloader is ubiquitous in computer systems, for example in a normal desktop PC a bootloader will be run before the operating system is loaded.

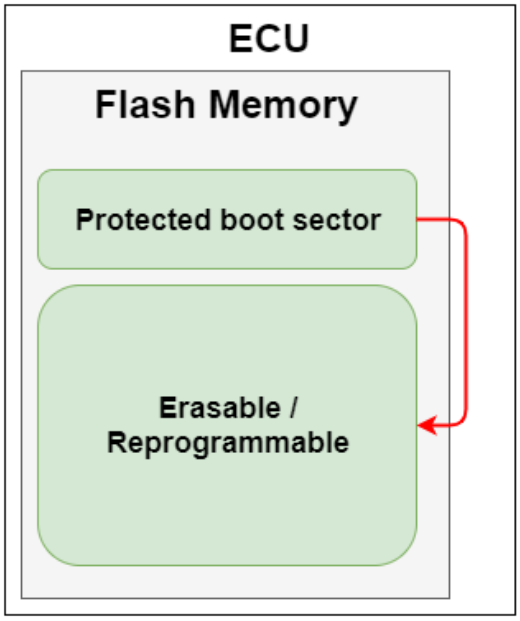

FIGURE 2.10: ECU is powered up the primary bootloader

2.4 On Board Diagnostic (OBD2) protocol

OBD2 is your vehicle's built-in self-diagnostic system. You may have noticed the malfunction indicator light on your dashboard. That is your car telling you there is an issue. If you visit a mechanic, he will use an OBD2 scanner to diagnose the issue.To do so, he will connect the OBD2 reader to the OBD2 16 pin connector near the steering wheel.This lets him read OBD2 codes aka Diagnostic Trouble Codes (DTCs) to review and troubleshoot the issue.

The OBD2 connector

The OBD2 connector lets you access data from your car easily. The standard SAE J1962 specifies two female OBD2 16-pin connector types (A & B). Figure 2.9 is an example of a Type A OBD2 pin connector (also sometimes referred to as the Data Link Connector, DLC).

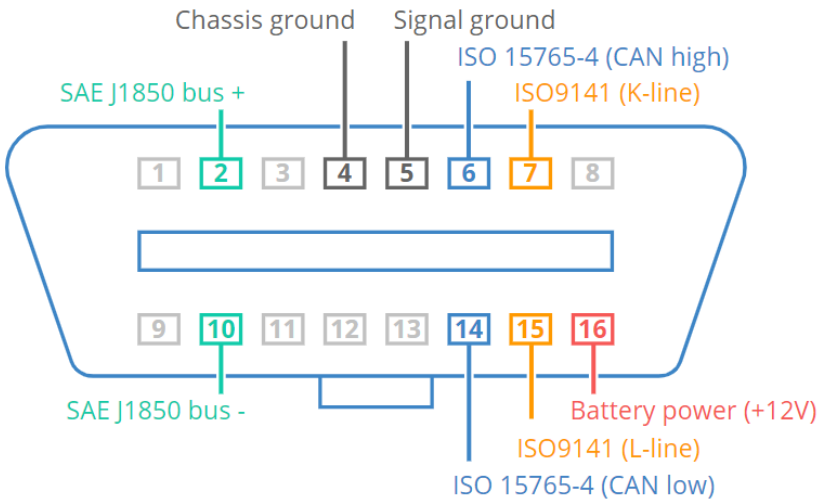

FIGURE 2.11: OBD2 connector pinout [2]

Following points to be noted regarding OBD2 connector:

- The OBD2 connector is near your steering wheel, but may be hidden behind covers/panels.
- Pin 16 supplies battery power (often while the ignition is off).
- The OBD2 pinout depends on the communication protocol.
- The most common protocol is CAN (via ISO 15765), meaning that pins 6 (CAN-H) and 14 (CAN-L) will typically be connected.

In practice, you may encounter both the type A and type B OBD2 connector. Typically, type A will be found in cars, while type B is common in medium and heavy duty vehicles. As evident from the figure 2.9, the two types share similar OBD2 pinouts, but provide two different power supply outputs (12V for type A and 24V for type B). Often the baud rate will differ as well, with cars typically using 500K, while most heavy duty vehicles use 250K (more recently with support for 500K).

To help physically distinguish between the two types of OBD2 sockets, note that the type B OBD2 connector has an interrupted groove in the middle. As a result, a type B OBD2 adapter cable will be compatible with both types A and B, while a type A will not fit into a type B socket.

Link between OBD2 and CAN bus

On board diagnostics OBD2, is a 'higher layer protocol' (like a language). CAN is a method for communication (like a phone).

In particular, the OBD2 standard specifies the OBD2 connector, incl. a set of five protocols that it can run on (see below). Further, since 2008, CAN bus (ISO 15765) has been the mandatory protocol for OBD2 in all cars sold in the US.

The five OBD2 protocols

As explained above, CAN bus today serves as the basis for OBD2 communication in the vast majority of cars through ISO 15765. However, if you're inspecting an older car (pre 2008), it is useful to know the other four protocols that have been used as basis for OBD2. Note also the pinouts, which can be used to determine which protocol may be used in your car [2].

ISO 15765 (CAN bus): Mandatory in US cars since 2008 and is today used in the vast majority of cars.

ISO14230-4 (KWP2000): The Keyword Protocol 2000 was a common protocol for 2003+ cars in e.g. Asia.

ISO9141-2: Used in EU, Chrysler & Asian cars in 2000-04.

SAE J1850 (VPW): Used mostly in older GM cars.

SAE J1850 (PWM): Used mostly in older Ford cars.

OBD2 parameter IDs (PID)

Mechanics obviously care about OBD2 DTCs (maybe you do too), while regulatory entities need OBD2 to control emission.But the OBD2 protocol also supports a broad range of standard parameter IDs (PIDs) that can be logged across most cars. This means that you can easily get human-readable OBD2 data from your car on speed, RPM, throttle position and more. In other words, OBD2 lets you analyze data from your car easily - in contrast to the OEM specific proprietary raw CAN data.

In principle it is simple to log the raw CAN frames from your car. If you e.g. connect a CAN logger to the OBD2 connector, you'll start logging broadcasted CAN bus data out-the-box. However, the raw CAN messages need to be decoded via a database of conversion rules (DBC) and a suitable CAN software that supports DBC decoding (like e.g. asammdf). The challenge is that these CAN DBC files are typically proprietary, making the raw CAN data unreadable unless you're the automotive OEM.

Car hackers may try to reverse engineer the rules, though this can be difficult. CAN is, however, still the only method to get "full access" to your car data - while OBD2 only provides access to a limited subset of data.

How to log OBD2 data?

OBD2 data logging works as follows:

- You connect an OBD2 logger to the OBD2 connector
- Using the tool, you send 'request frames' via CAN
- The relevant ECUs send 'response frames' via CAN
- Decode the raw OBD2 responses via e.g. an OBD2 DBC

In other words, a CAN logger that is able to transmit custom CAN frames can also be used as an OBD2 logger. Note that cars differ by model/year in what OBD2 PIDs they support. For details, see our OBD2 data logger guide [2].

Raw OBD2 frame details

To get started recording OBD2 data, it is helpful to understand the basics of the raw OBD2 message structure. In simplified terms, an OBD2 message is comprised of an identifier and data. Further, the data is split in Mode, PID and data bytes (A, B, C, D) as below.

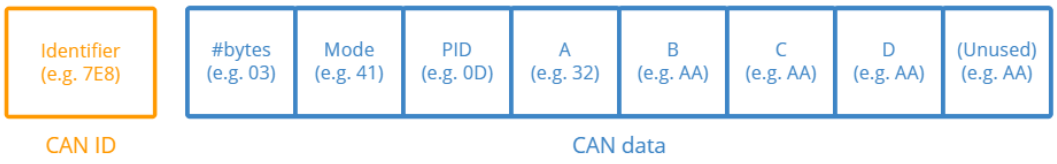

FIGURE 2.12: OBD2 Frame [2]

Identifier: For OBD2 messages, the identifier is standard 11-bit and used to distinguish between "request messages" (ID 7DF) and "response messages" (ID 7E8 to 7EF). Note that 7E8 will typically be where the main engine or ECU responds at.

Length: This simply reflects the length in number of bytes of the remaining data (03 to 06). For the Vehicle Speed example, it is 02 for the request (since only 01 and 0D follow), while for the response it is 03 as both 41, 0D and 32 follow.

Mode: For requests, this will be between 01-0A. For responses the 0 is replaced by 4 (i.e. 41, 42, ... , 4A). There are 10 modes as described in the SAE J1979 OBD2 standard. Mode 1 shows Current Data and is e.g. used for looking at real-time vehicle speed, RPM etc. Other modes are used to e.g. show or clear stored diagnostic trouble codes and show freeze frame data.

PID: For each mode, a list of standard OBD2 PIDs exist - e.g. in Mode 01, PID 0D is Vehicle Speed. For the full list. Each PID has a description and some have a specified min/max and conversion formula. The formula for speed is e.g. simply A, meaning that the A data byte (which is in HEX) is converted to decimal to get the km/h converted value (i.e. 32 becomes 50 km/h above). For e.g. RPM (PID 0C), the formula is (256*A + B) / 4.

A, B, C, D: These are the data bytes in HEX, which need to be converted to decimal form before they are used in the PID formula calculations. Note that the last data byte (after Dh) is not used [2].

OBD2 request/response example

An example of a request/response CAN message for the PID 'Vehicle Speed' with a value of 50 km/h can be seen in the illustration.Note in particular how the formula for the OBD2 PID 0D (Vehicle Speed) simply involves taking the 4th byte (0x32) and converting it to decimal form (50).

