Bachelor Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Automotive Ethernet Network Identification Tool

Martin Komínek

Supervisor: Ing. Jan Sobotka, Ph.D.
Field of study: Open informatics
Subfield: Software
May 2022

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Komínek** Jméno: **Martin** Osobní číslo: **483612**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Nástroj pro automatickou identifikaci sítě Automotive ethernet

Název bakalářské práce anglicky:

Automotive Ethernet Network Identification Tool

Pokyny pro vypracování:

1. Familiarize yourself with the 802.3bw (Automotive Ethernet), DoIP, UDS standards, and the IPv6 family protocols.
2. Perform a search within CAN bus security exploration tools (e.g. Caring Caribou) for a prospective tool for diagnostics services identification.
3. Perform a search within port scanners and network mappers suitable for automotive ethernet networks.
4. Design and implement a software tool for automated identification of an automotive ethernet network focusing on a single (pre-selected) ECU.
5. The tool should identify parameters of the data link layer, network layer, and Diagnostics services over DoIP.
6. Demonstrate your results by testing with a suitable ECU (provided by the supervisor).

Seznam doporučené literatury:

[1] MATHEUS, Kirsten; KÖNIGSEDER, Thomas. Automotive ethernet. Cambridge University Press, 2017.
[2] Craig Smith. 2016. The Car Hacker's Handbook: A Guide for the Penetration Tester (1st. ed.). No Starch Press, USA.
[3] Nicolas Navet, F. and Simonot-Lion, F.: Automotive Embedded Systems Handbook, CRC PressINC, 2009.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jan Sobotka, Ph.D. katedra měření FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **19.02.2024**

Ing. Jan Sobotka, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání	Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

Tímto bych chtěl poděkovat hlavně svojí přítelkyni, která mě svojí neuvěřitelnou pílí pomáhala motivovat psát tuto práci. Dále bych chtěl poděkovat mojí mamince za její neutuchající podporu abe jejíž podpory by tato práce nemohla nikdy vzniknout. Také bych chtěl poděkovat svojí nejlepší kamarádce, za to, že si našla čas na přečtení a kritiku mojí práce. Hlavní díky ale patří vedoucímu mojí práce. Ing. Janu Sobotka, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích a hlavně za vedení mojí bakalářské práce, která mi dala hodně zkušeností do mého života.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 20, 2022

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje vs souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 20. května 2022

Abstract

Aim of this thesis is to analyze current tools that perform automatic analysis of an ECU, provide theoretical information about DoIP and UDS, and implement a new tool that would automatically identify Automotive Ethernet network and scan some supported services on ECU. Firstly we explore what technologies are used and than we look at tools, that were developed for previous technology CAN. Next chapter describes methods to discover IP address of an ECU and in the rest of the thesis we describe how to design, implement and test this application. At the end, we evaluate results and suggest some possible future improvements.

Keywords: DoIP, Automotive Ethernet, UDS, diagnostic

Supervisor: Ing. Jan Sobotka, Ph.D.

Abstrakt

Cílem této práce je prozkoumat současné nástroje na automatickou analýzu funkcí řídící jednotky a implementovat nový nástroj, který bude identifikovat parametry současné techonologie Automotive Ethernet a provede scan podporovaných funkcí. Nejdříve se seznámíme s použitými technologiemi a podíváme se na nástroje pro sběrnici CAN. V další kapitole zjistíme jak odhalit IP adresu ECU a ve zbytku práce se podíváme na návrh, implementaci a testování nového nástroje. Na konci zhodnotíme výsledek práce a navrhnu případné možnosti na vylepšení.

Klíčová slova: DoIP, UDS, diagnostika,Automotive Ethernet

Překlad názvu: Nástroj pro automatickou identifikaci sítě Automotive ethernet

Contents

Figures

Tables

Acronyms

ARP Address Resolution Protocol.

CAN Controller Area Network.

DHCP Dynamic Host Configuration Protocol.

DID Data by identifier.

DNS Domain name system.

DoIP Diagnostic over Internet Protocol.

DTC Diagnostic trouble codes.

ECU Electronic Control Unit.

GUI Graphical User Interface.

IEEE Institute of Electrical and Electronics Engineers.

IP Internet Protocol.

OSI Open Systems Interconnection.

PAM Pulse-amplitude modulation.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UDS Unified Diagnostic Services.

UI User Interface.

XCP Universal Measurement and Calibration Protocol.

Chapter 1
Introduction

In recent years, modern cars have radically developed, and each generation has more electronics than the previous one. This development allows to bring new features from safety to entertainment and dramatically improves the capability of each car in terms of driving or monitoring space around the car. It is clear that all development aims to continue this trend and slowly change the whole industry from driving a car to being a passenger.

To achieve this goal, car will need to have a lot of sensors to monitor its surroundings at any situation. All of that data needs to be acquired by control units in the car to be processed and interpreted. Another example where data needs to be transferred is entertainment. Passengers want to do something during their ride, like watching movies or playing computer games, which requires a lot of data.

The automotive industry needs thin wire harness to reduce the cost and weight of a vehicle and at the same time have the capacity to transport all data. In history, CAN was used for this task, but with the ever-growing need for speed, CAN is quickly being replaced with Automotive Ethernet, which is designed to be cheap while offering more than 200x more bandwidth than CAN. DoIP and UDS are now implemented by some major manufacturers and it is becoming dominant in the market. With the new change from CAN to Ethernet, many open source projects based on CAN get outdated and need to be rewritten into new standard.

During my research, I did not find any paper that would cover this topic or an open-source project that would allow users to establish a connection to ECU and automatically diagnose what type of requests it supports. This thesis should help people understand what is happening in the background and help them further improve the capability of tools like this.

In this thesis, needed technologies will be introduced. Then I will try to search for CAN projects that could be helpful or used as parts for my tool for automatic network identification. After that, I will search for other usable tools and design and implement a tool with GUI to make the tool easy to use and accessible to a wider audience.

Chapter 2

Technologies

In this chapter, I will introduce needed technologies and a short summary of them. Many of these technologies are new and are used by a small number of people, so not many sources are available, and they could be difficult to read.

