Generalization of a Machine Learning Classifier of CAN Bus Signals

BY

ANDREA TRICARICO
B.S., Politecnico di Milano, Milan, Italy, 2017
M.S., Politecnico di Milano, Milan, Italy, 2020

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2020

Chicago, Illinois

Defense Committee:

Prof. Rigel Gjomemo, Chair and Advisor
Prof. Ugo A. Buy
Prof. Stefano Zanero, Politecnico di Milano

ACKNOWLEDGMENTS

I want to thank my parents, for always supporting and never stopped believing in me.

AT

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

RKE	Remote Keyless Entry
LSTM	Long Short-Term memory
RNN	Recurrent Neural Networks
CAN	Controller Area Network
DLC	Data Length Code
OBD	On-Board Diagnostics
UDS	Unified Diagnostic Service
ECU	Electronic Control Unit
CRC	Cyclic Redundancy Check
NN	nearest neighbor
DTW	Dynamic Time Warping
OVA	one-versus-all
AVA	all-versus-all
RPM	Revolutions per minute
CANH	CAN high
CANL	CAN low
PCI	Protocol Control Information

LIST OF ABBREVIATIONS (continued)

OBD-II PIDs	On-board diagnostics Parameter IDs
NIST	National Institute of Standards and Technology
ROC	Receiver Operating Characteristic
IDS	Intrusion Detection Systems
SOF	Start of Frame
UDS	Unified diagnostic services
SRR	Substitute Remote Request
IDE	Identifier extension bit
RTR	Remote Transmission Request
DLC	Data Length Code
CRC	Cyclic Redundancy Check
ACK	Acknowledgement
EOF	End of Frame
IMU	Inertial Measurement Unit

SUMMARY

In the last decades, the automotive field has evolved considerably. Modern cars are composed of a complex network of Electronic Control Units (ECUs). These ECUs are small computing units that elaborate data captured from different sensors and then perform actions throw actuators. They can optimize a variety of different actions performed by the driver (e.g. the fuel injection during acceleration) and activate safety mechanisms. Moreover, in the latest years, manufacturers have introduced new functionalities, like autonomous driving features and the possibility to control the car remotely, which require a complex network of computing units inside the vehicle.

All the ECUs are connected with an in-vehicle network which is usually designed following the CAN bus protocol developed by Bosch in the '80s (which is the de-facto standard for the internal networks of modern vehicles). This protocol is particularly suitable to design real-time networks that are simple and cheap.

The security against external attacks was not a problem at that time because the vehicles were not accessible from the external world. However, in the last years, more and more cars have introduced features to connect the vehicle to external devices (like the smartphone of the driver) and networks (like the Internet). These changes have considerably increased the attack surface and the lack of security in the CAN bus protocol is becoming a serious problem.

Many studies have demonstrated that it is possible, for an attacker, to gain control of some vehicle's functionalities through the injection of malicious packets in the CAN bus and that this can be done even remotely exploiting vulnerabilities of the computing units that are connected to external devices.

Unfortunately, manufacturers are still not paying enough attention to the security of the in-vehicles networks and they rely on a security-by-obscurity paradigm (keeping secret the detailed specification about the logic inside the ECUs and the syntax of the packets).

Researchers that study the security of modern vehicles and their vulnerabilities have to manually reverse engineer the packets sent from the ECUs to understand their internal logic and find their vulnerabilities.

In this work, we propose a new approach to the reverse engineering of CAN bus packets to reduce the effort and time needed for this process.

We analyze directly the raw logs that the researchers collect from their vehicles during a driving session. We extract the different signals that are encoded in the recorded sequence of CAN packets and process them.

We then use a machine learning classifier based on LSTM networks (a type of recurrent neural network particularly suited for the analysis of time-series) to find some specific physical signals without any prior knowledge on the vehicle architecture and without performing long experiments on the vehicle.

In our evaluation phase, we demonstrate that this model can drastically reduce the time needed to reverse engineer messages and signals of a vehicle without its proprietary specification. We then explain how to collect a suitable dataset to train the model and make it able to generalize the knowledge to classify signals of unknown vehicles.

CHAPTER 1

INTRODUCTION

1.1 Modern automotive vehicles

Talking about means of transportation, cars (and in general road vehicles) are one of the main ways that people use to move. We use cars almost every day, we think to them as mere mechanical vehicles and we expect them to be reliable.

The fact is that, nowadays, cars are not simple mechanical machines anymore. As time passes, vendors equip cars with new functionalities and technologies. Today's cars have lots of computing units that are all connected to offer services to the driver (things like infotainment or even autonomous driving functionalities), collect data from different sensors and perform physical actions after receiving some specific inputs.

Modern cars can have more than 100 Electronic Control Units (ECUs) and they are all connected thanks to one or more networks that are present in the car. There are different protocols for the design of these internal networks but the de facto standard protocol is the Controller Area Network (CAN). This protocol has been developed by Bosch [1] in 1986. The focus of Bosch in the design of this protocol was on providing a system that was able to transmit safety messages (e.g. those to activate the airbag in case of collision).

1.2 CAN network and possible attacks

The CAN network is composed of a bus that connects all the ECUs in a vehicle. Each message sent on the bus contains different information retrieved by some sensors and/or commands sent from a computing unit to another. The messages are broadcasted from the sender to every Electronic Control Unit (ECU) connected to the bus line.

The engineers at Bosch needed a system that was simple, cheap but also able to respect strict real-time requirements and the Controller Area Network (CAN) protocol meets all these requirements. The security against an external attacker was not a problem at that time because the cars were not connected to anything else, this has caused a lack of attention to common protection mechanisms that we have in other sensitive networks which are exposed to the world.

Following the improvement of technology, even the computing units inside the cars have become more and more advanced. The more computing power and cheap connections technology led to install new computing units that are complex and connected to external devices or even to the internet using the broadband cellular network. Modern cars have Bluetooth to connect mobile smartphones to the infotainment of the vehicle and, with some cars, it is also possible to take control of some vehicle's functionalities with the use of the vendor's app that connects to the car using an internet connection.

Now that the isolation is not effective anymore, the lack of security (e.g. authentication, encryption, access control) is becoming a serious threat for vehicles and people that uses them. The two main problems are:

- the new complexity of the computing units makes them more vulnerable to attacks

- the connection to the external world makes possible, and sometimes very easy, for an attacker to connect to the internal network of a vehicle and perform malicious actions sending some engineered packet to trick the ECUs.

1.3 Security studies of modern vehicles

There are lots of different studies where researchers were able to exploit all these computing units inside commercial vehicles to gain control of some parts of the vehicle and to make them perform actions (unwanted by the driver and that can be very dangerous) just by the injection of some particular instruction in the car network, even remotely [2].

To prevent these dangerous attacks it is necessary to study and increase the security of these machines. One of the main problems that the researchers face when they study the possible attacks and the security of modern vehicles is to understand the internal architecture of the vehicle that they are testing. The internal network of a vehicle is composed of more than one hundred ECUs that broadcast messages on the CAN bus. But the code of the ECUs is not publicly available and then, for the researches, it is impossible to understand the meaning of the messages that flow in the bus. In the definition of the CAN standard, it is only specified the general structure of a CAN packet, but it is up to each manufacturer to decide how to encode the data inside the data frames and the logic that rules the exchange of information and commands between different computing units.

One of the most difficult parts of studies of the security of a vehicle is the preliminary phase in which the researchers have to reverse engineer the data frames that the ECUs sends to understand how they behave and how it is possible to trick them to do malicious actions.

Car manufacturers rely on the ***security by obscurity*** [3] approach, they think that making the design and the implementation of their vehicles secret is a good method to obtain a secure product. Unfortunately, that is completely wrong, and even the National Institute of Standards and Technology (NIST) recommend an open design approach and discourage this practice "System security should not depend on the secrecy of the implementation or its components" [4]. Manufacturers should move from the current *security by obscurity* to a *secure by design* [5] approach.

The current lack of public specification about the vehicle's architecture for both attackers and researchers makes long and difficult the initial study of a vehicle. The great effort needed can discourage researchers and white hats [6] but for an attacker, it might be worth it.

1.4 Contributions of this work

The goal of our work is to reduce the effort needed to reverse engineer the internal architecture of a vehicle without the secret information in posses of the manufacturer. The tricky part of this process is to understand how each different unit sends and receives data and commands.

Both attackers and researchers focus on how each information is encoded in each packet because understanding how each piece of data and each command is encoded:

1. helps to understand the logic inside the different units that elaborate those packets.
2. makes possible to send fake information to exploit some vulnerabilities in the computing units and to make them do malicious actions.

This reverse-engineering process is long and difficult to do by scratch. Until now the state-of-art studies on reverse engineering of CAN packets are focused on the unsupervised analysis of the packet's traces and the use of statistical analysis to process the logs of the messages recorded from the CAN bus. This is not sufficient, these processing steps can't classify a signal encoded in a CAN packet without external information recorded by the researchers during the recording of the packets (which make longer and more difficult this phase).

We want to investigate the possibility to create a general machine learning classifier that can be trained with a group of vehicles and then can be used by researchers directly on raw logs from the CAN bus on different vehicles. With such a model it would be possible to find the signals very quickly and with minimal effort.

CHAPTER 2

BACKGROUND AND MOTIVATION

2.1 Controller Area Network

All the ECUs inside a vehicle need to communicate with each other to exchange data collected from the different sensors and make it possible to send commands to the actuators. Many different protocols can be used for the design of the internal network of vehicles but the CAN bus [7] has become the de facto standard for the majority of commercial vehicles and it gives the specification for a network that supports real-time systems and communication where different packets can have different priorities. The CAN bus protocol gives the specifications for the first two levels of the standard Open Systems Interconnection model (OSI model), the physical and data link layer.

In the CAN bus, all packets are broadcast to all the ECUs of the network. A CAN message does not have routing information like a receiver or a destination field, it only has an ID field. An ECU, upon receiving a message, reads the ID of the packet and, from that, it will understand if it has to process the packet or not.

A vehicle can have more than only one network, even with different protocols. But for commercial cars, it is common to have one or two (one for low-frequency messages and the other for high-frequency ones) internal CAN bus networks. If a car has two different CAN buses it usually uses them to separate low priority messages (e.g. for the infotainment systems) from the critical ones that are related to the driving functionalities. This is not always true and there can be some ECUs connected to both networks that work as a bridge [8]: it is possible to exploit those ECUs to gain the access to the inner network from the one connected to the external world.

2.1.1 Physical Layer

A CAN bus is composed of two or more nodes (the ECUs in the vehicle) connected by a two-wire bus. The wire buses are needed to transmit data bits using differential wired-AND signals: the transmission of data is done using two signals CAN high (CANH) and CAN low (CANL). To send a logic 0 bit CANH is set higher than CANL and to send a logic 1 bit CANH is set lower or equal to CANL.

The most common ISO standards that are used are:

1. ISO 11898-2 [9]: it is the standard for the high-speed CAN (the one used to connect all the safety-related ECUs). It is able to reach a transfer rate of 1 Mbit/s (it can reach 5 Mbit/s on CAN-FD [10] protocol in the newest vehicles). In modern cars, the transfer rate used in the high-speed bus is usually 500 kbit/s. The recessive voltage is 2.5V (to send a 1 logical bit) and the dominant voltage (to send a 0 logical bit) is 3.5V for CANH and 1.5 for CANL
2. ISO 11898-3 [11]: it is the standard for the low-speed and fault-tolerant CAN, with a transfer rate of 125 kbit/s and that uses larger voltage swings.

2.2 CAN packets

There are 4 different types of packets in the CAN standard protocol: Data frames, Remote frames, Error frames, and Overload frames.

1. Data frames: normal information exchange between the ECUs. These can be further divided in *normal CAN messages* and *diagnostic CAN messages.*
2. Remote frames: usually data are transmitted by each ECU periodically with a certain frequency but an ECU can use remote frames to request data from a source, this mechanism is rarely used.
3. Error frames: these frames are sent by an ECU upon the detection of an error in the network.
4. Overload frames: are used from an ECU to signal to the others that it is overloaded.

For our scope, we will focus our attention on the data frame packets.

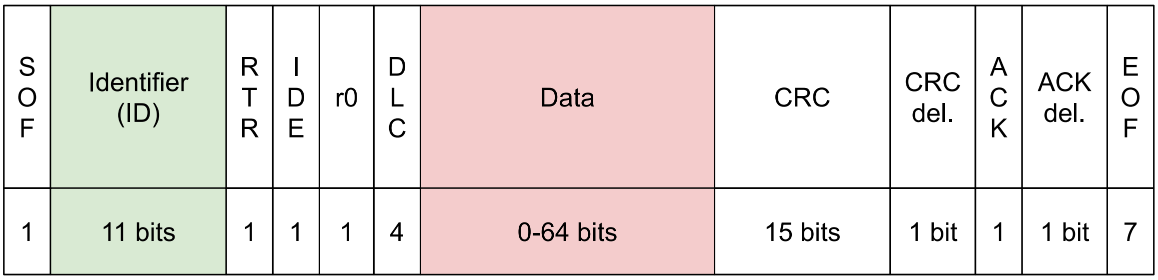
The latest version of the CAN protocol published by Bosch is CAN2.0, it reports two different formats for the data frames: one with an 11-bit identifier (CAN2.0A) and the other with a 29-bit identifier(CAN2.0B). Every vehicle that respects the standard must accept the packets formatted following the CAN2.0A and it is up to the vendors to decide to implement the extended version too[12].

The structure of the CAN packet (data link layer) is described in Figure 1 for the CAN2.0A format and in Figure 2 for the CAN2.0B format:

- Start of Frame (SOF)

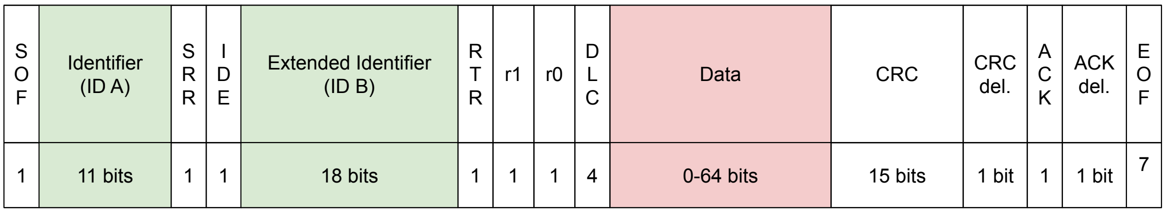
- Identifier (ID): unique identifier which also represents the priority of the message, 11 bits or 29 bits (11+18)
- Substitute Remote Request (SRR)
- Remote Transmission Request (RTR): muse be `0` for Data frames and `1` Remote frames reference.
- Identifier extension bit (IDE): must be `0` for standard format and `1` for extended format (29-bit for ID)
- r1,r0: reserved bits
- Data Length Code (DLC): number of bytes in the data field (`0`–`8` bytes)
- Data field: `0-8` bytes of actual data contained in the packet
- Cyclic Redundancy Check (CRC): error-detecting code
- CRC delimiter
- Acknowledgement (ACK) field
- ACK delimiter
- End of Frame (EOF)

The data field of the CAN packet contains `0` to `8` bytes of information sent by the ECU. The data is encoded in different ways depending on the particular vendors and the specif vehicle. This is one of the main problems that make it so hard to understand the meaning of the information that flows in the CAN bus without prior knowledge. The data field contains



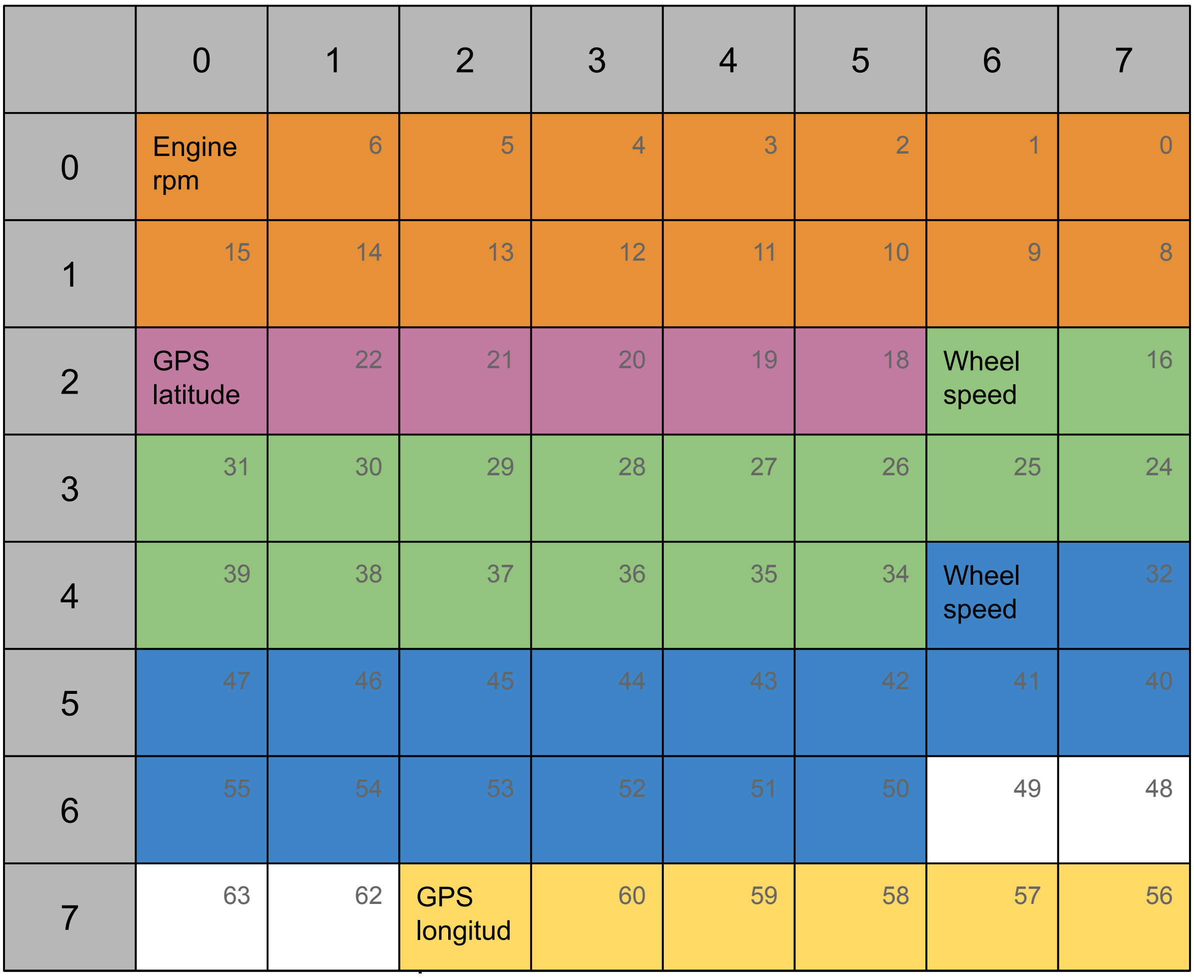
SOF	Identifier (ID)	RTR	IDE	r0	DLC	Data	CRC	CRC del.	ACK	ACK del.	EOF
1	11 bits	1	1	1	4	0-64 bits	15 bits	1 bit	1	1 bit	7

Figure 1: CAN 2.0 A data frame structure



SOF	Identifier (ID A)	SRR	IDE	Extended Identifier (ID B)	RTR	r1	r0	DLC	Data	CRC	CRC del.	ACK	ACK del.	EOF
1	11 bits	1	1	18 bits	1	1	1	4	0-64 bits	15 bits	1 bit	1	1 bit	7

Figure 2: CAN 2.0 B data frame structure



	0	1	2	3	4	5	6	7
0	Engine rpm	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	GPS latitude	22	21	20	19	18	Wheel speed	16
3	31	30	29	28	27	26	25	24
4	39	38	37	36	35	34	Wheel speed	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49	48
7	63	62	GPS longitud	60	59	58	57	56

Figure 3: Layout of the payload of a CAN packet in the DBC files

information related to the physical world like data from sensors and commands for the actuators but also unused bits and sequences of data used by the internal logic of the network.