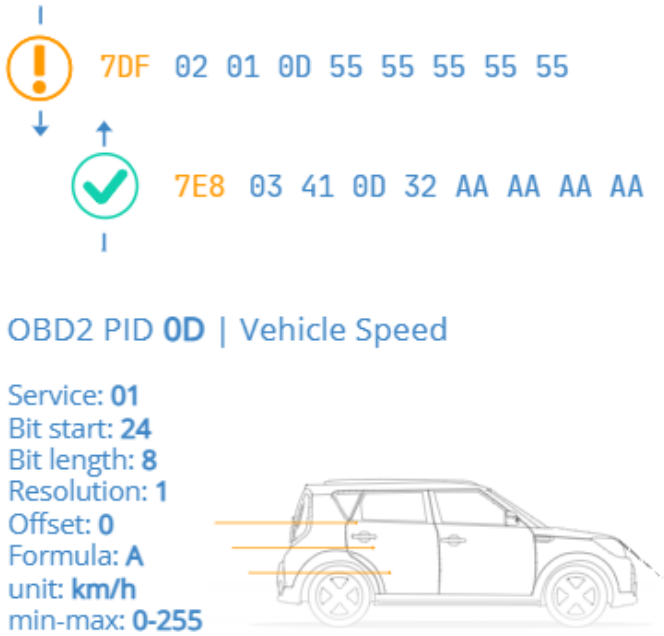

FIGURE 2.13: OBD2 request/response example [2]

In some vehicles (e.g. vans and light/medium/heavy duty vehicles), you may find that the raw CAN data uses extended 29-bit CAN identifiers instead of 11-bit CAN

identifiers. In this case, you will typically need to modify the OBD2 PID requests to use the CAN ID 18DB33F1 instead of 7DF. The data payload structure is kept identical to the examples for 11-bit CAN IDs.

If the vehicle responds to the requests, you'll typically see responses with CAN IDs 18DAF100 to 18DAF1FF (in practice, typically 18DAF110 and 18DAF11E). The response identifier is also sometimes shown in the 'J1939 PGN' form, specifically the PGN 0xDA00 (55808), which in the J1939-71 standard is marked as 'Reserved for ISO 15765-2' [2].

2.5 Unified Diagnostic Services(UDS) protocol

Unified Diagnostic Services (UDS) is an automotive protocol that lets the diagnostic systems communicate with the ECUs to diagnose faults and reprogram the ECUs accordingly (if required). It is called unified because it combines and consolidates all the standards like KWP 2000, ISO 15765 and others.

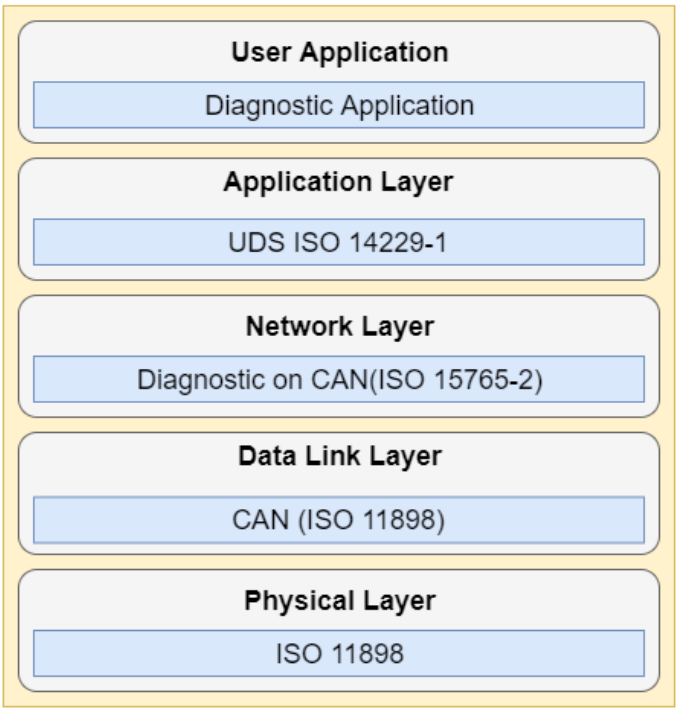

FIGURE 2.14: Architecture of UDS protocol

The Architecture of the UDS protocol is designed based on the Open System Interconnection (OSI) Reference Model. Hence, the UDS software stack has a layered architecture. One of the major functions of UDS software stack is to store the fault code in the ECU memory for every issue that occurs in the vehicle and transfer it (to the client side) as and when required. The diagnostic tester tool has a GUI that connects to the ECU, retrieves the fault code and displays it.

Why Diagnostics is needed in a vehicle?

The protocol is defined for two types of devices, namely, Server and Client. The vehicle will be the Server and the diagnostic device will be the Client. Recent vehicles

are equipped with a diagnostic interface, which makes it possible to connect a computer or diagnostics tool, to the communication system of the vehicle (ECU). UDS requests are sent to the controllers which provide a positive or negative response. With these responses provided from the controller, it is possible to diagnose faults and undesirable behavior inside vehicles such as:

- Data stored within the system
- Memory available in the individual control units
- Live vehicle data such as engine or vehicle speed
- Firmware updates
- Interaction with hardware I/O to turn the specific output ON or OFF based on the response to identify the fault
- Run specific functions to understand the environment and operating conditions of an ECU

As OEMs integrate/assemble automotive ECUs and components from different suppliers, the need for a standard diagnostic protocol was felt. This is because, prior to a Unified Protocol, OEMs and suppliers had to deal with compatibility issues between different diagnostic protocols like KWP 2000, ISO 15765, and diagnostics over K-Line.

- Unified Diagnostic Services (UDS) is the preferred choice of protocols for all off-board vehicle diagnostic activities. Off-board diagnostics refers to the examination of the vehicle parameters when the car is at servicing in the garage (while the vehicle is stationary).
- ECU flashing and reprogramming can also be performed efficiently with the help of a UDS stack.
- Additionally, UDS protocol is quite flexible and is capable of performing more detailed diagnostics as compared to other protocols like OBD and J1939.

List of Categories of Services Offered by an ISO 14229 UDS Protocol Stack

1. Data Transmission Capabilities

The data transmission capabilities of a UDS Protocol Stack enable the clients to read or write any information to or from the ECU. The data can be read or written on the basis of identifiers and periodic identifiers. The client can also read data from the physical memory at the specified address.

- The information can range from static info like ECU serial number to some real time data like the current status of the sensors, engine speed, etc.
- If the client wants the ECU to send periodic service values, then 'Read Data By Identifier Periodically' service will be required. The client can also write data by identifier and address. Using the write service, certain parameters such as threshold values and angles can be changed.
- Usually, the permission to write some sensitive data to the ECU can be controlled by restricting the access using 'Security Access Service'. Such permissions are reserved by the OEMs, as writing data to the ECU can interfere with the security and overall functioning of the vehicle.

2. Fault Diagnostics

One of the main services of the UDS protocol is fault diagnostics. Whenever an issue occurs in the vehicle, a diagnostic trouble code (DTC) corresponding to the fault is stored in the ECU fault code memory (FCM). The service personnel at the garage can retrieve these DTCs by using the ReadDTCInformation service.

- Fault Diagnostics service allows the client to read both emission related and non-emission related DTC information. The client can define a status mask based on which the DTC information will be displayed.
- DTC Snapshot data can also be retrieved using this service.

DTC Snapshot data gives additional information about the engine's parameters at the time of occurrence of the fault. The DTC information along with other data stored in the server can be erased if need be. ClearDiagnosticInformation service can be invoked to delete all such diagnostics data stored in the server.Once the fault codes are retrieved, the problem can be diagnosed efficiently, and repair work can follow.

3. Upload/Download Capabilities

As highlighted earlier, UDS protocol also supports ECU reprogramming. ECU reprogramming refers to updating the ECU software. This is required to resolve any existing bug or add newly developed modules in the ECU. Using the upload and download capabilities of UDS protocol, large packets of data can be sent and received from the car's ECU for ECU reprogramming purpose. The client can invoke RequestDownload and TransferData service to initiate a data transfer to the server (ECU) from the client (diagnostic tester) using a tester device.

- The server upon receiving the request will take all necessary actions to receive the data.
- A positive response message is sent when the server has successfully received the message.

Likewise, a RequestUpload service is used by the client to request data packets from the server.

- One of its practical examples can be configuring the parameters related to the vehicle's variant code. It implies that the client can download or upload the settings/configurations in order to change or adapt a particular variant.
- Suppose a car has two variants and one of them has Anti-lock braking system (ABS) and the other doesn't. The ECU of the variant with the ABS will need to be updated with configurations and settings to control the ABS. A task like this can be performed using this service.

4. Remote Routine Activation

Vehicle Diagnostics may require testing the faulty component in a given range of parameters. Moreover, during the testing phase of the vehicle, some system tests may be required to run over a period of time. For all such tasks, remote routine activation service of UDS protocol is used.

- In order to perform a test, a routine is triggered by the client in the server's memory. There are two methods for ending this routine – one is where the client interrupts the routine to stop it, and the other is when the server/ECU finishes the routine after a specified time frame.
- Using this service, the client can start a routine, stop a routine and also check the result that the routine produced after successful execution.

For instance, the service personnel at the garage may use this service to run the engine fan for a certain period of time and record the results. This would help him understand a particular issue well and rectify it without using any hit and trial method.

The UDS services that are frequently used in the software loading sequence are listed in following table.

SID	UDS SERVICES	DESCRIPTION
0x10	Diagnostic Session Control	Enable various diagnostics sessions within ECU
0x11	ECU Reset	Resetting the ECU to be back in the default session
0x27	Security Access	Limit access to data and services to prevent unauthorized access
0x3E	TesterPresent	Alert the ECU(s) that client is still connected so that diagnostic sessions remain active.
0x22	Read Data By Identifier	Request data from ECU(s)
0x2E	Write Data By Identifier	Write data onto ECU(s)
0x14	Clear Diagnostic Information	Clear diagnostic trouble codes (DTC) stored in the ECU
0x19	Read DTC Information	Read DTC from the ECU
0x2F	Input Output Control By Identifier	Control the input/output signals through the diagnostic interface
0x31	Routine Control	Control all the routine services (erasing memory, testing routines etc.)
0x34	Request Download	Request ECU to initiate download session based on request from the tester
0x36	Transfer Data	Manage actual transmission (upload and download) of data
0x37	Request Transfer Exit	Terminate and exit data transfer
0x28	Communication Control	Manage the exchange of messages in the ECUs
0x85	Control DTC Setting	Enable/disable updating of DTC settings in ECU
0x87	Link Control	Control the ECU- client (tester) communication to gain bus bandwidth for diagnostic purposes.
0x23	Read Memory By Address	Read memory data from the memory address provided

0x24	Read Scaling Data By Identifier	Read scaling data stored in the server using data identifier.
0x3D	Write Memory By Address	Write information into the server memory location
0x35	Request Upload	Request ECU to upload data

TABLE 2.5: Important software loading services specified in UDS

UDS message structure

UDS is a request based protocol. Following figure illustration outlined an example of an UDS request frame (using CAN bus as basis):

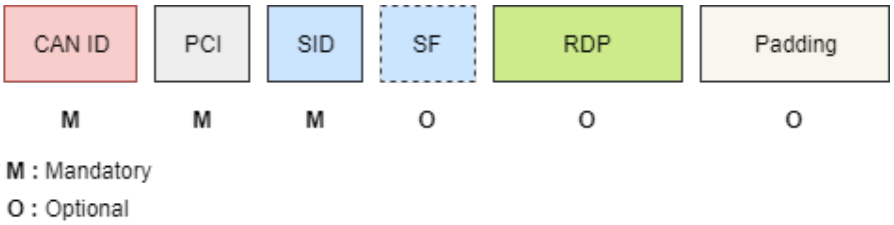

FIGURE 2.15: UDS request message format

A diagnostic UDS request on CAN contains various fields that we detail below:

Protocol Control Information (PCI)

The PCI field is not per se related to the UDS request itself, but is required for diagnostic UDS requests made on CAN bus. In short, the PCI field can be 1-3 bytes long and contains information related to the transmission of messages that do not fit within a single CAN frame. We will detail this more in the section on the CAN bus transport protocol (ISO-TP).