2.1 Automotive Ethernet

Automotive Ethernet is a physical layer specified in 802.3bw by IEEE. The most significant difference between standard Ethernet and Automotive Ethernet is the used cable. Standard Ethernet cable has eight wires. These wires are divided into four pairs of twisted wires. Standard Ethernet has dedicated transmit and receive paths and can transmit data up to 100 meters. Automotive Ethernet consists of an unshielded pair of copper wires, which reduces its price and weight, making it ideal for transport and competitive against current cheap technologies such as CAN while having higher bandwidth than FlexRay[1]. To achieve high speed, both cables receive and transmit data simultaneously. Automotive Ethernet uses PAM3 signalling at a speed of 66.667 Mb/s to achieve a transport speed of 100 Mb/s.The newer version standardised by IEEE 802.3bp is even more impressive, with PAM3 signalling at a speed of 750 Mb/s and modulation of 80B/81B, it can transmit up to 1000 Mb/s[2]. Automotive Ethernet has a maximum range of 15 meters.

2.2 DoIP

DoIP is short for Diagnostic over Internet Protocol and is standardised by ISO 13400-2. It serves as a fourth transport layer of OSI model to establish and maintain a connection. DoIP facilitates diagnostic communication between an external device and ECU using Internet Protocol, using both TCP and UDP[3]. It serves the same purpose as ISO-TP on CAN. DoIP enables to use of automotive diagnostic services exposed through UDS over IP. By standard, all equipment implementing DoIP features Vehicle announcement and discovery, error handling, vehicle basic status information retrieval, connection establishment, and data routing[4].

2.3 UDS

Unified Diagnostic Services (UDS) is a communication protocol used in automotive ECU to provide diagnostics, updates, testing, etc. It is standardized by ISO 14229, UDS is currently used by all tier 1 manufacturers. It provides various services like diagnostic and communications management, data transmission, remote activation etc. UDS uses 5th and 7th layer of OSI model[5][6].

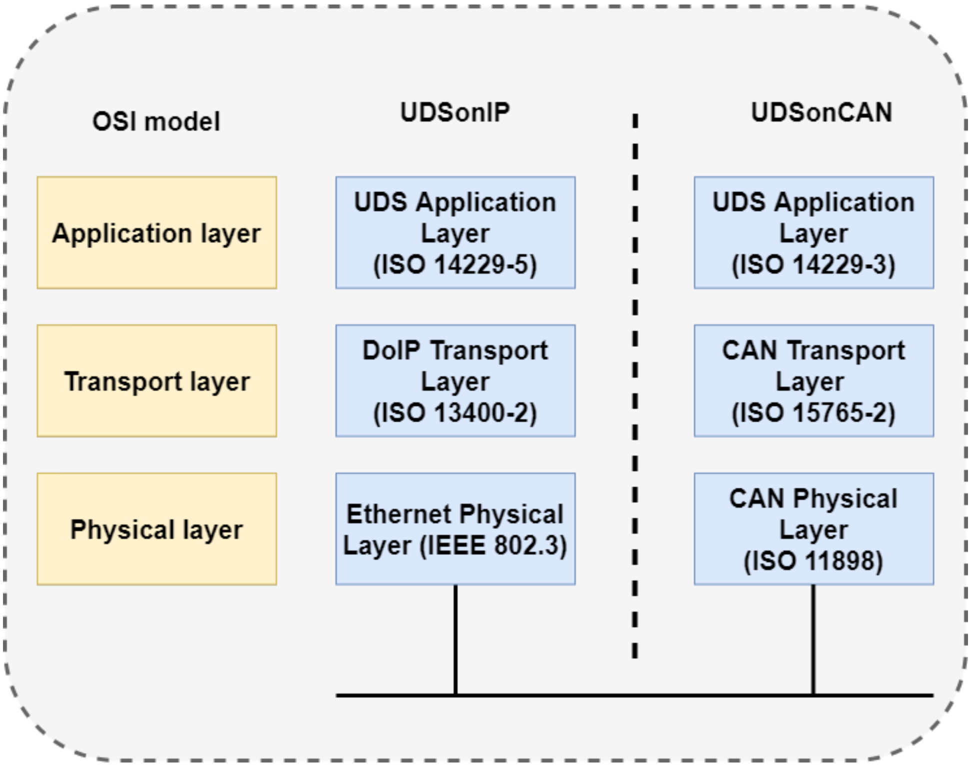
2.4 IPv6

Internet protocol version 6 (IPv6) is the latest version of Internet protocol. It provides a unique identification for each device in the network and further improves IPv4. The biggest change is the size of the address. IPv4 address is 32bit long, but IPv6 address is four times longer with 128bit[7]. That generates so many possible combinations that we will never run out of free addresses in our lifetime. Even though that IPv6 has larger addresses, header was reworked, and its length is 40 bytes fixed. For comparison, IPv4 header length ranges from 20 to 60 bytes. In addition to larger address space, it supports stateless address autoconfiguration, so devices do not require manual or DHCP configuration. Furthermore, it substitutes the ARP protocol with Neighbour Discovery Protocol instead. IPv6 does not support broadcast and instead uses multicast, which is more efficient in terms of bandwidth utilization. Security is also improved as opposed to IPv4. The use of IPSec is mandatory, and address space is so vast that it makes it impossible to scan the network for possible computers with security vulnerabilities[8].

Chapter 3

CAN tools

Before I start developing my system, I should find out if there are some projects that already solved this problem or would simplify development significantly. CAN projects are definitely the first place where I should start because the Application layer should be similar or the same, and I could just modify the transport layer, as you can see in this reference 3.1 to the OSI model.



OSI model
UDSonIP
UDSonCAN
Application layer
UDS Application Layer (ISO 14229-5)
UDS Application Layer (ISO 14229-3)
Transport layer
DoIP Transport Layer (ISO 13400-2)
CAN Transport Layer (ISO 15765-2)
Physical layer
Ethernet Physical Layer (IEEE 802.3)
CAN Physical Layer (ISO 11898)

Figure 3.1: DoIP and UDS with respect to OSI model[9]

3.1 CaringCaribou

CaringCaribou is a car security tool for CAN bus. It contains various modules such as XCP used for measurement and calibration of sensors, sending or dumping CAN massages and also UDS and newly even a DoIP, which is relevant for my thesis