2.2.1 Normal packets

The first category of data frames comprises the normal packets exchanged between the ECUs of the vehicle. At the application layer, these packets contain only the ID and the payload (sequence of bits in the data field).

Our dataset is a collection of messages of this category and, for our work, we use only this type of messages because these are the easiest to collect from a vehicle: it is just needed to connect a computer to the CAN bus and record every message that flows in the internal network (remember that all messages are broadcasted to all the nodes connected to the bus).

2.2.2 Diagnostic packets

Diagnostic CAN messages are designed for the communication between the diagnostic tools used by mechanics and the ECUs of the vehicle.

Diagnostic packets follow the specifications of the ISO-TP (see 2.2.3) protocol (we have not found the use of the protocol in CAN packets captured during our experiments). After the extraction of the ISO-TP header, the format of the actual data sent with diagnostic packets is described by the Unified diagnostic services (UDS) protocol and standardized in the ISO 14229 [13]. A list of available services is described in Table I.

These messages are more powerful than the normal ones and, to avoid that someone misuses them, the ECUs will usually ignore these messages if the car is moving at a certain speed. Unfortunately, Miller and Valasek showed [2] that for an attacker is possible to forge a fake speed signal and then enforce the ECUs to process the diagnostic packets even when it should be not possible.

Moreover, to do sensitive actions, it is necessary to authenticate to the ECU using the Security Access service. Sadly, even in this case, Miller and Valasek found many problems in their vehicles [2] because this access control mechanism is often not implemented in a right and robust way. For example in one of the ECUs the seed sent for the authentication is always the same and then the answer to authenticate is the same too and it is just necessary to sniff someone performing authentication or do a brute-force attack.

An example of the power of this kind of packets is the use of the services of ”Input Output Control”. With these services it is possible (after an authentication step) to send custom input to the ECU (instead of the real ones from its sensors) and monitor the response. This service is usually needed my mechanics to test if the responses to the input are correct but it is quite obvious how critical this feature is if an attacker can access it when someone is driving the car.

2.2.3 ISO-TP

The payload of each message contains `0` to `8` bytes of data but it is possible to send sequences of data longer than eight bytes thanks to the standard ISO 15765-2 (called also *ISO-TP*) [14] that describe how to use CAN packets to send payloads of arbitrary length. In this standard the first nibble (4 bits) of the data field is used to define the Protocol Control Information (PCI) type, this information is then used by the ECUs to control the flow of packets and to reconstruct the original payload that was sent. The possible values of the PCI are:

- 0 for single-frame packets, those that contain the entire payload in the data field on that message.
- 1 for the first packet of a sequence that contains the different parts of a payload.

Service ID	**Service**
0x10	Diagnostic Session Control
0x11	ECU Reset
0x14	Clear Diagnostic Information
0x19	Read DTC Information
0x22	Read Data By Identifier
0x23	Read Memory By Address
0x27	Security Access
0x28	Communication Control
0x2A	Read Data by Periodic ID
0x2E	Write Data By Identifier
0x2F	Input Output Control By Identifier
0x30	Input Output Control By Local Identifier
0x31	Routine Control
0x34	Request Download
0x35	Request Upload
0x36	Transfer Data
0x37	Transfer Exit
0x3D	Write Memory By Address
0x3E	Tester Present
0x83	Access Timing Parameters
0x84	Secured Data Transmission
0x85	Control DTC Setting
0x86	Response On Event
0x87	Link Control

TABLE I: List of UDS services

- 2 for all the consecutive packets of a sequence, for these packets the second nibble of bits is used as index inside the sequence.
- 3 for flow-control-frames, these packets are used as ACK of the first packet of a sequence.

```
ID = 127 DLC = 8 Data = 1A 13 56 BC 98 23 56 08

ID = 127 DLC = 8 Data = 30 00 00 00 00 00 00 00

ID = 127 DLC = 8 Data = 21 BB CC DD 12 34 56 78

ID = 127 DLC = 8 Data = 22 90 AB DF EG 46 53 76
```

Following the ISO-TP protocol the actual payload sent with this sequence of packets is:

```
A 13 56 BC 98 23 56 08 BB CC DD 12 34 56 78 90 AB DF EG 46 53 76
```

2.2.4 DBC files

What is used by the ECUs to decode each frame is the information that is collected in the DBC files of a vehicle.

The DBC files [15] of a vehicle (also called Communication Database for CAN) are the CAN message translation tables. For completeness, it is correct to point out that there are also other types of translation tables but the de facto standard is DBC.

In the DBC it is possible to find all the information needed to decode the information in the data frames, as it showed in Figure 3.

For each possible ID, they explain:

- which ECU is the sender of that packet and which are the ECUs that will receive (and process) it
- the length of the data section of the packet
- the separation in blocks of the data section:
  - start bit of the block
  - length in bits of the block
  - multiplicative factor
  - offset
  - encoding method (e.g. two's complement; representation with big-endian or little-endian)
  - max/min value
  - unit of measurement

In the DBC are listed all the signals sent by each unit connected to the internal network of the vehicle and each signal has its start bit and end bit (start + length) because in each CAN message it is possible to have more than one single signal (as it is shown in Figure 3 where each color describe a different signal in that particular message).

2.2.5 OBD-II

The On-Board Diagnostics (OBD)-II is a protocol used to make available to owners of the vehicles and technician emission-related information of the vehicle.

It is mandatory by law, for every vehicle produced after 1996, to implement the On-board diagnostics Parameter IDs (OBD-II PIDs) (which are standardized by the SAE J1979[16]) and to make that accessible through the standard port described in the SAE J1939 [17] which is showed in Figure 4. The only exception is electric cars which are not mandated to implement an OBD-II port and to support the OBD-II protocol (even if some of them implement it anyway).

With an OBD-II connector, it is possible to use the OBD-II PIDs codes to request standardized information from different ECUs. It is possible to use connectors with a built-in interpreter, like the ELM 327 [18], which can be used to send periodically request of information and then logs the answers.

The information available using this protocol is mainly emission-related and includes vehicle speed, engine speed, air-flow, internal temperatures among others.

This is a very small subset of the information available in the CAN bus but, differently to the correspondent signals sent normally inside the bus, the answers to the OBD-II PIDs are standardized (and equal for every vehicle) and they can be interpreted by anyone without the DBC files of the vehicle.

The OBD-II port is connected to the CAN bus because the OBD-II PIDs requests are sent using the normal bus in addition to the normal CAN packets. For this reason, this port is used not only to make requests using the homonymous protocol but as a standard way to connect to the CAN bus (to sniff the traffic and inject messages).

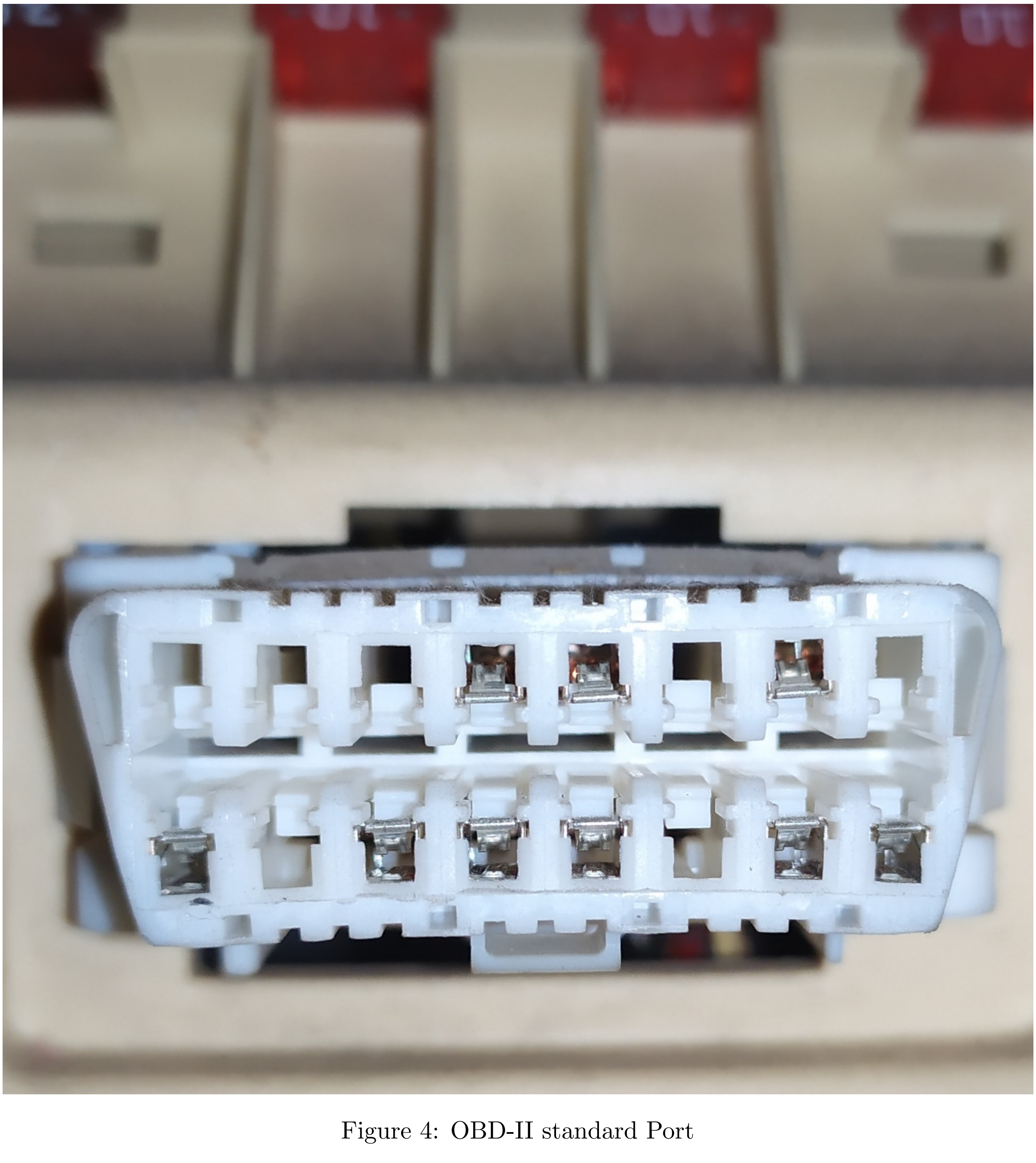


Figure 4: OBD-II standard Port

2.3 Automotive security

In this section, we want to give an overview of the researches that have demonstrated the fragility and lack of security in the internal architectures of our vehicles.

2.3.1 Attack surfaces

As well discussed by Checkoway et al. [19], an attacker can reach the internal network of a vehicle in different ways but these can be categorized in three different sets:

1. indirect physical access: the attacker can reach the internal network of our car using a physical intermediary. A physical device can be connected to the OBD-II port [20], which is present in all the vehicles by law, thanks to a temporary access to the vehicles. Otherwise, an attacker can reach the internal network using a counterfeit or malicious component sold online (FM radio) or infecting a device of the car's owner that will be connected to the car (an MP3 player, a CD, a USB drive).
2. short-range wireless access: this can be achieved mainly with the Bluetooth technology that is now used in the majority of vehicles to connect the smartphone to the car to reproduce music or to control the car from the manufacturer's mobile app. Other possible short-range wireless accesses uses two technologies that are present in almost every recent vehicle Remote Keyless Entry (RKE) (used to remotely open the doors, turn on light and other small actions) and RFID-based technology (used to implement tags inside the car keys that deactivate the vehicle immobilizers).
3. long-range wireless access: this category can be divided into broadcast channels and addressable channels. The former is accessible by sending messages using networks like GPS

or FM radio frequencies. In the latter, the access is possible thanks to the connectivity that many vehicles have using the mobile broadband network to connect to the internet their remote telematics systems.

An evident example that proves that the internal network of a car is accessible by external devices is given by all the automotive companion apps [21] that can send messages to remote control the vehicle (e.g. unlock the car and starting the engine) or by the software inside many OBD-II dongles that send OBD-II PIDs [22] to the CAN bus (but that can be counterfeit to send malicious packets too).

In some vehicles there is no a single internal network, it is common to have two different CAN buses: one high-frequency bus for safety-critical ECUs and the other for infotainment and other less critical features. The safety-critical bus is usually isolated from the outside, even if this has not usually done for security reasons but to improve bandwidth and integration [19]. Besides, even in those cases where the safety-critical network is not connected to other external networks, there is always one (or more) ECU which is connected to both the networks to support the exchange of messages between the all the ECUs and work as logical bridges between networks [19] [23]. Then it is possible to send malicious messages to safety-critical computing units by accessing the infotainment system and then infect the ECU that is connected to both the buses.

2.3.2 Causes of lack of security

If we think about the automotive field it is quite easy to understand the causes behind the lack of security in modern vehicles. The first one is, as we pointed out previously, that the base design of modern vehicles still relies on an architecture developed decades ago when the systems were secure because of their isolation and the complexity was much lower.

As suggested by Checkoway et al. [19] the lack of interest by manufacturers is caused by the absence of significant adversarial pressure. We have not seen many cases in which the vulnerabilities have bees exploited to make attacks that were a threat to the driver of the car. Until now almost all cyber-attacks had as a goal the car theft [24] and not the health of the driver.

The majority of the vulnerabilities found so far were at the interface boundaries between pieces of code developed by distinct companies [19]. This is a common problem in computer science but in the automotive field, it can cause a serious problem even because car manufacturers outsource the production of the computing units and then only integrate them in their vehicle: with this paradigm, different manufacturers do not have access to the code in the different units and it is very difficult to test the security of the vehicles as a whole after the integration of the different units.

Moreover, even when security measures are implemented we have seen that often it has been done in a wrong way that makes them ineffective (e.g. the security access control for diagnostic actions in some units, see 2.2.2).

Miller and Valasek have carried out different researches [8] [23] intending to get the eyes of car manufacturers on these problems. Unfortunately, car manufacturer underestimated all the problems because many experiments were conducted using an indirect/direct physical access to the vehicle and, in that case, there are many other attacks, not related to the injections of malicious messages in the car bus, which are much easier and dangerous (e.g. tampering with the brakes of the vehicle). To show the importance of focusing the attention to the attacks that can be carried out by exploiting vulnerabilities of the CAN bus and doing that remotely, Miller and Valasek showed [2] that it is possible, not only in theory, to send packets to an internal ECU of a car remotely, without physical access to the vehicle.

To understand the importance and the gravity of the situation, we report here some attacks that different studies have proved to be feasible:

- Change the speed and Revolutions per minute (RPM) displayed to the driver (which can make the driver accelerate and underestimate his real speed).
- Cause denial of service in the CAN bus that can cause a shut-down of the vehicle (which can cause an immediate loss of assistance in the steering).
- Make small adjustments to the steering of the wheels (that can be very dangerous if the car is moving at high speed)
- Slow down and completely stop the car exploiting the pre-collision system

We also need to understand that, with the new autonomous driving features of modern cars, these problems can only become worse (if manufacturers will not focus on security during the design phase).

