FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# ASSIGNMENT OF BACHELOR'S THESIS

| | |
|---|---|
| **Title:** | Automated tool for CAN bus message mapping |
| **Student:** | Duc Huy Do |
| **Supervisor:** | Bc. Martin Pozděna, MSc |
| **Study Programme:** | Informatics |
| **Study Branch:** | Computer Security and Information technology |
| **Department:** | Department of Computer Systems |
| **Validity:** | Until the end of summer semester 2020/21 |

## Instructions

The goal of the thesis is to research and develop a software framework to support CAN bus message mapping, similar to the functionality of nmap in TCP/IP network mapping. CAN bus message formats can be different among different car manufacturers and therefore the framework shall be implemented with scanning modules. One module for CAN bus message reconnaissance shall be developed to test the framework functionality.

1. Survey existing research on SW frameworks and tools for CAN bus message mapping.
2. Analyze potential SW framework design and propose an API designed for future plug-in modules.
3. Design, implement and test a software framework for CAN bus message monitoring and mapping. Discuss further framework design details including functional and non-functional requirements with the supervisor.
4. Design, implement, and test one CAN bus message reconnaissance module.

## References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 17, 2019

# Acknowledgements

First and foremost, I would like to thank my supervisor Martin Pozděna from Auxilium Cyber Security for his patient guidance and lots of insightful and sharp comments. I am also grateful to Thomas Sermpinis, who helped me find useful research material and other relevant resources. Finally, a huge thank you goes to my family and friends for their support and encouragement.

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for nonprofit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on June 2, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology


### Citation of this thesis

Do, Duc Huy. *Automated Tool for CAN Bus Message Mapping.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

# Abstract

In this bachelor's thesis, we first examine the topic of CAN bus message mapping and then create a tool to help with such a task. Control Are Network is a message protocol used in industrial domains, especially the automotive industry. With CAN bus we can create a cheap and robust communication network for devices and sensors in the vehicle. Although the CAN messages themselves are not encrypted by default, we are not able to understand what they mean or represent. The main reason behind that is the absence of a common standard for message identification and content formatting. Our goal was to create a tool that will make the arduous process of message mapping more effective and automated. The thesis begins with an introduction to the CAN protocol and the topic of message mapping. Then we walk through existing solutions and research dedicated to CAN bus analysis so that we can design and implement our tool. Result of the thesis is an automated framework for reconnaissance of the bus. What is more, the tool was designed and implemented in modular fashion allowing to implement additional functionality in form of modules. Lastly, it was succesfully tested using both simulator and a real car.

**Keywords** CAN bus, automotive, vehicles, nmap, reconnaissance, monitoring, security, message mappping

# Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací automatizovaného nástroje pro mapování zpráv sběrnice CAN, která se využívá v mnoha průmyslových odvětvích a především v automobilech pro komunikaci jednotlivých zařízení a senzorů ve vozidle. Přestože komunikace na sběrnici CAN je nezabezpečená, nelze přímo určit význam a data jednotlivých zpráv. V dnešní době neexistuje žádný obecný standard pro identifikaci a formátování dat ve zprávě a proto má každý výrobce aut nadefinované vlastní značení zpráv a strukturu dat, které v nich posílají. Jejím cílem bylo vytvořit nástroj, který by usnadnil a automatizoval proces dekódování těchto zpráv. Práce začíná teoretickým úvodem ke sběrnici CAN a problému mapování zpráv. Poté se zabývá existujícími nástroji a výzkumem, které se věnují analýze dat ze sběrnice CAN. Nabyté znalosti jsou využity pro návrh a posléze implementaci finálního řešení, které je výsledkem této práce. Tento nástroj umožňuje nejenom automatizovaný rozbor CAN zpráv a interakci se sběrnicí, ale i snadnou tvorbu vlastních modulů pro dosažení specifické funkcionality. Na závěr byl nástroj úspěšně otestován na simulátoru i skutečném automobilu.

**Klíčová slova** sběrnice CAN, vozidla, automobilový, nmap, odposlouchávání, bezpečnost, mapování zpráv

# Contents

# List of Figures

# Introduction

**Motivation** The Control Area Network protocol has various applications across different industries like automotive, manufacturing, construction and more. This thesis focuses primarily on the car industry, which may be considered as one of the most safety and security dependent business for customers. Nowadays, the main trends in the automobile industry do not move solely towards having faster and well-designed cars, but also getting a vehicle equipped with the high-end technology that often provides us with more safety, reliability and comfort. However, the usage of smart electronics within our car brought us the same challenges that we were used to dealing with only in ICT related domains. Modern cars hold many electronic modules, protocols or software. Hence it was only a matter of time when someone would start looking for the same issues that are known in a conventional computer system and try to exploit them.

**Problem statement** One of the most critical parts of the vehicle are ECUs – Electronic Control Units. They can be referred to as individual embedded systems and are responsible for handling electrical systems of the car such as engine, brakes, doors or windows and more. Although there are ways to attack ECUs directly, we are going to focus at the CAN protocol used for their communication. Interfering ECUs message transmission then allows us to alter information upon which an ECU will operate, thus invoke an unexpected behaviour of the whole car. In order to do so, we have to be able to read, understand and send CAN messages, which are unique for a particular car model. That itself is a quite challenging task because there are practical limitations of getting access to the bus in the first place. Secondly, we do not usually have prior knowledge regarding connected nodes and how they communicate. However, this work is dedicated mostly to the second part regardless of the physical access.

**Goals** To identify ECUs within the CAN bus network and to analyze messages sent between them, we will need a tool similar to network utility nmap with a capability to discover devices within the network and to analyze traffic. What is more, instead of a narrowly focused application, we are interested in creating a modular framework with an API, so that users can create their custom modules for different scenarios. Our goal is to develop the framework for monitoring and mapping and create a surveillance modules for security testing.

**Thesis outline** The thesis begins with a short introduction to CAN protocol and its architecture, including research of existing tools and solutions providing us needed capabilities. The following chapter presents a design and implementation of our framework with both monitoring and reconnaissance modules. Last but not least, the thesis continues with an experiment report about testing our tool on a simulator and a real car CAN bus. The final part is dedicated to a discussion of our results and future work directions. The appendix consists of source code and documentation.

Chapter 1

# Preliminaries

The starting point of the thesis consists of a summary of the CAN specification and presentation of related applications and tools. It covers the necessary basics and ideas for the framework design and development.

## 1.1 Control Area Network – CAN

Control Area Network is a message protocol used for communication between microcontrollers and devices within a so-called CAN bus. It was first introduced by Robert Bosch GmbH in 1986 and then in 1991 in version CAN 2.0 [1]. International Standard Organisation also published specification for CAN as a standard ISO 19898 [2]. CAN operates as a broadcast system, where data frames are not transmitted directly from sender to receiver, but distributed through to the network to every endpoint. Such implementation can be cheap, robust, efficient and flexible. In the context of the ISO/OSI communication model, CAN architecture takes place at the physical and data-link layer. Vehicles use CAN bus system to enable data exchange for control units or sensors installed in them.

### 1.1.1 Physical layer

Although CAN bus protocol does not specify a transmission medium to be used, only a bit timing requirements and synchronisation, we will discuss the most prevalent way of the CAN bus implementation at a physical level. In most cases, all nodes on the bus are interconnected through two signal line (CANH, CANL) bus, where each wire consists of a twisted pair terminated with 120 Ω characteristic impedance resistors to prevent signal reflections. CAN uses differential signalling providing high immunity to electrical interference in combination with the usage of twisted-pair cables. Both CANH and CANL are passively set on 2.5 V that is called recessive state. In a dominant

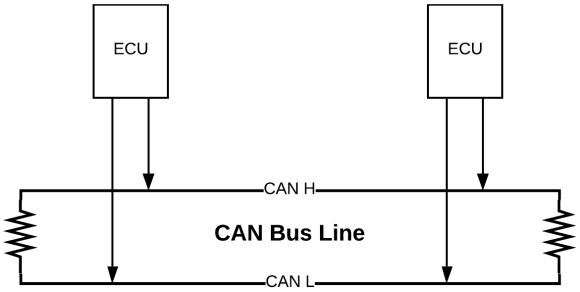


Figure 1.1: CAN bus circuit scheme

state, CANH is raised to 3.5 V, while CANL voltage is lowered to 1.5 V, creating a differential 2 V signal. This signal is then interpreted by driver input to 1-bit or 0-bit otherwise. However, the CAN bus might be realised in the form of a single wire or optical fibres as well [3].

### 1.1.2 Data-link layer

This layer can be divided into two sublayers – Logical link control (LLC) and Media access control (MAC). LLC layer provides Message filter, Overload notification and Recovery management functions. On top of it, MAC layer takes care of Message framing, Arbitration, Acknowledgment, Error detection and Signalling [1].

#### Bus access and arbitration

There are no explicit rules or access control for bus communication, but a method called bitwise arbitration. Inverted logic of driver input and output forms an original approach to the access arbitration, where the message IDs are used to decide its priority and who gets control of the bus. Signals for zeroes set the bus to the dominant state, which always overrides the recessive. Hence, the lower the device ID, the longer it keeps the CAN bus in the dominant state and wins over other messages. Such method is a very simple yet effective way to determine the order of messages sent to the bus. Feature like this favors the usage of CAN in real-time systems. This technique, in combination with an error handling, ensures data consistency, when a message is received by all or none nodes.[3].