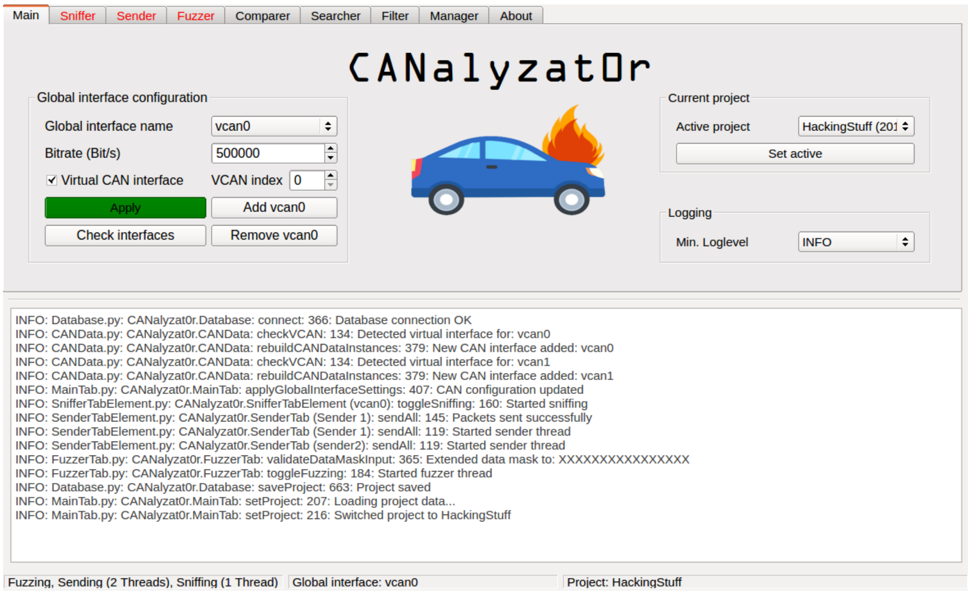
When I looked closely at the UDS module, I find out that it has five services. The first service is used to discover arbitration IDs, where ECU responds to an incoming diagnostic request. Otherwise, the ECU will dump arriving packets. The second service scans for services supported by the ECU. The third one requests ECU to reset, and the fourth one forces ECU in diagnostic mode to stay active. Both third and fourth services are not so crucial. The last one scans the range of DID and dumps their value to terminal, which is important, because these data can be useful. Although that UDS module of CaringCaribou has services I could use, its implementation is done for CAN, which has a different header, so I cannot use this module for Automotive Ethernet.

Caring Caribou has a new module called DoIP, which was added on the 23rd of March 2022, so during the writing of my thesis. Main services are the same as in the UDS module. Discovery, Scan services, and scan DID are the same functions as in the UDS module, just for a different transport layer. I tested this module as it supports the same services I want to implement. Discovery service works well, but it takes a lot of time. It takes ECU logical address from Vehicle Announcement message and scans a range of addresses to identify tester logical address. During my testing, Scan service didn't manage to identify any of the available services, suggesting that this service doesn't work properly, but I don't have another ECU to confirm this. Reading values from DID didn't work for me either, because it couldn't interpret data it received and crashed. Another small problem I encountered is that even though you don't use it, you have to install library python-can to run Caring Caribou. It is not a big problem, but as Caring Caribou was designed for CAN, I think it would be better if DoIP module would be a standalone program.

Caring Caribou has 14 contributors, and they added a new module this year, so the project is maintained, and there are not many issues. I can take inspiration from how they implemented these services and try to implement my own functions but overall DoIP module didn't match my expectation and I wouldn't use it to diagnose Automotive Ethernet.

3.2 CANalyzat0r

CANalyzat0r is a tool that has a GUI, so it should be easier to use. It also analyzes ECU via CAN bus and also uses UDS to diagnose available services. This tool is more complex and supports many other features. It enables logging of all action, basic UDS sniffing, threaded sniffing and fuzzing, packet filters and many more. I already have an inspiration for how the UDS module should be designed, but this project offers something else. GUI design and overall architecture is really good and can be an inspiration for my tool. Features like logging and saving your setup and results can be very helpful. Here 3.2 is example of how a CANalyzat0r GUI looks like. With six contributors and the last update in 2020, and no issues on GitHub, this tool is finished, and I don't expect any significant changes in the future.



Global interface name: vcan0
Bitrate (Bit/s): 500000
☑ Virtual CAN interface
VCAN index: 0
Active project: HackingStuff (201
Min. Loglevel: INFO

```
INFO: Database.py: CANalyzat0r.Database: connect: 366: Database connection OK
INFO: CANData.py: CANalyzat0r.CANData: checkVCAN: 134: Detected virtual interface for: vcan0
INFO: CANData.py: CANalyzat0r.CANData: rebuildCANDataInstances: 379: New CAN interface added: vcan0
INFO: CANData.py: CANalyzat0r.CANData: checkVCAN: 134: Detected virtual interface for: vcan1
INFO: CANData.py: CANalyzat0r.CANData: rebuildCANDataInstances: 379: New CAN interface added: vcan1
INFO: MainTab.py: CANalyzat0r.MainTab: applyGlobalInterfaceSettings: 407: CAN configuration updated
INFO: SnifferTabElement.py: CANalyzat0r.SnifferTabElement (vcan0): toggleSniffing: 160: Started sniffing
INFO: SenderTabElement.py: CANalyzat0r.SenderTab (Sender 1): sendAll: 145: Packets sent successfully
INFO: SenderTabElement.py: CANalyzat0r.SenderTab (Sender 1): sendAll: 119: Started sender thread
INFO: SenderTabElement.py: CANalyzat0r.SenderTab (sender2): sendAll: 119: Started sender thread
INFO: FuzzerTab.py: CANalyzat0r.FuzzerTab: validateDataMaskInput: 365: Extended data mask to: XXXXXXXXXXXXXXXX
INFO: FuzzerTab.py: CANalyzat0r.FuzzerTab: toggleFuzzing: 184: Started fuzzer thread
INFO: Database.py: CANalyzat0r.Database: saveProject: 663: Project saved
INFO: MainTab.py: CANalyzat0r.MainTab: setProject: 207: Loading project data...
INFO: MainTab.py: CANalyzat0r.MainTab: setProject: 216: Switched project to HackingStuff
```

Fuzzing, Sending (2 Threads), Sniffing (1 Thread) | Global interface: vcan0 | Project: HackingStuff

Figure 3.2: Example of GUI interface for CAN

3.3 CANToolz

CANToolz also aims to analyze the CAN network but was built using modules and the idea of chaining them together. So with a single install, it should be ready to use without the need to install several projects and try to make them work together. CANToolz supports ECU discovery, MitM testing, scanning, fuzzing and more. Unfortunately, with the last real update in 2017, 17 unresolved issues and the creator of the tool moving to another project, this tool is not suitable as a model for a new application.

3.4 Summary

In this chapter, I searched for various tools, that were developed for analyzing CAN bus and analyze their potential to be used in my tool for Automotive Ethernet. In short, the most promising tool was Caring Caribou with its implementation of the DoIP module, which implements UDS specified by ISO 14229 and can be used as an inspiration for my UDS module based on automotive Ethernet. Other tools are not so helpful, but I can take inspiration from their design. CANalyzat0r is great for its GUI, which makes it easier for inexperienced users to use. CANToolz is interesting for its modular design that enables to chain modules together, so only one tool is needed to test and research ECU, and both of these ideas should be kept in mind when designing my own tool. Here is a summary of all projects.