UDS Service ID (SID)

The use cases outlined above relate to different UDS services. When you wish to utilize a specific UDS service, the UDS request message should contain the UDS Service Identifier (SID) in the data payload. Note that the identifiers are split between request SIDs (e.g. 0x22) and response SIDs (e.g. 0x62). As in OBD2, the response SIDs generally add 0x40 to the request SIDs. See also the overview of all standardized UDS services and SIDs in the table 2.4. For example UDS service 0x22 is used to read data (e.g. speed, SoC, temperature, VIN) from an ECU.

UDS Sub Function Byte

The sub function byte is used in some UDS request frames as outlined below. Note, however, that in some UDS services, like 0x22, the sub function byte is not used.

Generally, when a request is sent to an ECU, the ECU may respond positively or negatively. In case the response is positive, the tester may want to suppress the response (as it may be irrelevant). This is done by setting the 1st bit to 1 in the sub

function byte. Negative responses cannot be suppressed.

The remaining 7 bits can be used to define up to 128 sub function values. For example, when reading DTC information via SID 0x19 (Read Diagnostic Information), the sub function can be used to control the report type.

UDS 'Request Data Parameters' - incl. Data Identifier (DID)

In most UDS request services, various types of request data parameters are used to provide further configuration of a request beyond the SID and optional sub function byte. For example, service 0x19 lets you read DTC information. The UDS request for SID 0x19 includes a sub function byte - for example, 0x02 lets you read DTCs via a status mask. In this specific case, the sub function byte is followed by a 1-byte parameter called DTC Status Mask to provide further information regarding the request. Similarly, other types of sub functions within 0x19 have specific ways of configuring the request.

Another example is service 0x22 (Read Data by Identifier). This service uses a Data Identifier (DID), which is a 2-byte value between 0 and 65535 (0xFFFF). The DID serves as a parameter identifier for both requests/responses (similar to how the parameter identifier, or PID, is used in OBD2). For example, a request for reading data via a single DID in UDS over CAN would include the PCI field, the UDS service 0x22 and the 2-byte DID. Alternatively, one can request data for additional DIDs by adding them after the initial DID in the request.

UDS DID	DESCRIPTION
0xF180	Boot software identification
0xF181	Application software identification
0xF182	Application data identifier
0xF183	boot software fingerprint
0xF184	Application software fingerprint
0xF185	Application data fingerprint
0xF186	Active Diagnostic session
0xF187	Manufacturer spare part number
0xF188	Manufacturer ECU software number
0xF189	Manufacturer ECU software version
0xF18A	Identifier of System supplier
0xF18B	ECU Manufacturing Date
0xF18C	ECU serial Number
0xF18D	Supported functional units
0xF18E	Manufacturer kit assembly part number
0xF190	Vehicle Identification Number (VIN)
0xF192	System supplier ECU Hardware number
0xF193	System supplier ECU Hardware version number
0xF194	System supplier ECU software number
0xF195	System supplier ECU software version number
0xF196	Exhaust regulation/type approval number
0xF197	System Name / Engine type
0xF198	Tester serial number
0xF199	Programming Date

0xF19D	ECU Installation Date
0xF19E	ODX File

TABLE 2.6: UDS Standardized DIDs

Positive vs. negative UDS responses

When an ECU responds positively to an UDS request, the response frame is structured with similar elements as the request frame. For example, a 'positive' response to a service 0x22 request will contain the response SID 0x62 (0x22 + 0x40) and the 2-byte DID, followed by the actual data payload for the requested DID. Generally, the structure of a positive UDS response message depends on the service.

However, in some cases an ECU may provide a negative response to an UDS request - for example if the service is not supported. A negative response is structured as in below CAN frame example:

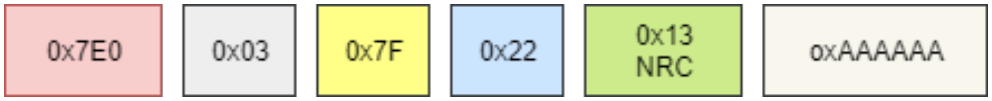

FIGURE 2.16: UDS Negative Response Format

In the negative UDS response, the NRC provides information regarding the cause of the rejection as per the table below.

UDS NRC	DESCRIPTION
0x01 – 0x0F	ISO SAE Reserved
0x10	General reject
0x11	Service not supported
0x12	Sub-function Not Supported
0x13	Incorrect Message Length Or Invalid Format
0x14	Response Too Long
0x21	Busy Repeat Request
0x22	Conditions Not Correct
0x24	Request Sequence Error
0x25	No Response From Sub-net Component
0x26	Failure Prevents Execution Of Requested Action
0x31	Request Out Of Range
0x33	Security Access Denied
0x35	Invalid Key
0x36	exceed Number Of Attempts
0x37	Required Time Delay Not Expired
0x70	Upload Download Not Accepted
0x71	Transfer Data Suspended
0x72	General Programming Failure
0x73	Wrong Block Sequence Counter
0x78	Request Correctly Received-Response Pending
0x7E	Sub-function Not Supported In Active Session
0x7F	Service Not Supported In Active Session

0x81/0x82	Rpm Too high/Low
0x83/0x84	Engine Is Running/Not Running
0x85	Engine Run Time Too Low
0x86/0x87	Temperature is Too High/Low
0x88/0x89	Vehicle Speed is Too High/Low
0x8A/0x8B	Throttle/Pedal is Too High/Low
0X8C/0x8D	Transmission Range Is Not In Neutral/Gear
0x8F	Brake Switch(es) Not Closed
0x90	Shifter Lever Not In Park
0x91	Torque Converter Clutch is Locked
0x92/0x93	Voltage is Too High/Low
0xF0 – 0xFE	Vehicle Manufacturer Specific Conditions Not Correct

TABLE 2.7: Negative Response Codes (NRC)

2.5.1 Standardized software loading sequence

Within the UDS-standard ISO 14229-1 there is a framework defined for "nonvolatile server memory programming process [[11], p. 303]" or software loading sequence for reprogrammable ECUs in other words. It contains specifications for the entire software loading sequence with mandatory; optional/recommended and vehicle manufacture specific steps to accommodate different vehicle manufacturers preferences but at the same time keep the flashing sequence relatively similar regardless of which vehicle manufacturer specific flashing sequence is used.

The software loading sequence defined in the UDS-standard is divided into two main programming phases. Programming phase 1 – download of application software and/or application data and programming phase 2 – server configuration that is an optional phase. These programming phases are in turn divided into three substeps: pre-programming, programming and post-programming (see Figure 2.15).

Pre-programming in phase 1

The pre-programming step in programming phase 1 is an optional step, which configures the ECU before the actual transfer of data. The mandatory steps included in this pre-programming step are sending a Diagnostic Session Control (0x10) UDS-request in order to enter the extended diagnostic session (0x03) on the ECU. When the ECU is in extended session it is possible for the tester to send UDS-requests Control DTC Setting (0x85) and Communication Control (0x28) to disable updating of DTC:s and disable non-diagnostic communication. Other optional steps include checking programming preconditions and disable fail-safe reactions with a Routine Control (0x31) UDS-request. Some manufacturers also use Read Data By Identifier (0x22) to read ECU data.

programming in phase 1

The actual downloading of application software and/or application data is done in the programming step in programming phase 1. The mandatory steps included in

the programming phase are to enter programming session (0x02) by using a Diagnostic Session Control (0x10) UDS-request. Then sending the main downloading sequence UDS-requests Request Download (0x34), Transfer Data (0x36) and Request Transfer Exit (0x37). There are also several optional/recommended steps such as Security Access (0x27) with the seed and key scheme, Erase Memory (0xFF00) and various checks on EEPROM or flash memory by using the Routine Control (0x31) UDS-service.

Post-programming in phase 1

The post-programming step of programming phase 1 only contains one step which can either be a ECU Reset (0x11) or a Diagnostic Session Control (0x10) request to put the ECU in the default diagnostic session.

Pre-programming in phase 2

The pre-programming step in phase 2 – server configuration is performed by sending a request to enter the extended diagnostic session to for example enable the control of updating of DTC:s and re-enable communication with the Control DTC Setting (0x85) and Communication Control (0x27) UDS-services.

programming in phase 2

In the programming step in phase 2 it is mandatory to clear diagnostic information, which might have been stored in the re-programmable ECU with a Clear Diagnostic Information (0x14) UDS-service. This step was, however, not performed by any of the projects at BW-PDS, which were analyzed in this project. There is an option to include vehicle manufacturer specific options in this step.

Post-programming in phase 2

The post-programming step of programming phase 2 only contains one step which can either be a ECU Reset (0x11) or a Diagnostic Session Control request to put the ECU in the default diagnostic session just like the post-programming step in phase 1.