#### Error handling

Several safety mechanisms prevent errors from occurring: error detection established by monitoring (comparing transmitted bit levels from the device and

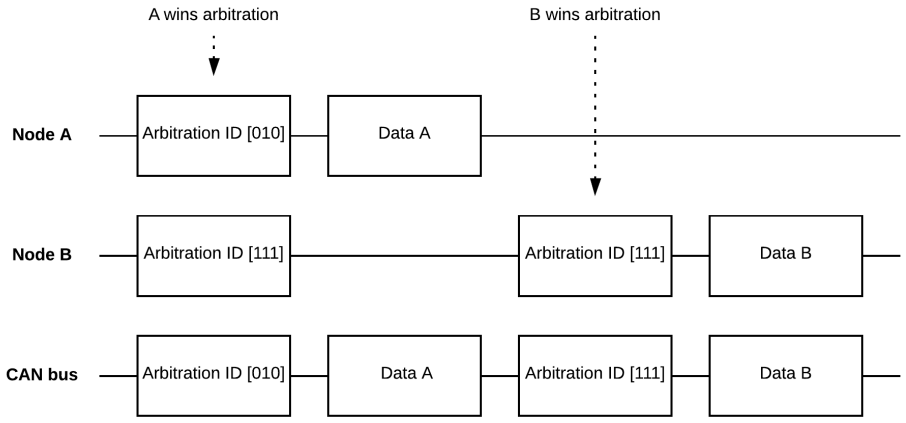


Figure 1.2: Message ID arbitration

on the bus) and checking data integrity with a control hash of the message. Error signalling marks faulty frames and requests its repeated transmission. Nodes causing permanent failures are switched off [1].

### CAN message

Every message, also referred to as a CAN frame, should be given in fixed format with a limited length which can vary depending on the size of data content. The message itself does not indicate any information about the sender or receiver. The identifier serves only as a descriptor of the data carried by the frame. Every ECU is individually programmed to receive and send a particular set of messages. Nowadays, two formats are described in ISO 11989 standard – Standard CAN and Extended CAN with the main difference being the length of the identifier. The message consists of several bit fields to store an identifier, data or control flags. One ECU might send and receive more messages with a different ID. CAN protocol defines four message types:

**Data frame** a frame containing node data for transmission

**Remote frame** a frame requesting the transmission of a specific identifier

**Error frame** a frame transmitted by any node detecting an error

**Overload frame** a frame to inject a delay between data or remote frame

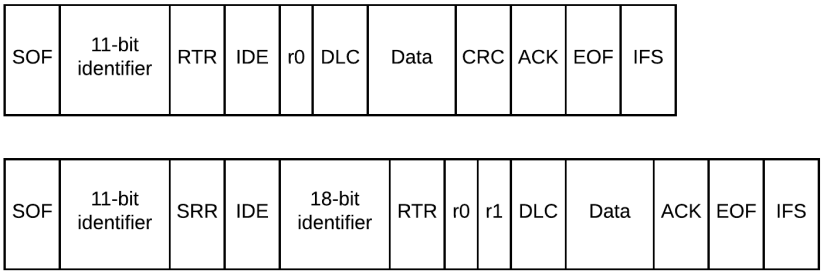


Figure 1.3: Standard CAN and Extended CAN data frame bit fields

**SOF** The Start Of Frame bit marks the beggining of a message and is used to synchronize transmissions on a bus

**Identifier** The identifier sets the priority of the message, lower binary value means higher priority

**RTR, SRE** The single remote transmission request bit is set for remotes frames

**IDE** The identifier extension bit signals if the frame has a standard CAN identifier with no extension

**r1, r0** Reserved bit

**DLC** Data length code represents number of transmitted bytes of data

**Data** The Data Field contains message payload of size up to 64 bits

**CRC** Cyclic redundancy check contains the checksum (number of bits transmitted) of the preceding application data for error detection

**ACK** Acknowledgment bit is sent by the receiver when accepting a valid message

**EOF** The end-of-frame marks the end of a CAN frame

**IFS** The inter frame space is a bit field to seperate data frames and error frames from preceeding frames

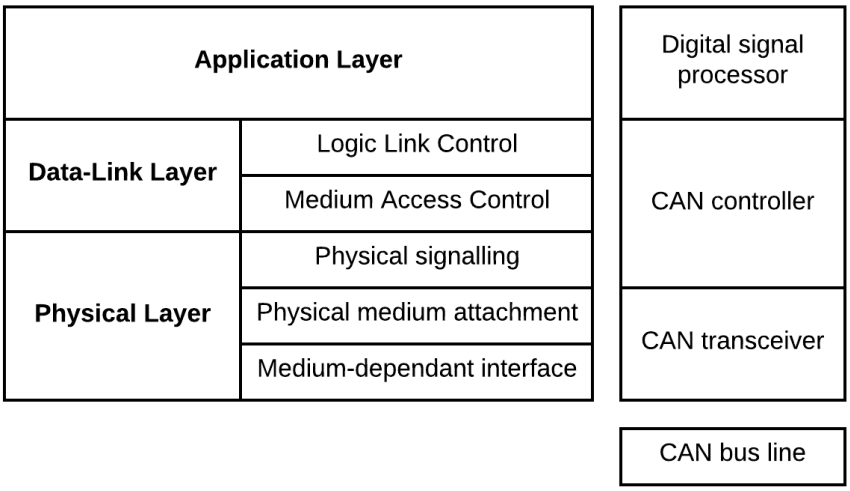


Figure 1.4: The Layered ISO 11898 Standard Architecture

### 1.1.3 Application layer

On top of lower levels of CAN, the application layer takes place in the form of various implementations depending on the utilization of the bus. Some examples might be flow control, message segmentation, security, or routing. A few noteworthy protocols are: ISO-TP - enables sending messages larger than frame size [4], CANopen – extends CAN with more complex communication models and protocol [5], UDS – standardized diagnostics services [6]. Besides, manufacturers themselves sometimes use their proprietary protocols. Generally speaking, certain functionality CAN protocol lacks is achieved within this layer.

#### CAN types

Nowadays, we can distinguish two versions of CAN bus: High-speed CAN (ISO 11898-2) and Low-speed fault-tolerant CAN (ISO 11898-3). The first one is capable of the transmission rate of 1 Mbps, while the latter has its rate limited to 125 kbps. In contrast, the low-speed variant operates with higher voltage swings and its CAN bus is terminated at every node, so that the communication can be established even if there is a wiring issue in the bus. Car manufacturers usually use a High-speed CAN for systems such as engine control unit or brakes, where higher transfer rate is required for im-

mediate respondency. Low-speed CAN usually connects comfort units like air-conditioner, windows or seats, etc. [7].

### 1.1.4 Security

By default, CAN protocol does not include any particular security measures. Therefore, message sniffing and spoofing or a denial of service may be accomplished if access to the bus is already granted. However, at least a certain level of security has been attained by "the security through obscurity" principle, because every vendor develops their own ECUs and its special firmware. Without knowledge of how a device is identified and what data it uses, it is very challenging and time-consuming to determine what is happening on the CAN bus. Nonetheless, with a thorough analysis and right tools, the bus is as vulnerable as any other computer network. For instance, well-known report by Miller and Valasek demonstrates remote exploitation of a Jeep Cherokee 2014 [8]. They were able to access the car remotely and gain control of an ECU. Afterwards, they just had to determine which CAN messages to inject, so as to affect physical systems.

In order to enhance the security of the CAN bus, some additional protective methods outside the ISO 19898 standards are adopted in the industry. Physical level solutions are sub-networks or gateways, contributing to necessary redundancy and isolation of the logical sets of connected nodes. Another way to supervise incoming messages is to employ an intrusion detection module, which should detect malicious frames and ECUs planted by an attacker. There are ongoing research efforts to design algorithms discovering CAN bus attacks [9]. Furthermore, encryption and authentication could be implemented as application layer protocols. Such an approach, however, has its limitations due to bus performance or requirements for additional hardware modifications [10].

### 1.1.5 Message analysis

#### Capturing CAN bus messages

For the task of reading CAN messages and processing them in a software application, we have to examine ways of bringing the data from the bus to a computer device and vice versa. The bus communication on the physical layer is handled by a CAN transceiver, which converts binary data to electrical signaling. On the data-link layer a CAN controller is responsible for serialization and deserialization of a data stream and error handling. A processor unit fetches the data from the controller and makes some use of them depending on the application. Usually, a microcontroller includes a CAN controller, and it is programmed to either process the data itself or forwards it to another interface like USB so that a computer with more sophisticated software can access it. In this work, we will design a tool operating only at the application

layer using any intermediary device for establishing a connection to the CAN bus. Depending on the interface, a corresponding device driver is required to deliver CAN messages to the program.

#### Message mapping

When it is possible to capture CAN messages from a CAN bus, the next step is to analyse them. Not only researchers but also car enthusiasts often attempt to reverse engineer CAN traffic from their car. While some are more successful than others, no universal method has been invented yet to reverse engineer a CAN message completely. One common, but quite elaborate technique is to record data from CAN bus and the car activity at the same time. Then it is possible to find some patterns in the log to match them with a certain function of the car. The need for effective automated methods, which will reduce manual efforts in this task is evident.

Huybrechts et al. [11] suggested using a machine learning method to facilitate the reverse engineering process. They were able to identify some interesting signals, but the conclusion was not very convincing in favor of machine learning. Others have also focused on analysing the structure and the behaviour of signals in a data field. Markovitz and Wool developed a classifier for signal fields in CAN message to recognise field types like sensors, counters, multi-values or constants [12]. In similar work, Marchetti and Stabili invented an algorithm using bit-flip rate of the message data to determine signal boundaries [13]. One common challenge mentioned in these papers was lack of real testing data from the CAN bus, both raw messages and their definitions needed to evaluate correctness.