Name	Features	Status	Last Update
Caring Caribou[10]	Modules	Active	1.4.2022
CANalyzat0r[11]	GUI, Managment, Logging	Finished	13.2.2020
CANToolz[12]	Modular design,Pipe	Abandoned	29.11.2017

Table 3.1: CAN tools summary

Chapter 4

Identifying network parameters and services

4.1 IP address discovery

To diagnose and analyse ECU, I first need to know its IP address. This section describes three ways to identify needed IP addresses and problems and solutions that come with it. I will explore different approaches in both IPv4 and IPv6 and how to establish a connection using DoIP.

4.1.1 Network scan in IPv4

When I want to scan for IPv4 addresses on a local network, it is pretty straightforward. In order to scan IPv4 network, I need to identify which part of IP address is the network address and which part is a host address. To do that, a subnet mask needs to be applied to the IP address using bitwise AND. As a result, I get the network address. Now I have two options. The first option is to brute force all combinations and try to connect to all possible addresses, and perform a port scan on each address. The second option is to add bits of 1 at the end of the address to get a broadcast address that will broadcast to all devices on the local network. For example, when subnet mask is 255.0.0.0 and the network address is 144.0.0.0, broadcast address would be 144.255.255.255.

4.1.2 Network scan in IPv6

Scanning an IPv6 local network is different. Thanks to its much larger address space of 128 bits, brute-forcing is impossible as a number of combinations are too high. I also cannot use broadcast because it's not supported in IPv6. Instead, I can use multicast. As I need to multicast to only local addresses, I will use the FF02::1 address. In IPv6, local addresses need to add scope id of the interface that I want to use. I can add this manually or I can extract all available interfaces and try all of them. Then I need to ping FF02::1%scopeID and wait for unique addresses that will respond.

4.1.3 Vehicle announcement

When an ECU boots, it is mandatory, according to specification, to send 3 Vehicle Announcement messages with 1-2 seconds between them. The message contains the ECU IP address, logical address and some other parameters to establish a connection and identify ECU. These messages are sent to the UDS discovery port. There I can receive the message and extract these parameters and try to establish a connection. This depends on the condition, that external equipment is ready to receive messages before the last message is sent. In case I missed the initial vehicle announcement messages, I can send a vehicle identification request to ECU on UDS discovery port using multicast or unicast. To that message, ECU is mandatory to respond with a vehicle response message. The diagram below 4.1 show possible sequences to establish a connection.

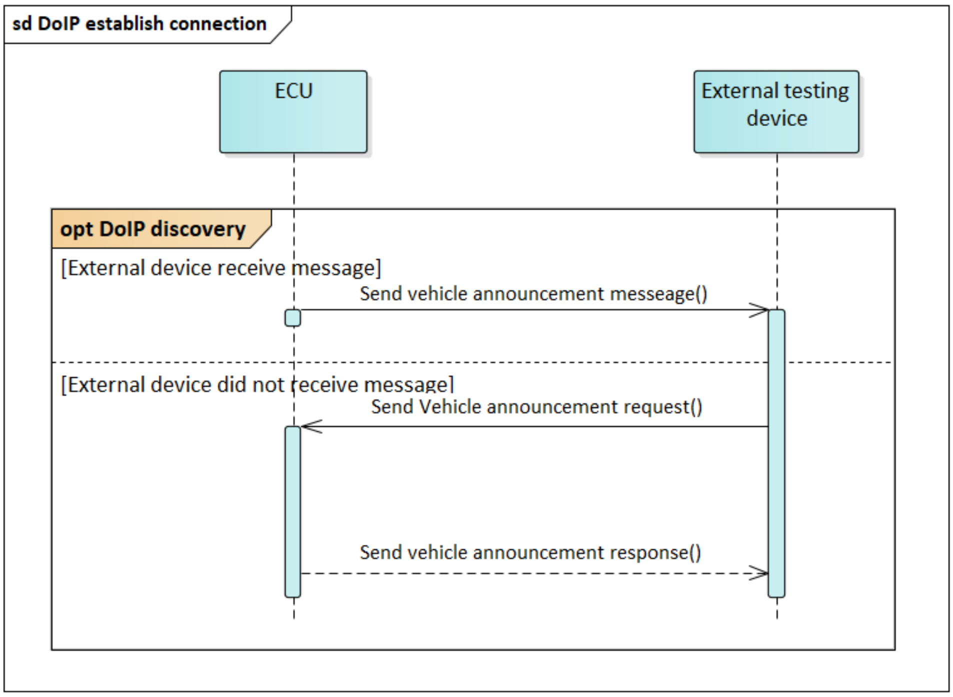

sd DoIP establish connection
ECU
External testing device
opt DoIP discovery
[External device receive message]
Send vehicle announcement messeage()
[External device did not receive message]
Send Vehicle announcement request()
Send vehicle announcement response()

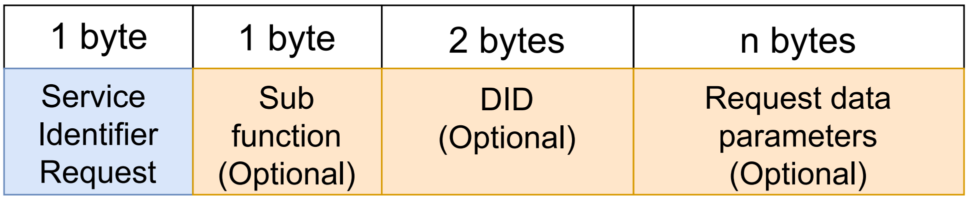
Figure 4.1: Structural diagram of DoIP discovery

4.1.4 Network scan

The last method to identify IP address of an ECU is listen to ongoing traffic on the interface, where I know the ECU is connected. If the ECU is connected to the network, there is a good chance that there is some traffic. It can broadcast some messages, when it is trying to resolve Neighbour Discovery Protocol or it already communicates with some device. I can use tools like Wireshark to intercept this communication and try to connect to this ECU.

4.2 UDS services

To understand how a UDS discovery can be done, I have to look at its request and response message structure.