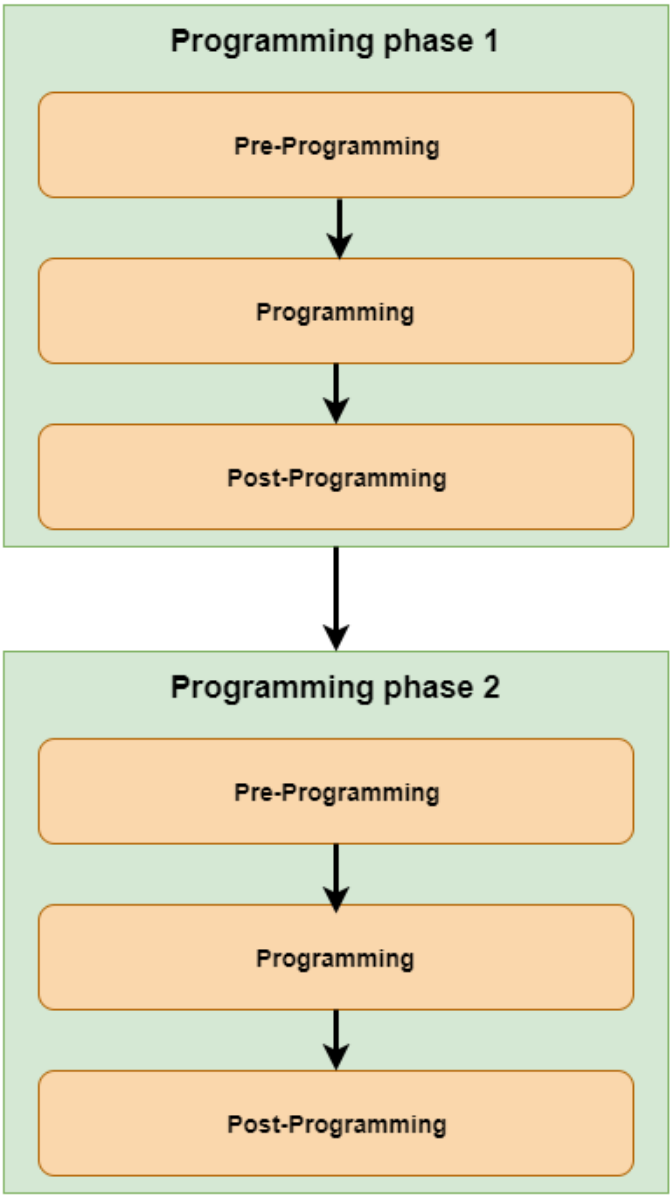

FIGURE 2.17: Non-volatile server memory programming process framework defined in the UDS-standard ISO14229-1

Chapter 3

Implementation

3.1 Proof of Concept

The Proof of Concept (PoC) sought to test and try breaking the security session of UDS protocol and extract meaningful information from the ECU. Analyze the extracted information, change it and dump it again to the ECU and change the original behaviour of the system.

The system used for the PoC was designed with the following considerations in mind:

- The UDS stack consists of a limited amount of services implemented from the defined services in ISO-14229.
- The hardware supports only CAN communication and not Ethernet hence the UDS on CAN is implemented.

The PoC was deemed successful if a tester ECU will be able to break the security session and extract firmware data from the main ECU.

3.2 System Architecture

Following figure 3.1 shows the system architecture used for this thesis.

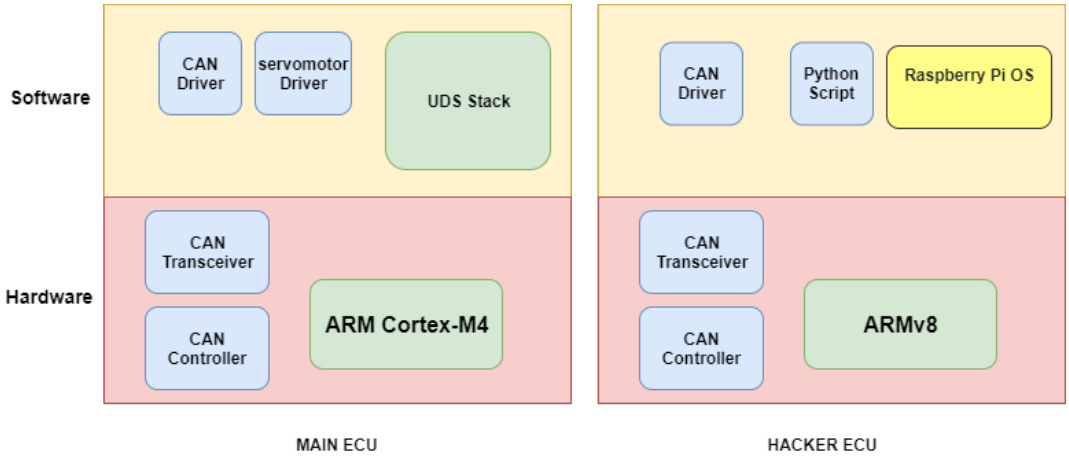

FIGURE 3.1: System Architecture

As shown in the system architecture, the ARM Cortex-M4 (MK20DX256) controller of the teensy 3.2 board has been used to run the program. Built-in CAN controllers have been utilized for CAN communication. CAN transceiver is used to send the CAN message on the CANbus. More on the hardware is described in the following section 3.3.The software mainly consists of the CAN drivers, Servomotor driver, and UDS stack. Arduino FlexCAN and Servo libraries were used for the CAN controller and servomotor. Detailed implementation of the UDS stack is described in section 3.4.

Raspberry pi 2 development board used as Hacker ECU here. The board has an ARMv8 processior and capable of running the raspberry pi OS and some of the device driver on it. CAN HAT waveshare board used as CAN controller and transceiver for establishing the CAN communication. Scocket CAN and CAN utils open source library have been used as CAN driver. pip library for the Python3 have been installed to run the python script on the hardware.

3.3 Hardware Setup

In order to test the UDS protocol, the Teensy 3.2 development board is used as the main ECU on which the UDS stack is installed. raspberry pi 3 has been used here as a tester tool. The following figure shows the hardware setup used in this thesis.

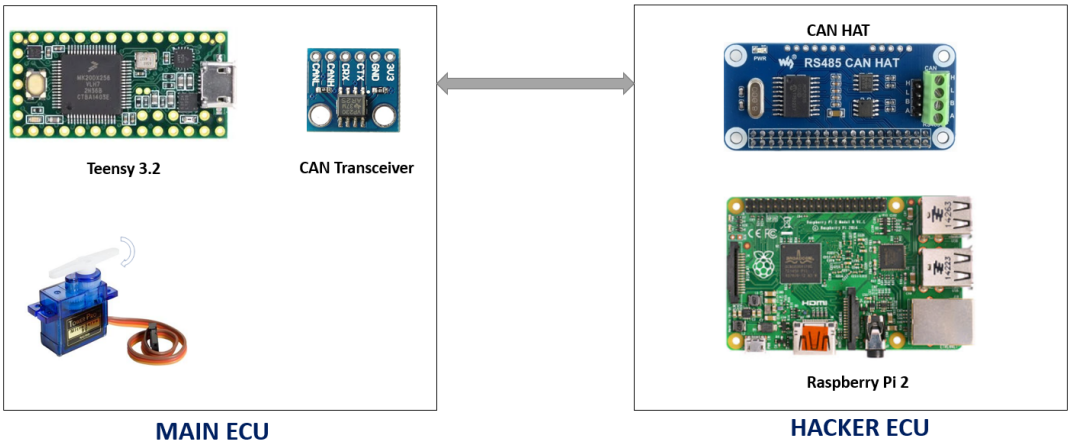

FIGURE 3.2: Hardware Setup

As shown in Figure 3.3, the default firmware program moves the Servo motor in the anti-clockwise direction. If hacker manage to access the firmware he can change the firmware and disturb the default behaviour of the motor moving direction. For the on road car ECU if such attack happens it may possibly danger the driver life.

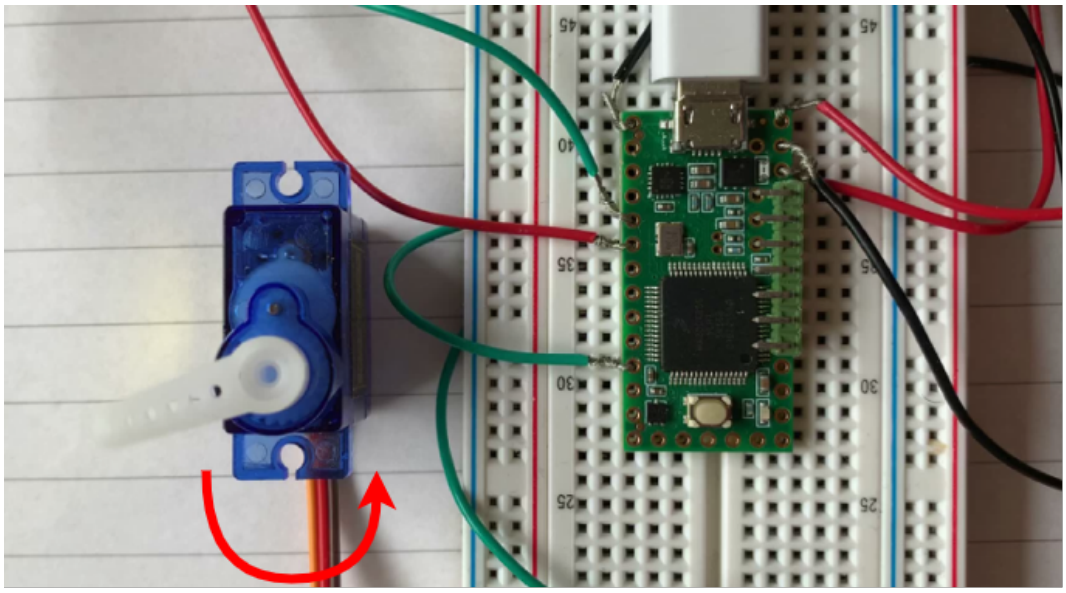

FIGURE 3.3: Default behaviour of the Motor

3.4 Development

Following two major development have been with respect to the software for this thesis.

- UDS stack development
- Development of Python script

3.4.1 UDS diagnostic session control

As the name suggests, this service is used to enable different sessions in the server/ECU. The ECU would be designed in such a way that certain diagnostic services can be accessed only in certain sessions and would be disabled for all other sessions.

List of Session and their sub functions

- Default session (0x01)
- Programming session (0x02)
- Extended diagnostic session (0x03)
- Safety system diagnostic session (0x04)
- Vehicle manufacturer specific session (0x40)

Important points to know about Diagnostic Session control

- There will always be one active session in a server/ECU at any point in time.
- The server/ECU will always start in the default session (0x01).
- Until another diagnostic session is requested by the client, the server/ECU will remain in the default session.