## 1.2 Existing solutions

The demand to analyze CAN bus communication is covered by various products ranging from robust commercial industry-oriented software to community driven, open-source or research solutions. This section offers a closer look at a few of them.

### 1.2.1 Proprietary

#### CANoe, Vector Informatik GmbH

CANoe [14], is full–fledged software tool used for development, automated testing and analysis of automotive electronics and entire ECU networks such as CAN bus. However, it supports only hardware interfaces available from Vector, so the user is bounded to use complete product stack by Vector to work with the CAN bus. Their products are mainly intended for companies, who are actually developing CAN networks and ECUs.

#### CAN BUS Analyzer, Microchip Technology Inc.

CAN BUS Analyzer Tool [15] is another example of comprehensive CAN bus analysis product consisting of both hardware and software components. Its primary function is monitoring CAN bus traffic via a hardware module and displaying it with a GUI application on the PC. In addition, the tool can transmit messages back to the nodes on the bus. It aims to be a lightweight, low-price alternative to more expensive tools like CANoe.

### 1.2.2 Non-proprietary

#### SocketCAN can-utils

Linux comes with a suite called can-utils [16], userspace utilities based on SocketCAN [17] – an implementation of CAN protocols for Linux contributed by Volkswagen Research. As they could be run only in the console, their main feature is simplicity and usability. Each tool has a specific function.

**candump** display or log received messages

**canplayer** replay logfiles

**cansend** send CAN message

**cangen** generate random messages

**cansniffer** show differences in incoming CAN messages

#### Wireshark

Wireshark is well-known software for capturing and filtering network traffic. In Linux, Wireshark can be used without any external modules, because it can treat any CAN device as a network interface thanks to SocketCAN. On the other hand, Windows version requires a plugin for each vendor-specific CAN interface.

#### Kayak

Kayak [18] is a GUI Java application with similar functionality as can-utils. Additionally, it uses Kayak CAN definition format to store CAN message descriptions. Rather than having a dedicated tool for one designated operation, it combines all the features in a single graphical environment. Its core library with complete CAN bus abstraction model could be utilized in other projects. The application core is also built upon SocketCAN layer.

### Python-can

This library provides a useful CAN abstraction for Python programmers. Its modular backend was designed to support numerous CAN interfaces while having a common API for sending and receiving CAN frames. A part of the library are console scripts for logging, replaying and viewing messages [19].

### Kvaser CANlib SDK

Kvaser [20] is primarily a hardware manufacturer and sells CAN interfaces and loggers. Not only its drivers are supported by Socket CAN, but Kvaser also offers the CANlib SDK compatible with its hardware platforms. The SDK comes with an open universal API for all Kvaser device drivers, but it can create a virtual CAN device for testing purposes as well.

### asammfd

Assamfd stands for ASAM (Association for Standardisation of Automation and Measuring Systems) and MDF (Measurement Data Format). Some loggers like CANedge [21] records and stores CAN bus messages in open standard MDF format. Such device delivers a different approach to examining CAN bus messages by collecting the data in internal or external storage for further evaluation instead of the way of real-time interaction. Assamfd tool can read and extract CAN log data in MDF format for later processing involving conversion, graphical plotting, editing and exporting [22].

Chapter 2

# Analysis and design

This part is dedicated to proposing a suitable architecture for our framework. Previous sections demonstrate several different approaches to CAN bus analysis to help us find the right ideas and meaningful baseline to start designing the tool. Next part describes application programming interface design and properties. Putting together all these segments should give us a blueprint for a modular framework with automated mapping features we intend to build.

## 2.1 Framework design

To begin with, we will define the mapping process from a conceptual level to identify the main requirements, both functional and non-functional and outline the scope of the project. Then, we will suggest an architecture to meet these requirements.

### 2.1.1 Mapping process analysis

As most generally available software nowadays can only read and send raw CAN messages, our effort is to offer more complex and detailed interaction with the CAN bus. By default, CAN protocol does not include any security measures, only relying on obfuscation. There is no explicit information indicating the number of ECUs and which messages they send and receive. Having just a CAN bus traffic, we are also unable to detect what is the meaning and context of a message without a manufacturer's specifications. Our goal is to create a framework to automate and simplify this process for CAN bus analysis. Users should be able to write custom modules for message context mapping so that for a specific message ID, we will be able to understand its meaning and content type. Such a task requires an API to access our processed data in a convenient and versatile way.

For simplicity, we will break down our project into smaller segments. Then we can find or create a solution for each of them separately.

1. Read CAN bus messages
    a) Live CAN bus monitoring
    b) Load CAN bus log
2. Assort messages and detect connected ECUS
3. Decode or encode CAN messages
4. Provide interface to manipulate with the processed data

This partition also hints at the structure of our architecture, where one layer deals with one of the tasks. For each problem, we will have a discussion to figure out the best way to solve it. If there is an already existing component, which can do the work for us, we will try to incorporate it into our framework. Otherwise, we must design and build it ourselves.

### Read CAN bus messages

There are two ways of fetching data from the CAN bus. Firstly, the bus must be connected to the computer for us to receive messages and handle them on the fly with our application. The second approach is post-processing when we only record some traffic and save it for later evaluation. When deciding between one or the other, it usually comes down to the hardware limitations and accessibility to the CAN network.

In the first chapter, we have demonstrated various platforms for creating a communication channel with CAN bus. Many hardware drivers are natively included in Linux kernel, therefore supported by SocketCAN. Because it gives us needed features to read messages conveniently, it makes perfect sense to use it in our implementation. Moreover, we can avoid programming and configuration sockets in low-level C/C++ environment with python-can library, which offers a complete CAN bus abstraction layer in Python above SocketCAN implementation.

Reading recorded messages from a file pose technical challenges regarding lack of standard data format. Every tool for logging CAN bus traffic stores messages differently. Implementing serialization for every possible format is neither needed nor feasible in the scope of the thesis. Therefore we will select the most important ones. Industrial CAN loggers mainly store messages in MDF, binary file format aiming for efficiency and high-performance. Another common way to read and log messages is SocketCAN utility candump. Being a part of Linux official package repositories has made candump generally adopted as a tool for quick and effortless experimenting with the bus. For simplicity, we can use only the can-utils log format, because it is possible to export MDF files to it with a converter.

### Assort messages and detect connected ECUs

Mapping requires a list of unique messages and the data they contain. We can create such a catalogue by registering message IDs to the database and assign them values received in the data field. This data could be later accessed through API calls for further operations. Another useful information we can obtain is a list of connected ECUs and related message IDs which can be achieved with CANvas mapping module [23]. CANvas by Kulandaivel et al. can identify transmitting ECUs using a pairwise clock offset tracking algorithm and receiving ECUs by forced ECU isolation. To understand the behaviour of a signal in the message and support the CANvas module, it is also necessary to keep the whole message as it was received from the CAN bus, including its timestamp, so that we can observe how its values changed in time.

### Decode or encode CAN messages

Mere knowledge of unique messages and ECUs do not tell us anything about their meaning and what they represent. If we are able to determine which physical action triggered a particular message and discover some patterns, we can create those associations by ourselves to some degree of accuracy. Even though message mapping is embedded in ECUs, thus not available publicly, some message definitions might be put together as a community effort or leaked from the manufacturer. Two most popular file formats are:

- DBC by Vector Informatik GmbH, proprietary [24]
- KCD – Kayak CAN definition, XML based, open-source [25]

They describe how information could be decoded in a CAN frame. Structure of a database consists most importantly of nodes and messages, including their signals. For every message, an ID, message name, message size and transmitter name is defined. Each message also has a list of signals placed in it. A signal description consists of starting position, size, byte order, value type, linear function, value range, unit and a receiver.

For illustration, we will show you an example of a DBC definition. As can be seen in Figure 2.1, we have two nodes: Engine and Gateway. Message EngineData with ID 100 and size of 8 bytes is sent from the Engine node. One of its signals – EngTemp has an offset of 16 bits and length of 7 bits. Value byte order is big-endian, and it is unsigned. To make the linear transformation from a physical (real) value to a raw value, one must subtract $-50$ from physical value and divide the result by 2. The expected range of values goes from $-50$ to 150 degrees Celsius. Finally, the receiver of the message is the Gateway node.

At this point, it should be clear how difficult it is to map a raw CAN message to its actual interpretation without any syntactical nor semantical information given apriori.

```
BU_: Engine Gateway
BO_ 100 EngineData: 8 Engine
        SG_ PetrolLevel : 24|8@1+ (1,0) [0|255] "l" Gateway
        SG_ EngPower : 48|16@1+ (0.01,0) [0|150] "kW" Gateway
        SG_ EngForce : 32|16@1+ (1,0) [0|0] "N" Gateway
        SG_ IdleRunning : 23|1@1+ (1,0) [0|0] "" Gateway
        SG_ EngTemp : 16|7@1+ (2,-50) [-50|150] "degC" Gateway
        SG_ EngSpeed : 0|16@1+ (1,0) [0|8000] "rpm" Gateway
```

Figure 2.1: DBC definition example [24]

A conversion tool called CANBabel serves to translate data from DBC to KCD format. Moreover, both formats could be parsed with CAN BUS Tools utility and used for decoding and encoding CAN messages [26].