1 byte	1 byte	2 bytes	n bytes
Service Identifier Request	Sub function (Optional)	DID (Optional)	Request data parameters (Optional)

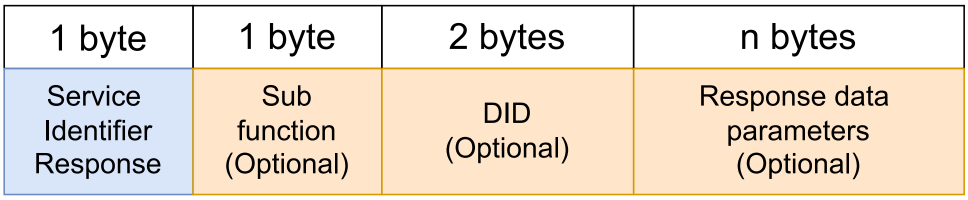
Figure 4.2: Structure of UDS request message

First byte of this message identifies what UDS service I request. This can be, for example Diagnostic Session Control, ECU Reset, Read DID and many others. I created an overview of UDS services 4.1. Response of the ECU will depend on the passed parameters, like what sub-function or data I pass, but not all services has sub-function as seen below.

Service name	SID	Available in Default	Sub-function
Diagnostic session control	0x10	Yes	Yes
ECU reset	0x11	Yes	Yes
Clear diagnostic information	0x14	Yes	No
Read DTC	0x19	Yes	Yes
Read DID	0x22	Yes	No
Read memory by address	0x23	Yes	No
Security access	0x27	Yes	Yes
Communication Control	0x28	Yes	Yes
Read data by periodic identifier	0x2A	Yes	No
Dynamically define DID	0x2C	Yes	Yes
Write data by DID	0x2E	Yes	Yes
Input Output control by Id	0x2F	No	No
Routine control	0x31	Yes	Yes
Request download	0x34	No	No
Request upload	0x35	No	No
Transfer data	0x36	No	No
Write memory by address	0x3D	Yes	Yes
Tester present	0x3E	Yes	Yes

Table 4.1: Overview of UDS services[13]

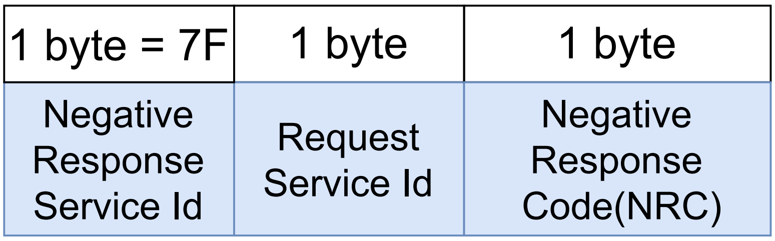
When ECU supports service it will response with Positive response message, which you can see on this image 4.3. It has the same structure as request. Difference is in the first byte, which identifies Service response instead of request. This ID is usually a request ID plus 0x40. For example for 0x10, which is a Diagnostic session control, response starts with 0x50. Other two parameters stay the same, followed by the requested data.



1 byte	1 byte	2 bytes	n bytes
Service Identifier Response	Sub function (Optional)	DID (Optional)	Response data parameters (Optional)

Figure 4.3: Structure of UDS positive response message

When for some reason ECU cannot send positive message, it will send a Negative response message, which structure you can see on 4.4 image. First byte is always 7F. After that comes the service ID of request, so that the tester knows what request message it responds to. Last byte is Negative response code. This represents why is the message negative. Example of some of the NRC codes is seen in table 4.2



1 byte = 7F	1 byte	1 byte
Negative Response Service Id	Request Service Id	Negative Response Code(NRC)

Figure 4.4: Structure of UDS negative response message

To perform diagnostic, I will send UDS request and then wait for a response. When it is positive, I know that ECU supports this service, and I can try to extract data. If it sends a Negative response, I can look at NRC. If it is 0x11, I know ECU does not support this service as it is a code for a service not supported. In other cases, I cannot decide, as there are too many possibilities. Message can have wrong combination of parameters, service can be supported only in extended sessions, ECU can be busy, etc.

UDS NRC code	Description
0x10	General reject
0x11	Service not supported
0x12	Sub-function nor 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	Exceeded 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, but response is pending
0x7E	Sub-function not supported in active session
0x7F	Service not supported in active session

Table 4.2: Overview of NRC codes[13]

Chapter 5

Tool selection

After analysing existing solutions for Automatic diagnostic on CAN in chapter 3, none of the existing solutions can be reused in my program, so I have to write a new application from scratch. To do that I need to choose tools, that I will use to create a GUI interface and diagnostic of ECU.

5.1 GUI Framework

I choose Python as a programming language, I need to look for available GUI Frameworks that work with Python. All of the frameworks are free to use for non-commercial applications.

5.1.1 PyQt5

Pyqt5 is a python interface for Qt, which is a set of C++ libraries for creating GUIs using widgets. It runs as cross-platform applications that run on Windows, Linux or macOS. Because QT is written in C++, it is fast and takes less space. PyQt enables to take this speed and, at the same time, makes it easier to develop an application by using Python. It enables to design of UI with drag and drop tools, which speeds up the creation of applications. It supports many built-in widgets that look modern and tools to develop custom widgets, and there are plenty of tutorials on how to use it. With that said, you have to install this library to use it. It's harder to use for beginners, and each widget takes more lines in code as each setting is on the new line, so the code takes more space, and it is harder to navigate. There is also a lack of Python-specific documentation.

5.1.2 Tkinter

Tkinter is the standard GUI library for Python, so in order to use it, you don't have to install anything else. It works across all platforms, and it is easy to understand and master. It has a very simple syntax and is stable. There are a lot of tutorials on Tkinter. Widgets look old as the Tkinter's first release was in 1991. Compared to other GUIs, there are not so many widgets, and some more complex widgets are lacking completely. Changing the scene in a window is, in my opinion, harder than it should be.

Figure 5.1: Example of Tkinter[14]

5.1.3 CustomTkinter

Custom Tkinter is a new GUI framework based on Tkinter. It is very new, as development started in 2021, and it has a very modern look and has the same syntax as Tkinter. The only difference is naming. For example, instead of a Label, a new name is CTkLabel. It is also written in Python, but unlike Tkinter, you have to install an additional library.