- The system continues to perform its business logic when in default session.
- When the client requests for a different session, a positive response is first sent to the client, and then transition to the requested session is internally initiated.
- If the client requests for a service which the server/ECU is already in, the server/ECU will send a positive response and continue to be in the same session.
- If the server/ECU is in a non-default session and if there is diagnostic inactivity for 5 seconds, the server/ECU transits back to the default session.

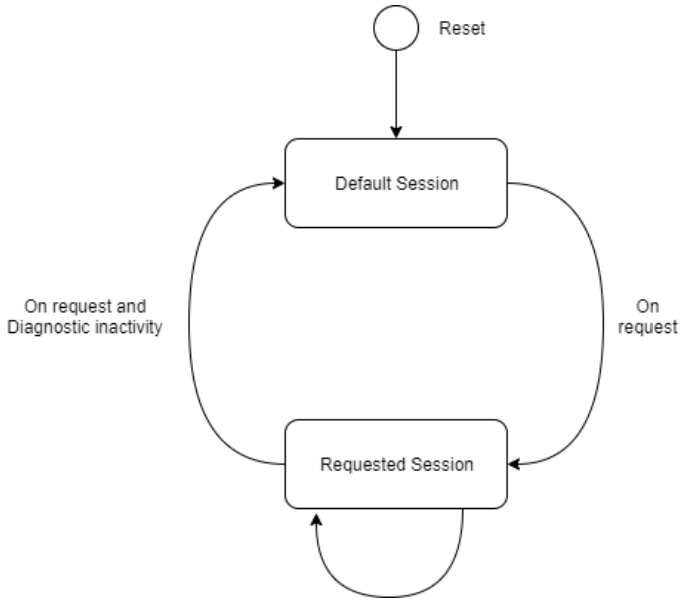

FIGURE 3.4: UDS Sessions Change

Note: Sub functions default session, programming session, and extended diagnostic session of the diagnostic session control(0x10) have been implemented in the UDS stack. Additionally, the P2 timing set for diagnostic inactivity is 6 seconds.

3.4.2 Security algorithm

UDS defines how to proceed with the key exchange. This process consists of 2 exchanges of request/response between the client and the server. It goes as follows:

1. First, the client requests for a seed to unlock a specific security level (identified by a number). This seed is usually a random value that the client must use in order to compute the key. It is meant to prevent someone from recording the CAN bus message exchange and then gaining privileges by blindly sending what was recorded. In cryptography terms, the seed is a nonce used to avoid replay attacks.
2. Once the client gets the seed, it must compute a key using an algorithm that is defined by the ECU manufacturer and known by the server.
3. The client then sends the key to the server, the server verifies it and, if it matches the server's value, the security level is unlocked and a positive message is responded to the client.

3.4.3 Security Access Seed Request

Security Access Service Identifier (0x27) mostly supports Extended diagnostic sessions, so before you request it you should know the diagnostic session that how to jump from the default session to the extended diagnostic session. When the client will request a seed first to the ECU, the ECU will receive it, and as per the ECU behavior, it will send the response as positive or negative. If it is negative then the client should take the decision of what action needs to be done. But if it is positive, then the Client will receive the seed key sent by the server and then the client will generate a security key from that seed key sent by the ECU. Following figure 3.4 shows the seed request frame format and Its responses.

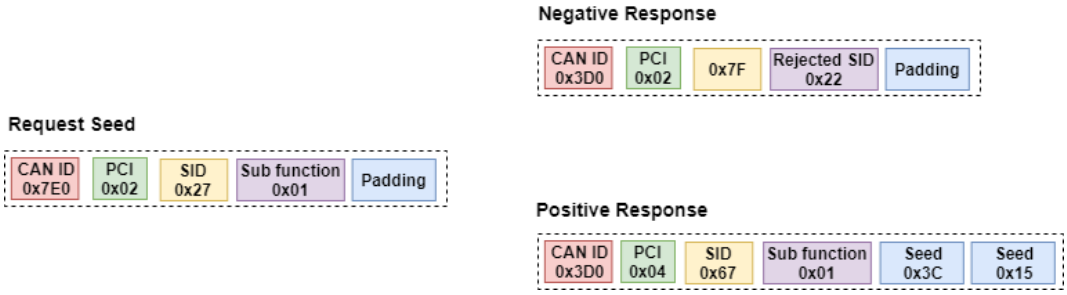

FIGURE 3.5: Seed Request frame and it's responses

3.4.4 Security Access Key Request

When the client will send the "sendkey" request message with a valid security key, the server will check to compare that key with the sake key generated by the client and if it will match then the server will unlock the ECU. Then after unlocking the ECU will send a positive response message as shown in figure 3.5.

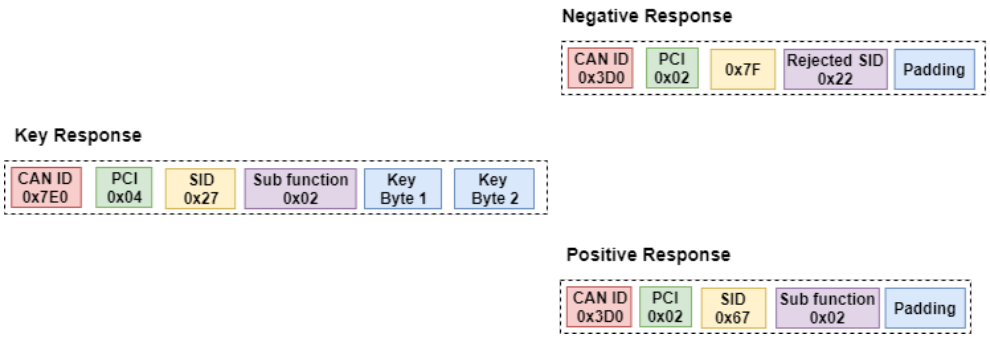

FIGURE 3.6: Key respond frame and it's responses

3.4.5 Read and Write Data by addresses

The ReadMemoryByAddress service (0x23) allows the client to request memory data from the server via a provided starting address and to specify the size of memory to be read. The ReadMemoryByAddress request message is used to request memory data from the server identified by the parameter memory address and memory size. The number of bytes used for the memory address and memory size parameter is defined by addressAndLengthFormatIdentifier (low and high nibble). It is also possible to use a fixed addressAndLengthFormatIdentifier and unused bytes within the

memory address or memory size parameter are padded with the value 00 hex in the higher range address locations.

The WriteMemoryByAddress service (0x3D) service allows the client/Tester to write information into the server/ECU at one or more contiguous memory locations. The tester sends a memory address, the number of bytes, and a data string (according to the number of bytes). The ECU writes the data string into its memory. The addressAndLengthFormatIdentifier parameter in the request specifies The number of bytes used for the memory address and memory size parameter.

Following figure 3.6 shows the frame format for the read and write data by memory address.

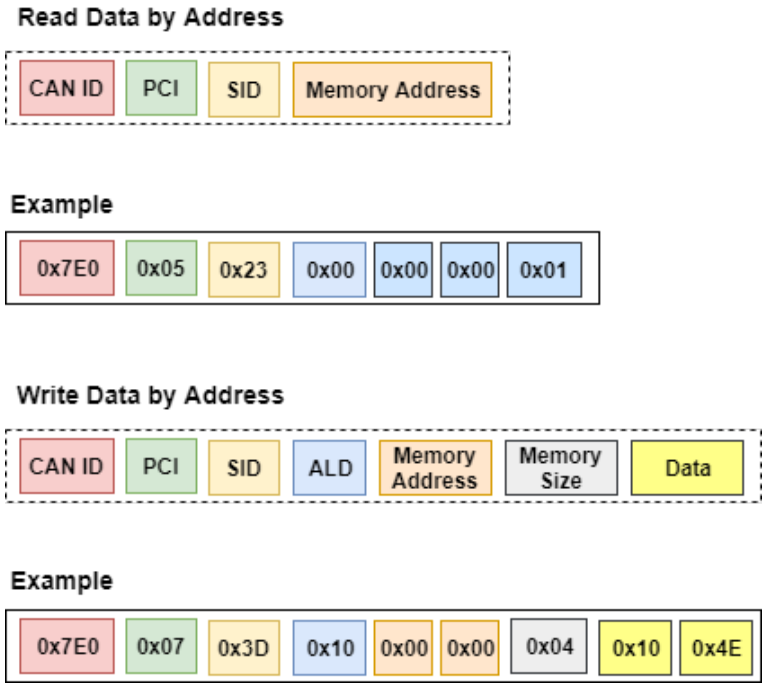

FIGURE 3.7: Read and write data by address

3.4.6 Read and Write Data by Identifier

The Read Data By Identifier (0x22) service is used to read the scaling information of a record identified by a provided data identifier from the ECU. The client request message contains one data identifier value that identifies data record(s) maintained by the server. The format and definition of the data record shall be vehicle-manufacturer-specific and may include analog input and output signals, digital input and output signals, internal data, and system status information if supported by the server.

The Write Data By Identifier (0x2E) service allows the client/Tester to write information into the server/ECU at an internal location specified by the provided data identifier. The Write Data By Identifier service is used by the client to write a data Record to a server. The data is identified by a data identifier and may or may not be secured.

Dynamically defined data identifier(s) shall not be used with this service. It is the vehicle manufacturer's responsibility that the server conditions are met when performing this service. Possible uses for this service are:

- programming configuration information into the server (e.g. VIN).
- clearing non-volatile memory.
- resetting learned values.
- setting option content.

Following figure 3.7 shows the frame format for the read and write data by Identifier.