#### Provide interface to manipulate with the processed data

Previous points have shown us ways to receive or read CAN frames and sort them out to make mapping efforts more effective. Now we have to design an interface for a user to manage this framework and perform CRUD operations with our internal structures. It is evident that our framework combines many independent components, and we must create a core module to bind them all together and an API to access and set up all this functionality in a unified and practical manner.

### 2.1.2 Requirements

#### Functional

- *Automated message mapping*: Message mapping nowadays requires lots of manual work in order to figure out what does a message represent. We aim to delegate as much workload as possible from a user to our framework.

- *Live/Offline execution*: Getting direct physical access to a CAN bus with a computer may not often be possible, especially for a meaningful amount of time to collect enough data. That is why mapping for both live data stream and stored data logs must be achievable.

- *Modularity*: Users should be able to write own modules, which would be automatically loaded to the framework so that they can run their custom functions using our framework.

**Non-Functional**

- *CLI*: The framework by itself does not need any graphical user interface. Visual elements could be implemented in the form of modules. Command–line interface also suits better for our command based approach. Furthermore, it also takes up fewer system resources and does not require any additional libraries.

- *Open-source distribution*: We want our tool to be available for everyone at no costs. Inherently, the way of open-source will encourage crowdsourcing and cooperation, which plays a vital role in the CAN messages definitions disclosure.

- *Hardware independent*: Our tool shall not be dependent on any specific device for CAN bus logging or its drivers. Users should be free to choose any hardware they seem fitting their needs and resources.

### 2.1.3 Architecture

For our architecture, the logical partition of the framework reflects the layered model discussed in the previous chapter. Our model consists of three tiers – physical, backend and frontend. CAN bus physical and data-link layer, which in reality involves the CAN bus, CAN transceiver, CAN controller and

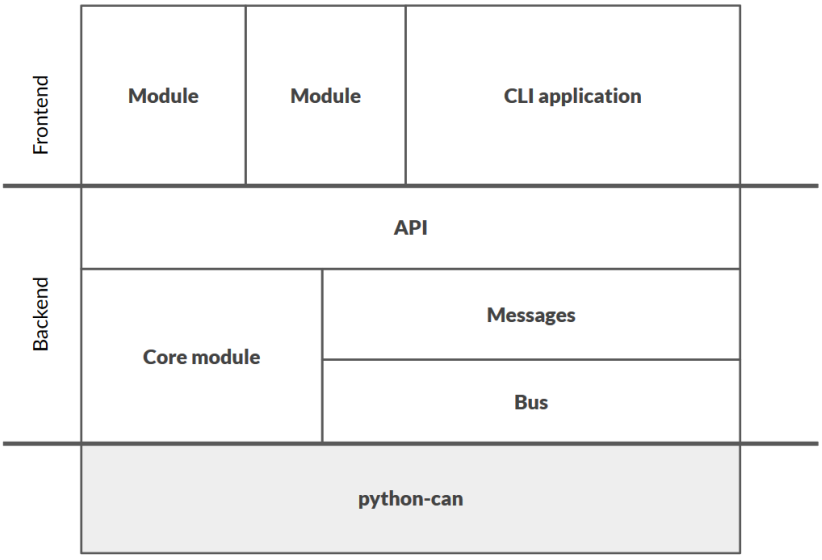


Figure 2.2: Framework architecture

a hardware interface to a computer will be implemented by python-can. Backend part deals with an integration of external components and translation of the interface and raw data to the internal representation exposed via API. Frontend consists of user modules using the API and a console application.

## 2.2 API design

API should provide a convenient way to interact with our framework and access useful information for mapping. We also consider having an API layer essential for modularity. Users do not have to be familiar with the framework internals, we want them to effectively create interesting and useful modules, which will add the main mapping functionality to our framework. Based on the previous process analysis, we defined four tasks the API must support.

- *Read messages*: return messages for a requested message ID. Having a complete history of a message is necessary to study changes in the data field and its structure. That could help users recognize particular signals and its values.
- *List messages*: return all unique message, including statistics and values. The output might resemble a DBC database without a context. The method should serve as an overall picture of a CAN bus traffic providing information such as all detected unique message IDs, message count, message interval or received values for each byte in the data field.
- *Send messages*: send message to the CAN bus. Injecting messages to the bus can also be very helpful in mapping efforts. This approach is only useful when being connected to the real CAN bus so that we can see if our spoofing triggered anything in the car. Another use-case is security testing when we would flood a bus with a message, which can lead to the Denial of Service in an ECU or even an interruption of the communication.
- *Decode message*: decode a CAN message using imported definitions. Our framework will support importing DBC or KCD databases. Thus we should provide an option to decode the message if there is a definition for it.

Chapter 3

# Implementation

Next comes the practical part, where we describe the development of the framework we named as CANdy. In the beginning, we present used technologies and the reasoning behind our decisions. Afterwards, we will go through the main segments and challenges we faced when creating them. Last section is dedicated to modules, which form the main reconnaissance functionality of the framework.

## 3.1 Technology

### 3.1.1 GNU/Linux platform

We decided to write the tool in the first place for GNU/Linux operating systems, because of their native support of SocketCAN.

### 3.1.2 Socket CAN

With SocketCAN, CAN device drivers, most often realized as character devices, are turned to network interfaces and could be accessed through socket interface as an ordinary network device. Many third-party drivers are already included in the library. Using the socket abstraction has several useful implications:

1. There are no limitations for number of processes using the device simultaneously in comparison with character device or serial interface.
2. Programmers can develop applications regardless of a CAN controller drivers as there is no abstraction layer with unified API for them.
3. Using Berkeley socket API should be easier and more familiar than interacting directly with the drivers unless they provide some high-level SDK.

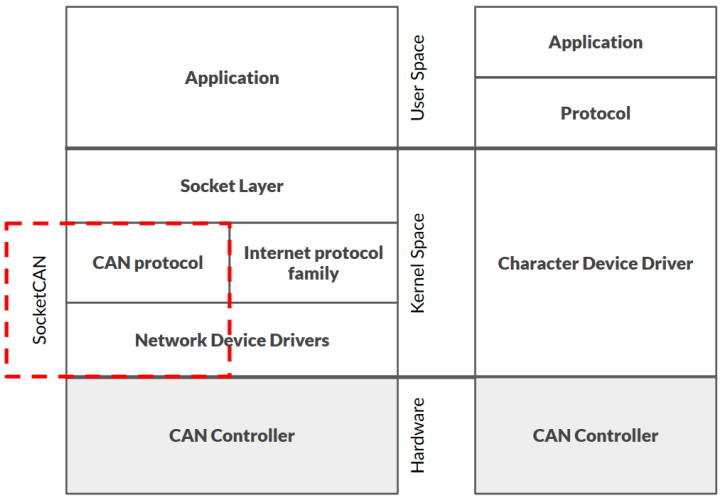


Figure 3.1: CAN communication layer - with SocketCAN (left) or conventional (right)

### 3.1.3 Python

Python is an interpreted, high-level programming language often used for data processing and analysis, where CAN bus message mapping certainly belongs. Being embedded in most Linux distributions makes it even more favorable for our project. For its popularity and relative simplicity for beginners, it is also the right choice for writing user modules. Another advantage is that many microcontrollers such as Arduino devices can run Python code too, which would give our tool more versatility. Also, there are some useful Python libraries that we can use in the framework.

#### python-can

This library provides Python implementation of CAN bus abstraction over SocketCAN interface and other device-specific interfaces. Though the library does not have any mapping functionality, we can easily configure and execute read/write operations on the CAN bus. What is more, python-can fulfils the requirement for hardware independency, most importantly, its modular backend, which should make our tool applicable for most hardware interfaces.

#### cantools

We can easily parse both DBC and KCD files using cantools. When imported to some application, the library can decode captured CAN message if a definition for it exists in the database.

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;

strcpy(ifr.ifr_name, "can0" );
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
bind(s, (struct sockaddr *)&addr, sizeof(addr));

struct can_frame frame;

nbytes = read(s, &frame, sizeof(struct can_frame));
nbytes = write(s, &frame, sizeof(struct can_frame));
```

Figure 3.2: SocketCAN - read a CAN frame [17]

### CANvas

CANvas, automotive network mapper aims to improve security testing methods regarding CAN bus networks. The tool is capable of identifying connected ECUs and matching them with sets of outgoing and incoming message IDs, despite having no previous information about the network. Researchers designed the tool to be fast, cheap and as non-intrusive as possible. At the moment, only core modules for source and destination mapping are published. We deemed this program worth integrating into our framework, because ECUs are the building blocks of the CAN network and as such we should be aware of them as much as possible. However, due to the technical limitation, we can apply only source ECUs identification that is conducted algorithmically whereas detection of the destination ECUs requires a hardware module.

### venv

The virtual environment allows us to use isolated site directory with its Python binaries, pip package installer and external packages. With venv, there is no need to install requirements for our tool on the whole system, and this independency from system installation sustains compatibility as well.

## 3.2 Base modules

### 3.2.1 Core

The central part of our framework is the Core module. It is responsible for the orchestration of the framework, managing modules and initializing internal objects.

#### Initalization

Given the arguments from the user, the core module initializes the framework with a CAN bus interface or without it. The latter, what we call an offline mode, must load a CAN dump file and serves solely for the purpose of post–processing without any interaction with a CAN bus. Launching the framework in a live monitoring mode will setup Bus instance on a user defined interface and capture messages sent to it. The Message database instance is also created for storing information about received or loaded messages.

#### Modules management

Next responsibility of the core is to deal with loading and running user modules. We use importlib package to find modules and invoke them when requested. When the module is launched, an API object is linked to it which exposes API methods for a module to use. Every module must follow a defined structure to be plugged into our framework.

#### Listener daemon

Like any network communication, listening to the CAN bus is an asynchronous process as we ought to wait for data from an I/O device. That is why we decided to run the message listener function in a separate thread next to the CLI application as a daemon. While the listener is processing incoming messages in one thread, the main application and the modules run independently in the other. This implementation decision logically leads to achieving a level of concurrency and a cleaner structure of the code. For the thread-based paralellism we used Python *threading* library.

### 3.2.2 Bus

Bus class represents the CAN bus and extends Bus class from python-can. Our abstraction takes care of configuration of the python-can Bus instance and adds a custom listener for recording and storing CAN bus traffic. It also provides an interface for interaction with the rest of the framework.