2.4 State of the art of CAN packets reverse engineering

Many researchers have focused their work on finding countermeasures of possible attacks to the CAN bus, in particular, many Intrusion Detection Systems (IDS) have been proposed in recent years [25]. To carry out many of these researches and evaluate their results the researchers must understand the internal architecture of the vehicles used in the experiments and to do so it is necessary to reverse engineer the packets that are sent and received from the ECUs connected to the CAN bus.

In this section, we describe the existing works on the reverse engineering of CAN communications.

In the last years, researches proposed different methods that can be of some help during the reverse engineering process of CAN packets, which is important in those situations where the DBCs files are not available. Unfortunately, there is still no truly resolutive approach to this problem.

Wen et al. described and implemented **CanHunter**[21], a system that can reverse engineer CAN commands analyzing the car's companion app. As we have discussed, modern vehicles have incremented their connections to the external world to give to the drivers more functionalities, examples of this are all the companion apps that vendors make available for the modern smartphone and that can be connected to the car to make them execute different actions triggering them remotely from the app. Wen at al. have described a system that reverses engineer the companion app of a vehicle to retrieve the CAN bus messages used to trigger the possible actions. The limitations of this work are that only a few vendors have already produced companion apps for their cars and, even if the app exists, the commands sent from the app are just a little fraction of the possible signals sent and received in the car's network.

Another source of possible information is the data that can be extracted using the OBD-II PIDs. The **ACTT** algorithm [26] developed by Verma et al. extracts extra signals using that protocol and then uses them to find the corresponding signals in the CAN bus. This work is mainly limited by the very small number of the signals that are standardized in OBD-II PIDs, moreover, the work has been evaluated on old vehicles that have much fewer signals in the internal network respect to newer ones.

Young et al. [27] suggested a very different approach. They proposed to use a machine learning classifier to group messages which are strongly related to each other.

1. The first step of their work is the reverse engineering of signals correlated to the speed and the action of the brake system by monitoring the changes of the bits in CAN messages during acceleration and the activation of the brakes.
2. In the second step, they used a hierarchical clustering algorithm to group together similar sequences of signals (using Euclidean distance as distance metric).
3. Finally the results of the hierarchical clustering are visualized to the researchers using a dendrogram that should then determine the right number of clusters. The label assigned to each cluster is determined by the labels assigned in the first step to some of the IDs.

This model has many limitations, mainly in the first step. The authors of the work have tested their algorithm against three different logs of CAN messages that do not represent a generic modern vehicle (a simulated dataset with just 10 different IDs; an old tractor; logs from a vehicle positioned on a treadmill in a lab to simulate a real drive). In a real situation, there are too many changes in the bits of the CAN data frames during every action (many of them changes randomly even when the vehicle is still and no action is performed). Anyway, this work shows that machine learning models can be applied with good results to the sequences of packets recorded from the CAN bus.

2.4.1 READ: Reverse Engineering of Automotive Data Frames

The READ algorithm [28], developed by Marchetti and Stabili, processes raw CAN logs to find the boundaries of the different signals enclosed in each CAN packet. Moreover, it can recognize physical signals and label some data blocks inside the packets which are used by the ECUs as control mechanisms. These results are obtained without any prior knowledge about the different signals inside the sequences of packets in input and then it can be applied with no extra work to any vehicle.

The algorithm analyzes the packets sent in the CAN bus as an ordered sequence of payloads grouped by the ID of the correspondent packets and then observes the evolution of their bits over time. The raw logs of sniffed packets from the CAN bus are a sequence of data frames ordered by timestamps (which is added from the computer that records them), so they have to be grouped by their IDs in different sequences (still ordered by timestamps). The different sequences can be then processed by the algorithm in parallel, the processing of one sequence is not influenced by the others.

The boundaries of the signals and the labels are assigned using two types of metadata calculated for each sequence (identified by an ID) at the beginning: the *bit-flip rate* and the *magnitude* arrays with a length equal to the number of bits in each sequence (determined by the value of the DLC field).

The bit-flip rate is calculated by counting the number of bit-flips that occur in the sequence (how many times each bit change from 0 to 1 and vice-versa in consecutive packets) and then dividing these values by the number of data frames that compose the sequence.

The magnitude array is then obtained from the bit-flip rate array applying the equation Equation 2.1 to any bit-flip b.

$$magnitude(b) = \lceil \log_{10}(b) \rceil \tag{2.1}$$

Then the algorithm is divided into two phases. It uses the magnitude array to find a preliminary list of boundaries between the bits positions in the sequences and, in the second phase, it uses the bit-flip rates to find the final boundaries and to assign a label to each of them. The overall process of the READ algorithm is represented in Figure 5.

The first phase of the algorithm identifies the preliminary boundaries using only the magnitude array of each sequence. A preliminary boundary is identified by a drop in the magnitude of two consecutive bit positions, this is done because this drop of the magnitude shows that a less significant bit of a signal (with a high magnitude because it changes many times during the change of values of the signal) is followed by the most significant bit of another signal (which will change much less) or by a constant / multi-value signal.

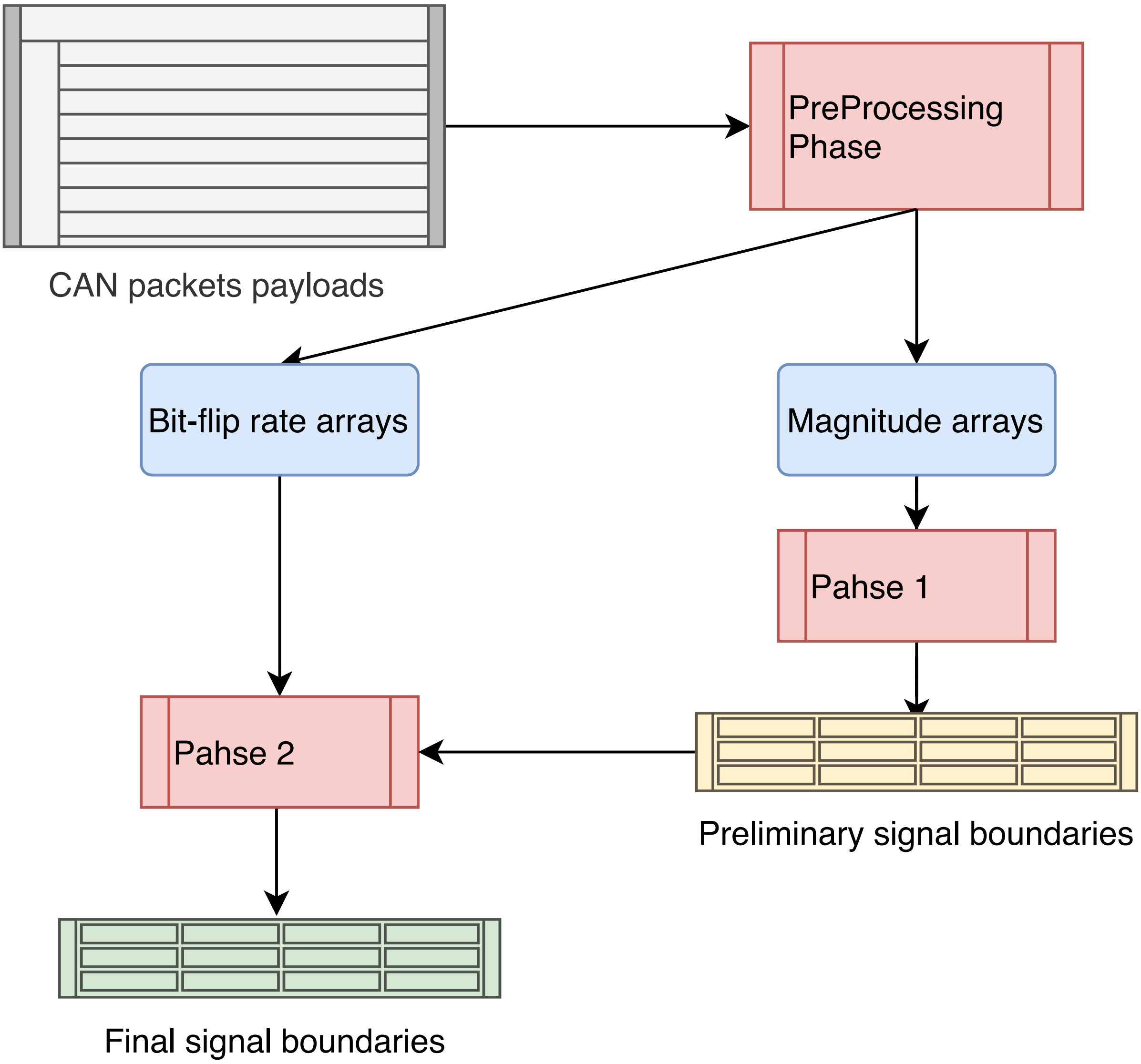


Figure 5: Phases of the READ algorithm

In the second phase, the algorithm looks for two types of information that manufacturers include inside the payload of the packets to implements a simple security mechanism against basic attacks: counters and CRCs.

- Counters are, as the name suggests, counters that identify the order of the messages with the same ID, they help the nodes to order the packets and identify retransmissions. They are identified by a magnitude of the bit position corresponding to the least significant bit equals to zero and a bit-flip rate that doubles at each position from the most significant bit to the least one.
- CRCs are error detecting messages used to find errors in the bit transmitted in safety-related frames. These signal have the same goals of the CRC field in the CAN packet structure 2.2 but they are not the same thing: they coexist and the one that is included in the payload is computed following proprietary (and secret) rules, on the other hand, the CRC field uses a check that is calculated following the public standard. Marchetti and Stabili found from empirical studies that a CRC signal is identified by a magnitude equal to 0 for each bit position and the bit-flip rate of its bits follows a Gaussian distribution centered in 0.5.

By applying these empirical results the READ algorithm can find and label counters and CRCs, after removing these signals it outputs the final boundaries of the physical signals.

2.4.2 LibreCAN: Automated CAN Message Translator

The approach adopted by LibreCAN [29] is not a simple analysis of CAN messages but it is a structured design of how to collect data from the vehicles during the experiments and how to use all these data in the following analysis.

The approach uses a different source of data:

1. raw logs of CAN packets.
2. information collected using the OBD-II PIDs.
3. Inertial Measurement Unit (IMU) data collected using the sensors of a gyroscope in a smartphone

The first step of this approach uses a slightly modified version of the READ algorithm (2.4.1) to extract the different signals from the data frames.

In the first phase of the process, they run an xcorr (normalized cross-correlation) for each signal from those collected from the IMU sensors and OBD-II PIDs with all the signals extracted from the CAN bus. The CAN signals with a high correlation value (they decided a threshold that maximizes precision and recall) are labeled as kinematic-related signals in the CAN bus. Moreover, for those signals with a very high correlation value, it is possible to match the CAN signal to the one from the smartphone or the PIDs and then we can find the offset and the multiplier of the signal (as mentioned in section 2.2.4, the CAN signals do not encode absolute values but a signal with a linear relation to the real value determined by an *offset* and a *multiplier*).

The second phase aims to find the signals of body-related events (e.g. opening/closing of doors and windows, heating). To recognize these signals, a 3-stage filtering process is applied to snippets of the recorded data from the CAN bus (where the researchers do actions to trigger each different event).

The three stages consist of:

1. filtering out from the event snippet all the sequences of messages with the same ID where the bits remained constant during the event.
2. filtering out all the messages with ID and payload that match a couple of ID-payload of the messages present in a snippet of CAN data recorded when no body-related event was taking place.
3. filtering out all the signals found during the first phase (that should be related to kinematic signals).

This work shows the most recent approach to reverse engineering of CAN signals and to the labeling each different ID with the corresponding signal.

This work, unlike READ, cannot be applied directly to raw CAN logs but it needs to collect data from different sources (OBD-II PIDs and IMU data) and it requires more complicated experiments to collect different snippets for each body-related event. Moreover, the paper is not clear enough on some steps and it does not explain how the labeling of the real kinematic signals is done because:

- OBD-II PIDs contain only a very small fraction of the kinematic signals of the CAN bus

- it is not specified how the IMU data from the gyroscope of a smartphone are processed and used and how can they be used to label a specific signal of the CAN bus.

2.5 Goals and Challenges

In this section, we want to explain which are the main goals of our work and how we want to overcome the state-of-art shortcomings

2.5.1 Goals

Our main goal is to improve the tools and modern approaches to the reverse engineering of CAN communications. We want to suggest to the researches a better approach that can reduce the time needed to manually reverse engineer the internal architecture of their vehicles.

We want to achieve this goal without asking them to perform long and tedious experiments and without the use of external pieces of hardware required to collect additional data (e.g. external IMU sensors).

Our approach suggests using a trained machine learning model that just needs as input the logs of the CAN bus of the vehicles. In this way the collection of these logs can be done during a normal driving session, the data can be collected by common operators that do not have specific knowledge and old CAN logs can be analyzed without repeating the experiments in particular conditions.

The suggested model uses a statistical algorithm (slight modification of [28]) to preprocess the packets recorded from the internal network of the vehicle and a supervised machine learning model to label the IDs of some important signals flows in the vehicles internal network. Doing this, we do not need special equipment for collecting the data and need the input needed includes only the raw logs of the CAN packets.

2.5.2 Challenges

The security by obscurity paradigm of modern vehicles makes difficult to reverse engineer the signals encoded in the data frames of a generic vehicle. This process, without the information provided by the DBC files from the vendor, is very long and it should be done for every different vehicle because different vehicles have different architectures and implementations.

The biggest challenge we face is that we have to create a dataset to train and evaluate our model without the use of any prior information, we do not have the DBC files of the vehicles used in our experiments.

Moreover, our goal is to understand if a machine learning classifier can be trained using a set of vehicles and then used to classify signals in different vehicles with logs recorded in different conditions. This is a very tricky step, in the other works that we have presented([29], [26], [27]) the data used to label the CAN signals was collected from the same vehicle of the CAN signals and during the same driving experiment. In our work, we have to overcome this limitation because our model needs to be as generic as possible and we want the researcher to use a pre-trained model to classify the signals in their vehicles with no extra effort.

CHAPTER 3

APPROACH

3.1 Introduction

Our objective is to create a model that directly analyzes CAN logs. In this section, we describe our research and the steps that are necessary for the creation of such a model (see Figure 6).

We start by explaining how to collect the dataset to train and test the model. The dataset is created by collecting only raw logs (formatted appropriately), which means a mere record of CAN packets sniffed from the bus.

Following the creation of the dataset, we describe how and which algorithms we have used to extract the different signals from the list of messages (each message can have multiple signals inside) and how we have cleaned the data from useless metadata (CRCs and counters).

Removing those, we remain with a time series for each experiment and each signal. We preprocess these time series and then use a part of them to train our model.

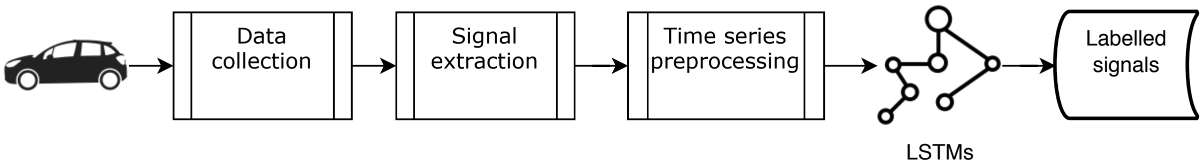


Figure 6: Overall architecture

Finally, we describe the machine learning architecture used for our model and discuss the evaluation process.

The steps to create the machine learning classifier are:

1. Dataset creation
2. Dataset exploration [30] and Manual reverse engineering of the signals
3. Identification of different signals inside the data frames
4. Preprocessing
5. Training of the deep learning model
6. Evaluation

3.2 Dataset creation

To train and evaluate our model we need logs of data frames from different vehicles and vendors. It is important to have very different vehicles in the dataset to be able to train a model that can be as general as possible and not fit on one particular vehicle or manufacturer. Moreover, it is necessary to test the model against a test-set that can be considered as generic as possible. One of the main problems of other researches in this area is the lack of available vehicles for experiments and evaluations.

As we have discussed in section 2.4, some previous work did not have reliable data for their experiments and that led to problems in the evaluation phase.

Even in the lucky case, in which researchers have a partnership with a specific vendor for the use of DBC files, it can be a problem to evaluate the work accurately because the evaluation is usually done using only vehicles of the same vendors [29] and this does not allow to understand how their results will translate in different settings.

To create a good dataset that can be considered generic and representative of a variety of different vehicle it is necessary to collect data from different vehicles, produced from different manufacturers. This is necessary because similar vehicles from the same manufacturers will probably have some components in common and this will lead to having a bias of the machine learning model that will learn only how that specific set of components represents the signals.

The input data of our model is composed of recorded sequences of the CAN packets that flow in the CAN bus network during a generic driving session. Since we want a model that can be used from the researcher on data collected during a generic experiment and we do not want to force them to follow a particular track for the collection of their data, we collect the data used to train and test the model with many city/highway driving sessions, in different conditions and with different tracks.

As we will explain in detail in the next sections and similarly to what is done in the READ algorithm [28], our model analyzes how the bits change from one message to the next one. This process can have slightly different results depending on the driving conditions, then we also repeat the data collection from some of our vehicles doing more than one driving session with each of them.

During the driving, we sniff all the data frames that flow in the internal network of the vehicle. The data that is recorded from the CAN bus contains a sequence of packets and the timestamps of our computer when it received each packet.