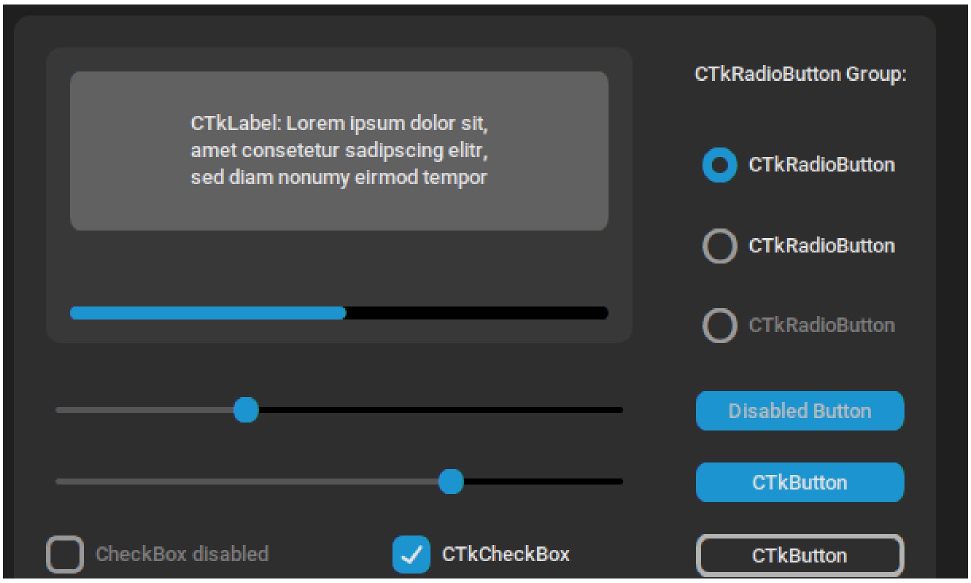

CTkLabel: Lorem ipsum dolor sit, amet consetetur sadipscing elitr, sed diam nonumy eirmod tempor

Figure 5.2: Example of CustomTkinter[15]

5.1.4 Used framework

I decided to use Tkinter to create GUI, as the size of the project is relatively small, and I had no previous experiences with these GUI frameworks, so Tkinter was the easiest to learn, and it was part of the standard library, so that makes deployment of application easier. Development was slower because Tkinter didn't have a drop and drag tool. After I finished all development, I found CustomTkinter, so I decided to try it, and manage to implement it into my application. Application now has more modern look and only downsize is that tool requires to install a new library.

5.2 DoIP Transport layer

Implementation of DoIP has only one library in Python. DoIP client is a new library started in 2020. It is a library that implements all necessary functions according to standard ISO 13400-2. The library is still in development as they added support for IPv6 just 16.2.2022, so at the beginning of writing this thesis.

5.3 UDS

As every major manufacturer of ECUs uses UDS, I need to choose a library that would help us create and send UDS Requests and Responses. I found three libraries that implement UDS in Python. Udsoncan, python-uds and py-uds. Python uds could work with DoIP client, as creator of DoIP client created fork to run it, but it is still not merged, and library has many unresolved issues, so I will not use this library. Py-uds is a new library that started its development last year and wants to implement futures such as Client and Server simulations. Today it is still under development and it does not have support for Automotive Ethernet, so I will not use this library too, which lefts only one library left.

5.3.1 udsoncan

Udsoncan started its development in 2017 and today fully implements UDS standard. It offers four layers of sending a UDS message, starting with raw connection of hex data, and ending with services that create a request, send it, wait for a response, and interpret it. For lack of alternatives I explained above, I choose to use this library.

5.4 Architecture

Main purpose of my application is to perform an Automatic diagnostic of an ECU with a GUI. I based my application as a desktop application so everybody can download it, connect the ECU and perform analysis.

I chose to develop this application with Model-View-Controller(MVC) architecture. This enables separation of how data are stored, used and viewed. In this architecture when user creates an event in View, Controller will handle this event, decides what function to use and uses Model to get stored data. When data are loaded, it returns them to Controller to change View as needed.

Advantages of MVC[16]:

- Separation of functionality. Each section does only what some operations.
- Easy to maintain.

- This architecture helps to test components independently.
- Easy to swap components. If I want to have a different GUI, I will just change view section.

Disadvantages of MVC:

- It is harder to watch flow of application.
- More complexity.

Chapter 6
Implementation

In this chapter, I will describe how was my application built, how I implemented necessary functions and what problems couldn't be solved and the limits of my application. Before I started programming, I chose Python as a programming language. It's easier to write code with it, and it has many libraries that make my work easier. I developed this application for Linux as I used some commands from the bash terminal, and anybody can install it for free. I used Git, so I could version my application, and I used Pycharm as my IDE.

6.1 Interface

The first thing I needed to do was to identify available network interfaces of the computer, so the user of the program could choose the interface on which the program could analyze network traffic. To do that, I used a command from terminal **ifconfig** that will list all available interfaces together with information about that interface. I only need the name of the interface, so I used a library called ifconfig-parser that takes the output from ifconfig and divides it into attributes. Here you can see an example from the program.

```python
from ifconfigparser import IfconfigParser

def identify_interfaces():
    process = subprocess.run(['ifconfig'], capture_output=True)
    console_output = process.stdout.decode('UTF8')
    interfaces = IfconfigParser(console_output=console_output)
    return interfaces.list_interfaces();
```

6.2 IP address discovery

In Chapter 4, I described three ways to get the IP address of an ECU. In this section, I will describe how I implemented these methods and some limitations and problems with them.

6.2.1 IPv6 network scan

I need to send a multicast message to a local network. Before I can do that, I need an interface on which I will send this message. How to get the interface I showed in the previous section. Now I need to send multicast message ff02::1 using terminal command ping. After I do that, the program will wait for responses and parse the output to identify the unique address that responded to the ping.

```
def scan_net(laninterface):
    address = 'ff02::1%' + laninterface
    process = subprocess.Popen(['ping6', '-c 3', address],
    ↪  stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    network_list = []
    while True:
        output = process.stdout.readline()
        temp = output.strip().split()
        if len(temp) > 5:
            address = temp[3].decode('UTF8')[:-1]
            if address not in network_list and
            ↪  address.startswith("fe80"):
                network_list.append(address)
        return_code = process.poll()
        if return_code is not None:
            break
    return network_list
```

6.2.2 Network listener

Another method to get the IP address of the ECU is to simply monitor network traffic. To do that, I used Pyshark, which is a Python library that wraps tshark, which is a terminal version of a Wireshark. Pyshark enables to read packets from a file or live capture ongoing traffic. It can be done for some amount of time or for some number of packets. Unfortunately, capturing for some amount of time did not work, and it is an unresolved issue on GitHub. So I choose to capture some amount of packets and identify all unique IP addresses. This means that until some amount of packets is received, the application freezes for time, which depends on ongoing network traffic.

```
def listen_network(interface):
    address_captured = []
    try:
        capture = pyshark.LiveCapture(interface=interface)
        for packet in
        ↪  capture.sniff_continuously(packet_count=5):
            address = None
            if "IPV6" in packet:
                address = packet["IPV6"].src
            else:
                continue
            if address not in address_captured:
                address_captured.append(address)
        return address_captured
    except:
        raise StopIteration
```

