POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in
Computer Science and Engineering

POLITECNICO
MILANO 1863

Explorative Techniques and Vulnerability Assessment on Automotive Networks

Supervisor:
PROF. STEFANO ZANERO

Assistant Supervisor:
STEFANO LONGARI

Master Graduation Thesis by:
LIVIU RAZVAN CALIN
Student Id n. 859091

Academic Year 2017-2018

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AE Extended Address

AI Address Information

ASCII American Standard Code for Information Interchange

AUTOSAR AUTomotive Open System ARchitecture

BS Block Size

CAN Controller Area Network

CF Consecutive Frame

CGM Central Gateway Module

DID Data IDentifier

DLC Data Length Code

ECM Engine Control Module

ECU Electronic Control Unit

FC FlowControl Frame

FF First Frame

FF-DL First Frame Data Length

FS	Flow Status
ID	IDentificator
ISO-TP	ISO Transport Layer
JSON	JavaScript Object Notation
LIN	Local Interconnect Network
Mtype	Message type
NRC	Negative Response Code
OBD	On-Board Diagnostics
OEM	Original Equipment Manufacturer
OSI	Open Systems Interconnection
OTA	Over The Air
PCB	Printed Circuit Board
PCI	Protocol Control Information
PDU	Protocol Data Unit
RPM	Revolutions Per Minute
SA	Source Address
SAE	Society of Automotive Engineers
SF	Single Frame
SF-DL	Single Frame Data Length
SID	Service IDentifier
SN	Sequence Number
STmin	Separation Time minimum
TA	Target Address
TAtype	Target Address Type
TCM	Transmission Control Module
UDS	Unified Diagnostic Services
VIN	Vehicle Identification Number

ABSTRACT

Modern automobiles incorporate tens of electronic control units (ECUs), driven by, according to estimates, as much as 100,000,000 lines of code. They are tightly interconnected via internal networks, mostly based on the CAN bus standard. Past research showed that, even remotely, an attacker could control safety-critical inputs such as throttle, steering or brakes. One of the first time-consuming activities that a researcher has to perform when approaching to make a security analysis of an automobile is to realize which systems it has on board, how these are implemented, which ECU is responsible for performing each functionality and consider if there are known vulnerabilities that may be exploited by an attacker. Being a relative new field, the availability of tools which facilitate such kind of work is quite scarce, and always relies on brute-force or databases specific for each manufacturer or automobile model. Researchers have to do a lot of manual reverse engineering to perform those tasks.

This thesis presents and analyzes a technique for performing ECU discovering on CAN networks without relying neither on brute-force nor on existing databases, but exploiting the until now underestimated functional addressing feature of UDS diagnostic protocol. We provide a methodology which is able for each discovered ECU to perform advanced information gathering and we propose a structured framework in order to implement tests for known vulnerabilities and perform automated vulnerability assessment.

We traduce our approach in the implementation of the cmap tool and in order to prove its effectiveness and efficiency we perform the ECU discovery test on four modern unaltered vehicles, obtaining positive results on tree of them.

SOMMARIO

Il mondo automobilistico e continuamente soggetto a cambiamenti e miglioramenti. Dalla scelta di materiali strutturali sempre più leggeri ma capaci di assorbire una maggiore quantità di energia in caso di incidente, agli sviluppi della struttura dello pneumatico allo scopo di garantire aderenza in tutte le possibili condizioni del manto stradale; dall'evoluzione della meccanica del motore ai fini di migliori consumi di carburante senza sacrificare le prestazioni, all'applicazione di studi di aerodinamica per raggiungere contemporaneamente maggiore efficienza e deportanza.
Nonostante sia imprescindibile l'apporto dato dai progressi sopracitati, l'aspetto che ha maggiormente rivoluzionato il mondo dell'automobile negli ultimi 40 anni è stata l'integrazione sempre più massiccia di elettronica e software. Partendo dai primi sistemi di iniezione del carburante e proseguendo verso i primi sistemi di antibloccaggio delle ruote in frenata (ABS), l'automobile si è dotata in maniera sempre più evidente di computer predisposti a gestire ogni suo singolo aspetto. Il risultato visibile su una moderna automobile è la disposizione di decine di centraline elettroniche di controllo (ECU), connesse tra loro tramite una o più reti interne tendenzialmente basate sul protocollo CAN, e di centinaia di milioni di righe di codice.
Con l'aumento della complessità dei sistemi presenti sulle automobili le classiche tecniche di diagnosi dei problemi finora impiegate si sono rivelate inefficaci. La diagnostica in generale determina, verifica e classifica i sintomi al fine di ottenere un quadro generale che permetta di trovare la causa di un problema nel veicolo. Dunque è stato necessario sviluppare ed introdurre nuovi strumenti elettrici ed elettronici di diagnosi, in grado di fornire informazioni sulle caratteristiche elettroniche dei vari componenti e misurazioni dei valori di parametri come la pressione dell'olio, il volume d'aria apportata al motore e le varie temperature.
Oggi avanzati strumenti diagnostici sono la normalità. Lo sviluppo e l'introduzione di nuovi concetti e soluzioni di diagnosi offrono ai produttori e più in generale ai fornitori del modo automobilistico un enorme potenziale per migliorare l'efficienza della produzione e la qualità del prodotto finale. L'applicazione di tecniche avanzate di diagnosi spazia dal processo di sviluppo al controllo della qualità durante la produzione e fino all'impiego post-vendita nelle officine meccaniche.
I sistemi diagnostici moderni richiedono che gli strumenti informatici

siano in grado di comunicare con i vari sistemi presenti su un'automobile per eseguire le varie attività di diagnosi, generalmente basandosi sul connettore standard per automobili OBD-II. La comunicazione in sé si basa su regole formalmente definite da protocolli di diagnosi, i quali definiscono lo standard e le modalità di interazione fra il dispositivo di diagnosi e le centraline stesse, e convenzioni di implementazione adottate dai produttori.
Seppur riconosciuti e statisticamente rilevanti i miglioramenti e progressi apportati a tali sistemi, l'aggiunta sempre più ingente di computer per la gestione di aspetti anche critici del veicolo ha inevitabilmente portato con sè conseguenze in materia di sicurezza informatica; problematiche tutt'altro che ignote al mondo dell'ICT, ma inedite nel mondo dell'automobilismo. Una combinazione di pressioni economiche per il rilascio sempre più solerte di nuovi prodotti sul mercato, esigenze di integrazione di componenti di diversi fornitori, adempienza alle norme di legge, l'aggiunta sempre più preponderante sulle centraline della vettura di interfacce comunicanti con il mondo esterno, una scarsa o persino totalmente assente metodologia d'approccio verso il problema della sicurezza informatica ed una frenetica ricerca del profitto necessaria per la sopravvivenza in un mercato estremamente competitivo hanno portato negli ultimi anni alla comparsa sui veicoli di vulnerabilità informatiche sempre più concretamente sfruttabili per un eventuale aggressore.
Una delle prime attività che un ricercatore deve svolgere durante l'analisi di sicurezza di un veicolo è quella di identificare quali sistemi sono presente a bordo, come sono implementati, quale centralina è responsabile per ogni funzionalità e considerare se sono presenti vulnerabilità note che potrebbero essere sfruttate da un aggressore. Essendo quello della sicurezza informatica automobilistica un ambito relativamente recente, la disponibilità di strumenti che facilitino tali attività è molto limitata. I metodi implementati oggi sono generalmente basati sull'utilizzo di brute-force o basi di dati specifiche per produttore e modello automobilistico, tecniche che comportano nel primo caso problemi di tempo (in determinate circostanze il brute-force diventa impraticabile) e nel secondo caso risultano essere di difficile reperibilità e spesso riportano informazioni frammentate. I ricercatori sono quindi obbligati ad applicare ogni volta, manualmente, diverse tecniche di reverse engineering, le quali sono onerose in termini di tempo.
Nel seguente lavoro di tesi viene presentata e analizzata una tecnica automatizzata per individuare le ECU presenti sulla rete CAN di un'automobile, senza l'impiego di brute-force o basi di dati esistenti. Il proto-