Our dataset can be represented as a table in which each row is a recorded CAN packet and the columns are:

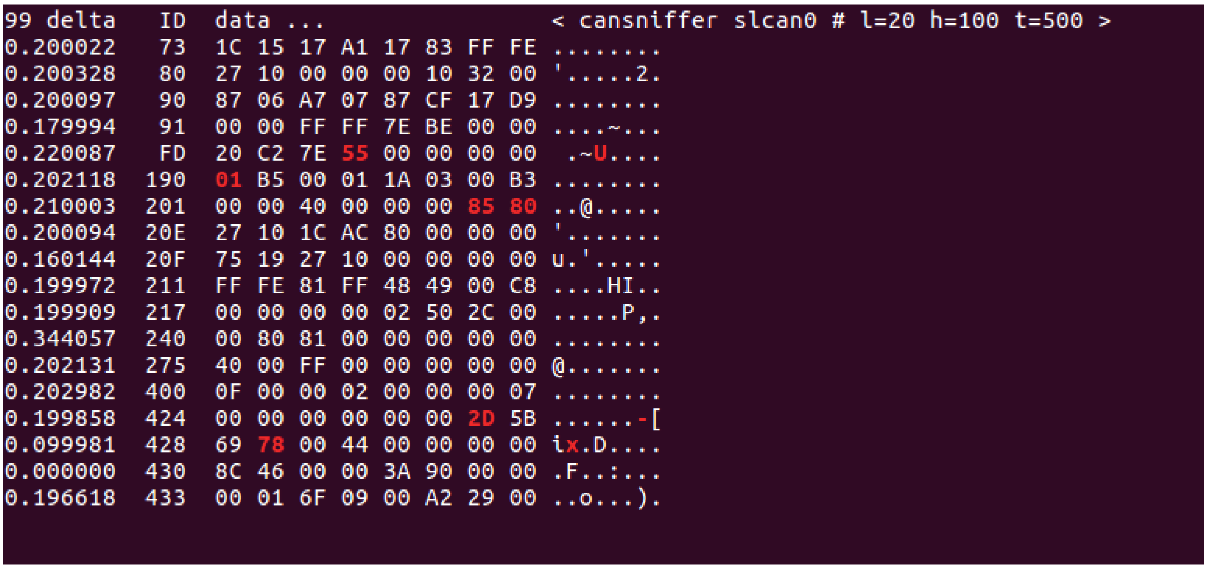
- Vehicle
- Driving session in which the packets has been recorded
- ID of the packet
- CAN bus line (if we have found more than one CAN bus line in the vehicle)
- Timestamp
- Length (in bytes) of the data frame's payload
- Actual payload

To make the data easier to store and access, the dataset is divided into different tables (saved in csv format) grouping the messages by vehicle and by driving session.

3.3 Dataset exploration and Manual reverse engineering of the signals

Data exploration is always the first step in data science, it is the preliminary exploration of the dataset which is done to identify the most relevant characteristics of the collected data. It helps to understand the right preprocessing steps and it exploits the ability of the researchers of recognizing patterns not captured by automatic algorithms.

During the driving sessions, we visualize how bits change in each frame (identified by an ID) in real-time (see Figure 7) and take note of those that change accordingly with the movements of the vehicle. We now have a set of possible signals, which are identified by the couple (ID,



```
99 delta    ID  data ...                < cansniffer slcan0 # l=20 h=100 t=500 >
0.200022    73  1C 15 17 A1 17 83 FF FE ........
0.200328    80  27 10 00 00 00 10 32 00 '.....2.
0.200097    90  87 06 A7 07 87 CF 17 D9 ........
0.179994    91  00 00 FF FF 7E BE 00 00 ....~...
0.220087    FD  20 C2 7E 55 00 00 00 00  .~U....
0.202118   190  01 B5 00 01 1A 03 00 B3 ........
0.210003   201  00 00 40 00 00 00 85 80 ..@.....
0.200094   20E  27 10 1C AC 80 00 00 00 '.......
0.160144   20F  75 19 27 10 00 00 00 00 u.'.....
0.199972   211  FF FE 81 FF 48 49 00 C8 ....HI..
0.199909   217  00 00 00 00 02 50 2C 00 .....P,.
0.344057   240  00 80 81 00 00 00 00 00 ........
0.202131   275  40 00 FF 00 00 00 00 00 @.......
0.202982   400  0F 00 00 02 00 00 00 07 ........
0.199858   424  00 00 00 00 00 00 2D 5B ......-[
0.099981   428  69 78 00 44 00 00 00 00 ix.D....
0.000000   430  8C 46 00 00 3A 90 00 00 .F..:...
0.196618   433  00 01 6F 09 00 A2 29 00 ..o...).
```

Figure 7: Screenshot of cansniffer, highlighted in red the bits that have changed

bit positions), that seems related to the physical signals that we were looking for (speed of the vehicle and RPM)

Find the signal of the steering wheel angle is much easier, we just turn on the vehicle (without turning on the engine) and we move the steering wheel: we plot all the bits changing during this experiment (which are only a fraction of those that change during a real drive with the engine turned on) and we take notes of those IDs where the bits change with behavior that follows our movements.

To confirm our manual reverse engineering done during the driving sessions, we then plot the signals found and check our hypothesis by exploring the set of signals and the relationship between them. For example in Figure 8 it is shown the plot of:

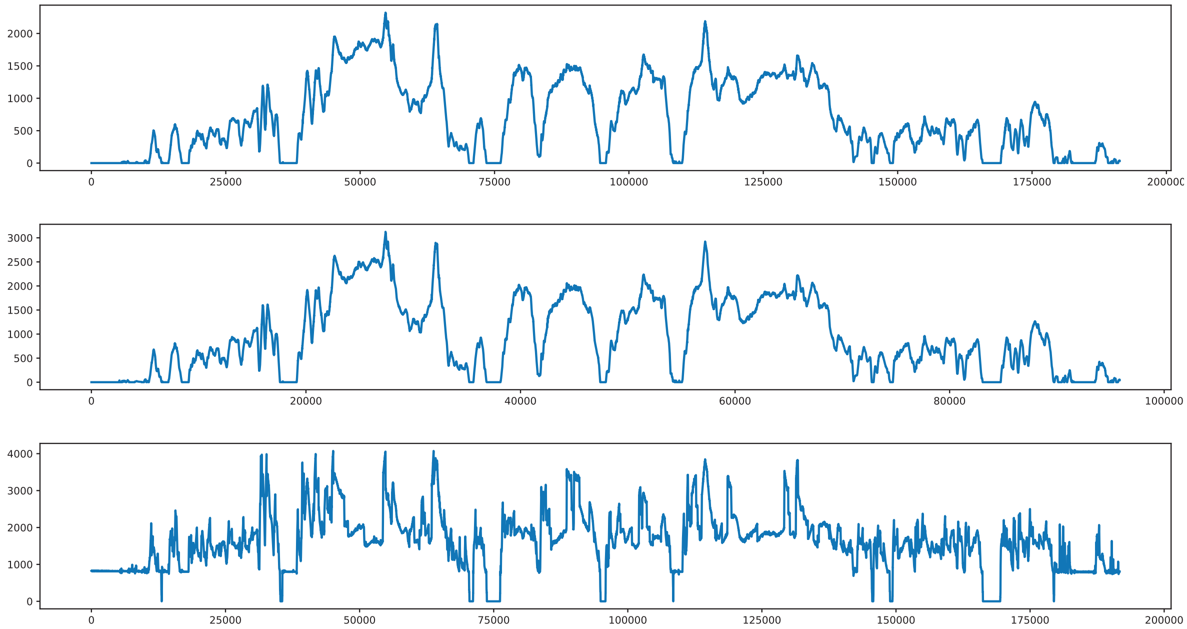


Figure 8: Plotting of speed signals and RPM of the Alfa Romeo Giulia in the first experiment

1. the 4 signals of the speed of the wheels
2. the signal of the speed of the vehicle (which has a frequency lower than the others)
3. the signal of the RPM

After manual reverse engineering, we can confirm that some trucks do not have a signal for the steering wheel angle in the CAN bus.

3.4 Identification of different signals inside the data frames

In this section, we explain how to obtain the different signals recorded by the ECUs of a vehicle starting from the packets sniffed during the driving experiments.

The log of CAN messages, for each driving of a vehicle, is the sequence of sniffed packets ordered by their timestamps. To record the packets we use candump. The recorded CAN packets look like:

```
(1580823978.993305) can0 546 [8] 8F CE 80 00 00 00 00 00

(1580823978.993499) can0 0DE [6] 1B A1 97 D0 08 F8

(1580823978.993747) can0 0FC [8] 0C E8 C0 02 00 3E 8F 4B

(1580823978.993873) can0 2FA [3] 10 05 63

(1580823978.994167) can0 5A5 [8] 7F FC 00 00 00 00 40 00

(1580823978.994346) can0 41A [7] C6 CF 74 90 A2 84 E0

(1580823978.994907) can0 0EE [8] 00 00 00 00 00 00 0A D8
```

Starting from these sequences we then group the packets by their ID in smaller sub-sequences.

As discussed in section 2.2, in each packet are usually encoded information regarding more than one single signal. It is then necessary to divide the sequences of packets with the same ID in traces that correspond to the same signal, separating the sequence of bits of the payloads in sub-blocks (one for each signal encoded in a packet).

Looking at how the bits change in the data frames it is possible to analyze the internal structure and try to divide the different signals that are contained in a unique packet. This is suggested by Marchetti and Stabili[28] and our procedure is a slightly modified version of the READ algorithm that we discuss in detail in [30].

To chose how to group the bits in a trace we calculate two arrays of values: BitFlip and Magnitude. Each element `i`th of the arrays is the value calculated with the bits in the `i`th position in the data frame. We calculate the bitflip b as the count of times in which a bit change value from one packet to the next one divided by the number of packets in the trace. The magnitude is calculated with the equation Equation 3.1 (it differs from the original one because of the use of *floor* instead of *ceiling*):

$$magnitude(b) = \begin{cases} -\infty & b \leq 0 \\ \lfloor \log_{10}(b) \rfloor & \text{otherwise} \end{cases} \tag{3.1}$$

We use magnitudes and bitflips to find the start and end of blocks of bits that contain different signals in each data frame.

To separate these blocks, the traces go through a two-step process.

In the first step, we produce a set of preliminary boundaries for the data-block using as delimiter those positions where the bitflip is `0` (when we find a sequence of one or more bits that has never changed value it is should be a signal separator and constant signals are dropped). Then we look at the array of magnitudes: when we find a drop in the value of the magnitude we put a delimiter there because these drops are the consequences of the presence of a less significant bit of one signal that is followed by the most significant one of another signal.

The blocks found so far are still not clean signals: they can include extra metadata because of some naïve safety mechanism implemented by vendors like CRCs and counters, described in detail in section 2.4.1.

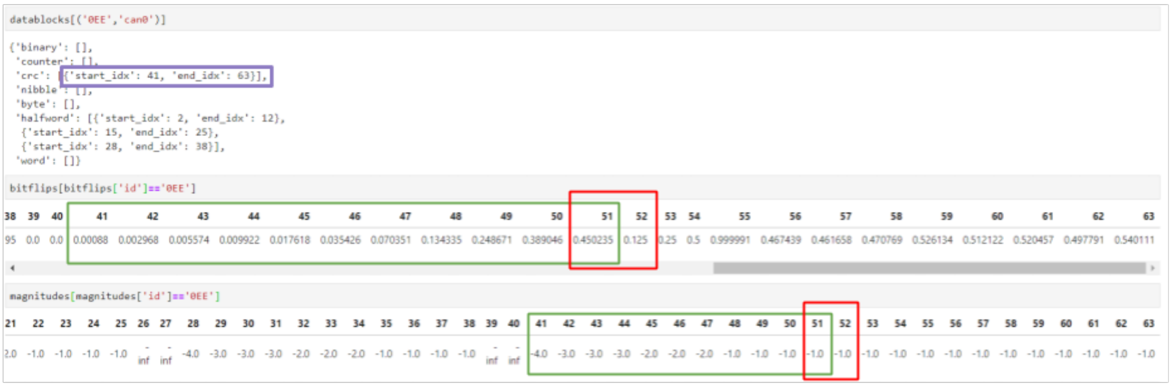
In the second step, we analyze the preliminary delimiters of blocks that we have found so far and find and drop the sequences of bits that are recognized as CRCs or counters. A sequence of bits is considered a counter if the bitflip rate double at each position (going from the most to the least significant bit) and it is considered a CRCs if the magnitudes of all the bits positions are 0s and the bitflip rates follows a Gaussian distribution with the center in 0.5.

The sequences of resulting blocks of bits are those that represent a physical signal and that we analyze in the following sections.

With respect to the original READ algorithm we have used *floor* function instead of the *ceiling* one because this makes us able to separate those bitflips that change constantly. However, a deeper study on the heuristic used to split the payloads of the packets in separate blocks could find a more accurate function.

An evident example of this inaccuracy of the magnitude function has been found in the recognition of the 4 signals of the wheel speed in the Alfa Romeo Giulia. By exploring the signals in the Giulia we have found that the 4 signals of the speed of the wheels are in the packets with ID `0EE` and the blocks should be (the start and end bits change be slightly different because some bits can be constant just because in our experiments we have not reached a speed high enough):

```
{start-idx: 2, end-idx: 12}
{start-idx: 15, end-idx: 25}
{start-idx: 28, end-idx: 38}
```



```
datablocks[('0EE','can0')]
```

```
{'binary': [],
 'counter': [],
 'crc': [{'start_idx': 41, 'end_idx': 63}],
 'nibble': [],
 'byte': [],
 'halfword': [{'start_idx': 2, 'end_idx': 12},
  {'start_idx': 15, 'end_idx': 25},
  {'start_idx': 28, 'end_idx': 38}],
 'word': []}
```

```
bitflips[bitflips['id']=='0EE']
```

38	39	40	41	42	43	44	45	46	47	48	49	50
95	0.0	0.0	0.00088	0.002968	0.005574	0.009922	0.017618	0.035426	0.070351	0.134335	0.248671	0.389046

51	52	53	54	55	56	57	58	59	60	61	62	63
0.450235	0.125	0.25	0.5	0.999991	0.467439	0.461658	0.470769	0.526134	0.512122	0.520457	0.497791	0.540111

```
magnitudes[magnitudes['id']=='0EE']
```

21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
2.0	-1.0	-1.0	-1.0	-1.0	-inf	-inf	-4.0	-3.0	-3.0	-3.0	-2.0	-2.0	-2.0	-1.0	-1.0	-1.0	-1.0	-inf	-inf

41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
-4.0	-3.0	-3.0	-3.0	-2.0	-2.0	-2.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0

Figure 9: Example of Magnitudes and Bitflips of ID 0EE Alfa Romeo Giulia

```
{start-idx: 41, end-idx: 51}
```

But the algorithm can find only the first three because the magnitude does not change between the bit position 51 and the following one. Even if the bitflip rate of the bit in position 52 is lower the difference is not enough to have a drop in the magnitudes. This is shown in Figure 9.

3.4.1 Two's Complement

The time series that compose our dataset, that we have after the identification and separation of data blocks in the different packets, are identified by ID-CanLine-Variable-Vehicle-Experiment (where "variable" indicate the specific data block between those identified in the previous step). As they are, some of these time series, are not ready to be processed by the deep learning model and need a further processing step.

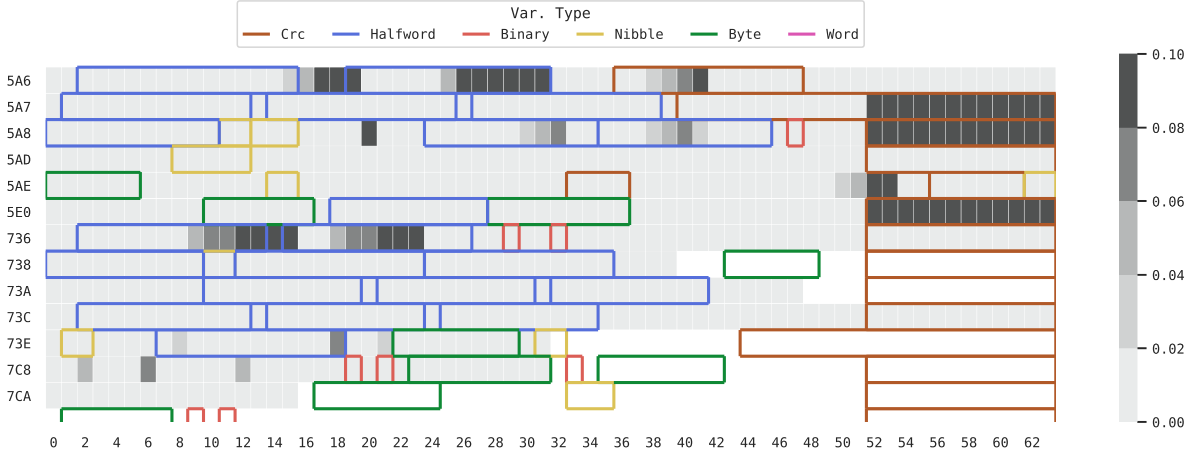


Figure 10: Heatmap representation where each cell value represents the magnitude of that bit-flip rate in that bit position, the signals identified are highlighted.

The time series that we retrieve from the CAN bus are binary sequences, the values encoded in these sequences are, usually, unsigned integers but that is not always the case.

As the first step, we convert our binary values (interpreted as unsigned-int) into decimal integers.