```
self.can_bus = can.ThreadSafeBus(
        bustype="socketcan",
        channel="can0",
        bitrate=500000,
        receive_own_messages=True,
)
```

Figure 3.3: python-can bus initialization in the Core module

#### Read CAN messages

When the python-can Bus instance is all set and ready, we can start receiving messages. Every message is kept in the Bus history object to preserve the history for further analysis. Every message is also saved into the Message database.

#### Send CAN messages

We can create a CAN message and transmit it to the CAN interface using python-can Bus API. There are also options to send a message periodically within the specified amount of time.

#### Filter

Bus manages a set of filters which are applied to the CAN bus. One filter rule consists of an ID and mask. Message ID matches the filter when following condition is met:

```
received_id & filter_mask == filter_id & filter_mask
```

In other words, if the mask bit is set to zero, the corresponding ID bit will automatically be accepted, regardless of the value of the filter ID bit. When the mask bit is set to one, received ID bit is compared with the filter bit.

### 3.2.3 Messages

Messages class implements messages and definitions database with functions to analyze them.

#### CAN message

For every unique message ID, we keep its count number, all the values appeared in each byte of the message, a label and an interval estimation. Although we may not know the position and length of the signals, saving unique

```
# Raw CAN message
2C1#08FF4D0000BA00D9
1C4#03681F0834003ED1

# Decoded CAN message
0x2C1     {'GAS_RELEASED': 0, 'GAS_PEDAL': 0.665}
0x1C4     {'RPM': 1578.90625}
```

Figure 3.4: Message decoding using cantools and DBC definitions

values of every byte of the data field could give us some clues about the message. What is more, we are keeping a change flag for every message, which tells us whether the message has a new value or not.

#### Definitions

Using cantools library, we can import definition databases and decode messages we have received.

## 3.3 API

API methods were implemented to offer functionality discussed in the design section. Users can call them to manipulate with the Bus and Message objects in our framework. Parameters marked with * are optional, message IDs, message data and filters use hexadecimal values.

- `read()` return one message from the message queue (FIFO)
- `get_messages(msg_id*)` return a single message by its ID or a list of all unique messages
- `get_message_log(msg_id, n*)` return all or n last received messages with a certain ID
- `send_message(msg_id, msg_data)` send a CAN message
- `send_periodic_time(msg_id, msg_data, period, limit*)` periodically send CAN messages for a specified time limit
- `send_periodic_count(msg_id, msg_data, period, number)` - periodically send a specified number of CAN messages
- `decode_message(msg)` decode message using imported DBC/KCD definitions

- `label_message(msg_id, label)` set a label for a message ID
- `set_filter_rule(msg_id, mask)` set a filter for CAN messages and return existing filter rules
- `reset_filter()` reset a filter for CAN messages
- `find_nodes()` detect nodes using CANvas module
- `get_nodes()` returns detected nodes

## 3.4 CLI application

We created the command-line application with the help of python *cmd* framework, which takes care of prompt mechanism and formatting, generating help section or command history and adding autocomplete feature.

According to given parameters, the application sets up internal structures and then looks for user plugins in a specified folder. Then a command line prompt is issued, and the user can start entering commands or modules in it for execution. A preview of the application can be seen in Figure 3.5.

```
CANdy - Automated tool for CAN bus message mapping
Version: 1.0
Starting monitoring session on vcan0
Loaded modules: dos liveplot newsig plot
Use 'help' for commands or 'mod' for modules
>>
>> help

Documented commands (type help <topic>):
========================================
filter  get  help  import  label  mod  monitor  nodes  quit  send

>>
```

Figure 3.5: CLI application preview

### 3.4.1 Inbuilt modules

We wanted our framework to provide a basic functionality by default, which is why we incorporated them into our CLI application. We consider these modules as the most elementary tasks users should be able to do.

- `Get` This module shows either all unique messages, their count and available definitions or detailed information about one message.

- `Filter` Users can setup filtering for incoming messages and view used filters.
- `Monitor` Like a candump from can-utils, monitor prints incoming messages to the screen, but is also able to translate using imported definitions.
- `Nodes` Integrated CANvas module is able to compute existing source nodes on the CAN bus.

### 3.4.2 Custom modules

One of the main advantages of our framework is the possibility to write user modules and use them in combination with others. Modules are loaded at the start of our CLI application, and they can be invoked as a command in our CLI application. We will present some modules we created for the framework to support message mapping and reconnaissance.

- `Plot` One of the best ways to understand how some data changed in time is visualisation. The time series graph may give us some interesting observations about the message signals. As suggested in [12] a signal can hold one constant value or multiple. It can also serve as a counter or carry a physical value from a sensor. We can distinguish these types from the shape of a plotline in the graph as can be seen in Figure 3.6. The module is based on the Python library *Matplotlib* [27].
- `Liveplot` When analysing live CAN bus, one might be interested in plotting a graph of a signal to see how it changes when performing a physical action like opening a door, pressing a gas pedal or brakes, increasing a car speed etc. User can choose a message ID and a signal byte they are curious about, and the module will create a plot that updates itself when a new message is received.
- `Newsig` In order to find message ID which could be related to using turn signals, we created this module to indicate messages that were received with a new signal value. This can help us see which messages had changed when something happened in the car so that we can link that event with a message ID.
- `DoS` Arbitration based on message ID and the broadcast nature of CAN protocol makes the bus susceptible to DoS attacks. This module performs such attack by periodically sending a specified amount of messages to the bus. One security testing scenario is using a low message ID, which will prevent ECUs from receiving real messages. Secondly, we can try to target a single ECU and force it to the error state or even take control of it. There are more DoS variations [28], but for now, the module offers only basic functionality and could be improved.

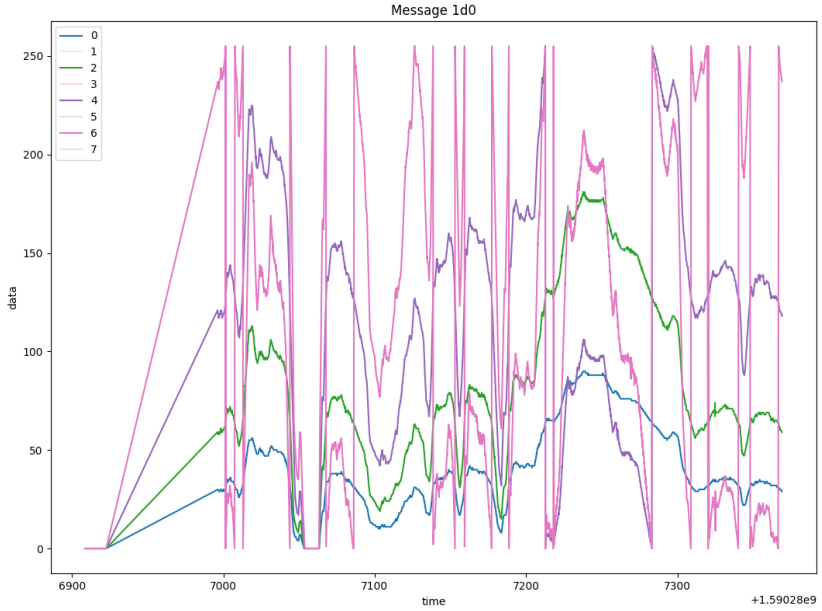


Figure 3.6: Plot module

Even though we implemented only a few module ideas for the scope of the thesis, we want to share some suggestions for the future. Firstly, modules using mentioned algorithms to classify and detect message signals [12, 13] would make a beneficial addition for message mapping. As of security testing, we can imagine having a fuzzing module to test the robustness of the bus and connected ECUs against excessive values. Moreover, with the knowledge of message definitions, we can create a dashboard simulating a car control panel to show a speedometer, RPM gauge, door status indicator and more. If we can identify a message with GPS data, we might be able to render a map showing a car route. All of this should be possible with our framework and the API.

Chapter 4

# Testing

For the evaluation of our completed tool, we needed to acquire some actual CAN bus data in the first place. We searched for some CAN dump files posted online, but at the end, we made some ourselves and conducted the testing with real cars and a CAN bus simulator. Firstly, we will describe how was the testing prepared and carried out. Next, the results and our experiences will be assessed at the end of the chapter.

## 4.1 ICSim

Instrument Cluster Simulator is a training utility for CAN bus hacking by OpenGarages project [29]. The tool simulates basic actions in the car like pressing the gas pedal, opening and locking the door or using turn signals. Such actions trigger sending CAN messages for the user to reverse engineer them. ICSim can be used with either keyboard or a game controller to control simulated car systems. Even though the tool generates only synthetic data, which may not be sufficient for more advanced analysis as reported by Schappin [9], this tool offers an interactive way to create interesting CAN bus traffic. Moreover, using a virtual CAN bus system like ICSim comes in handy during the development phase, because we can manipulate with the data source. In contrast, establishing a physical connection to a real CAN bus every time we need to is impractical for obvious reasons.

During our testing phase, we followed some advice from the renowned book The Car Hacker's Handbook: A Guide for the Penetration Tester [30], which demonstrates various techniques for hacking cars systems. Chapter 5 – Reverse engineer the CAN bus describes a few ways how to identify a message ID for a specific function of the car. Regardless of the used tool, these are the general steps for message recognition:

1. Find out which message ID appears when a particular event occurs.
2. Analyze its payload to determine, how is the event represented and the format of the information.
3. Describe discovered signals and test the results.

At the start, we were interested in mapping messages for every action possible with ICSim. Secondly, we wanted to obtain some additional information about the bus itself. For the first step, we used a `newsig` module, which prints out message IDs that were received with a new value. After launching CANdy, we waited for a few minutes to register as many message IDs and its values as possible, so they would not be marked as changed. Thus, the newsig output should be empty when no special action is done. On the other hand, triggering something new would be caught by the module and shown to us. For instance, we started off with detecting door lock/unlock the message. When we began monitoring the bus, the only value in the message, yet unknown to us, must have indicated that all doors are locked. That would not have changed until we opened them manually. So before doing that, we also run the newsig module, which revealed that the only message changed when we opened the door was 0x19b. The ID became our candidate carrying information about the car doors. After acquiring the message ID, there are two ways to proceed next. The first is to set a filter for this ID and run a monitor mode to see if the message is sent when the door opens or closes. This might also hint us the position of the data, see 4.1. The second way is to run a plot module for the ID and see if the changes in the graph correspond to our activity. The plot module will also indicate which byte holds the information we are looking for, see 4.2.

From both figures, we can see that the state of all doors is encoded with four bits starting at 20th bit of the data field. In our experiment, we observed this behaviour in the third byte of the message:

- **0f** (1111) – all doors locked
- **0e** (1110) – left front door unlocked
- **0b** (1011) – left back door unlocked
- **0d** (1101) – right front door unlocked
- **07** (0111) – right back door unlocked

Now we could easily describe the message and its signals regarding the doors, shown in 4.3. We used a DBC format for a more structured look. There is no linear transformation, therefore factor is 1 and offset 0. Also, we do not know who is the receiver of the signals, hence the XXX denoting an unknown node.

| Timestamp | ID | DLC | Data | Channel |
|---|---|---|---|---|
| 1590774572.068368 | 019b | 6 | 00 00 0e 00 00 00 | vcan0 |
| 1590774573.665996 | 019b | 6 | 00 00 0a 00 00 00 | vcan0 |
| 1590774598.761461 | 019b | 6 | 00 00 0e 00 00 00 | vcan0 |
| 1590774659.227013 | 019b | 6 | 00 00 0f 00 00 00 | vcan0 |
| 1590774660.521353 | 019b | 6 | 00 00 0b 00 00 00 | vcan0 |
| 1590774712.555436 | 019b | 6 | 00 00 0f 00 00 00 | vcan0 |
| 1590774728.632295 | 019b | 6 | 00 00 07 00 00 00 | vcan0 |
| 1590774748.907103 | 019b | 6 | 00 00 0f 00 00 00 | vcan0 |
| 1590774750.406659 | 019b | 6 | 00 00 0d 00 00 00 | vcan0 |
| 1590774945.864442 | 019b | 6 | 00 00 0d 00 00 00 | vcan0 |
| 1590774946.519650 | 019b | 6 | 00 00 05 00 00 00 | vcan0 |
| 1590774948.134589 | 019b | 6 | 00 00 07 00 00 00 | vcan0 |
| 1590774948.466710 | 019b | 6 | 00 00 0f 00 00 00 | vcan0 |
| 1590774986.484967 | 019b | 6 | 00 00 0e 00 00 00 | vcan0 |

Figure 4.1: CAN log for door lock message


Figure 4.2: Plot module: graph of door lock message signals

Finally, we created some fake messages and sent them to the bus, which worked for the simulator as our message saying all doors are unlocked made the display panel show that they are opened.

Using the same technique, we were able to detect message IDs and signals for speed and turn signals. Regarding the speed, it was quite interesting to watch the liveplot module graph changing as we were either accelerating or slowing down. Also, raw values for the speed ranged from 0 to 56, while the actual speed went up to 100 mph. We assumed we could scale the value range from (0, 56) to (0, 100), which gave us a multiplication factor 1.78571 and