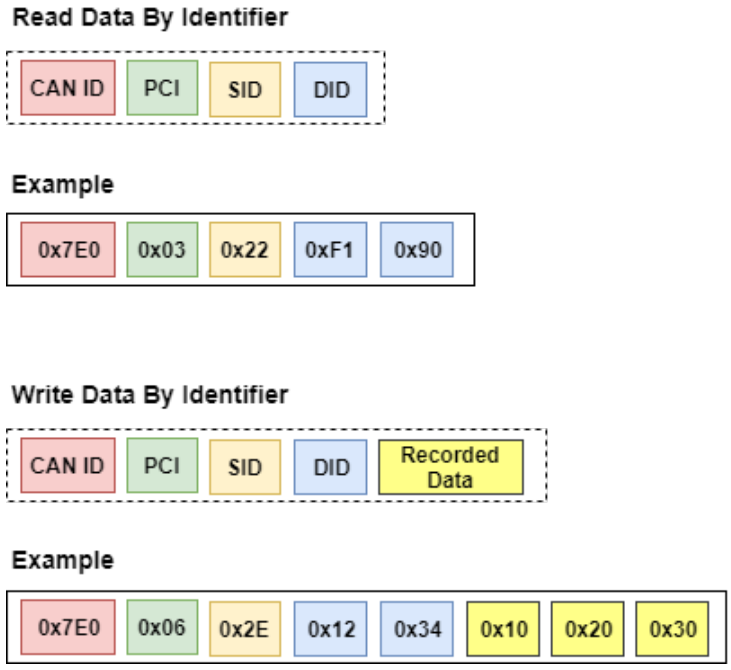

FIGURE 3.8: Read and write data by Identifier

3.4.7 Development of Python script

In determining the strength of an ECU's security, there are several interdependent variables at stake. For example, the length of the cryptographic seed in authentication and the number of allowed failed authentication attempts in combination determine the amount of time that a brute-force attack would take on average. It is the combination of different variables that gives the security heuristic, not necessarily the specific values on their own.

The shortest observed seed and authentication code were 2 bytes long. This length makes a brute-force attack feasible, as per scenario 3 of section 4.2.1. The aim of this attack is to get one-time access to Programming Session, using the brute-force approach.

Course of the Experiment
The attack consists of querying Security Access seeds and always responding with

a pre-chosen authentication code. It is likely that at some point, authentication code A will be accepted as the correct reply to a seed. The purpose of this experiment is to explore the possibility of such an attack and to measure the timespan until the attack succeeds. To determine some repeatability, this test is conducted twice. Given the time expectations of this experiment, repeating it as many times as would suffice for making statistical conclusions would be too demanding within the time scope of carrying out this thesis research. The course of the experiment is as follows. Figure 3.8 explains the automation steps involved which is developed using the python script.

- Beforehand, one random 2-byte combination is chosen as the value of authentication code. Programming Session is entered.
- Security Access request Seed message is sent. If a negative response is received (i.e. time delay is on), repeat the message until the ECU sends a positive response (i.e. until the time delay is over).
- A Security Access send Key message is sent with authentication code A as its content.
- If the ECU responds negatively to authentication code A, repeat the process beginning from the Security Access request Seed message.
- If the ECU accepts the authentication code and grants access to the unlocked state and try reading the memory data by address.
- Download the binary data from the ECU firmware memory region and do the reverse engineering.

Result

The experiment was performed several times. The amount of time needed for the attack may be considered feasible, but not short. Theoretically, this attack is realistic to some extent, but it only provides one-time security access (instead of uncovering a secret key, which would provide access for all future occasions). If the details of the authentication code calculation algorithm are known to the attacker, the uncovered legitimate seed-authentication code pair may, however, be subjected to a much more speedy offline brute-force attack to find out the secret key.

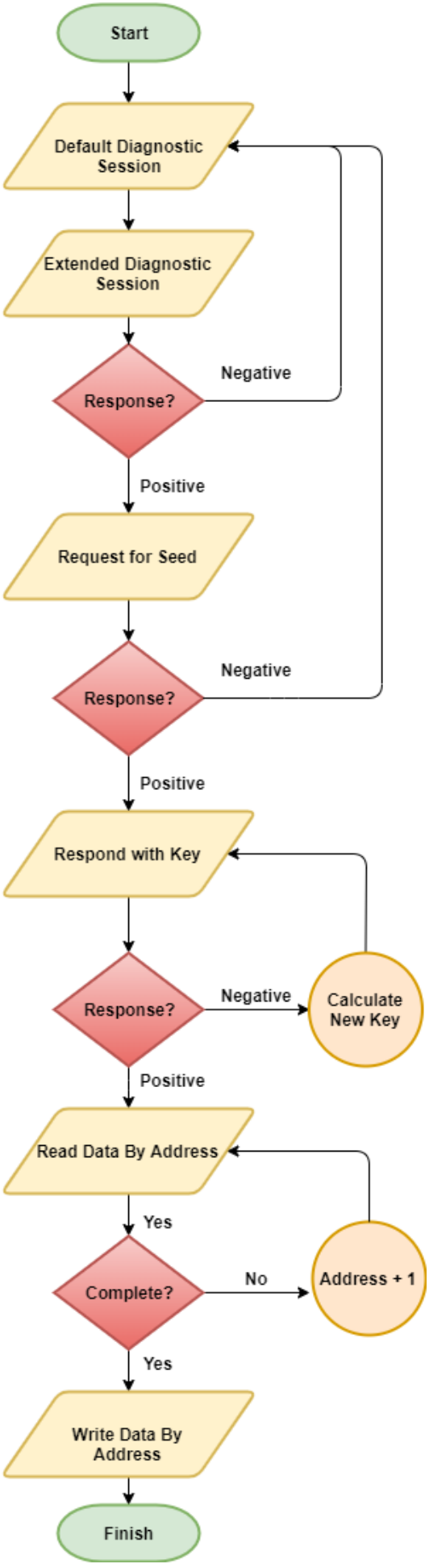

FIGURE 3.9: Flow Chart

Chapter 4

Testing and Results

4.1 Software Testing Methods

Although diagnostics standards may prescribe a fairly secure system, the real security lies in the implementation. As previous research suggests, serious vulnerabilities may be found in a faulty or insufficient implementation. Therefore, software testing methods are used in this research for data collection and analysis. This section introduces the main concepts and methods used in software testing, and how they are applied in this research.

4.1.1 Positive and Negative Testing

For testing every requirement or need, several test cases are designed. A requirement of a function normally states some input and a respective output. A positive test runs the function with an input stated in the requirements and sees whether it results in the respective output. It is called “positive" since it tests what the program is supposed to do. A negative test sees what happens when the function is run with unexpected input. It is “negative" in the sense that the test case tests what happens when the user tries to make the program behave in a way that it is not supposed to behave.

For example, consider testing a key and a lock. A positive test would involve trying if a correct key opens the lock, when turned counterclockwise in the keyhole, and locks it, when turned clockwise. A negative test would involve inserting a different key in the lock, turning it both ways, or trying to pick the lock with a piece of wire - inputs that are not in conformance with what the lock has to do in a usual case, but the complete product has to handle anyway.

4.1.2 Integration Testing

Integration testing is a term used for the testing of interfaces - the boundaries of information exchange for different components of a computer system. An ECU has an external interface for communicating with another node on the CAN network - meaning that the software running on the ECU reacts to a specific set of commands sent to it from the CAN (such as commands to program the ECU with new software). In this thesis, this interface is tested with regard to the software update process.

In integration testing, all interactions between the communicating nodes need to be tested. Some of these interactions are documented (explicit) and some are not (implicit). In the case of ECUs, interfacing is achieved with a communication protocol (UDS). The explicit interactions are documented in public standards specifications,

which are available for anyone to study. However, there are also implicit interactions. These protocol details are internal to the vehicle manufacturing company. This does not mean that these interactions cannot be used by an adversary. It only means that in an everyday situation, the adversary is assumed to not know about them.

4.1.3 Grey-box Testing

A software tester may perform tests acting as an end user of the product, or they may instead go through the program code. In black-box testing, the tester has no access to the code - the focus is on the software's external functionality. White-box tests are based on an analysis of the internal code structure of the software. Grey-box testing refers to a mixture of white-box testing and black-box testing. Figure 4.1 illustrates these different methods, treating the software as a box with code inside, and externally usable functions (illustrated as F1 and F2). Grey-box testing is performed mostly on the external functions, but sometimes parts of the internal structure are also studied for clarity and to make further decisions.

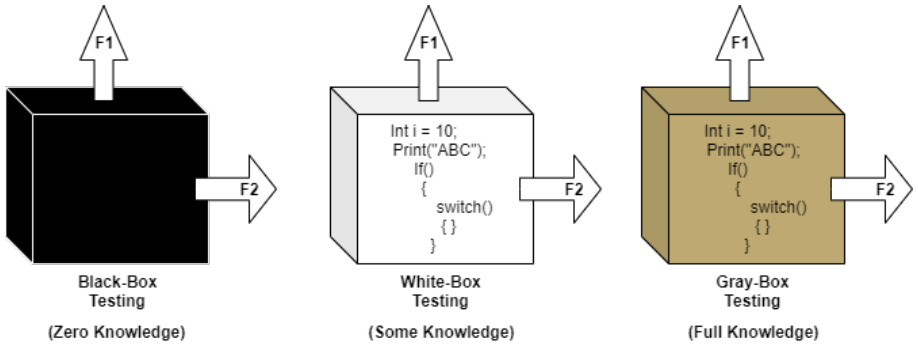

FIGURE 4.1: White-box, black-box and grey-box testing

When testing ECU security, the role of a hacker is assumed. Normally, they would not have any knowledge of the software's inner workings and treat the interface as a black box. However, there are two reasons why it is a good idea to not only perform black-box testing within this research but to mix in some white-box testing as well. Firstly, since the research is carried out on-site in the company, then internal documentation and software code is available for use. This reduces the need for a trial-and-error search for answers about the interface functionality and thereby saves time. Secondly, security through obscurity cannot be relied on - if functionality or a bug is present, it can theoretically be found and used, even when it is not publicly documented.

4.1.4 Test Automation

In some cases it is better to automate the testing process with software tools, instead of performing tests manually. This involves following testing activities which:

- need to be repeated many times over.
- are very slow to carry out manually.
- cannot be performed accurately manually.

One could buy an appropriate tool, or develop one on their own. Sometimes, open-source options are available, which can be modified to the tester's needs. In this research, to break the security session and download and upload software python script is implemented and used for testing purposes. This option allows for speedy development, and is very flexible.

4.2 Test Plan

Security vulnerabilities may lie in faulty or insufficient implementation of security requirements. Security testing employs classical software testing methods to find any such vulnerabilities, if present. Several ECUs are tested against the UDS standard documents. The aim is to collect meaningful data from the server ECU by breaking the implemented security mechanisms as per the standard. The following subsections describe which security requirements are tested, why, and how. Some of them have been chosen based on the results of previous research, and others represent the remaining security requirements in the standards.