During our experiments, we have observed that some signals are encoded using Two's complement, in particular those signals with a physical value which is zero centered (e.g. the steering angle position). In these cases, we observe different changes in the sign of the values along with the time series. Those time series values (in two's complement and zero centered) after the conversion of their values from binary to decimal integer have a behavior similar to square waves because of the sign changes: the negative values are converted in very high positive numbers (depending on the length of the binary number) because the last bit is `1` and the

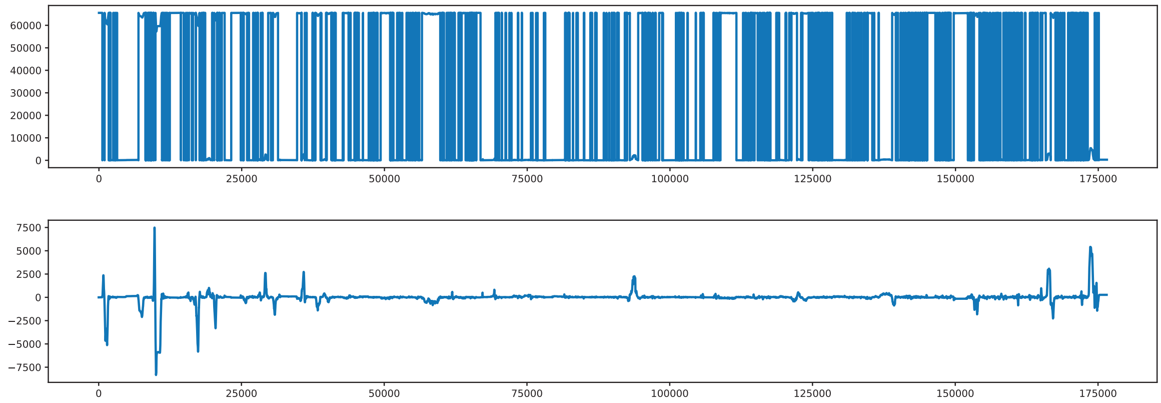


Figure 11: Plots of the signal of the steering wheel angle of the Opel Corsa decoded as Unsigned integer and then as a Two's complement number

positive number are converted in much smaller numbers because the most significant bit is `0`. We need to find and convert properly these time series because the square waves are a wrong representation and the original behavior of the signal is not correctly represented.

We look for two's complement time series by searching for changes in sign in the binary number. To do so we count how many times the first x bits of a time series changes from all `1`s to all `0`s, if the counter exceeds a certain threshold (for us it was 5) then the sequences are recognized as a two's complement encoding and it converted from binary to integer in the right way.

Figure 11 shows the plots of the *steering angle position* signal of the Opel Corsa, the first plot is the signal interpreted as unsigned integers and the same signal but with the correct decoding.

3.5 Time series preprocessing

Our model has to find patterns in the behavior of the time series that we have produced in our experiments.

Exploring our dataset we noticed that a fair number of time series and in particular those that seem directly related to the movements of the vehicles have long sequences of `0`s, these sequences were probably caused in the moments in which the car was still or even when the engine was turned off. The frequency of packets is very high (in particular for those sensors that send information related to the driving). These sequences are a great problem for the classification, the neural network labels them incorrectly because of the absence of any pattern in them. So we remove those sequences from the time series in the dataset.

The time series, as they are now, have two problems caused by their length. A single time series has the data of an entire experiment with the vehicle, which is a relatively long drive where many different behaviors have been registered (e.g. city driving followed by a highway driving) and different traffic conditions along the same drive can make difficult the learning process of the neural networks. To avoid this problem we split our long time series into smaller pieces, in this way we have signals that were captured during a smaller period of time and so it is more likely that the driving conditions were more uniform along that time. Splitting the time series in smaller sequences is also the first step to mitigate the problem of having a strongly imbalanced dataset, we will talk about this problem soon. Moreover, using shorter input sequences in our neural network makes the training phase much faster and it helps a lot the model during the training process.

3.5.1 Deal with imbalanced data

In the CAN bus, there are usually more than 100 signals (and the number increase as the technology improves) and our goal is to find a specific signal among all the others. In our dataset, we have just a sample of each signal for each driving experiment.

Learning with an imbalanced dataset is a common problem in lots of different applications of machine learning. The imbalanced data problem refers to those situations where the positive examples (those of a particular signal that we are searching) are outnumbered by the negative ones [31]. If a machine learning model is trained with a great number of negative examples will just learn to predict always negative because it is the easiest way to minimize the error in predictions. To deal with this problem we try to make our training dataset more balanced with two common techniques: over-sampling (with SMOTE) and under-sampling (see Figure 12).

SMOTE (Synthetic Minority Over-sampling Technique) [32] is a particular over-sampling technique that generates new synthetic examples of the minority class, in our case the signals that we want to recognize. Common over-sampling techniques consist of just train the model using as input a dataset with more copies of some examples of the minority class, in this way the model cannot reduce the error predicting only the majority class. On the other hand, SMOTE creates new samples and does not repeat the existing ones. To do so, SMOTE takes k random nearest neighbors (in our case k=5) of the samples in the minority class and then the synthetic samples are picked by choosing points in the segments that join the samples in those K-NNs.

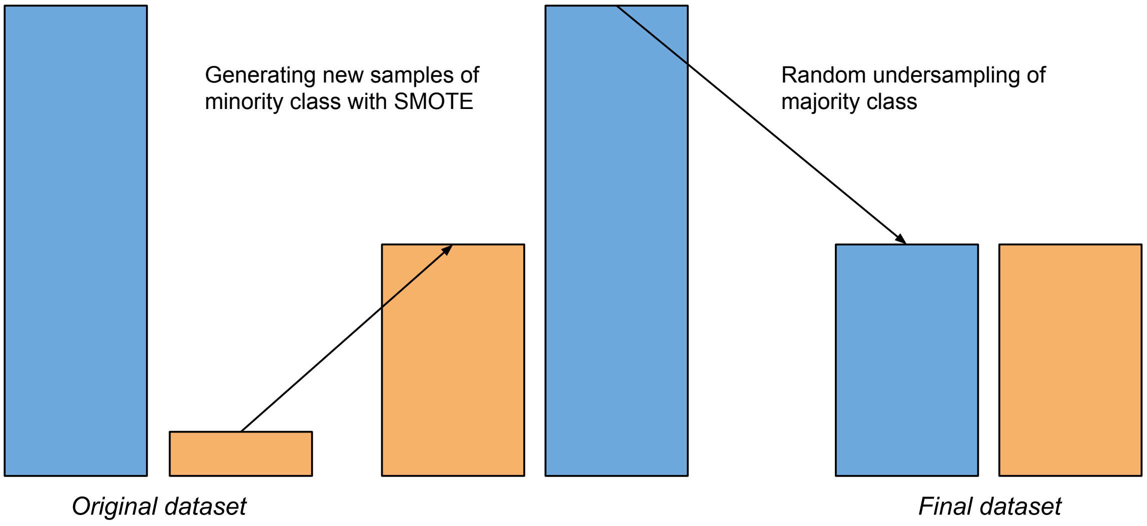

Generating new samples of minority class with SMOTE
Random undersampling of majority class
Original dataset
Final dataset

Figure 12: Deal with imbalanced data of the training set.

After over-sampling we apply a random under sampler to the majority class (all the other signals in the dataset). As suggested by Chawla et all [32] the use of sub-sampling after SMOTE improves the performances of the classifier. Our dataset is strongly unbalanced, then we use both techniques to increment the ratio of the samples in the minority class over the samples in the majority class but we decide not to reach a balanced dataset because this leads to overfitting too much with SMOTE on the signals of our experiments. To avoid over-fitting we apply SMOTE to reach a ratio (number of minority class samples over the number of majority class samples) of 0.25 and then under-sampling the majority class to reach a ratio of 0.4.

3.6 Deep Learning Model

Our main goal is to find an easy and quick solution to find a specif car signal among all the others. We want to achieve this by just looking at the time series without any prior knowledge on the specific car and external data (DBCs or data recorded during the experiment by other devices like in [29] or [21]). We have preprocessed CAN logs to produce a dataset of time series and we want to classify them.

A common and difficult to beat approach is to use 1-nearest neighbor (NN) [33] using Dynamic Time Warping (DTW) as distance function [34]. But unfortunately, this approach can't create a generalized model for different vehicles (as we explained, all the vehicles have a different and secret representation of data in their networks) and for data collected in different ways (different tracks and driving behaviors).

We use a deep learning algorithm that can be trained using a group of different vehicles. The resulting model can be then used to classify the same signals in other vehicles receiving as input only the logs of the CAN bus messages (preprocessed as we have described in the previous chapters).

Deep neural networks [35] are now commonly used for classification problems and Recurrent Neural Networkss (RNNs) (they are based on the work of Rumelhart [36]) can handle time series data thanks to internal loops in their architecture that are used as a sort of internal memory. The internal memory of RNNs is used to take into account the behavior of the data in input along with the different time instants. We use Long Short-Term memorys (LSTMs) [37] which are a particular evolution of the common RNN architecture that can deal with the vanishing and exploding gradient problems [38]. LSTMs are particularly suited for time series classification ([39], [40]), moreover Saleh et al. [41] already applied them in the automotive area for the classification of driving behavior with good results.

We use one LSTM network for each signal that we want to find because we prefer to have different binary classification problems rather than a single multi-class classification problem than can be easily created with an one-versus-all (OVA) or all-versus-all (AVA) approach. The reason behind this choice are:

- in a multi-class classification problem, errors in one of the underlying binary classifiers can compromise the entire prediction [31]
- we prefer to have results with different confidences for each signal. This work aims to give to researchers help in the reverse engineering process so we think that it's better to give them all the results for each signal because even false positive or signals that are predicted as the negative class but with a result that is just a bit lower of the threshold can be useful (e.g. if the classifier of the RPM finds another signal that behaves similarly then this information can be useful to understand what signal it is). Moreover, it is easier to analyze the results of many binary classifiers rather than the results of a multi-class one.

To classify the three signals that we have labeled in our dataset, we train three different LSTMs (see Figure 13).

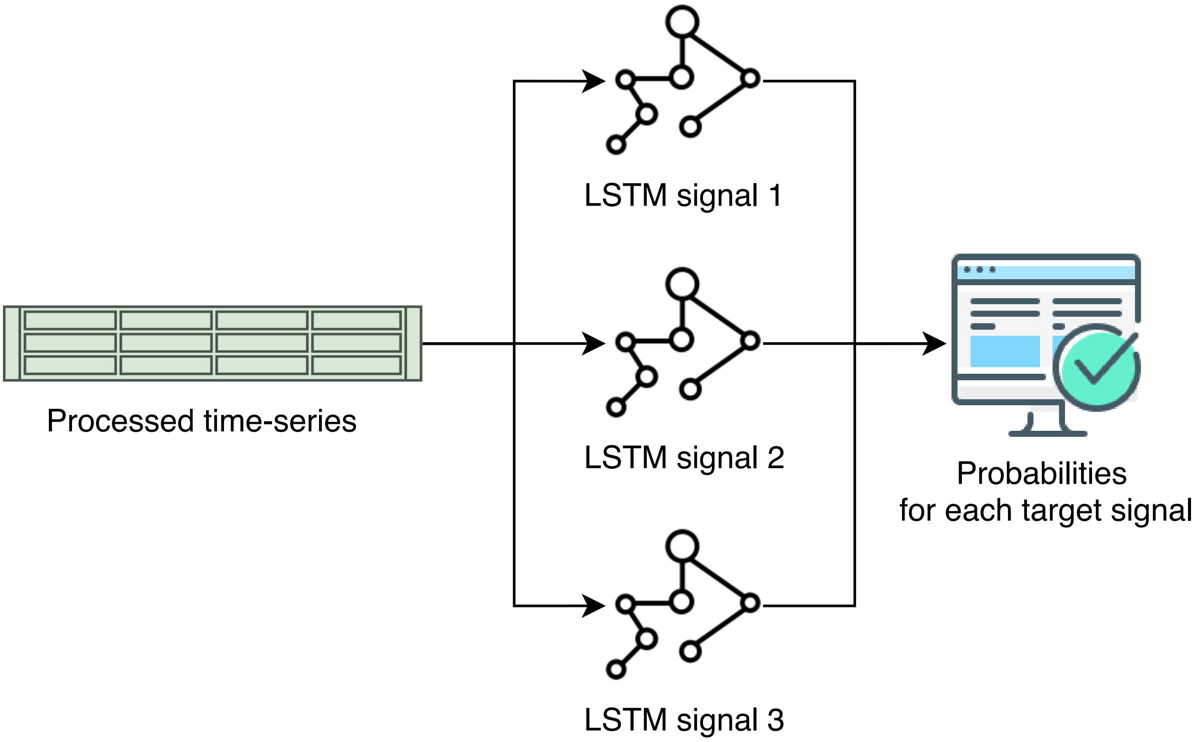


Figure 13: LSTM classifiers

We feed each LSTM binary-classifier with all the processed time series that represent our CAN signals. Each classifier gave us, in output for each input signal, the probability that the signal is the target signal that the classifier is trained to classify.

The final output that we want to obtain from our model is a subset of the set of CAN signals, identified from the recorded messages of a vehicle, for each target signal that our model can classify. These subsets should contain a small number of signals (compared to the original dataset) that the researchers will manual inspect to find the right signals.

To obtain the subset, for each target signal, we select the signals to which the correspondent LSTM has assigned a high probability. By default, the threshold to select the signals is 0.5 but this can be then lowered by the researchers if during the manual inspection it is not possible to find the signals they were looking for.

In our preprocessing step we have split the time series in smaller sequences and then our LSTMs will return a probability for each sub-sequence. The final label assigned to each original signal is obtained by merging all the labels assigned by the model to the corresponding sub-sequences with the majority voting rule.

3.7 Evaluation

For the evaluation of the model's results we split the initial dataset in a training-set and a test-set using different vehicles for the two sets.

The vehicles that we used:

- are very different from each other (we have commercial trucks of very different sizes and functionalities, a city car and a sporty car).

- have been produced by different vendors.
- have produced data for the experiment in different driving settings.

Splitting the data collected with these vehicles make us confident that the results that we obtain can be generalized to other vehicles and in other experimental settings.

We take into consideration different metrics to evaluate the results of our model. In our problem we cannot rely on accuracy (Equation Equation 3.2).

$$accuracy = \frac{number\ of\ samples\ classified\ correctly}{total\ number\ of\ samples} \tag{3.2}$$

Our goal is to find few signals among more than 100 signals that flow in the car internal network, this means that if our model will predict every input as the negative class (which include almost all the signals in the dataset) the accuracy will be very close to 1 even in those cases where it is unable to find our signals.

First of all, it is necessary to look at the confusion matrix, Table II, which represents all the possible outcomes of a classification problem. Using the values of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) we can introduce other metrics that suits well our problem: Precision (Equation Equation 3.3), Recall (Equation Equation 3.4), False Positive Rate and True Positive Rate.

$$Precision = \frac{TP}{TP + FP} \tag{3.3}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.4}$$

		Actual Class	
		Target Signal	Other Signal
Predicted Class	Target Signal	True Positive	False Positive
	Other Signal	False Negative	True Negative

TABLE II: Confusion Matrix.

Precision measures how many samples, between those retrieved by the model, are of the class that we were looking for. *Recall*, on the other hand, measures how many samples of the target class are classified correctly.

Finding exactly only the target signals is almost impossible, we already explained the reasons that make signals of different vehicles so different but we have also many signals of the same vehicle that are highly correlated with each other and this makes it difficult for a machine learning classifier to distinguish them.

Considering that we cannot reach the perfect classifier and that we will have misclassifications, we decide to focus on the Recall metric and to reduce those signals of the target class that are not classified correctly. Then we prefer to have a higher number of False Positive rather than a high number of False Negative.

This model can be used to select a small group of signals (TP+FP), among the entire CAN log, that has to be explored [30] that researches can then analyze to find the target signals. If we have a high number of FN this cannot be done because the researchers will always need to look to the entire dataset because many target signals will be part of the False Negative group.

CHAPTER 4

EXPERIMENTAL VALIDATION

4.1 Goals

With our work, we aim to show that:

1. our model can find the target signals used to train the model. We focus on the Recall value and we care less about Precision and False Positive Rate.
2. the procedure to collect the necessary data that has to be given as input to our tool is simple and does not require great effort and prior knowledge.
3. the tool is relatively quick in the classification (we don't care much about the time required for the training phase) and help researchers reducing the time required in the reverse engineering process.

The first goal is the one that arises the greatest difficulties in the evaluation. Unfortunately, we miss ground truth, we don't have the DBCs files of the vehicles that we used to collect our data and this means that we cannot translate the entire set of signals in the CAN bus. So we can validate our model only against the signals that we have manually reverse engineered, we are very confident of those signals but unfortunately, we cannot understand the label of the other signals. That means that we cannot derive any conclusion about the False Positive signals (we can't say how much they are related to the original signal that we were searching).

In modern vehicles, many signals are directly correlated like the pedals positions and the actions triggered by that (change in the gas flow, brakes, etc..) but without the ground truth, we cannot include these types of considerations in our evaluation. Moreover, even the algorithm [28] that we used in the identification is not perfect, for example, we found that it was not able to find all the counters and it has not recognized some important signals, and even here the DBCs files would have been precious.

4.2 Dataset

The creation of this dataset has two main goals:

1. to provide us a dataset to train and test our model and that contains vehicles with very different characteristics.
2. to provide to future researches an additional dataset to evaluate their models.

We made this dataset publicly available [30].