collo che abbiamo studiato approfonditamente per raggiungere gli scopi di questo lavoro è 'Unified Diagnostic Services (UDS)', il quale è stato sviluppato e supportato per diventare un nuovo standard nell'ambito automobilistico, unificando ed estendendo le caratteristiche di due suoi predecessori: ISO 15765-3 e KWP2000. Questa tecnica sfrutta l'indirizzamento funzionale messo a disposizione dal protocollo di diagnosi UDS, caratteristica di quest'ultimo fino ad ora molto sottovalutata. Questo tipo di indirizzamento permette una comunicazione uno a molti, tra il dispositivo esterno e le varie centraline presenti sull'automobile. Oltre alla presentazione di questa tecnica proponiamo una metodologia di raccolta avanzata di informazioni per ogni centralina individuata, al fine di identificare la funzionalità svolta dalla stessa all'interno dei sistemi dell'automobile.
Infine definiamo una struttura di classi e metodi volti all'implementazione di test per l'individuazione automatizzata di vulnerabilità note all'interno dei sistemi dell'auto. Al fine di comprovare la validità di quanto proposto e realizzato abbiamo individuato una nuova vulnerabilità nelle implementazioni del protocollo UDS. Tale vulnerabilità è costituita dall'accesso non protetto a servizi UDS critici ed in grado di disabilitare i normali pacchetti CAN inviati dalle centraline. Quest'operazione, come dimostrato da precedenti lavori, favorisce la realizzazione di attacchi basati sull'immissione nella rete CAN di pacchetti contraffatti. Per permettere di rilevare la sua presenza sui sistemi automobilistici è stato realizzato un test basato sulla struttura proposta.
Abbiamo trasformato il nostro approccio e le tecniche discusse nella realizzazione dello strumento cmap, un software che implementa le funzionalità di ricerca delle ECU e la struttura per la realizzazione e l'applicazione dei test di vulnerabilità note. Per comprovare l'efficacia e l'efficienza del nostro strumento abbiamo eseguito il test di individuazione delle centraline su quattro autovetture moderne e inalterate, presentando l'elenco delle ECU trovate, discutendo i risultati positivi ottenuti su tre di queste e analizzando i problemi e le limitazioni riscontrate sulla quarta vettura. Abbiamo inoltre eseguito il test per individuare la presenza della vulnerabilità discussa su un'autovettura, provando la sua correttezza e mostrando gli effetti temporanei riscontrati sui sistemi della vettura stessa.

1 INTRODUCTION

1.1 THE EVOLUTION OF THE AUTOMOTIVE ARCHITECTURE

By far, road vehicles are the most popular mode of transport in the world. According to official statistics from Eurostat [1] and from USA Bureau of Transportation [2], cars, motorcycles, trucks and coaches are on average adopted three times more than rail, air or sea for passenger transfers Figure 1.1. Among these, cars are the most preferred conveyance. Considering only legally registered automobiles, in 2015 there were 1,776,136,357 vehicles circulating on Earth [3], and this number is expected to duplicate by 2040 [4]. Cars have massively evolved over the years. The well known saying "four wheels and an engine", by which automobiles are sometimes referred, is today by all means completely inadequate to describe the technological state to which cars have leaped, thanks to decades of research. Improvements in active and passive safety systems have almost halved road casualties in the USA with respect to 40 years ago [5]. Though by no means diminishing other paramount

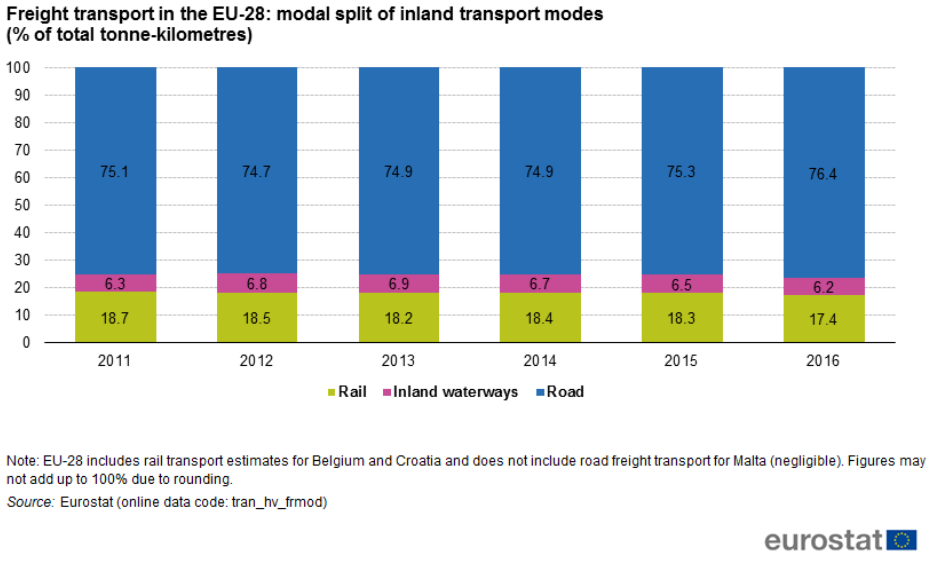


Figure 1.1: European modal split of inland transport modes

enhancements in the car universe, the most radical changes cars have witnessed in the last four decades are owed to the ever increasing ad-

dition of electronics and software. Starting from the late seventies with the first and relatively simple (at least if compared to modern standards) Electronic Control Unit (ECU), responsible for regulating fuel injection systems in order to meet the increasingly stringent laws on emissions, embedded systems have rapidly pervaded throughout the car and now, by communicating among each other via an internal network - the most common of which, today, is CAN bus -, supervise every aspect of it, from telematics units to steering, from airbags deployment to central locking, from heating, ventilation and air conditioning to autonomous active safety systems. Figure 1.2 shows an overview of the most common systems presents on a modern automobile.