4.2.1 Attack scenarios

There are several possible attack scenarios, which could be tested if an ECU with weak enough security is found. Here, these scenarios are defined generally. Once the security mechanisms of each ECU are mapped, it is possible to define how exactly each attack could be implemented.

scenario 1: there is no effective authorization mechanisms in place to stop the attacker from simply connecting with the ECU and loading new software on it.

scenario 2: the attacker reads the ECU software from memory, reverse engineers it and recovers secret keys.

scenario 3: is a brute-force attack where the attacker chooses a value to use as the authentication code. This value is given to every received seed, until a seed is received, to which this authentication code happens to correspond to. Then, access is granted.

scenario 4: is also a brute-force attack. In this one, the attacker has prior knowledge of a valid seed and authentication code pair. They will query seeds without replying, until the known seed is received. Then, the known authentication code is sent as a reply, and the higher security level is unlocked.

4.2.2 Security Requirements

The experiment focuses on finding flaws in the implementation of standard security features. The following security requirements will be testing.

1. The ECU is unlocked only if a SecurityAccess service is called and successfully responded to.
2. Flashing operations on the ECU can only be performed in the unlocked state.
3. Reading sensitive information (e.g. secret cryptographic keys) from the ECU memory is functionally impossible or possibly only in an unlocked state.

4. The SecurityAccess seed is non-static and random.

5. The SecurityAccess seed, key and authentication code are so long that any brute-forcing would be rendered infeasible. This means that trying through all possible seeds or keys would take so much time that the brute force method could not possibly be of benefit.

6. A time delay of sufficient length to mitigate brute-force attacks is implemented after a certain amount of failed SecurityAccess attempts.

4.2.3 Test Cases

The security requirements listed in Section 4.2.2 are tested with the following test cases. There is one or more test cases corresponding to each requirement. Some of the requirements are tested only with negative test cases because it is important to approach those points from a hacker's perspective. A hacker does not approach the ECU interface with the desire to go through the security controls as is stated in a use case for updating ECU software, because they are unlikely to have the necessary secret keys right away. Instead, they try to see if there is any other possible sequence of actions that will eventually give them access to the software updating functions. Therefore, some functions are tested with unexpected input to see whether the security still holds.

Each test case consists of a sequence of steps and an expected result. The expected result is what a well enough secured system would give, if the described steps are followed. If the given steps have the expected result, the test is considered passed, otherwise the test is failed. The test case may also define measuring something, the value of which will play a role in determining the result of the test. The measurements are used for calculating the feasibility of the brute-force attack scenario (scenario 3 described in Section 4.2.1). The results of carrying out these tests are presented in Section 4.3.

Test Case 1: Changing Sessions

Step 1: Enter Programming Session.
Step 2: If got negative response from step 1, Enter Default Session.
Step 3: Enter Programming Session.
Step 4: wait for 10 seconds after getting positive response form step 3.
Step 5: Send request Seed message.

Expected Result: After executing step 5, Server ECU should reply with negative response if the delay and change of session functionality is implemented.

Test Case 2: Long Seed

Step 1: Enter Programming Session.
Step 2: Send request Seed message.

Expected Result: a seed is acquired, which is so long that a brute-force attack is rendered infeasible.

Test Case 3: Seed Unpredictability

Step 1: Enter Programming Session.
Step 2: Send request Seed message. (should be repeated for above 20 times in order to determine how random or static the seed is).

Expected Result: all seeds are different and appear not to have an obvious pattern.

Test Case 4: Reading Memory

Step 1: Enter Programming Session.
Step 2: Send Read memory by address message.

Expected Result: security access denied.

Test Case 5: Default session after the ECU reset

Step 1: Enter Enter Default Session.
Step 2: Enter Programming Session.
Step 3: Reset ECU or Power off/On ECU.
Step 4: Enter Programming Session.

Expected Result: After the reset it should not give positive response to programming session request. It should give positive response to the default session.

Test Case 6: Unlocking

Step 1: Enter Enter Default Session.
Step 2: Enter Programming Session.
Step 3: Send request Seed message.
Step 4: Send generate Key using the python script.

Expected Result: After breaking the security key the python script will shows the correct key for the seed. If the script is not working you will notice only the negative responses from the server ECU.

Test Case 7: Writing Data To Memory

Step 1: Enter Enter Default Session.
Step 2: Enter Programming Session.
Step 3: Send request Seed message.
Step 4: After breaking the security session in previous test case 6. Send Read memory by address message.
Expected Result: security access denied.

Test Case 8: Failed Attempt Count

Step 1: Repeat the first 3 steps of Test Case 6.
Step 2: Send different keys.

Expected Result: Measure the amount of allowed failed attempts.

4.3 Test Results

Following figure shows the some of the results captured during the testing. Figure 4.2 shows the messages generated by the server ECU. These messages are logged to analyze them to extract meaningful information.

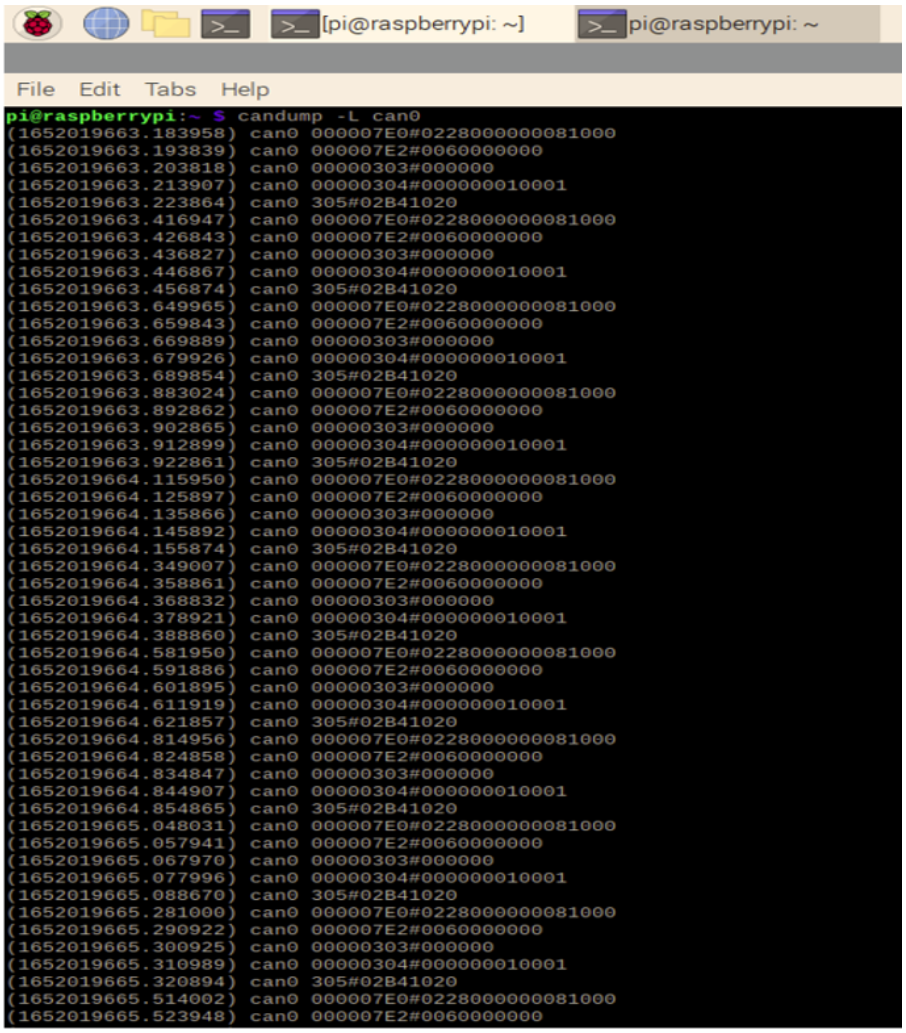

FIGURE 4.2: CAN messages from the server ECU

Figure 4.3 shows the result for the test cases related to session change and requiting the seed and seed length used for security implementation in the server ECU.

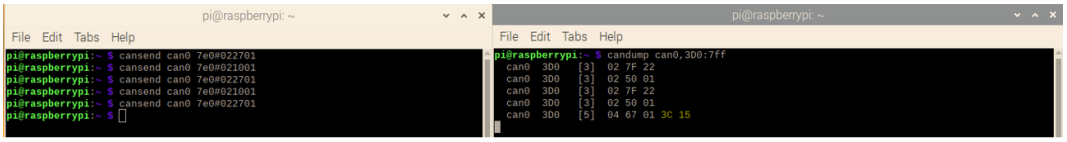

FIGURE 4.3: Session change and Seed request

Figure 4.4 shows the python script braking the security session and display the correct key for the corresponding seed. Above, test result shows the requested seed is "3C 15" and the response we got with the python script is positive and the corresponding key is "AA 0A".