The dataset is composed of CAN bus data logged during different driving conditions. We collect logs from five vehicles, produced by five different vendors. The vehicles have a very different design (and we then expect that also the behavior and design of their ECUs is not too similar): two of them are cars and the other three are commercial trucks:

- Alfa Romeo Giulia
- Opel Corsa
- Mitsubishi Fuso Canter
- Isuzu M55

- Piaggio Porter Maxi

Table III and Table IV describe the different vehicles and test done.

For training and evaluating our model we use only the data from the tests that include a driving session (see the description column in Table IV).

The dataset is composed of data extracted from five different vehicles but, for the two cars, we have more than one driving session. This has been done because, as it is evident from the column "IDs" in Table IV), in each session the sniffed packets can be different because not all the events have been triggered every time. Moreover, repeating the driving sessions can lead also to some slightly different results in the identification of the boundaries of each signal in the CAN packets: to recognize these boundaries we analyze bitflips and magnitudes of each bit position, then the results of that process can have small variations in very different driving sessions. Unfortunately, doing different driving tests was not possible with the three trucks.

We have monitored each session to explore the recorded signals and double-check our manual reverse engineering comparing the signals with the events and the path traveled (in every instant of time). To record the sessions we have used an app installed in a smartphone[42], Figure 14 shows the track followed during the last driving session and Figure 14 shows the statistics of that session.

To train our dataset we have manually reverse engineered three signals (vehicle speed, engine speed, and steering wheel angle) and then, for each vehicle and each signal, we have recorded in the dataset:

- ID of the packet that contains the signal

ID	Vehicle	Type
`C-1`	Alfa Romeo Giulia Veloce	Car
`C-2`	Opel Corsa	Car
`T-1`	Mitsubishi Fuso Canter	Commercial Truck
`T-2`	ISUZU M55	Commercial Truck
`T-3`	Piaggio Porter Maxi	Commercial Truck

TABLE III: Dataset composition according to vehicle type

- Start bit of the data-block of the signal
- End bit of the data-block of the signal

4.3 Experimental Setup

4.3.1 Hardware

To collect the data from the different vehicles (see Table III) we use common tools and open-source software. The necessary pieces of hardware to connect a laptop to the vehicles are a *CANtact* board (Figure 16) and a standard *CAN connector* (Figure 4). A *ELM327* interface Figure 17 and the "OBD Auto Doctor" software have been used to collect the OBD-II PIDs that we used as help during the manual reverse engineering phase.

The ***CANtact*** [43] is a board that works as an interface between the CAN bus and a computer. It connects to the computer using the USB port and to the CAN bus using the standard OBD-II port or by direct wire access to the bus. The CANtact is an open-source project and both the firmware and hardware are publicly available, it is a low cost and cross-platform and these are the reason that makes it the perfect choice for these experiments.

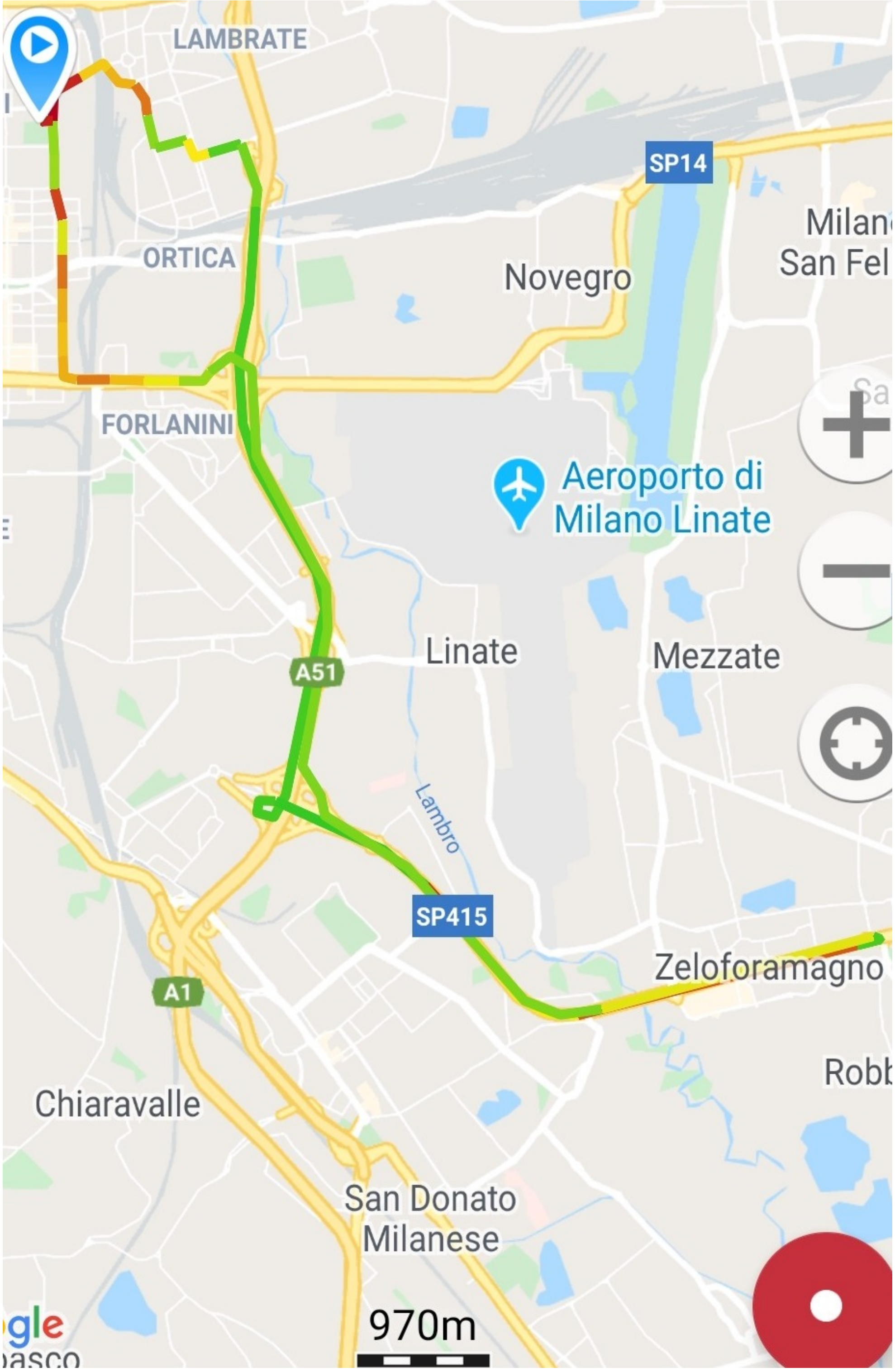
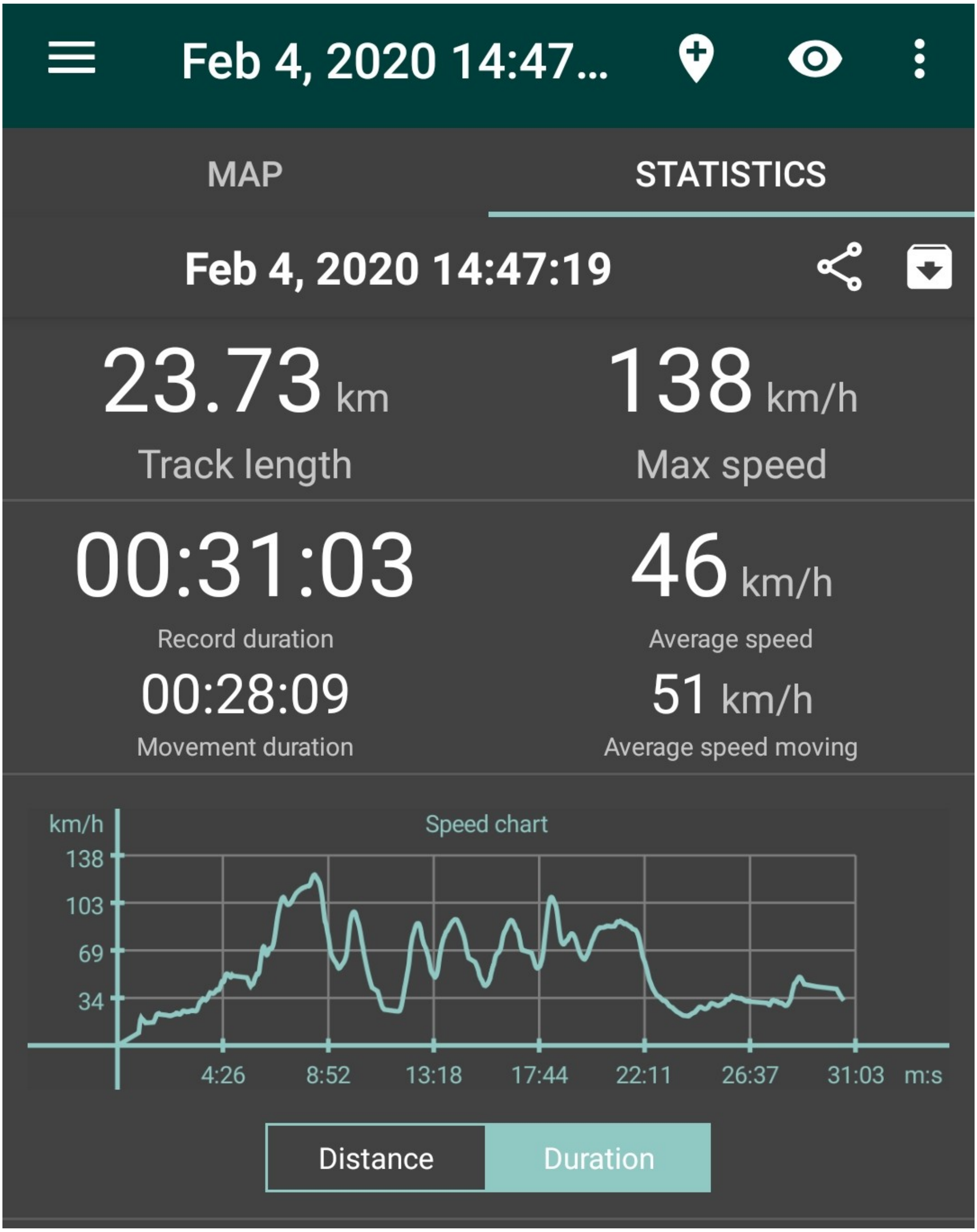


Figure 14: Recorded path travelled in the last driving session of the Alfa Romeo Giulia



Feb 4, 2020 14:47...

Feb 4, 2020 14:47:19

23.73 km
Track length

138 km/h
Max speed

00:31:03
Record duration

46 km/h
Average speed

00:28:09
Movement duration

51 km/h
Average speed moving

Figure 15: Recorded path travelled in the last driving session of the Alfa Romeo Giulia

Vehicle	Experiment time			IDs	Frames	Description
	Date	Start	End			
`C-1`	2018-07-26	15:15:58	15:35:20	77	3,062,691	city driving
`C-1`	2018-07-26	15:46:13	15:48:32	76	364,863	city driving
`C-1`	2018-07-26	15:49:10	15:49:23	76	33,005	brake tests
`C-1`	2018-07-26	15:50:29	16:10:54	83	3,227,315	city driving
`C-1`	2018-07-26	16:10:57	16:20:16	83	1,473,625	city driving
`C-1`	2018-07-26	16:20:20	16:30:59	83	1,684,769	city driving
`C-1`	2018-07-26	16:53:17	17:10:31	83	2,723,484	city driving
`C-1`	2019-02-01	16:31:01	16:40:58	82	1,569,776	city driving
`C-1`	2019-02-01	15:18:55	16:30:36	88	10,942,747	city driving
`C-1`	2020-02-04	13:46:18	14:18:15	88	5,113,676	city driving
`C-2`	2019-10-02	08:54:16	09:22:40	78	3,467,855	city driving
`C-2`	2020-02-13	13:45:35	14:15:00	78	3,592,112	city driving
`T-1`	2019-02-20	16:04:06	16:35:04	31, 47*	1,798,602*	city driving
`T-2`	2019-11-08	14:51:57	15:07:43	22	498,721	city driving
`T-2`	2019-11-08	14:34:33	14:43:20	22	263,269	not moving
`T-3`	2019-11-08	11:48:56	12:14:58	23	1,729,623	city driving
`T-3`	2019-11-08	11:16:55	11:23:42	19	2,795,321	not moving 1
`T-3`	2019-11-08	12:57:48	13:42:52	23	2,795,321	not moving 2

* For this experiment, there are included both `can0` and `can1` lines.

TABLE IV: List of experiments per vehicle

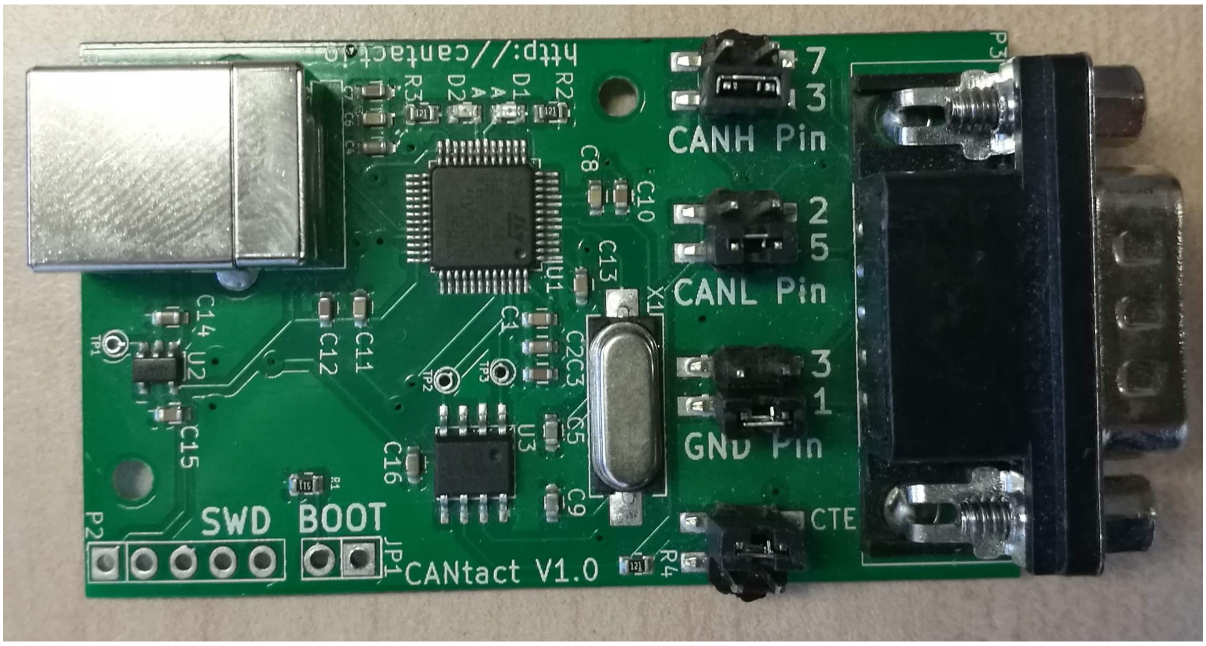


Figure 16: CANtact board

4.3.2 Software

The software used to connect to the CAN bus is part of the ***CAN Utils*** library [44] which is open-source and it is included in the Linux kernel. In particular, we used two tools of the library: candump and cansniffer.

As first step, it is necessary to create a network interface with the command `slcand [options] <tty> [CAN interface]`, we used the options `-o -c -s6`. The most important option is the last one that depends on the frequency of the bus in the vehicle (-s6 is for a frequency of 500kbit/s, which is the most common one).