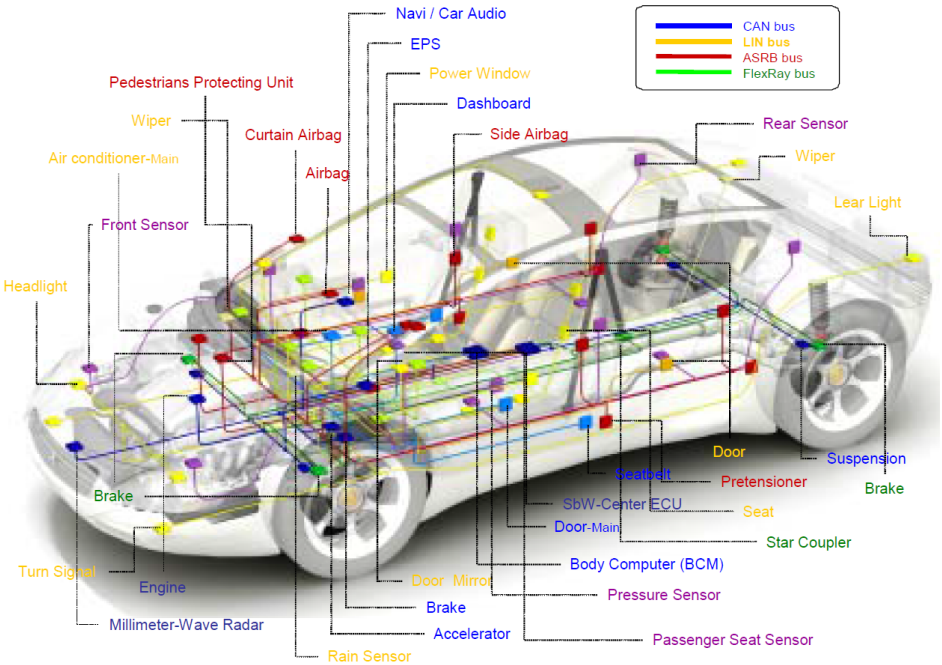


Figure 1.2: Overview of the features typically controlled by ECUs in a modern premium sedan [6]

1.2 AUTOMOTIVE DIAGNOSTIC OVERVIEW

Prior to the 1950s, most discovery of problems in automotive systems was done with just a few gauges and the rest by hand, listening to awkward sounds, smelling strange odors, and otherwise using the senses to detect problems. As vehicles became more complex, we entered a «Middle age» in the 1960s, where there was more of a reliance upon instrumentation. Video display terminals could show the electronic characteristics of the electric system, and as computers became more readily

available, other measurements, such as vacuum, oil pressure, and various temperatures could be integrated into a machine-based diagnostic system.
Diagnostic determines, verifies and classifies symptoms aiming to get an overall picture in finding the root cause of a problem in a vehicle. The detection, improvement and communication strategies applied to abnormal operation of systems is monitored by electrical and electronic devices. Therefore, the purpose of diagnostic is to identify the root cause of abnormal operation so a repair can be performed.
Today powerful diagnostic tools are an everyday reality, for example the Bosch KTS 590 showed in Figure 1.3. The development and introduction of new diagnostic concepts and diagnostic solutions offers significant potential to automotive OEMs and suppliers for realizing efficiency gains and quality improvement. The applications of diagnostic solutions range from the development process, to quality assurance in production and finally diagnostics in the service garage.

- During the development process, correct functionality of the components must be validated. Diagnostic subsystem then takes role at reading out ECU's internal sensor and actuator's values;
- during the production process the diagnostic system is used for transferring calibration data and software updates to the non-volatile memory of the ECUs, programming and tests included;
- aftersales, diagnostics are mainly used for error detection. Detected errors are stored to a persistent fault memory and read at the service station in order to make troubleshooting feasible.

Back in 1990 automotive OEMs created their own tools in-house, precisely to their requirements and specific applications. This produced individual in-house solutions within each Original Equipment Manufacturer (OEM) and even for different process steps. This trend continued until open, non proprietary diagnostic tools became available on the market in the year 2000. For OEMs, the route of product licenses is noticeably more cost-effective than assuming responsibility for developing and maintaining proprietary tools in-house. Moreover, there are shared benefits due to synergistic effects of the combined experience of other market participants. In 2005 the tools began to support general standards [8], [9].
Diagnostic systems require that computer tools can access a communication protocol and perform all diagnostic tasks based on the diagnostic

Figure 1.3: Example of a diagnostic professional tool, Bosch KTS 590 [7]

connector. A diagnostic protocol defines a set of conventions used for diagnostic communication between a diagnostic device and an ECU in the vehicle. The protocol that we deeply studied to achieve the purposes of this work is the Unified Diagnostic Services (UDS), which is claimed to replace older protocols such as ISO 9141 and KWP2000.

1.3 TOWARDS AUTOMOTIVE SECURITY

Albeit irreproachable the contribution of embedded computers to overall automobiles improvement, the unavoidable consequence of this increased complexity and co-presence of electronic and computer based components is a wider digital attack surface. A combination of time to market pressures, integration needs of components from manifold distinct suppliers, laws abidance, outside world interfaces requirements, poor or even absent security concerns and frantic cost reduction originated a totally new set of vulnerabilities (new to the car industry, yet most of them well known by the majority of IT companies) that a potential adversary might exploit for malicious intents. Indeed, in the last decade the number of automotive security studies and reported possible exploit vectors - even completely remote and Internet based - has increased exponentially [10]–[16], to the point that «car hacking» is now

being taken into serious consideration by, for instance, US government agencies [17], [18], government acts for strengthening automotive cybersecurity regulations [19] and bills for specifically sanctioning unauthorized access to vehicles have already been proposed [20].
The majority of the attacks published so far share a common point: the leverage of a vulnerability (or a chain of vulnerabilities) with the aim of indiscriminately sending messages into the internal car network and proving that it is possible to alter the behavior of safety-critical elements such as engine, brakes or steering.
One of the first time-consuming activities that a researcher has to perform when approaching to make a security analysis of an unknown automobile is to realize which systems it has on board, how these are implemented and which ECU is responsible for performing each functionality of the automobile. Whereupon he inspects and analyse each system and relative ECU (or ECUs) in order to find out known weaknesses and resulting vulnerabilities in its design or implementation.
In the classical IT industry, there are several and consolidated tools which facilitate such kind of work, like for example nmap [21], which is a free and open source utility for network discovery and security auditing. On the contrary, being a relative new field, the availability of such tools for the automotive industry is quite scarce. Researchers have to do a lot of manual reverse engineering to perform those tasks since they have at disposal just a few little automated tools. The available tools always rely on brute-force or existing databases with ECU codes employed by each specific manufacturer and automobile model.
The tools based on brute-force in some circumstances can perform with acceptable time performances, around 4 minutes, while in other they are definitely out of maximum time, more than 2 years. On top of that brute-force procedure is easily detectable and any effort in performing an ECU discovery procedure by using it could be made ineffective by protection systems.
The existing databases of the systems and ECU codes are very limited and incomplete, since such information is proprietary and hard to access by the perspective of an independent researcher. The existing ones were once again obtained by manual reverse engineering and brute-force of real automotive systems. Moreover, none of the existing tools provides additional information on the discovered ECUs, neither performs automatic vulnerability auditing.