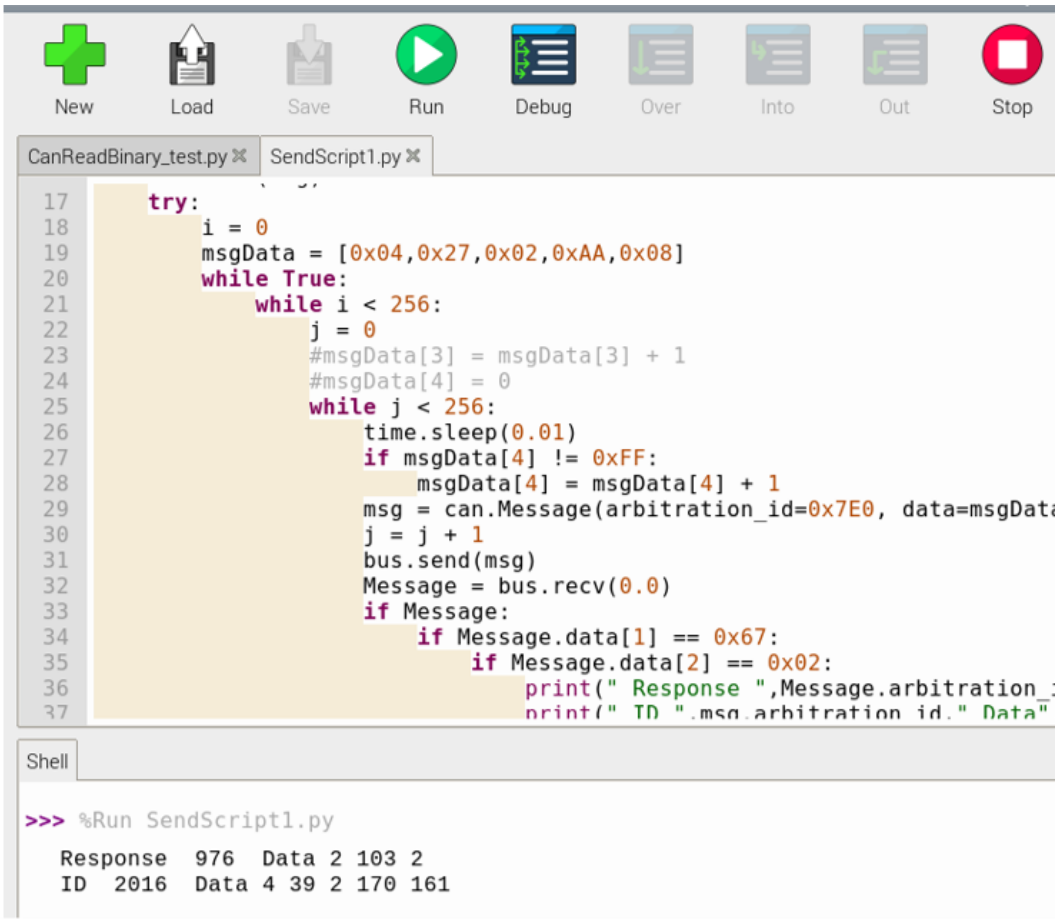

FIGURE 4.4: Key response using Python Script

Figure 4.5 shows the firmware downloaded using the python script. Read data by address uds service (0x23) have been used.

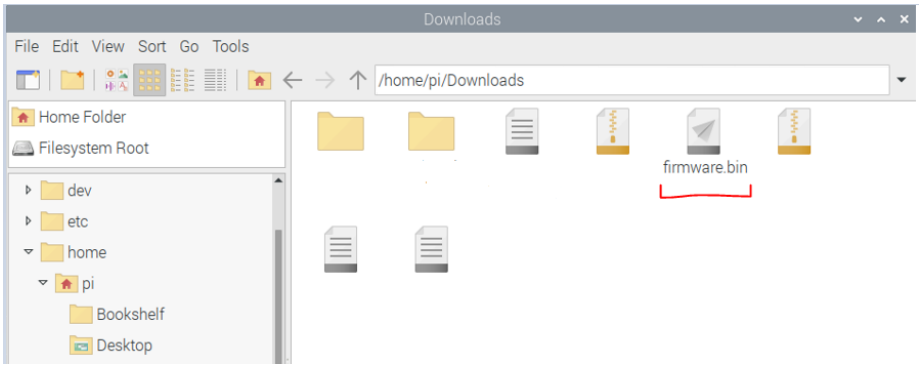

FIGURE 4.5: Downloaded Firmware File

A test is considered passed, if the results matches the expected result described in the respective test case in Section 4.2. Otherwise, the test is considered failed. Find the test overview in the following table 4.1.

Test Case No	**Questions**	**Answer**	**Result**
1	Is time delay affected by changing sessions?	Yes	Passed
2	How Log is the Seed?	2 Bytes	Passed
3	Is seed predictable?	Yes	Failed
4	Can ECU memory be read in locked state?	No	Passed
5	After the reset ECU by default goes to Default session?	Yes	Passed
6	Is Security Access required for Unlocking	Yes	Passed
7	Data writing is possible after braking security session?	Yes	Passed
7.4	Is there time delay after failed Security Access attempts?	Yes	Passed
8	How many attempts are allowed before time delay?	Not implemented	Failed

TABLE 4.1: Fields used in standard CAN frame format

4.3.1 Summary

As per the result we can say that if the security algorithm is not that strongly implemented in the ECU, Using the UDS protocol its easy to break security session and get useful information from the ECU and alter the data and re-flash it with wrong data to malfunction the ECU.

Chapter 5

Conclusion

Modern vehicles are controlled by a distributed computer system of embedded devices - Electronic Control Units (ECUs). Customising the functionality of a vehicle comes down to changing the software on ECUs. To avoid problems with road safety, as well as legal issues, it is desirable that only the vehicle manufacturer can make software alterations to ECUs controlling critical modules. This goal requires for appropriate security measures to be in place so that external parties would be unable to get access to an ECUs self-reprogramming functionality. Previous research has already identified several potentially exploitable vulnerabilities in the diagnostic interfaces of ECUs used in automobiles. Also, general-purpose attacks have been demonstrated.

This thesis continued the research by moving the focus to ECUs used in modern car, and specifically the reprogramming functionality of the ECUs diagnostic interfaces (mainly UDS over CAN). The purpose was to identify the possibly hacking the ECU software through the UDS standard for the process of performing software updates on different ECUs from a security perspective. As a result, it reports security vulnerabilities, which may lead to an unauthorized person flashing ECUs with arbitrary software. To thoroughly present vulnerabilities and their impact, the thesis has only one goal: to identify vulnerabilities and demonstrate attacks. The research was carried out as a quantitative study, using experimental research methods and a deductive approach. An experimental methodology was used to test ECU and derived test results were performed to arrive at conclusions. To evaluate the severity of the vulnerabilities found, the results were validated by performing experimental attacks.

To start of the security evaluation and meet the goal - identifying vulnerabilities - formal software testing methods were used. The subject of testing was the implemented UDS diagnostic interface of chosen ECU. The vulnerabilities pointed out in previous research, as well as the security requirements defined in the UDS standards documents, were built upon to compile a set of test cases. Each test aimed to verify the correct or sufficient implementation of a functional or cyber security requirement. Positive tests were executed to verify the presence of standardized security requirements, and negative testing was added to identify any security bugs. As the specification documents and software code used internally in the Matrickz was available for this study, then penetration testing and brute force methods were applied to save time. The brute force method was automated with a python script to save time and avoid manual errors.

Identified Vulnerabilities

The results of the security evaluation reveal several exploitable vulnerabilities in ECUs with varying security strength. The main identified problems were the following:

- Short seed and authentication code used in authentication. It is feasible to perform a brute-force attack on an ECU with a seed and authentication code length of merely 2 bytes, and an average-length failed access attempt delay.
- No authentication or encryption of messages carrying ashing data. Since CAN is a message-based protocol, then lack of higher-level authentication and sending plain text messages makes it vulnerable to man-in-the-middle attacks. After getting access to the software its easy to reverse engineer it and flash the malicious software to the ECU.

The found vulnerabilities are in line with previous research, which emphasizes similar or related problems in ECUs used in automobiles. To determine and demonstrate the significance of these vulnerabilities, they were further experimented with.

Brute-Force Attack

Previous research suggests that short seeds and keys might only provide a temporary protection from malicious security access attempts. As in the course of testing an ECU was found, which featured a 2-byte seed and authentication code, the realistic feasibility of a brute-force attack could be verified.

A random 2-byte combination was chosen and repeatedly used as the authentication code. New seeds were queried until the authentication code was accepted, and a higher security level was unlocked. This experiment was conducted a few times and succeeded in a feasible amount of time on every occasion.

Gaining access to a higher security level makes it possible to successfully download and send flashing commands with the wrong malicious software to an ECU.

5.1 Future Outlook

In future research, the other parts of the ECU software update process could also be evaluated in terms of security. This includes the management and distribution of secret keys, as well as the production, signing and distribution of flash files. These topics include a wider subject area, spanning multiple computer systems in the vehicle manufacturing organization.

An important security aspect that went completely undiscussed in this project is the algorithm for generating random numbers in ECUs. What was verified in Test case 6, was that the seed is not static or that its pattern is not obvious to the eye. However, white-box testing of the algorithm would reveal, whether the input used for calculating the seed is predictable. A predictable input also makes the seed predictable, given that the adversary knows the seed generation algorithm.

To comprehensively validate these attacks, they should be tested on a real vehicle. Recreating these attacks on a complete system may uncover restrictions, which went unnoticed in this research.

Bibliography

[1] andro Amendola. “Improving automotive security by evaluation from security health check to common criteria. White paper, Security Research & Consulting GmbH, 176”. In: (2004).

[2] CSS Electronics. “UDS Explained - A Simple Intro”. In: (2020). URL: https://www.csselectronics.com/pages/uds-protocol-tutorial-unified-diagnostic-services.

[3] ISO IEC. “ISO IEC 15408-1-2009 - Evaluation Criteria for IT Security”. In: (2009).

[4] National Instruments. “ECU Designing and Testing using National Instruments Products”. In: (2009). URL: http://www.ni.com/white-paper/3312/en/.

[5] National Instruments. “uberblick uber den Kommunikationsbus FlexRay fur den Automobilbereich”. In: (2021). URL: https://www.ni.com/de-de/innovations/white-papers/06/flexray-automotive-communication-bus-overview.html.

[6] Franziska Roesner Karl Koscher Alexei Czeskis and Shwetak Patel. “Experimental Security Analysis of a Modern Automobile”. In: (2010).

[7] AndreWeimerskirch MarkoWolf and ThomasWollinger. “State of the Art- Embedding Security in Vehicles. EURASIP Journal on Embedded Systems”. In: (2007).

[8] MDPI.com. “Sensor Technologies for Intelligent Transportation Systems”. In: (2018). URL: https://www.mdpi.com/1424-8220/18/4/1212/htm.

[9] Hiroaki Takada Ryo Kurachi and Kentaro Takei. “Evaluation of Security Access Service in Automotive Diagnostic Communication”. In: (2019).

[10] Mohan Trivedi Shane Tuohy and Liam Kilmartin. “Intra Vehicle Networks- A Review”. In: (2015).

[11] International Organization for Standardization. “Road vehicles - Unified diagnostic services (UDS) - Part 1”. In: (2013).

[12] RF WIRELESS. “Standard/Extended CAN frame format”. In: (2012). URL: https://www.rfwireless-world.com/Terminology/Standard-CAN-frame-vs-Extended-CAN-frame.html.

[13] Physics World. “How to hack a self-driving car”. In: (2020). URL: https://physicsworld.com/a/how-to-hack-a-self-driving-car/.

[14] Markus Zoppelt and Ramin Tavakoli Kolagari. “What Todays Serious cyberattacks on Cars Tell Us- Consequences for Automotive Security and Dependability”. In: (2019).