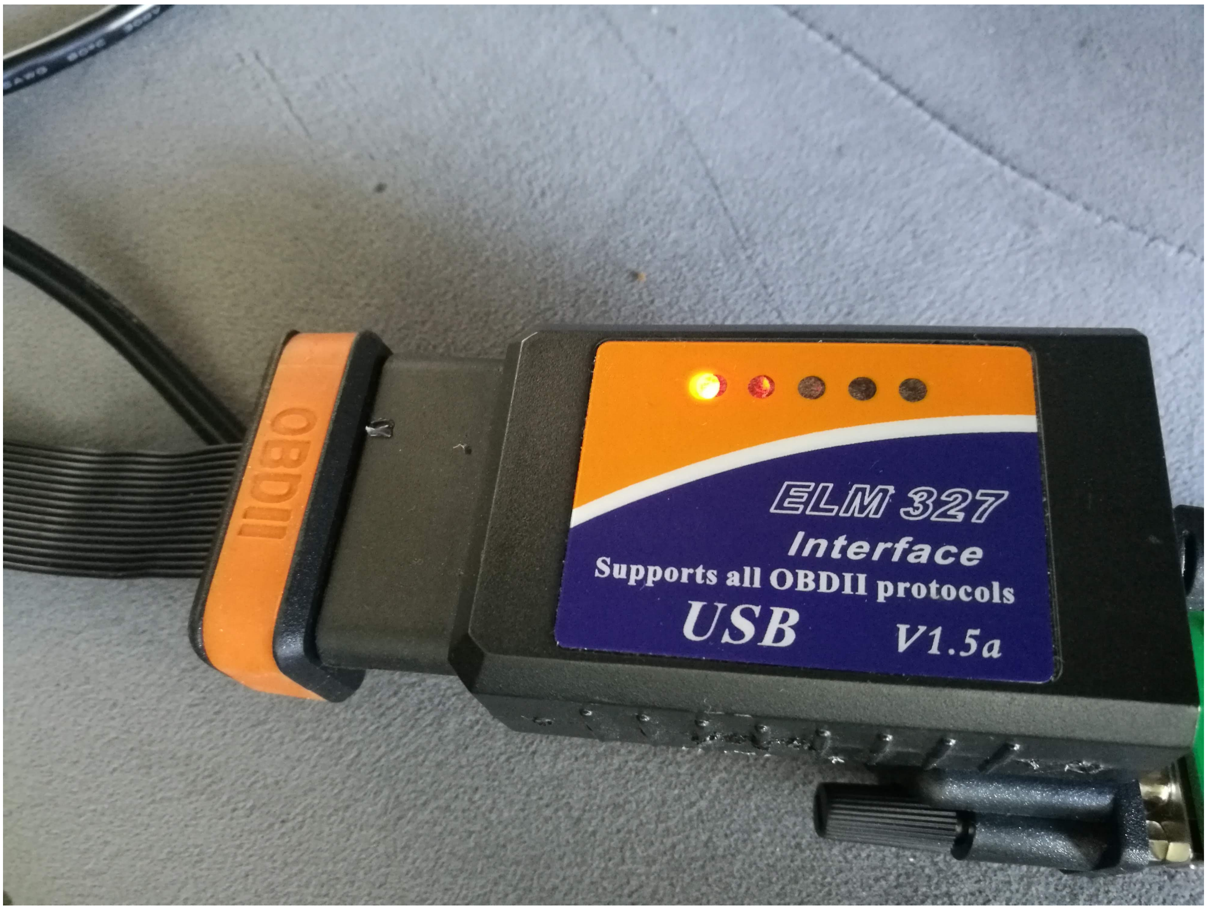


Figure 17: ELM 327

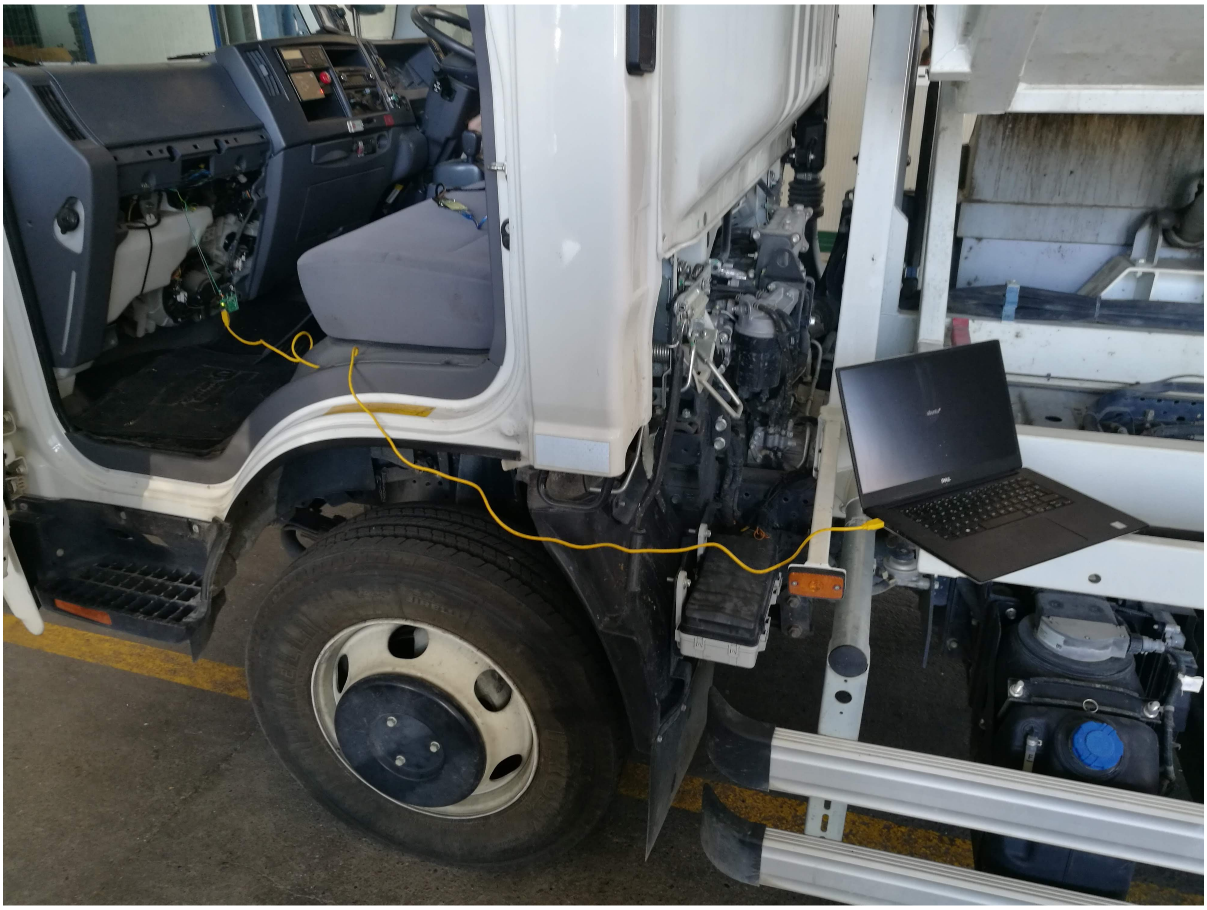


Figure 18: Connection to the vehicle

Candump is used to display and dump all the packets that flow in the bus. When the network interface is up (`ifconfig <CAN interface> up`) we can start logging the CAN packets with Candump with the command `candump -ta <CAN interface> > <output file>`. The option `-ta` is necessary to save the POSIX timestamp for each packet. A data frame collected in this way has the following structure:

```
(<timestamp> canLine ID [length in bytes] <actual data>
```

For example:

```
(1573208215.472159) can0 300 [8] 64 00 00 00 00 00 00 00
```

Cansniffer has been used as help during the manual reverse engineering of our target signals. We have stressed our vehicles in many experiments to recreate strange behavior in the signals that we wanted to find, during these experiments we used cansniffer with the command `cansniffer <Can interface> -c -B` that shows in real time only the data frames where the bits are changing. Doing this it was possible, repeating some action different times, to select groups of ID and bits that seem to change accordingly to our actions (and then exploring them looking at the logs of the packets).

4.4 Performance Evaluation

In this section, we want to present the results obtained with our model and analyze how this approach can help the researchers in comparison to the other approaches proposed in the state-of-art (see section 2.4).

4.4.1 Comparison with the related works

Up to now, the best results in reverse engineering of CAN packets have been obtained from the READ algorithm [28] and the approach proposed by LibreCAN [29].

The *READ* algorithm is the base of the phase in which we identify the different signals encoded in the CAN packets and we have proposed different analyses and graphical representations of the results of that phase in [30]. In our paper, we have implemented a framework that easily extrapolates the boundaries of the different signals and shows statistical information in a graphic representation.

In this work, starting from the results that are possible to obtain with an algorithm like READ, we want to enrich the information that can be extrapolated from the logs of CAN messages by labeling specific signals in the logs.

LibreCAN [29] is the only work in the state-of-art with a similar idea and approach. The first phase of the algorithm, like in our case, uses a modified version of READ to identify the boundaries of the signal starting from the raw logs.

A direct comparison between our work and the LibreCAN algorithm does not make much sense because the goals of the two works are different:

- we want to study the feasibility of a completely automatic model that can analyze directly raw logs of CAN messages and, from these, it can give the most complete analysis to researchers that can then use them in the reverse engineering process. The model that we propose, after being trained, will be then able to classify signals of even unknown vehicles.
- the LibreCAN paper does not aim to create a model that can directly classify CAN signal, it is a guide to a different approach to manual reverse engineering of CAN messages that explain to researchers how to carry out the data collection from each vehicle and how to set up the experiments. This approach leads to repeat every step for each vehicle that needs to be studied.

We do not take into consideration the work of Young et al. [27] in this section because of the evident limitations discussed in section 2.4.

4.4.2 Performance metrics

To understand in deep the results of our model we use different performance metrics.

First of all, it is important to understand that there are two possible kinds of binary classification problems[31]:

1. **X versus Y**, these are common classification problems like classifying a set of images as cats or dogs pictures. In this kind of situation, we usually want to maximize accuracy (Equation Equation 3.2).
2. **X versus not-X**, in this case, the classification problem is focused on spotting all the X-samples. In this kind of problem, the focus is not on the classification of the two classes

anymore, but the important thing is to find all the samples of one of the two classes. A good example of this problems are medical tests: if we are testing a patient for a virus infection we want to be sure that he has not the virus (even making some errors and saying that he has the virus when it is not true) more than classifying him as healthy when that is not true. Moreover, in this case, we usually have to deal with imbalanced data (when the samples of one class outnumber the samples in the other class) and accuracy is not a good metric anymore. We have to use other metrics like recall (Equation Equation 3.4) and/or precision (Equation Equation 3.3).

Our problem is an evident ”X versus not-X” case, we want to find a signal (which can be the speed of the vehicle or another physical signal present in the CAN bus) among all the possible signals that are registered by the ECUs of a vehicle.

Since is almost impossible to create a perfect classifier, we have to decide where to focus our attention and which is the metric that is more relevant to us.

Our objective is to give help to the researchers in the reverse engineering work but we cannot replace them, the human intervention at the end is still needed and then our results are just a further help for the humans to make the process easier.

Then we have to take into consideration the four possible outcomes of our classification process, well described in the confusion matrix represented in Table II, that is trying to find a target signal among the entire set of signals that flows in the CAN bus. Then the possible outcomes are:

1. True Positive (TP): the signal is classified correctly as the signal that we were looking for

2. True Negative (TN): the signal is correctly recognized as not one of the target signals
3. False Positive (FP): the signal is classified incorrectly as one of the target signals when it is not
4. False Negative (FN): the signal is not classified as the target signals when it is

We want to be able to select a smaller set of possible signals from which the researchers have to pick the correct signal that they are looking for, to reduce the effort needed.

In this perspective, our goal is to filter as many signals as possible from the set of possible signals where we are looking for a specific signal but without removing accidentally the target signal because in this case the researchers will not find the target signal among the selected set and they will need to analyze all the possible signals (and this means that our tool has not made the process faster).

We should, therefore, focus on having a very low number of false negative outcomes and this is well captured by the **recall** metric. After that, we evaluate our model using false positive rate (see Equation Equation 4.1) and the *F1 score* to understand how much can we reduce the set of possible signals (the number of false positive outcomes extends the time needed to the researchers to find the right signal between those selected by the model).

$$FalsePositiveRate = \frac{FP}{FP + TN} \tag{4.1}$$

The **F1 score** is a popular metric to visualize in a unique number the quality of the solution by leveraging recall and precision (this is also called the *balanced f-score*) and it is defined as described in Equation Equation 4.2. The values of the F1 score in our tests are low because this metric leverages recall and precision (and not the false positive rate). Both *Precision and False Positive Rate* are metrics that measure the quality of a model looking at the number of false positive samples. The difference is that the former compares them to the number of true positives and the latter to the number of true negatives. In our case, the classification will always have a very small number of true positive samples for each signal (in many cases only one signal) and the majority of the cases will be true negative samples.

Moreover, we want to measure how much the model can decrease the number of signals that the researchers have to inspect manually. Therefore, we want to compare the number of misclassified signals, that they have to eliminate from the final set given by our model, with the number of all signals that they should have eliminated from the entire set of signals without applying our model.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{4.2}$$

4.4.3 Results analysis

We have tested our model with a cross-validation approach: we have done three different tests, separating training and testing sets each time in different ways. To avoid over-fitting, we have grouped the data by vehicle and used, every time, complementary sets of vehicles to train and then test the model. In this way, we are sure that the input signals of the training phase do not have some correlation with those in the testing set.

Test	Training Set	Testing Set
Test 1: Speed-RPM	C1,T1,T2	C2,T3
Test 1: Steering Angle	C1,C2	T3
Test 2: Speed-RPM	C2,T1,T3	C1,T2
Test 2: Steering Angle	C1,T3	C2
Test 3: Speed-RPM	C1,T1,T3	C2,T2
Test 3: Steering Angle	C2,T3	C1

TABLE V: Training and testing sets for each test.

Training and testing sets for each test are described in Table VI.

The classifiers for the *speed* and *RPM* signals have been trained with the same sets of signals in each test. For the classifier of the *steering angle position* signal a different choice has been made because this signal was not present in the CAN bus of the Mitsubishi Fuso Canter (T1) and of the Isuzu M55 (T2).

4.4.3.1 **Speed signal classifier**

The outcomes of the speed signal classifier for each test are shown in Table VI.

Except for the last test, all the speed signals have been correctly classified and we have only a few false positive samples (10 on average). The third test is the one that had the worst results, in this case, we had 5 false negative samples.

Since, in most cases, researchers know if a signal should be present in the CAN bus when they do not find the target signal in the set found by the model they should lower the threshold used to classify the signals. We have then reported the results obtained in the same test but lowering the threshold.

		Actual Class	
		Target Signal	Other Signal
Predicted Class Test 1	Target Signal	9	7
	Other Signal	0	134
Predicted Class Test 2	Target Signal	7	24
	Other Signal	0	178
Predicted Class Test 3	Target Signal	1	1
	Other Signal	5	179
Predicted Class Test 3*	Target Signal	5	15
	Other Signal	1	165

* The results of Test 3 but lowering the threshold of classification.

TABLE VI: Confusion Matrix of the outcomes of the Speed Signal classification

Test	Recall	F1 score	False Positive Rate
Test 1	1.0	0.72	4.96%
Test 2	1.0	0.37	11.88%
Test 3	0.17	0.25	0.55%
Test 3*	0.83	0.38	8.33%

* The results of Test 3 but lowering the threshold of classification.

TABLE VII: Metrics of the Speed Signal Classifier

Inspecting the misclassified signals, we found that some of them are very easy to detect through human inspection (some of them are just counters that were not recognized from the READ algorithm and others have a very strange behavior which cannot be mistaken for the signal of the speed) and then the selection of the right signals in the subset proposed by the model should be easy and fast.

The metrics for this classifier in each test are summarized in Table VII.

		Actual Class	
		Target Signal	Other Signal
Predicted Class Test 1	Target Signal	2	7
	Other Signal	0	141
Predicted Class Test 2	Target Signal	2	50
	Other Signal	0	157
Predicted Class Test 3	Target Signal	2	25
	Other Signal	0	159

TABLE VIII: Confusion Matrix of the outcomes of the RPM Signal classification

Test	Recall	F1 score	False Positive Rate
Test 1	1.0	0.36	4.96%
Test 2	1.0	0.07	24.14%
Test 3	1.0	0.138	13.58%

TABLE IX: Metrics of the RPM Signal Classifier

4.4.3.2 RPM signal classifier

The outcomes of the RPM signal classifier for each test are shown in Table VIII.

The results for this signal were pretty good and, in all the tests, we have no false negative samples.

The metrics for this classifier in each test are summarized in Table IX.

4.4.3.3 Steering wheel angle signal classifier

The outcomes of the Steering Wheel Angle signal classifier for each test are shown in Table X.

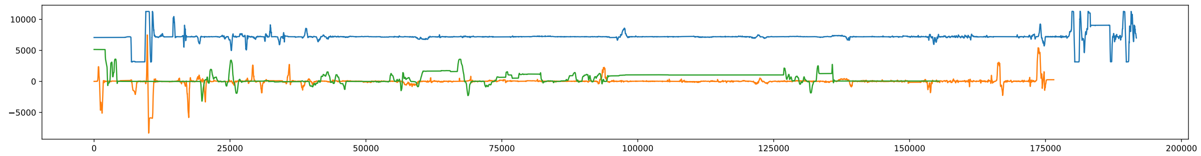


Figure 19: Steering wheel signals of Alfa Romeo Giulia (blue signal), Opel Corsa (orange signal) and Piaggio Porter Maxi (green signal).

As we have just explained, these tests were a bit different because we do not have this signal on all the vehicles. Then the sets used to train and test the model are smaller.

In Figure 19 we have plotted the three steering signals in the three vehicles that we have. From the plots, it is evident that in the Piaggio Porter Maxi (T3) and the Opel Cosa(C2) the internal representation of the ”Steering wheel angle” is almost the same.

The model is then able to recognize easily the signals (and with just a very small number of false positive) of the two vehicles which use a similar representation of the signal. But it has problems recognizing the signal in the Alfa Romeo Giulia (C1): in the third test, in order to reach a recall of 1, it was necessary to set a lower threshold and we have reached a false positive rate of 30% in that case. This difficulty is due to the limited number of vehicles available for this signal. The same model trained with a greater variety of vehicles would have performed better.

The metrics for this classifier in each test are summarized in Table XI.

		Actual Class	
		Target Signal	Other Signal
Predicted Class Test 1	Target Signal	1	2
	Other Signal	0	33
Predicted Class Test 2	Target Signal	1	5
	Other Signal	0	108
Predicted Class Test 3	Target Signal	0	2
	Other Signal	1	134
Predicted Class Test 3*	Target Signal	1	41
	Other Signal	0	95

[*] The results of Test 3 but lowering the threshold of classification.

TABLE X: Confusion Matrix of the outcomes of the Steering wheel angle Signal classification

Test	Recall	F1 score	False Positive Rate
Test 1	1.0	0.5	5.74%
Test 2	1.0	0.286	4.42%
Test 3	0.0	0.0	1.47%
Test 3*	1.0	0.05	30.14%

[*] The results of Test 3 but lowering the threshold of classification.

TABLE XI: Metrics of the Steering wheel angle Signal Classifier

4.4.4 Final considerations on the results

These results show that our model can be of great help to the researchers in the classification of the signals of the CAN bus because it can, from the entire set of signals that flows in the internal network of a vehicle, select a much smaller subset of candidates and then reduce the time needed for this work in their research.