1.4 THE CONTRIBUTION OF THIS WORK

This thesis presents a novel technique which allows to list the ECUs present on an automobile CAN bus. It is based on the functional addressing feature of UDS diagnostic protocol, it does not employ neither brute-force nor existing databases of ECU codes and thus it overcomes the time, detectability and narrowness limitations of the aforementioned techniques. We widely discuss the characteristics of UDS protocol comparatively with other diagnostic protocols and its diffusion across different manufacturer and OEM suppliers. For each discovered ECU, we provide a set of information useful to establish the exact purpose of the ECU in the automobile systems.
We then propose a framework which facilitates the development and putting in practice of tests for known vulnerabilities. The framework allows to apply to each discovered ECU the available set of tests to reveal the presence of known vulnerabilities.
As a concrete result of this work, we have implemented the cmap tool, a new discovery and vulnerability assessment utility for the automotive CAN networks. It is a python-based tool whose procedures works locally, through the standard diagnostic port, which is mandatory in essentially every country [22].
As a proof of concept of the usage of the vulnerability test framework we have identified a novel vulnerability in the UDS protocol implementations, which allows the usage of critical UDS services by unauthorized clients and which in order may be exploited for performing other kinds of attacks. We proceeded to implement a test based on our framework for revealing its presence in automotive systems.
In order to validate our technique and tool we have performed the discovery test on four different vehicles with a successful result in three of them. We have performed the vulnerability test on a single car, the Audi A3 Sportback, and its execution revealed the presence of the discussed vulnerability.
In summary, this thesis makes the following contributions:

- it provides a technique and a tool for performing quick ECUs listing on a CAN network, without employing brute-force and with advanced information gathering about each discovered ECU;
- it provides a framework for implementing and executing tests in order to reveal known vulnerabilities in automotive systems;

- it discusses a novel vulnerability in UDS implementations and provides a test which reveals it if present.

1.5 WORK STRUCTURE

This thesis is structured in five principal chapters. Chapter 2 provides an introduction to CAN bus, an in-depth summary of the ISO-TP and UDS protocols. Finally, it describes the investigated problem and discusses the actual available solutions. Chapter 3 provides the motivations of why we have chosen the UDS diagnostic protocol and how we exploited it in order to develop our tool. The same chapter then analyzes the Unsecured UDS Critical Services vulnerability and describes how an attacker could exploit it. Chapter 4 gives the details of the contributions done to the open source pyvit library, the details about how the cmap tool was designed and implemented and how the test for the mentioned vulnerability was implemented. Chapter 5 shows the tests done on modern, unaltered production vehicles and discusses the encountered problems and the results concerning both: discovery procedure and know vulnerability test framework. Chapter 6 discusses the possible future directions of the research in this purview and the limitations of this work.

2

BACKGROUND AND MOTIVATION

This chapter provides an introduction to CAN bus and in-depth summary of the ISO-TP and UDS protocols. First it gives a brief history of CAN protocol and portrays its fundamental properties, subsequently presents the ISO-TP protocol, which is a Transport Layer protocol that allows the transmission of messages that exceed 8-byte maximum data size of a CAN frame, and the UDS protocol, which is an Application Layer diagnostic protocol for the automotive industry on top of which is based the more well-known OBD standard. Afterwards we provide an analysis of the security investigation problem of discovering the ECUs present on a CAN bus and examine them for the presence of vulnerabilities. We describe the currently available tools and their shortcomings which are overcame by the cmap tool which is the solution proposed by this work and which is introduced in the last section.
During the protocols presentation, we pay particular attention to the features which allowed to implement a discovery method without relying on brute-force, in order to get a list of the most significant number of ECUs present on the automobile.

2.1 BACKGROUND

2.1.1 *CAN*

The CAN bus standard was designed by the automotive industry to provide a way for the various ECUs in an automobile to communicate. It forms a simple bus topology network, as showed in Figure 2.1, in which messages are tagged with identifiers that are associated with specific functions.
Most important applications of CAN bus are to be seen in the automotive world, which is the specific domain for which CAN has been designed. CAN has relentlessly been adopted by more and more car manufacturers, to the point that almost all automobiles today on the market feature at least one CAN network as a backbone for embedded systems communications. CAN allows for extremely cost-effective component integration thanks to the general purpose ability of carrying data for a

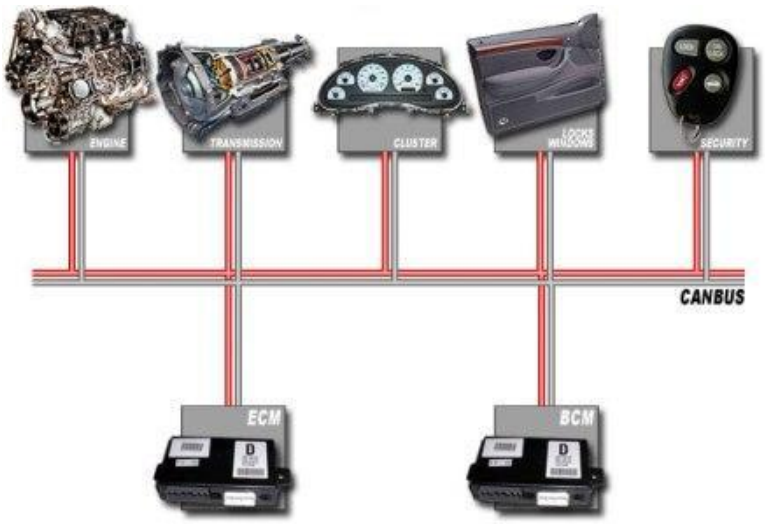


Figure 2.1: CAN bus topology

great variety of applications, the remarkable reduction of wire harnesses due to the bus topology with respect to direct hard wired connections and extremely competitive chip prices. In addition to that, it allows fast data transfer rates, up to 1 Mbps.
The CAN protocol covers two layers in the Open Systems Interconnection (OSI) Reference Model, Physical Layer and Data Link Layer, all layers above are implementation specific. The Data Link Layer consists of two sub layers, Logic Link Control sublayer, which for example decides which messages from the ones received are accepted, and Medium Access Control sublayer, which controls framing, arbitration, error checking, error signaling and fault confinement. The Physical Layer handles the actual transfer of bits between the nodes with respect to all electrical properties. [23]
Development of the CAN bus started in 1983 at Robert Bosch GmbH [24]. The protocol was officially released in 1986 at the Society of Automotive Engineers (SAE) [25] conference in Detroit, Michigan. The first CAN controller chips, produced came on the market in 1987. Released in 1991 the Mercedes-Benz W140 was the first production vehicle to feature a CAN-based [26].
Bosch published several versions of the CAN specification and the latest is CAN 2.0 published in 1991. The CAN was standardized in 1993 by the International Organization for Standardization as ISO 11898:1993 [27] and after several revisions reached his currently standardized version ISO 11898:2015 [28].
According to the original Bosch specification, the main properties of

CAN are: prioritization of messages; guarantee of latency times; configuration flexibility; multicast reception with time synchronization; system wide data consistency; multimaster; error detection and signaling; automatic retransmission of corrupted messages as soon as the bus is idle again; distinction between temporary errors and permanent failures of nodes and autonomous switching off of defect nodes.