6.2.3 VLAN identifier

Some ECUs communicate only at a specific VLAN. To find out if ECU has VLAN, I can again use Pyshark and find out if there is another layer with VLAN and return its id. Then I can run a script that will add a new interface with the VLAN id added. Unfortunately, this is not a regular command and requires sudo access. I could do that inside the app, but the script needs to be owned by root. Also, sudo setting needs to be changed to run without the need for a password. This has to be done manually and is too complicated for an average user. On the other hand, after setting this configuration, the application can run the script by itself, and a new interface is added automatically. The second option is that after the user gets the VLAN id from the application, he needs to run the script manually with the VLAN id as a parameter. This is, in my opinion, easier for an average user.

```
def get_vlan(interface):
    capture = pyshark.LiveCapture(interface=interface)
    try:
        for packet in
         ↪  capture.sniff_continuously(packet_count=20):
            if "VLAN" in packet:
                return packet['VLAN'].id
    except:
        raise StopIteration
```

6.2.4 Vehicle announcement

In my opinion best method how to get only ECU address is simply to use the Vehicle announcement message that ECU sends. It contains all important data about ECU as IP address, port and ECU logical address. To receive this message and communicate with ECU, I used the python-doipclient library. In the example below, you can see how to extract this information using DoIPClient. As I want to use IPv6, I have to specify that via bool parameter, so my application doesn't work with IPv4 ECU.

```
address, announcement =
 ↪  DoIPClient.await_vehicle_announcement(ipv6=True)
logical_address = announcement.logical_address
ip, port,_,_ = address
```

6.3 Additional info about IP address

Before I will connect to ECU and start UDS diagnostic, it would be useful to know additional info about chosen IP address. Discovering what ports are running on an IP address can be useful, as many services use default ports. To do that, I can use nmap tool, which is a bash command in linux. "Nmap is used to discover hosts and services on a computer network by sending packets and analyzing the responses."[17] By default, Nmap scan only TCP ports, but parameters can be added to scan both UDP and TCP ports.

```
def scan_address(address):
    if ":" in address:
        process = subprocess.run(['nmap', '-6', address],
        ↪  capture_output=True)  # add 'p' to scan all ports
    return process.stdout.decode('UTF8')
```

Ports are not the only thing I can identify about IP addresses. By listening to ongoing traffic and analyzing packets, I can get MAC addresses as well. Another piece of information I can get is to simply ping the IP address to find out if it responds to ping and how long it takes. The last information I try to find about the IP address is its domain name on DNS server. Example from the application of extracting this information.

```
 for packet in capture.sniff_continuously(packet_count=5):
    if "IPV6" in packet:
        address = packet["IPV6"].src
        if address in ip:
            mac_address = packet[0].src_resolved
        else:
            mac_address = packet[0].dst_resolved
 output = process.stdout.readline()
    temp = output.strip().split()
    while len(temp) > 0 or output.decode('UTF8') == "\n":
        output = process.stdout.readline()
        if output.decode('UTF8') != "\n":
            temp = output.strip().split()
            avg = temp[1].decode('UTF8')
            if "avg" in avg:
                ping_responses = temp[3].decode('UTF8')
                split_ping = ping_responses.split('/')
                ping_avg = split_ping[1]
                it_pings = True
                break
    strip_ip = ip
    if '%' in ip:
        result = ip.find('%')
        strip_ip = ip[:result]
    process = subprocess.Popen(['nslookup', strip_ip],
     ↪  stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    output = process.stdout.readline()
    temp = output.strip().split()
    name_of_ip = temp[3]
    return mac_address, it_pings, ping_avg, name_of_ip
```

6.4 Connection to ECU

Now when I have an IP address, I can try to connect to an ECU. To establish a connection, I use DoIP Client, which is a library that I described in the chapter 5. The connection can be simply established like this.

```
is_doip = False
try:
    doipclient = DoIPClient(ip, 0x0000, activation_type=None)
    is_doip = True
except:
    messagebox.showerror("Error", "Couldn't connect probably
     ↪  not an ECU")
```

The problem is that ECU has a logical address that I don't know. Standard takes care of this by forcing ECU to send a logical address together with a Vehicle announcement response message. This message is broadcasted three times after turning on the ECU. I can restart the ECU so I can receive this message, or I can use connection and send a Vehicle announcement request on which the ECU must respond with a Vehicle announcement response. During my testing I discovered, that ECU supports this request and response with Vehicle announcement response, but only with when DoIP Client has correct ECU logical address, which I wanted to get from response. Only way I can extract a logical address from ECU is to close the old connection, restart the ECU manually and await Vehicle announcement response.

```
doip_client.close()
#manually restart ECU
address,announcement =
 ↪  DoIPClient.await_vehicle_announcement(ipv6=True)
logical_address = announcement.logical_address
```

Now that I have a new connection, I can finally perform some diagnostic using UDS. The problem is that some ECUs will not respond to UDS requests. If the client's logical address in the DoIP client is not right, ECU will deny access. To solve that, I can send a simple UDS request and, each time change the client's logical address, which ranges from 0x0000 to 0xFFFF, until I receive a positive response. Now I can create the final DoIP Client with all needed parameters set.

```
while send_address < max_address:
    doip_client = DoIPClient(ip, logical_address,
    ↪   client_logical_address=send_address)
    conn = DoIPClientUDSConnector(doip_client)
    with Client(conn,request_timeout=timeout) as client:
        response = client.change_session(
        DiagnosticSessionControl.Session.defaultSession)
    if response.positive:
        self.tester_logical_address = send_address
        break
    send_address += 1
```

6.5 Scan DID and DTC

Application can scan a range of DIDs and diagnose which ones return a positive response with some data. Application returns only IDs of these DID as there is no standard for decoding these data. In this exmaple the application creates a request for a DID and waits for a response from the ECU. Than, it checks if response was positive and adds it to a list.

```
my_hex_data = did_id.to_bytes(2, 'big')
req = Request(services.ReadDataByIdentifier, data=my_hex_data)
self.uds_connection.send(req.get_payload())
payload = self.uds_connection.wait_frame(timeout=1)
response = Response.from_payload(payload)
if response.code == Response.Code.PositiveResponse:
    supported_did_codes.append(did_id)
else:
    if textbox is not None:
        continue
return supported_did_codes
```

I can also get DTC codes, which stands for diagnostic trouble codes. When some problem occurs, it will be stored in the ECU and can be later retrieved by a tester. Unlike DID, DTC has some structure and I can retrieve how sever the error is, its ID and its status. Example of implementation.

```
status = Dtc.Status(True, True, True, True, True, True, True,
↪   True)
response = client.get_dtc_by_status_mask(status)
save = response.service_data.dtcs
text_insert = "DTC with id " + hex(save[i].id) + " status " +
↪   hex(
save[i].status.get_byte_as_int()) + " severity " +
↪   hex(save[i].severity.get_byte_as_int()) + "\n"
```

Chapter 7
Conclusion

The main aim of this thesis was to design and implement a tool for Automotive Ethernet that could automatically identify Automotive Ethernet network and services supported by an ECU, as older projects are not usable in modern cars.