We have applied this model just to the signals that we have found manually but we have found no reasons to not extend, in the future, this model with other signals (hopefully with the help of the DBCs files of the vehicles used for the training phase).

The results obtained are also limited by the number of only five vehicles used in the experiment (three in the case of the steering signal) which cannot be used all together to train the model to avoid over-fitting and then invalidate the results of the testing phase.

Moreover, we have worked mixing cars and trucks which have a very different internal network: in two of the trucks one of the target signals is not even present and the number of IDs (as shown in Table IV) and number of signals in the trucks is much lower than in the cars (which have much more computing units in their network).

Thanks to this we have shown that a generalized model can be obtained even working with very different vehicles but, probably, even better results can be obtained training and then using the model with vehicles more similar to each other.

4.5 Choice of the vehicles to train the model

In this section, we want to analyze the results obtained training the model with different sets of vehicles to understand how the choice of the training set impacts the ability of the model to classify the signals.

The results change a lot depending on the differences between the vehicles in the training set and the vehicles in the test set.

A great example of this is the case of the classification of the Steering Wheel angle signal (see section 4.4.3.3). The results for that signal are very high for the first two tests: in those tests, in the training sets, there was a vehicle where the signal was represented in a similar way to the one in the test set. In the third case, instead, the results were lower because of the great difference between the vehicles in the training and testing sets.

We have then done various experiments combining all possible training sets for the classification of the Speed and RPM signals, without considering the possibility of lowering the threshold of classification. The results are summarized in Figure 20 for the Speed signal and in Figure 21 for the RPM signal.

We have 5 vehicles, then, we can train the model with 1, 2, 3 or 4 vehicles. For each number of vehicles used in the training phase, we have experimented with each possible combination of the vehicles chosen for the training.

The final results are obtained by averaging the values of each metric obtained by testing the models with different vehicles grouping the tests by the number of vehicles in the training sets.

As we can see in the graphs reported in Figure 20 and Figure 21, as expected, the higher the number of vehicles in the training set the better the results obtained from the model.

We have reached the best results training the model with all the vehicles but the one used to test the model, but it is important to note that the results vary a lot for each choice of the training set. As we have seen in the previous section, the number of false positive can rise a lot or it can be necessary to lower the classification threshold if none of the vehicles used in the training set has a signal that is encoded similarly as the one that we want to classify.

We have chosen to use very different vehicles for our work (trucks and cars with very different characteristics) to put ourselves in the worst-case scenario and prove that the model can be used as help even in these cases. But to have the best results it is very important to train the model with vehicles as similar as possible to those to which we want to apply the model.

Another important factor is the number of driving sessions done with each vehicle used to train the model. As we can see in Table VII and Table IX, the tests with the best results are those in which we have the Alfa Romeo Giulia in the training sets. That is related to the much higher number of driving sessions done with that vehicle, this helps the model to do not focus on particular driving behaviors and generalize better.

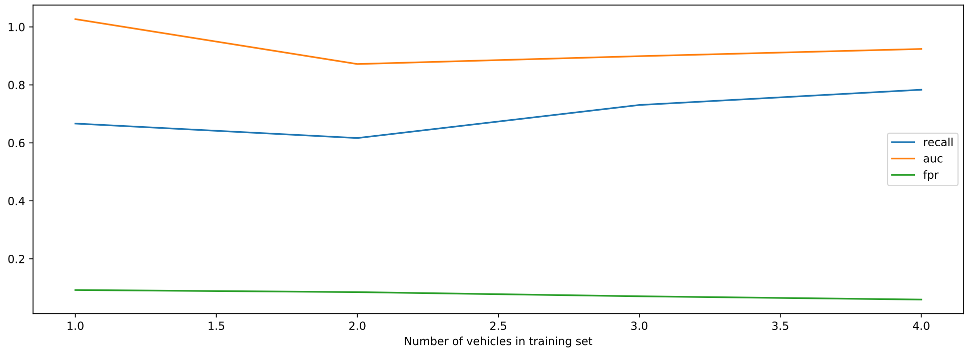


Figure 20: Results of tests on different vehicles with models trained with an increasing numbers of vehicles - Speed Signal

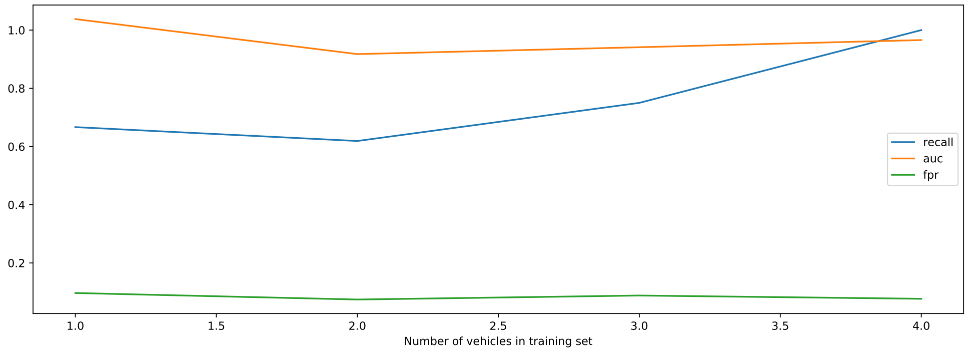


Figure 21: Results of tests on different vehicles with models trained with an increasing numbers of vehicles - RPM Signal

CHAPTER 5

LIMITATIONS & FUTURE WORKS

5.1 Limitations

The lack of available DBC files of the vehicles under study, which are industrial secrets, led to the main limitations of this work:

1. We have a limited set of signals that we were able to reverse engineer manually from all the vehicles (speed signal, engine rotation speed, steering wheel angle). With the DBCs files that describe the locations of all the signals in the CAN packets it would have been possible to train and test our model with many other different signals.
2. The lack of ground truth leads us to be less precise in the construction of the dataset. Our ground truth has been obtained by manual reverse engineering of the signals in our vehicles. Although we tried to be as precise as possible, there's still room for human error and no solution to double-check its correctness.
3. Without a translating table for the entire set of packets, it is impossible to understand the reasons behind misclassifications (both false positives and false negatives). With a complete ground truth, it would have been possible to derive more conclusions (for example it would have been possible to understand what are the relations between the target signals and those that are incorrectly classified).

5.2 Future Works

Our main limitation can be overcome in future work with a partnership with a vehicle manufacturer that provides new vehicles with the corresponding DBCs files. With the translation tables, we will be able to study the misclassifications and improve the model for those cases. Moreover, it will be possible to research other signals that this model can classify.

During the identification of different signals inside the data frames we use a slightly modified version of the READ [28] algorithm, this uses the magnitude function to find the boundaries between different features. This function has been chosen empirically and it sometimes fails in finding the correct boundaries or even finding different results in different driving logs of the same vehicle. A deeper study is necessary to investigate if a better function can be used to split the variables inside a single frame.

CHAPTER 6

CONCLUSIONS

Security researchers have started to study the vulnerabilities of in-vehicle networks but their work is made difficult by the security-by-obscurity paradigm adopted by the automotive industry.

Since researchers often do not have access to the complete specifications of their vehicles (described in the DBCs files), in this work, we have presented a solution to speed up the reverse engineering process of CAN packets.

At first, we collected a detailed dataset of CAN packets and made it publicly available [30] in order to help researchers in the future to train and test their works (which is particularly useful for machine learning models).

We then proposed to apply a machine learning classification model to the labeling of different CAN signals. Our main goal is to make the process simple and the model fast to use. To accomplish this, we proposed a model that can classify signals of a generic vehicle taking as input directly the logs of CAN packets sniffed during a normal driving session, without prior knowledge on the vehicle and without the need for further data collected with external devices during the driving sessions.

The proposed model processes directly the CAN packets collected from a vehicle, identifies the different signals encoded in them and then selects only a small subset of possible signals using LSTM networks where the researchers can find the target signal they were looking for.

Using a trained model like this it is possible to reduce the effort and time needed to reverse engineer the messages and signals of a vehicle without the proprietary specification of the in-vehicle network.

Future works may involve using DBC files to expand the set of possible target signals that the model can classify.

In conclusion, the main contribution of this work is a methodology that enables a fast and partially automated classification of signals from generic and unknown vehicles. It can be trained using labeled signals from different vehicles and then is able to classify signals of vehicles completely different from those used in the training phase without prior knowledge about them.

As far as we know, this is the first approach that produces a model that can analyze and label signals of unknown vehicles using as input only recorded packets from the CAN bus.

CITED LITERATURE

1. CAN in Automation: History of can technology.

2. Miller, C. and Valasek, C.: Remote exploitation of an unaltered passenger vehicle. Black Hat USA, 2015:91, 2015.

3. Wikipedia contributors: Security through obscurity — Wikipedia, the free encyclopedia, 2020. [Online; accessed 24-March-2020].

4. Scarfone, K., Jansen, W., and Tracy, M.: Guide to general server security. NIST Special Publication, 800(s 123), 2008.

5. Wikipedia contributors: Secure by design — Wikipedia, the free encyclopedia, 2019. [Online; accessed 24-March-2020].

6. Wikipedia contributors: White hat (computer security) — Wikipedia, the free encyclopedia, 2020. [Online; accessed 24-March-2020].

7. Robert Bosch GmbH: Can specification, 1991.

8. Miller, C. and Valasek, C.: Adventures in automotive networks and control units. Def Con, 21:260–264, 2013.

9. ISO, I.: 11898-2, road vehicles controller area network (can) part 2: High-speed medium access unit. International Organization for Standardization, 2003.

10. Robert Bosch GmbH: Can with flexible data-rate specification version 1.0, 2012.

11. Standard, I.: 11898-3: Road vehicles–controller area network (can)–part 3: Low-speed, faulttolerant, medium-dependent interface. International Organization for Standardization (www. iso. org), 2006.

12. Wikipedia contributors: Can bus — Wikipedia, the free encyclopedia, 2020. [Online; accessed 11-March-2020].

CITED LITERATURE (continued)

13. ISO: Iso 14229-3:2012 road vehicles — unified diagnostic services (uds) — part 3: Unified diagnostic services on can implementation (udsoncan). International Organization for Standardization, 2012.

14. ISO: Iso 15765-2:2016 road vehicles — diagnostic communication over controller area network (docan) — part 2: Transport protocol and network layer services. International Organization for Standardization, 2016.

15. CSS Electronics: Can dbc file - convert data in real time (wireshark, j1939), 2020.

16. Standard, S.: Sae j1979: E/e diagnostic test modes. Vehicle EE Systems Diagnostics Standards Committee. SAE International, 2002.

17. Prasad, B., Tang, J.-J., and Luo, S.-J.: Design and implementation of sae j1939 vehicle diagnostics system. In 2019 IEEE International Conference on Computation, Communication and Engineering (ICCCE), pages 71–74. IEEE, 2019.

18. ELETRONICS, E.: Elm327 obd to rs232 interpreter. ELM Electronics Datasheets, 2015.

19. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., and Others: Comprehensive experimental analyses of automotive attack surfaces. In USENIX Security Symposium, volume 4, pages 447–462. San Francisco, 2011.

20. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Others: Experimental security analysis of a modern automobile. In 2010 IEEE Symposium on Security and Privacy, pages 447–462. IEEE, 2010.

21. Wen, H., Zhao, Q., Chen, Q. A., and Lin, Z.: Automated Cross-Platform Reverse Engineering of CAN Bus Commands From Mobile Apps.

22. Wikipedia contributors: Obd-ii pids — Wikipedia, the free encyclopedia, 2020. [Online; accessed 23-March-2020].

23. Miller, C. and Valasek, C.: A survey of remote automotive attack surfaces. black hat USA, 2014:94, 2014.

24. Jupp, E.: Stolen in seconds: keyless new cars that fail security tests, Aug 2019.

CITED LITERATURE (continued)

25. Nova, D. H.: Literature Review : Intrusion Detection Systems for CAN networks.

26. Verma, M., Bridges, R., and Hollifield, S.: ACTT: Automotive CAN tokenization and translation. In 2018 International Conference on Computational Science and Computational Intelligence (CSCI), pages 278–283. IEEE, 2018.

27. Young, C., Svoboda, J., and Zambreno, J.: Towards Reverse Engineering Controller Area Network Messages Using Machine Learning.

28. Marchetti, M. and Stabili, D.: READ: Reverse engineering of automotive data frames. IEEE Transactions on Information Forensics and Security, 14(4):1083–1097, 2018.

29. Pesé, M. D., Stacer, T., Campos, C. A., Newberry, E., Chen, D., and Shin, K. G.: LibreCAN: Automated CAN Message Translator. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 2283–2300, 2019.

30. Zago, M., Longari, S., Tricarico, A., Carminati, M., Pérez, M. G., Pérez, G. M., and Zanero, S.: ReCAN–Dataset for reverse engineering of Controller Area Networks. Data in brief, 29:105149, 2020.

31. III, H. D.: A Course in Machine Learning. 2017.

32. Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P.: Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research, 16:321–357, 2002.

33. Lines, J. and Bagnall, A.: Time series classification with ensembles of elastic distance measures. Data Mining and Knowledge Discovery, 29(3):565–592, 2015.

34. Bagnall, A., Lines, J., Bostrom, A., Large, J., and Keogh, E.: The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. Data Mining and Knowledge Discovery, 31(3):606–660, 2017.

35. Heaton, J.: Ian goodfellow, yoshua bengio, and aaron courville: Deep learning, 2018.

36. Rumelhart, D. E., Hinton, G. E., and Williams, R. J.: Learning representations by back-propagating errors. nature, 323(6088):533–536, 1986.

CITED LITERATURE (continued)

37. Hochreiter, S. and Schmidhuber, J.: Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

38. Pascanu, R., Mikolov, T., and Bengio, Y.: On the difficulty of training recurrent neural networks. In International conference on machine learning, pages 1310–1318, 2013.

39. Karim, F., Majumdar, S., Darabi, H., and Chen, S.: Lstm fully convolutional networks for time series classification. IEEE access, 6:1662–1669, 2017.

40. Huybrechts, T., Vanommeslaeghe, Y., Blontrock, D., Van Barel, G., and Hellinckx, P.: Automatic reverse engineering of CAN bus data using machine learning techniques. In International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, pages 751–761. Springer, 2017.

41. Saleh, K., Hossny, M., and Nahavandi, S.: Driving behavior classification based on sensor data fusion using lstm recurrent neural networks. In 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC), pages 1–6. IEEE, 2017.

42. Geo tracker - gps tracker.

43. Evenchick, E.: CANtact, Jan 2017.

44. Linux Kernel: Can utils, Feb 2018.

45. Fawaz, H. I., Forestier, G., Weber, J., Idoumghar, L., and Muller, P.-A.: Deep learning for time series classification: a review. Data Mining and Knowledge Discovery, 33(4):917–963, 2019.

46. Wikipedia contributors: On-board diagnostics — Wikipedia, the free encyclopedia, 2020. [Online; accessed 27-March-2020].

47. Tricarico, A.: A Long Short-Term Memory based approach for reverse engineering and classification of CAN signals. 2020.

VITA

NAME	Andrea Tricarico
EDUCATION	
	Master of Science in Computer Science, University of Illinois at Chicago, USA
	Master of Science in Computer Science and Engineering, Apr 2020, Polytechnic of Milan, Italy (110/110)
	Bachelor's Degree in Computer Science and Engineering, Sep 2017, Polytechnic of Milan, Italy (106/110)
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
SCHOLARSHIPS	
Spring 2019	Research Assistantship (RA) position (20 hours/week) with full tuition waiver plus monthly stipend
TECHNICAL SKILLS	
Languages	Expert in Java, competent in Python and SQL, basic knowledge of C
Software	Unity3D, Anaconda, Android Studio, Intellij, Pycharm, NS-3, Linux, UML (StarUML)
WORK EXPERIENCE AND PROJECTS	
2020-Present	Software Engineer at Bending Spoons
2019	CODE@FLOW: a challenge engineered by Flow Traders in Amsterdam to get to experience the developer life in the trading industry. One-week event with high-tech discussions with specialists at Flow Traders in software development and trading and experiencing challenging cases based on real-life situations in the trading industry.
	Research Assistant at UIC: Research Assistant for the professor Jon Solworth to apply Machine Learning to the anonymity network Fasor

VITA (continued)

	Android Projects: Five small Android applications to explore the main concepts and components of the Android OS, focusing on services, fragments, broadcast receivers and applications with multiple threads.
2018	Replicated Data Storage: Distributed Systems project. Implementation of a replicated data storage. The system provides a causal consistency model. Implemented using Java.
	Data Mining Project: Project done in collaboration with the Bip company in Milan. The goal of the project is to provide a working forecast model that can be used by retailers to optimize promotions and warehouse stocks.
	PixelVR: Virtual reality project using Unity3D and the HTC Vive. Implementation of a virtual reality work environment where it is possible for the user to create 3D models made of voxels. Presentation of the project, video and code: https://atrica2.people.uic.edu/Project3/.
	Collaborative Robotics Modeling: Apply formal methods, in particular the TRIO language, to model the behavior of a robot in an industry where it has to work in collaboration with humans focusing on both the productivity of the robot and the safety of humans.
2017	ATHENS project at Universidad Politcnica de Madrid: Completed a merit-based program, organized by a network of European universities. The course was based on physical computing, providing fundamental skills in Arduino programming
	LorenzoThePST: Complete Java implementation of the board-game Lorenzo il Magnifico. Including all the features of the board-game, a command line interface, a graphical interface and the possibility to play it with many players in a LAN (using RMI - Remote Method Invocation).