ECUs are essentially embedded devices, networked together on the

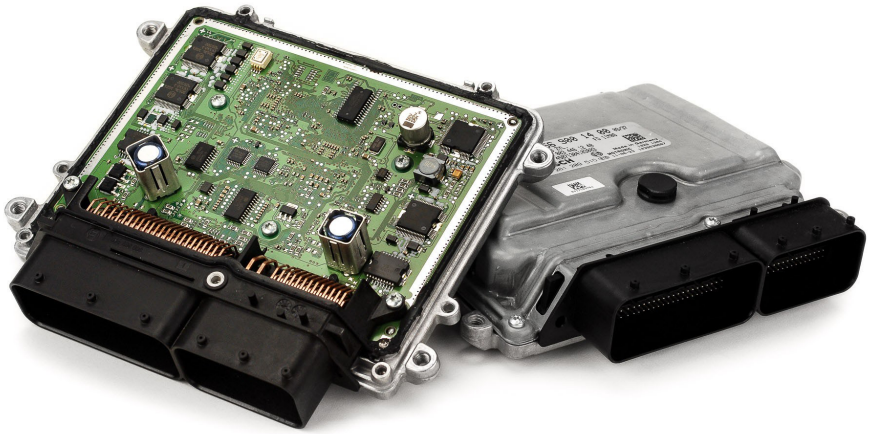


Figure 2.2: ECU

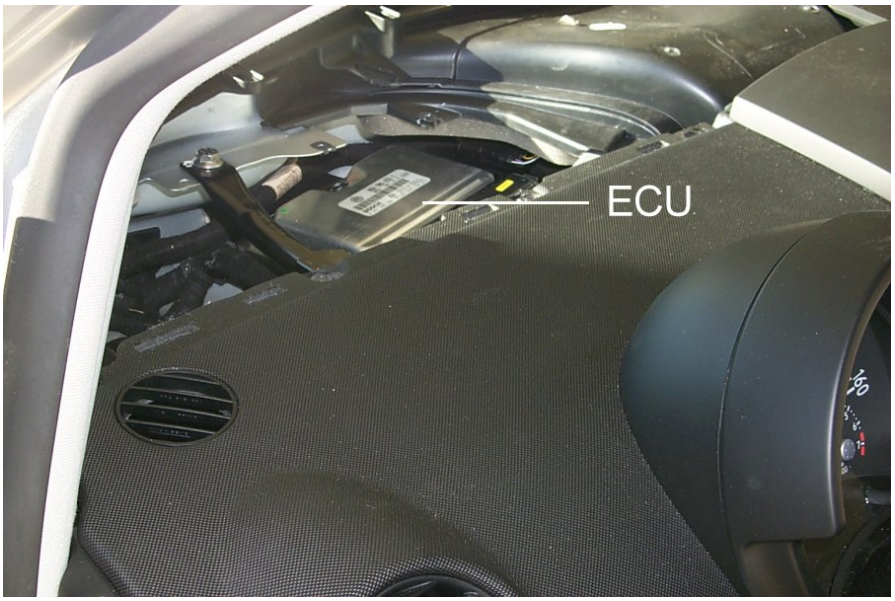


Figure 2.3: ECU mounted on car

CAN bus. Each is powered and has several sensors and actuators attached to them, see Figure 2.2 and Figure 2.3 below. The sensors provide input to the ECUs so they can make decisions on what actions to take.

The actuators allow the ECU to perform actions. These actuators are frequently used as mechanisms to introduce motion, or to clamp an object to prevent motion.
In summary, ECUs are special embedded devices with specific purposes to sense the environment around them and act to help the automobile. Each ECU has a purpose to achieve on its own, but they must communicate with other ECUs to coordinate their behavior. For this our automobiles utilize CAN messages. Some ECUs periodically broadcast data, such as sensor results, while other ECUs request action to be taken on their behalf by neighboring ECUs. Table 2.1 provides a summary of most common used ECUs across all automobiles manufacturers. Other CAN messages are also used by manufacturer and dealer tools to perform diagnostics on various automotive systems.

Component	Functionality
ECM	**Engine Control Module** Controls the engine using information from sensors to determine the amount of fuel, ignition timing, and other engine parameters.
EBCM	**Electronic Brake Control Module** Controls the Antilock Brake System (ABS) pump motor and valves, preventing brakes from locking up and skidding by regulating hydraulic pressure.
TCM	**Transmission Control Module** Controls electronic transmission using data from sensors and from the ECM to determine when and how to change gears.
BCM	**Body Control Module** Controls various vehicle functions, provides information to occupants, and acts as a firewall between different subnets if present.
Telematics	**Telematics Module** Enables remote data communication with the vehicle via cellular link.
RCDLR	**Remote Control Door Lock Receiver** Receives the signal from the car's key fob to lock/unlock the doors and the trunk.
HVAC	**Heating, Ventilation, Air Conditioning** Controls cabin environment.
SDM	**Inflatable Restraint Sensing and Diagnostic Module** Controls airbags and seat belt pretensioners.
IPC/DIC	**Instrument Panel Cluster/Driver Information Center** Displays information to the driver about speed, fuel level, and various alerts about the car's status.
Radio	**Radio** In addition to regular radio functions, funnels and generates most of the in-cabin sounds (beeps, buzzes, chimes).
TDM	**Theft Deterrent Module** Prevents vehicle from starting without a legitimate key.

Table 2.1: ECU examples

Most CAN buses are characterized by the topology specified in ISO 11898-2 [29], also called «high speed CAN»: a two wire, CANH (high)

and CANL (low), differential balanced signaling scheme featuring a termination at each end by means of a 120-ohm resistor. The differential signaling allows for noise immunity; the balanced signaling means that the current flowing in each signal line is equal but opposite in direction, resulting in the necessary field-canceling effect to obtain low noise emissions [30]. Each CAN node generally comprises three elements, as

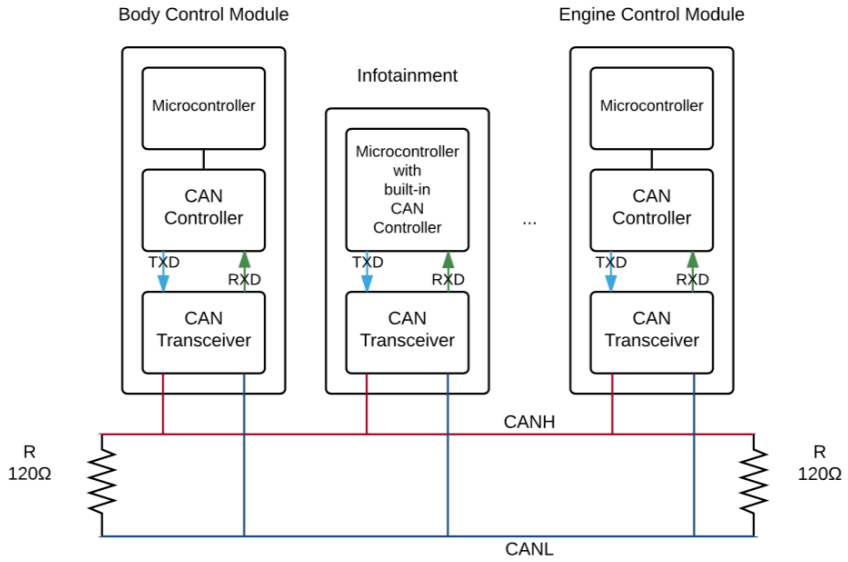


Figure 2.4: Example architecture of an ISO 11898-2 CAN network

showed in Figure 2.4:

1. **Microcontroller:** is responsible for sending and processing complete CAN frames to and from the CAN controller and supervising the CAN controller operation;

2. **CAN controller:** s in charge of correctly implementing the CAN specifications. It synchronizes with the CAN signal, sends and receives logical data to and from the CAN transceiver, automatically adds stuff bits, performs error handling and actualizes the error modes finite state machine;

3. **CAN transceiver:** serves as an interface between the CAN controller and the physical bus by translating logical signals coming from the CAN controller into bus electrical levels.