```
BU_: CLS (Central locking system)
BO_ 19b DOOR_STATE: 6 CLS
        SG_ RIGHT_BACK : 20|1@1+(1,0) [0|1] "" XXX
        SG_ LEFT_BACK : 21|1@1+(1,0) [0|1] "" XXX
        SG_ RIGHT_FRONT : 22|1@1+(1,0) [0|1] "" XXX
        SG_ LEFT_FRONT : 23|1@1+(1,0) [0|1] "" XXX

VAL_ 19b DOOR_STATE 0 "unlocked" 1 "locked";
```

Figure 4.3: DBC definition for ICSim Central locking system message

offset 0. Given the fact the transformation is linear, we tested our theory by finding the raw value of the speed 50 mph, which was 28 as expected ($50 \div 1.78571 \approx 28$). Therefore, we are confident enough to say that we figured out the linear factor used in the conversion from raw to physical value and vice versa. The last byte was periodically oscillating between 0 and 255. From the graph in Figure 4.4 we can see some correlation between the acceleration in the speed signal (red line) and the unknown signal (purple line), but we can hardly tell what exactly does it represent. Remaining messages for the rest of ICSim actions are listed in Figure 4.5

We have detected 36 unique messages out of which we were able to map three of them - speed, doors and turn signals, which are the only messages we were able to interact with. CANvas module did not find any ECUs, probably because of the synthetic nature of the data.

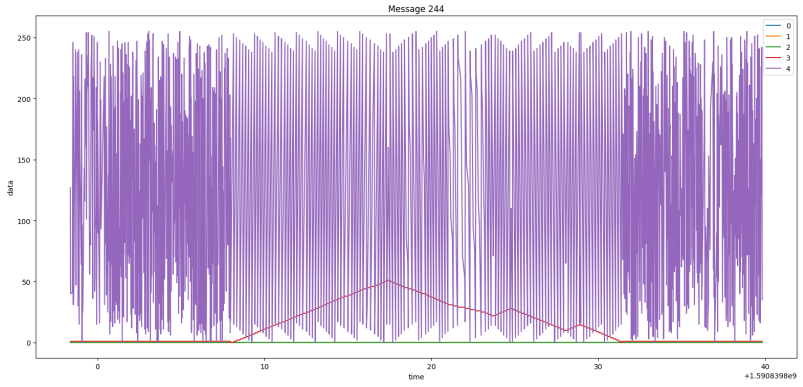


Figure 4.4: Plot module: a graph of speed message signals

```
BO_ 188 STEERING_LEVERS: 4 XXX
        SG_ TURN_SIGNALS : 6|2@1+(1,0) [0|3] "" XXX

BO_ 244 SPEED: 5 XXX
        SG_ SPEED : 24|8@1+(1.78571,0) [0|100] "" XXX
        SG_ UNKNOWN : 32|8@1+(1,0) [0|255] "" XXX