In the beginning, I explained the necessary terms regarding Automotive Ethernet and diagnostic. After that, I conducted an analysis of current tools that are open-source and perform a similar task for the CAN bus. During this analysis, I discovered that one CAN tool has a module for DoIP. I tested it and discovered that two main services did not work with an ECU I had, but I looked at what services they try to scan and how. Another CAN tool that I looked at had a nice GUI, and I tried to design my tool to have a similarly simple and nice GUI.

In the next step, I described three methods to get IP address and other parameters of IPv6 network. Later I described UDS message structure, particularly response and request. I explained how to perform diagnostic of supported services and their limitations. Based on previous research, I presented tools that I considered and explained why I decided to use that tool over another. Based on performed analysis, I designed and implemented a new tool with GUI that performs automatic network identification and can scan two of the UDS services, DID and DTC.

7.1 Ideas for further improvements

I have several ideas that could be implemented in the future to further improve my tool's usefulness. Here are the most significant ones.

7.1.1 More services

Currently, only two services are supported by my tool. Reading of DID and DTC. It would be useful to add support for more services such as ECU reset, read Memory by address, Tester present, and others. Unfortunately, these services could not be implemented and tested because the tested ECU did not support these features or due to a lack of time on my side. This will not take much time, and I am already working on support for service discovery and adding an option to periodically send tester present message.

7.1.2 Backward compatibility

I designed my tool with IPv6 as its network layer. It wouldn't be hard to modify my tool to work with IPv4.

7.1.3 Interpret DID codes

DID codes contains various data like VIN code,hardware number and many others that depends on manufacturer. Decoding these data would be nice. I actually tried to decode some of the DID codes, as some of them are stored in ASCII, but I had to add how long the string is, and I did not manage to get this info automatically.

7.1.4 Implement VLAN into application

Currently application can detect VLAN on network, together with its id, but it cannot create a new interface with VLAN to communicate with an ECU. Currently outside script that requires sudo access can be run. It would be better to implement this into the tool, ask for password and run the script from application.

Bibliography

[1] CAN and alternatives — Navixy. GPS Tracking Platform – Navixy [online]. Copyright © 2005 [cit. 31.03.2022]. Available at: https://www.navixy.com/docs/academy/can-bus/can-and-alternatives/

[2] From Standard Ethernet to automotive Ethernet. (2019) [online] [cit. 31.03.2022]. Available at: https://www.keysight.com/us/en/assets/7018-06530/flyers/5992-3742.pdf

[3] (DoIP) Diagnostics Over Internet Protocol Explained | AutoPi. IoT platform for your car, built on the Raspberry Pi | AutoPi [online]. Copyright © Copyright [cit. 31.03.2022]. Available at: https://www.autopi.io/blog/diagnostics-over-internet-protocol-explained/

[4] DoIP Software Solution for Remote Vehicle Diagnostics | UDSonIP v/s UDSonCAN. Product Engineering Services | Magento | Hybris | Ecommerce Website Development [online]. Copyright © 2021 Embitel. All Rights Reserved [cit. 31.03.2022]. Available at: https://www.embitel.com/blog/embedded-blog/how-uds-on-ip-or-doip-is-enabling-remote-vehicle-diagnostics

[5] UDS Explained - A Simple Intro (Unified Diagnostic Services) [online]. [cit. 31.03.2022]. Available at: https://www.csselectronics.com/pages/uds-protocol-tutorial-unified-diagnostic-services

[6] Unified Diagnostic Services - Wikipedia. [online]. [cit. 31.03.2022] Available at: https://en.wikipedia.org/wiki/Unified_Diagnostic_Services

[7] IPv6 overview, IBM Docs. [online]. Copyright © Copyright IBM Corporation 1998, 2010 [cit. 31.03.2022]. Available at: https://www.ibm.com/docs/en/i/7.2?topic=6-ipv6-overview

[8] IPv4 vs IPv6: Comparing Their Security & More. WisdomPlexus - "Top Destination for Updated Techscoop" [online]. Copyright © Copyright [cit. 31.03.2022]. Available at: https://wisdomplexus.com/blogs/ipv4-vs-ipv6-security/

[9] Embitel *how uds on ip or doip is enabling remote vehicle diagnostics*[online] https://www.embitel.com/blog/embedded-blog/how-uds-on-ip-or-doip-is-enabling-remote-vehicle-diagnostics

[10] Caring Caribou [online] https://github.com/CaringCaribou/caringcaribou

[11] CANalyzat0r [online] https://github.com/schutzwerk/CANalyzat0r

[12] CANToolzl [online] https://github.com/CANToolz/CANToolz

[13] Softing Automotive | Expertise of diagnostics and testing in automotive electronics [online]. Copyright ©U [cit. 18.05.2022]. Available at: https://automotive.softing.com/fileadmin/sof-files/pdf/de/ae/poster/UDS_Faltposter_softing2016.pdf

[14] Frank. Medium. 2020. Create desktop applications with Python tkinter. [online] Available at: https://medium.com/@frank_43640/create-desktop-applications-with-python-tkinter-bb7a0e073f0c [cit. 16.05.2022]

[15] Custom Tkinter [online] https://github.com/TomSchimansky/CustomTkinter

[16] Svirca, Z., 2020. Everything you need to know about MVC architecture. [online] Available at: https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1 [cit. 31.03.2022]

[17] What is Nmap And Why You Should Use It? - The Hack Report. Top Software Testing Services Compant | The Hack Report - The Hack Report [online]. Copyright © [cit. 10.05.2022]. https://thehackreport.com/what-is-nmap-and-why-you-should-use-it/

Appendix A
List of appendixes

Part of this thesis is an attachment in zip format. This attachment has these folders.

- EthernetIdentificationTool - Source code of implemented tool
- overleaf - source code of this thesis