In the recent years, the trend which has characterized automotive specific microcontrollers is to embed the CAN stack on chip, for both cost effectiveness and space saving reasons [31]. Thus, there exist microcontrollers

with embedded CAN controllers and microcontrollers which even feature the entire stack on chip (i.e. CAN controller and CAN transceiver incorporated inside a microcontroller [32]). Though less popular, CAN buses can also be single wire (e.g., standard SAE J2411 [33]). Because of their better error resiliency, two wire buses are preferred, especially in safety-critical applications.
The layout of a sample CAN frame is showed in Figure 2.5, Figure 2.5 (a) shows a frame which uses 11-bit identifier, commonly called CAN A, while Figure 2.5 (b) shows a frame that uses 29 bit identifiers, commonly called CAN B. However, there is no enforced addressing or message source identification and thus any device on the CAN network can send any message in a manner indistinguishable to the target ECU. The identi-

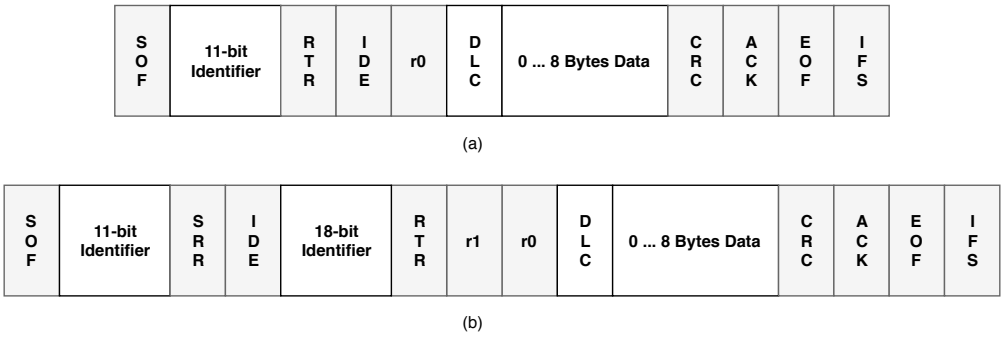


Figure 2.5: CAN frame structure

fier is used as a priority field, the lower the value, the higher the priority. The CAN standard mandates two different signaling states that can be written on the bus: dominant and recessive, with the former capable of anytime overwriting the latter; that is, whenever a dominant bit is sent at the same time as a recessive bit, the bus state and thus the logical signal perceived by all other CAN nodes is dominant. Most CAN bus implementations feature a wired-AND configuration, hence the dominant bit is the logical 0 whereas the recessive bit is the logical 1. The distinguishing factor between a dominant state and a recessive state is the differential voltage between the CANH and the CANL lines. In case such difference does not exceed a threshold value (usually 0.9 V in ISO 11898-2 networks), a recessive state (thus 1) is presumed, else a dominant state (0).
During a recessive state, the signal lines and resistors remain in a high impedance states and voltages on both CANH and CANL tend weakly toward a midway value, usually 2.5 V. During a dominant condition, the signal lines and resistors move to a low impedance state so that current flows through the resistor, CANH voltage tends to 3.5 V and CANL

tends to 1.5 V.
The identifier is also used in order to help ECUs determine whether they should process it or not. This is necessary since CAN traffic is broadcast in nature. All ECUs receive all CAN packets and must decide whether it is intended for them. This is done with the help of the CAN packet identifier.
The ISO 11898-2 standard supports communications transfer rates up to 1 Mbps (i.e. minimum nominal bit time of 1 us). However, because of unavoidable skews due to the physical required time for the signal to travel in the transmission medium from one end to the other and the bus nature of CAN which requires all nodes to be synchronized, transfer speeds are restricted by cable length. Approximately, 500 kbps are achievable only in buses up to 100 meters, 250 kbps up to 200 m, 125 kbps up to 500 m and only 10 kbps up to 6 km.
The cable impedance is required to be 120-ohm, though values in the interval of [108;132] ohm are still permitted.

2.1.2 *ISO-TP Protocol*

2.1.2.1 *Protocol Overview*

ISO 15765-2, or better known as ISO-TP, is an international standard for sending data packets over a CAN bus. It is last specified in ISO 15765-2:2016 [34]. It defines a way to send arbitrary length data over the CAN bus. The internal operation of the Transport Layer provides methods for segmentation, transmission with flow control, and reassembly. It is the protocol specified for Transport and Network layers of OSI model, its main purpose is to transfer messages that might or might not fit in a single CAN frame. Messages that do not fit into a single CAN frame are segmented into multiple parts, where each can be transmitted in a CAN frame. In Figure 2.6 and Figure 2.7 we show respectively an example of unsegmented communication and one of a segmented communication.
ISO-TP prepends one or more metadata bytes to the beginning of each CAN packet. These additional bytes are called the Protocol Control Information (PCI). Employing those, the protocol allows to transmit a maximum payload size of 4095 bytes. Flow control is used to adjust the sender to the capabilities of the receiver.
The communication between the peer protocol entities of the network layer in different nodes is done by means of exchanging frames called Protocol Data Unit (PDU). ISO 15765 specifies four different types of transport ayer protocol data units: single-frame (SF PDU), first-frame (FF