VAL_ 188 TURN_SIGNALS 0 "none" 1 "left" 2 "right" 3 "both;
```

Figure 4.5: DBC definition for ICSim turn signals and vehicle speed

## 4.2 Automobile CAN bus

We also wanted to try out the tool using a real car, which required more effort than may be expected, because of several limitations. Every car model has a unique specification, and it is not trivial to find a functional entry point to the CAN bus. Thorough research is recommended to get familiar with the target system. Finding some detailed technical information might be quite challenging as well because they are kept secret by the vendors. Furthermore, not every car has their CAN bus exposed via the Onboard Diagnostic port, referred to as OBD-II, which is the first place to look at when finding a way to the CAN bus. However, some car manufacturers put a gateway between the OBD-II port and the CAN bus subnetworks to filter outcoming traffic. Passthrough can be requested, but the handshake procedure or request methods are again not publicly available for apparent business reasons. Another way is to find the bus wires and get on the bus directly.

Either way, having a car for solely testing purposes would be great but is not feasible given the circumstances. We had to use only cars in our personal possession or ask around our friends. For instance, our first attempts with cars like Škoda Fabia or Volkswagen Golf did not yield any data from the CAN bus as we have found out they both have a gateway module behind the OBD-II. BMW X1 sends only one CAN message to the OBD port by default indicating whether the engine is on or off. Ultimately, we were given an opportunity to have a test drive with Toyota Auris and Honda CR-V that do not restrict listening to the CAN bus traffic via OBD-II. Apart from recording the live CAN bus traffic, there are also CAN bus datasets from Opel Astra and Renault Clio available at 4TU Centre for Research Data repository [31]. In every testing session, we were using Korlan USB2CAN converter from 8devices [32], shown in Figure 4.6, which connects a CAN bus device to the computer via USB port. Luckily for us, we also found an online DBC database [33] containing message definitions for a variety of car models. These definitions are part of the opendbc project – an open-source programme for enabling self-driving capabilities in a personal car. Therefore, all of the DBC definitions are

Figure 4.6: Korlan USB2CAN converter [32]

slightly modified for their self-driving agent. Besides, we must not forget they did not come from an official resource, the vendor, but a collection put up by a community. Hence they are not complete and may contain inaccuracies. Nevertheless, replaying a log from our ride and mapping the messages using DBC definitions helped us a lot to verify the functionality of our modules.

Once again, there are two ways how to process a CAN log without being connected to a live CAN bus. First of all, we can import the log directly into our framework, which will preserve original timestamps but will not give us the same live experience as if we were connected to the bus. That could be achieved with canplayer from can-utils, which can replay messages from the log to a virtual can interface, acting like a real CAN bus. However, the timestamps will be different, even though it sends the messages in the same intervals. Some modules such as CANvas could be affected by that, because they are using algorithms working with the transmission time.

Every testing should start with looking up the existing message database for the target car model. Unfortunately, we could not find any DBC or KCD definitions for Toyota Auris messages, so instead, we used definitions for Toyota Corolla available at opendbc repository. As we have later found out, Auris is only a rebranded version of Corolla hatchback model for Europe and some other countries [34], so the database should be applicable to some certain degree.

Out of 98 unique messages, we mapped DBC definitions to 18 of them, see Figure 4.7. However, the database contains 38 definitions, so there are another 20 definitions that did not match any message ID and 80 message IDs without any definition. One apparent reason for such discrepancy is simply that Toyota Auris 2016 is a different car than Toyota Corolla 2017. While Auris is a hatchback, Corolla is a sedan, and they were also released in different years. What have we learned though, is which messages and equipment they

| ID | count | period (ms) | message name |
|---|---|---|---|
| 24 | 2206 | 20 | KINEMATICS |
| 25 | 4419 | 10 | STEER_ANGLE_SENSOR |
| aa | 4396 | 10 | WHEEL_SPEEDS |
| b4 | 2201 | 20 | SPEED |
| 1c4 | 1869 | 23 | ENGINE_RPM |
| 224 | 1466 | 30 | BRAKE_MODULE |
| 228 | 2199 | 20 | ACCELEROMETER |
| 260 | 2201 | 20 | STEER_TORQUE_SENSOR |
| 262 | 1097 | 40 | EPS_STATUS |
| 2c1 | 1393 | 31 | GAS_PEDAL |
| 399 | 43 | 1024 | PCM_CRUISE_SM |
| 3b7 | 150 | 295 | ESP_CONTROL |
| 3bc | 44 | 1001 | GEAR_PACKET |
| 413 | 44 | 1000 | TIME |
| 611 | 60 | 986 | UI_SETTING |
| 614 | 6 | 8094 | STEERING_LEVERS |
| 620 | 204 | 294 | SEATS_DOORS |
| 622 | 62 | 970 | LIGHT_STALK |

Figure 4.7: Identified message IDs in Toyota Auris 2016 using DBC definitions from opendbc

```
BO_ 705 GAS_PEDAL: 8 XXX
        SG_ GAS_RELEASED : 3|1@0+ (1,0) [0|1] "" XXX
        SG_ GAS_PEDAL : 55|8@0+ (0.005,0) [0|1] "" XXX
```

Figure 4.8: DBC definition for Toyota Auris gas pedal message

have in common. Regarding the testing, we shall present one example for demonstration – gas pedal message. According to the DBC definition 4.8, there are two signals: GAS RELEASED and GAS PEDAL. The first only indicates whether the pedal is pressed (0) or not (1) and the second tells how much. Looking at the graph in Figure 4.9 where the blue line is for GAS RELEASED signal and pink line for GAS PEDAL signal, we can clearly see that this functionality works as expected.

Testing our tool on Honda CR-V 4th Gen would be similar to the previous case, so we decided to analyze the CAN bus from a new angle. This time, we found a DBC database directly for Honda CR-V, which contained 26 message definitions. In the CAN log, we identified 50 unique messages, and 16 of them were given a definition. It is possible that some messages were not sent at all during the test drive. Furthermore, instead of mapping the messages, we

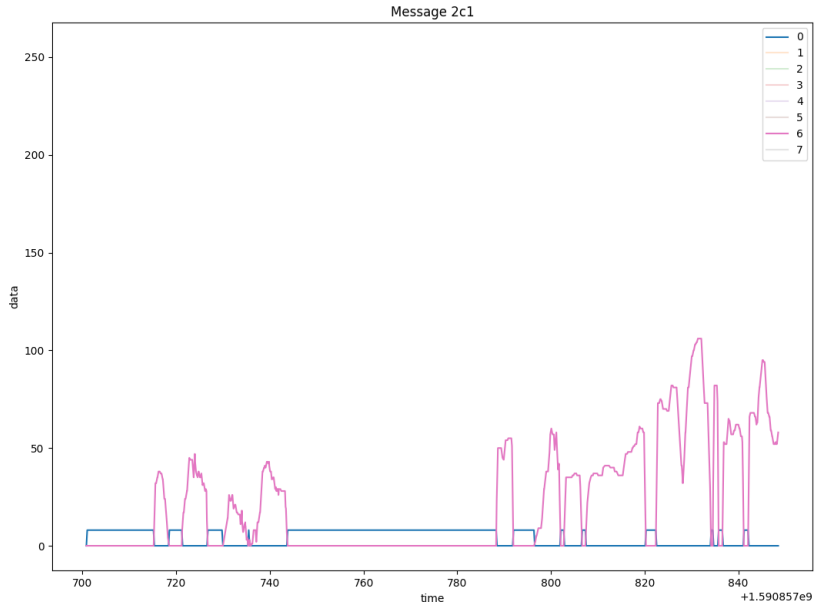


Figure 4.9: Plot module: a graph of gas pedal signals

focused on mapping ECUs connected to the bus. The DBC definition listed these nodes: EBCM, ADAS, PCM, EPS, VSA, SCM, BDY, EPB, EON. As you can see, it would be helpful to know what exactly each node name stands for, which is described next. We also added messages they transmit according to the DBC file to get a better idea about their purpose.

**EBCM** Electronic Brake Control Module

**ADAS** Advanced Driving Assistance System

- 506 BRAKE_COMMAND
- 780 ACC_HUD (Adaptive Cruise Control - Head-up display)
- 829 LKAS_HUD (Lane Keeping Assist System)

**PCM** Powertrain Control Module

- 344 ENGINE_DATA
- 380 POWERTRAIN_DATA
- 419 GEARBOX
- 804 CRUISE

- 892 CRUISE_PARAMS

**EPS** Electric power steering

- 342 STEERING_SENSORS
- 399 STEER_STATUS
- 427 STEER_MOTOR_TORQUE

**VSA** Vehicle Stability Assistance

- 420 VSA_STATUS
- 432 STANDSTILL
- 464 WHEEL_SPEEDS
- 487 BREAK_PRESSURE
- 490 VEHICLE_DYNAMICS
- 597 ROUGH_WHEEL_SPEED

**SCM** Suspension Control Module

- 422 SCM_BUTTONS
- 660 SCM_FEEDBACK

**BDY** Body Control Module

- 773 SEATBELT_STATUS
- 1029 DOOR_STATUS

**EPB** Electronic Parking break

**EON** Device containing range of sensors required for self-driving – this was probably added for opendbc project [35] and we will ignore this node in our analysis

Given this information about nodes, we employed the CANvas module to detect source ECUs and compare the results. We had to load the whole CAN log file instead of replaying it to use original timestamps. The output from the module, see Figure 4.10., consists of enumerated ECUs with a list of transmitted messages. We can tell from the result that ECU number 1 could relate to the BDY, number 3 to the SCM and number 6 to the PCM, because they have overlapping groups of messages. The rest remains unknown as the linked messages are not defined in the DBC file. However, this information help us assign new message IDs to already existing nodes like in case of SCM and PCM.