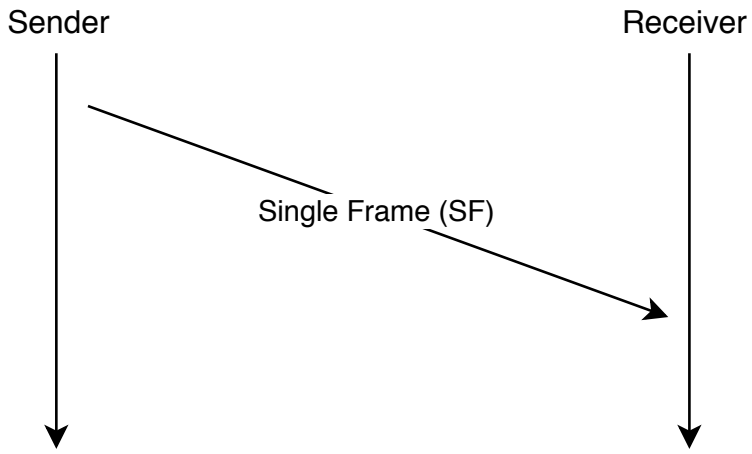


Figure 2.6: Example of an unsegmented message

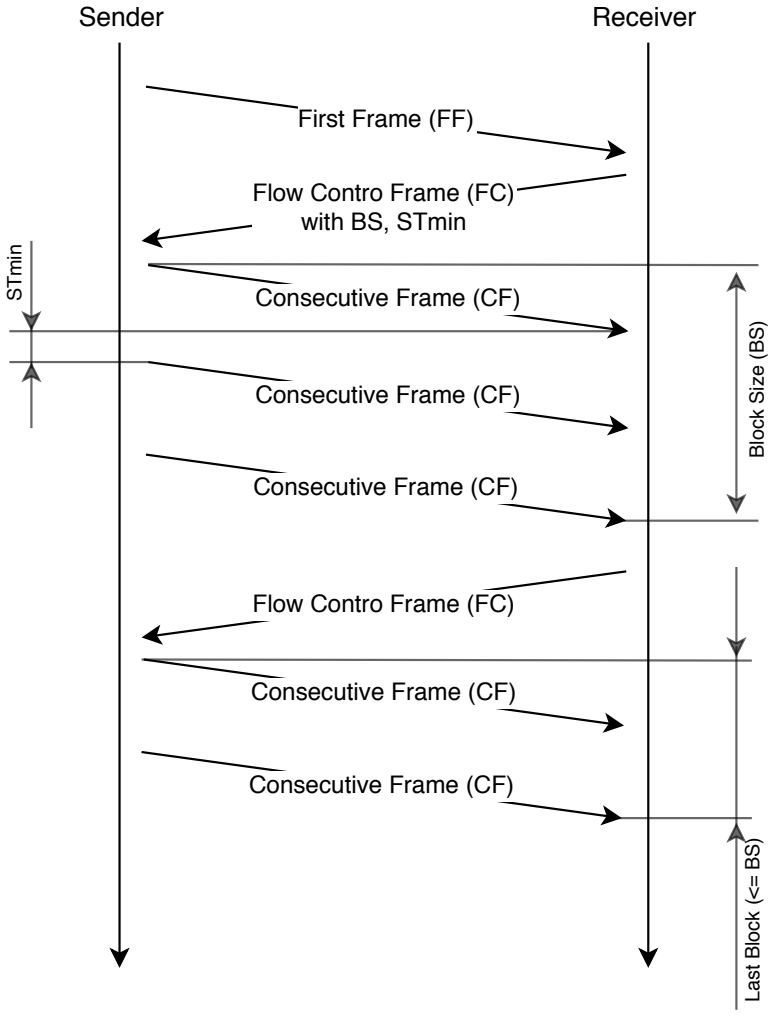


Figure 2.7: Example of a segmented message

PDU), consecutive-frame (CF PDU) and flow-control (FC PDU); which are used to establish a communication path between the peer Network Layer entities, to exchange communication parameters and to transmit user data. All PDUs consist of three fields, as given in Table 2.2.

Address information	Protocol control information	Data field
AI	PCI	Data

Table 2.2: PDU format

The Address Information (AI) field is used to identify the communicating peer entities as well as the communication model for the message and the optional address extension. AI information field is composed by: Source Address (SA), Target Address (TA), Target Address Type (TAtype), Extended Address (AE) (optional). The set of these parameters contributes to the determination of the CAN arbitration ID, according to different addressing formats as described in Section 2.1.2.3;
The PCI field identifies the type of PDUs exchanged, its use and structure are described in Section 2.1.2.2;
The Data field is all or part of the user data received which is being transmitted with the PDU. The size of the Data field depends on the PDU type and the address format chosen.

2.1.2.2 *PDU Types*

The ISO-TP protocol defines four types of PDUs, the type of each PDU is defined by the first 4 bits of the PCI. The type in order influences the length and the meaning of the next fields contained in the PCI, as reported in Table 2.3.

PDU bytes				
Byte #1				
Bits 7-4	Bits 3-0	Byte #2	Byte #3	PDU Name
0	SF-DL	N/A	N/A	Single Frame (SF)
1	FF-DL		N/A	First Frame (FF)
2	SN	N/A	N/A	Consecutive Frame (CF)
3	FS	BS	STmin	FlowControl Frame (FC)

Table 2.3: Summary of PCI bytes meaning

SINGLE FRAME (SF) shall be used to support the transmission of messages that can fit in a single CAN frame. It is an optimized implementation of the protocol with the message length embedded in

the PCI first byte only. In fact, Single Frame Data Length (SF-DL) states the length of the data contained in the PDU in number of significant bytes, the allowed values are:

- 0x1 - 0x6 for all the addressing methods;
- 0x7 only allowed with normal addressing.

FIRST FRAME (FF) shall only be used to support the transmission of messages that cannot fit in a single CAN frame, i.e. segmented messages. On receipt of an FF, the receiving entity shall start assembling the segmented message. First Frame Data Length (FF-DL) states the length of the data contained in the PDU in number of significant bytes, the allowed values are:

- 0x7 only allowed with extended or mixed addressing;
- 0x8 - 0xFFF for all the addressing methods. The maximum segmented message length supported is equal to 4 095 bytes of user data.

CONSECUTIVE FRAME (CF) shall be used for sending the remaining data after a FF was correctly sent. On receipt of a CF, the receiving entity shall assemble the received data bytes until the whole message is received. The parameter Sequence Number (SN) is used to specify the following:

- the numeric ascending order of the Consecutive Frames;
- that the SN shall start with zero for all segmented messages. The FF shall be assigned the value zero. It does not include an explicit SN in the PCI field but shall be treated as the segment number zero;
- that the SN of the first CF immediately following the FF shall be set to one;
- that the SN shall be incremented by one for each new CF that is transmitted during a segmented message transmission; that the SN value shall not be affected by any FC frame;
- that the SN value shall not be affected by any FC frame;
- that when the SN reaches the value of 15, it shall wraparound and be set to zero for the next CF.

FLOW CONTROL (FC) shall regulate the rate at which CF PDUs are sent to the receiver.

- Block Size (BS) parameter states the absolute number of CF PDUs per block, or rather between two FC PDUs. Only the last block of Consecutive Frames in a segmented data transmission may have less than the BS number of frames.
- Separation Time minimum (STmin) parameter states the minimum time gap allowed between the transmissions of two Consecutive Frame PDUs.

Three distinct types of FC are specified to support the regulation function. The parameterFlow Status (FS) indicates whether the sending network entity can proceed with the message transmission. It may assume the values:

1. 0x0 - ContinueToSend. It shall cause the sender to resume the sending of CF. The meaning of this value is that the receiver is ready to receive a maximum of BS number of Consecutive Frames.
2. 0x1 - Wait. It shall cause the sender to continue to wait for a new FC PDU. The values of BS and STmin in the Flow Control message are not relevant and shall be ignored.
3. 0x2 - Overflow. It shall cause the sender to abort the transmission of a segmented message. This PCI FS parameter value is only allowed to be transmitted in the Flow Control PDU that follows the First Frame PDU and shall only be used if the message length FF-DL of the received First Frame PDU exceeds the buffer size of the receiving entity. The values of BS and STmin in the Flow Control message are not relevant and shall be ignored.

2.1.2.3 *Addressing Formats*

The meaning of the parameters that compose the address information is:

- SA parameter shall be used to encode the sending entity;
- TA parameter shall be used to encode one or multiple (depending on the TAtype: physical or functional) receiving entities;
- TAtype is an extension to the TA parameter. It shall be used to encode the communication model used by the communicating peer entities. Two communication models are specified: 1 to 1 communication, called **physical addressing**, and 1 to n communication,

called **functional addressing**. The second one is supported only in case of Single Frame PDU type;

- AE parameter is used to extend the available address range for large networks, and to encode both sending and receiving Network Layer entities of sub-networks other than the local network where the communication takes place.