```
0 ['1024', '1036', '1108', '1125', '983']
1 ['1029', '1064', '1296']
2 ['1365', '57']
3 ['422', '542', '660', '661']
4 ['538', '597']
5 ['777', '882', '884', '888']
6 ['803', '804', '808', '892']
```

Figure 4.10: CANvas module: ECU mapping for Honda CR-V 4th Gen

# Conclusion

In the final chapter, we will sum up our achievements and discuss how much were our goals fulfilled and propose improvements for the future.

## Summary

The main objective of our project was to develop a tool that would aid CAN bus testing and CAN message analysis. In order to design and create such software, we divided the thesis in a theoretical and a practical part.

The former begins with an introduction of the CAN bus protocol and the topic of message mapping. Then we researched solutions for CAN bus monitoring and analysis. Although there are many existing applications and tools, many of them require proprietary hardware devices and have a high price-tag. Also, a lot of utilities served only a single purpose in the process of CAN bus analysis. Given the obtained knowledge from the research, we were able to identify requirements for our tool and find possible components, which could be employed in it.

The second part consists of the design and implementation sections. First of all, we described the procedure of mapping CAN messages and how to carry out every step of it. This approach gave us an idea of what must be implemented and which functionality could be achieved using existing components. Our tool is designed to work as a modular framework, allowing users to write their own modules fitting their specific needs and efforts in a simple and efficient manner. There is yet no universal way to decode raw CAN messages, and with our framework users can use or create different methods to unfold the context and meaning of certain messages. Such modularity is possible with an API provided in the framework to interact with CAN bus and the data from it effortlessly. The framework is controlled with CLI giving the user access to the core functionality as well as user modules. Implementation of this framework consisted of building core modules and integrating useful libraries for

operating CAN bus or fulfilling smaller tasks related to CAN message analysis, developing API layer and CLI application and lastly creating modules for reconnaissance. Experimenting with both simulation environment and real drive testing data rendered our work successful and shown the potential of the tool in various CAN bus testing scenarios.

## Future work

Here are a few ideas on how to improve this project apart from the suggestions proposed at the end implementation and testing chapter. Inspired by the Metasploit framework, a module database could be managed to collect interesting modules created by the users and share them with others. Many car enthusiasts cooperate together when trying to analyze CAN message for a car model. Still, these efforts are usually conferred casually on a variety of internet forums, which is not really convenient in terms of sustainability and accessibility. We can also collect and share CAN message dumps from various cars so that others can use them and help each other decoding messages.

Secondly, the tool now fully supports only the Standard CAN message format. However, another two extensions to the original protocol exist today: part B of the CAN 2.0 specification describes Extended CAN message format and CAN FD (CAN with Flexible Data-Rate) from 2012 [36]. As CAN FD is expected to replace classic CAN protocol in new car models since this year, some revision of the framework will be unavoidable in a few years.

Furthemore, we would like to conduct an interactive testing with a real vehicle using the valuable experience from both ICSim and offline CAN log analysis. Every car has a different difficulty in terms of the CAN bus access and the more challenging ones require more invasive procedures. However, acquiring a car as a testing object was not financially viable in this setting.

# Bibliography


1. ROBERT BOSCH. *CAN Specification 2.0*. 1991. Available also from: http://esd.cs.ucr.edu/webres/can20.pdf.
2. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO 11898-1:2015 Road vehicles - Controller area network (CAN) - Part 1* [online]. 2015 [visited on 2020-03-03]. Available from: https://www.iso.org/standard/63648.html.
3. TEXAS INSTRUMENTS. *Introduction to the Controller Area Network* [online]. 2002 [visited on 2020-03-01]. Available from: http://www.ti.com/lit/an/sloa101b/sloa101b.pdf.
4. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO 15765-2:2016 Road vehicles — Diagnostic communication over Controller Area Network (DoCAN) - Part 2* [online]. 2016 [visited on 2020-03-10]. Available from: https://www.iso.org/standard/66574.html.
5. CSS ELECTRONICS. *CANopen Explained - A Simple Intro* [online]. 2020 [visited on 2020-03-10]. Available from: https://www.csselectronics.com/screen/page/canopen-tutorial-simple-intro.
6. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO 14229-1:2020 Road vehicles - Unified diagnostic services (UDS) - Part 1* [online]. 2020 [visited on 2020-03-10]. Available from: https://www.iso.org/standard/72439.html.
7. NATIONAL INSTRUMENTS. *Controller Area Network (CAN) Overview* [online]. 2019 [visited on 2020-05-25]. Available from: https://www.ni.com/cs-cz/innovations/white-papers/06/controller-area-network--can--overview.html.
8. MILLER, Chris; VALASEK, Charlie. Remote Exploitation of an Unaltered Passenger Vehicle [online]. 2015. Available also from: http://illmatics.com/Remote%20Car%20Hacking.pdf.

9. SCHAPPIN, C. N. I. W. *Intrusion detection on the automotive CAN bus*. 2017. Master's thesis. TU Eindhoven.

10. RADU, Andreea-Ina; GARCIA, Flavio. LeiA: A Lightweight Authentication Protocol for CAN. *Computer Security – ESORICS 2016*. 2016. ISBN 978-3-319-45743-7.

11. HUYBRECHTS, Thomas et al. Automatic Reverse Engineering of CAN Bus Data Using Machine Learning Techniques. *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*. 2017. ISBN 978-3-319-49108-0.

12. MARKOVITZ, Moti; WOOL, Avishai. Field classification, modeling and anomaly detection in unknown CAN bus networks. *Vehicular Communications*. 2017, vol. 9.

13. MARCHETTI, Mirco; STABILI, Dario. READ: Reverse Engineering of Automotive Data Frames. *IEEE Transactions on Information Forensics and Security*. 2019, vol. 14, no. 4.

14. VECTOR INFORMATIK. *CANoe* [online]. 2020 [visited on 2020-02-20]. Available from: `https://www.vector.com/int/en/products/products-a-z/software/canoe`.

15. MICROCHIP TECHNOLOGY. *CAN BUS Analyzer Tool* [online]. 2020 [visited on 2020-02-25]. Available from: `https://www.microchip.com/Developmenttools/ProductDetails/APGDT002`.

16. LINUX-CAN. *can-utils* [online]. 2020 [visited on 2020-01-20]. Available from: `https://github.com/linux-can/can-utils`.

17. OLIVER HARTKOPP AND URS THUERMANN AND OTHERS. *SocketCAN - Controller Area Network* [online] [visited on 2020-02-08]. Available from: `https://www.kernel.org/doc/html/latest/networking/can.html`.

18. MEIER, Jan-Niklas. *Kayak* [online]. 2020 [visited on 2020-02-15]. Available from: `https://dschanoeh.github.io/Kayak`.

19. THORNE, Brian. *python-can* [online]. 2020 [visited on 2020-02-15]. Available from: `https://github.com/hardbyte/python-can`.

20. KVASER. *Kvaser CANlib SDK* [online]. 2020 [visited on 2020-02-15]. Available from: `https://www.kvaser.com/developer/canlib-sdk`.

21. CSS ELECTRONICS. *CANedge2* [online]. 2020 [visited on 2020-02-15]. Available from: `https://www.csselectronics.com/screen/product/can-lin-logger-wifi-canedge2`.

22. HRISCA, Daniel. *asammdf* [online]. 2020 [visited on 2020-02-15]. Available from: `https://github.com/danielhrisca/asammdf`.

23. KULANDAIVEL, Sekar et al. CANvas: Fast and Inexpensive Automotive Network Mapping. *28th USENIX Security Symposium*. 2019.

24. VECTOR INFORMATIK. *DBC File Format Documentation*. 2007. Available also from: `http://read.pudn.com/downloads766/ebook/3041455/DBC_File_Format_Documentation.pdf`.

25. MEIER, Jan-Niklas; KRUEGER, Jens. *kcd* [online]. 2017 [visited on 2020-02-20]. Available from: `https://github.com/julietkilo/kcd`.

26. MOQVIST, Erik. *CAN BUS tools* [online]. 2019 [visited on 2020-04-28]. Available from: `https://cantools.readthedocs.io/en/latest/`.

27. THE MATPLOTLIB DEVELOPMENT TEAM. *Matplotlib: Visualization with Python* [online]. 2020 [visited on 2020-04-02]. Available from: `https://matplotlib.org/3.2.1/index.html`.

28. MURVAY, Pal-Stefan; GROZA, Bogdan. DoS Attacks on Controller Area Networks by Fault Injections from the Software Layer. *ARES '17: Proceedings of the 12th International Conference on Availability, Reliability and Security*. 2017, vol. 29.

29. OPENGARAGES. *Instrument Cluster Simulator for SocketCAN* [online]. 2020 [visited on 2020-03-30]. Available from: `https://github.com/zombieCraig/ICSim`.

30. SMITH, Craig. *The Car Hacker's Handbook: A Guide for the Penetration Tester*. San Francisco: No Starch Press, 2016. ISBN 978-1-59327-703-1.

31. DUPONT, Guillaume; LEKIDIS, Alexios. *Automotive Controller Area Network (CAN) Bus Intrusion Dataset v2* [online] [visited on 2020-04-10]. Available from: `https://data.4tu.nl/repository/uuid:b74b4928-c377-4585-9432-2004dfa20a5d`.

32. 8DEVICES. *Korlan USB2CAN* [online] [visited on 2020-01-25]. Available from: `https://www.8devices.com/products/usb2can_korlan`.

33. COMMA.AI. *opendbc* [online] [visited on 2020-04-09]. Available from: `https://github.com/commaai/opendbc`.

34. AUTOEVOLUTION. *TOYOTA Auris 5 Doors 2013 - 2015* [online]. 2020 [visited on 2020-05-20]. Available from: `https://www.autoevolution.com/cars/toyota-auris-5-doors-2013.html`.

35. COMMA.AI. *FAQ - EON* [online]. 2019 [visited on 2020-05-21]. Available from: `https://community.comma.ai/wiki/index.php/FAQ#EON`.

36. ROBERT BOSCH. *CAN with Flexible Data-Rate*. 2012.

Appendix A

# Acronyms

**API** Application programmable interface

**CAN** Control Area Network

**CANFD** Control Area Network with Flexible Data-Rate

**CLI** Command line interface

**ECU** Electronic control unit

**GPS** Global Positioning System

**GUI** Graphical user interface

**ISO** International Standard Organisation

**I/O** Input/Output

**KCD** Kayak CAN definition

**LLC** Logical link control

**MAC** Media access control

**MDF** Measurement Data Format

**SDK** Software development kit

**XML** Extensible markup language

Appendix B

# Contents of enclosed USB flash drive

- `README.md` ..... the file with USB drive contents description
- `src` ..... the directory of source codes
  - `CANdy` ..... implementation sources
    - `candy_app.py` ..... CANdy application launcher
    - `setup.sh` ..... install script
    - `setup_vcan.sh` ..... script for setting up virtual can
    - `base` ..... folder containing the Core module
    - `lib` ..... CANdy libraries
    - `modules` ..... CANdy modules
    - `misc` ..... testing files and other
  - `thesis` ..... the directory of LaTeX source codes of the thesis
  - `demo` ..... demo video of using CANdy with ICSim
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format