The parameter Message type (Mtype) shall be used to identify the type and range of address information parameters. ISO 15765 specifies a range of two values for this parameter:

1. diagnostic, then the address information AI shall consist of the parameters SA, TA, and TAtype;
2. remote diagnostic, then the address information AI shall consist of the parameters SA, TA, TAtype, and AE.

Depending on how the AI parameters are composed to obtain the arbitration ID and which kind of arbitration ID we want to obtain, the standard defines 4 different addressing formats: normal addressing; normal fixed addressing; extended addressing; mixed addressing. Each addressing format requires a different number of CAN frame data bytes to encapsulate the addressing information associated with the data to be exchanged. Consequently, the number of data bytes transported within a single CAN frame depends on the type of addressing format chosen.
Next 4 sections specify the mapping mechanisms for each addressing format, based on the Data Link Layer services and service parameters defined in ISO 11898-1[28].

NORMAL ADDRESSING

For each combination of SA, TA, TAtype and Mtype, a unique CAN identifier is assigned. Correspondence between AI parameters and the CAN arbitration ID is left open. PCI and Data are placed in the CAN frame data field. The mapping showed in Table 2.4 is generally valid when TAtype is physical, while only the first line regarding SF is valid for functional.

CAN ID	CAN frame data field byte pos. 1	2	3	4	5	6	7	8	PDU type
AI	PCI	Data							Single Frame
AI	PCI		Data						First Frame
AI	PCI	Data							Consecutive Frame
AI	PCI		N/A						FlowControl Frame

Table 2.4: Mapping of PDU parameters into CAN frame - Normal addressing

NORMAL FIXED ADDRESSING

Normal fixed addressing is a sub format of normal addressing in which the mapping of the address information into the CAN identifier is further defined. Only 29-bit CAN identifiers are allowed. Table 2.5 defines normal fixed addressing mapping where TAtype is physical, while Table 2.6 defines normal fixed addressing mapping where TAtype is functional.

29-bit CAN ID bit position						CAN frame data field byte pos.								
28-26	25	24	23-16	15-8	7-0	1	2	3	4	5	6	7	8	PDU type
110 b	0	0	218 dec	TA	SA	PCI	Data							Single Frame
110 b	0	0	218 dec	TA	SA	PCI		Data						First Frame
110 b	0	0	218 dec	TA	SA	PCI	Data							Consecutive Frame
110 b	0	0	218 dec	TA	SA	PCI		N/A						FlowControl Frame

Table 2.5: Mapping of PDU parameters into CAN frame - Normal fixed addressing - physical

29-bit CAN ID bit position						CAN frame data field byte pos.								
28-26	25	24	23-16	15-8	7-0	1	2	3	4	5	6	7	8	PDU type
110 b	0	0	219 dec	TA	SA	PCI	Data							Single Frame

Table 2.6: Mapping of PDU parameters into CAN frame - Normal fixed addressing - functional

Note that for this addressing format the difference between physical and functional addressing can be spotted from the CAN arbitration ID, looking to bits from 23 to 16. In first case, we will find a decimal value equal to 218, while in second case we will find it equal to 219.

EXTENDED ADDRESSING

For each combination of SA, TAtype and Mtype, a unique CAN identifier

is assigned. TA is placed in the first data byte of the CAN frame data field. PCI and Data are placed in the remaining bytes of the CAN frame data field. Table 2.7 defines the mapping of PDU parameters into CAN frame where the addressing format is extended

CAN ID	CAN frame data field byte pos. 1	2	3	4	5	6	7	8	PDU type
AI, except TA	TA	PCI	Data						Single Frame
AI, except TA	TA	PCI	Data						First Frame
AI, except TA	TA	PCI	Data						Consecutive Frame
AI, except TA	TA	PCI			N/A				FlowControl Frame

Table 2.7: Mapping of PDU parameters into CAN frame - Extended addressing

MIXED ADDRESSING

Mixed addressing is the addressing format to be used if Mtype is set to remote diagnostics.

Table 2.8 and Table 2.9 define the mapping of the address information (AI) into the 29-bit CAN identifier scheme and the first CAN frame data byte, depending on the target address type (TAtype). PCI and Data are placed in the remaining bytes of the CAN frame data field.

29-bit CAN ID bit position						CAN frame data field byte pos.								PDU type
28-26	25	24	23-16	15-8	7-0	1	2	3	4	5	6	7	8	
110 b	0	0	206 dec	TA	SA	AE	PCI	Data						Single Frame
110 b	0	0	206 dec	TA	SA	AE	PCI		Data					First Frame
110 b	0	0	206 dec	TA	SA	AE	PCI	Data						Consecutive Frame
110 b	0	0	206 dec	TA	SA	AE	PCI			N/A				FlowControl Frame

Table 2.8: Mapping of PDU parameters into CAN frame - Mixed addressing - 29b CAN ID - physical

29-bit CAN ID bit position						CAN frame data field byte pos.								PDU type
28-26	25	24	23-16	15-8	7-0	1	2	3	4	5	6	7	8	
110 b	0	0	205 dec	TA	SA	AE	PCI	Data						Single Frame

Table 2.9: Mapping of PDU parameters into CAN frame - Mixed addressing - 29b CAN ID - functional

Note that also for this addressing format the difference between physical and functional addressing can be spotted from the CAN arbitration ID, looking to bits from 23 to 16. In first case, we will find a decimal value equal to 206, while in second case we will find it equal to 206.
Table 2.10 define the mapping of the address information (AI) into the 11-bit CAN identifier scheme. For each combination of SA, TA and TAtype, a unique CAN identifier is assigned. AE is placed in the first data byte of the CAN frame data field. PCI and Data are placed in the remaining bytes of the CAN frame data field.

	CAN frame data field byte pos.								
CAN ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**PDU type**
AI	AE	PCI	Data						Single Frame
AI	AE	PCI		Data					First Frame
AI	AE	PCI		Data					Consecutive Frame
AI	AE	PCI			N/A				FlowControl Frame

Table 2.10: Mapping of PDU parameters into CAN frame - Extended addressing - 11b CAN ID

2.1.2.4 *Padding*

The DLC of the CAN frame is always set to 8, even if the PDU to be transmitted is shorter than 8 bytes. The sender must pad any unused bytes in the frame. This can be the case for an SF, FC frame or the last CF of a segmented message. The DLC parameter of the CAN frame is set by the sender and read by the receiver to determine the number of data bytes per CAN frame to be processed by the Network Layer. The DLC parameter cannot be used to determine the message length; this information shall be extracted from the PCI information at the beginning of a message.

2.1.3 *UDS Protocol*

2.1.3.1 *Protocol Overview*

ISO 14229-1, or better known as Unified Diacgnostic Services (UDS), is an international standard which defines a diagnostic protocol in the electronic control unit (ECU) environment within the automotive electronics, last documented in ISO 14229-1:2013 [35]. Unified in this context means that it is an international and not a company-specific standard. It is de-

rived from ISO 14230-3, better known as KWP2000 [36], and ISO 15765-3 (Diagnostic Communication over Controller Area Network (DoCAN)) [37]. It is developed with the goal of standardizing different implementations of the predecessor standards and new requirements stemming from further developments in technology and new standards to form one generally valid diagnostic protocol [38].
UDS, when mapped to the OSI model, represents the definition of the Application Layer. When the complete diagnostic communication is mapped on the OSI model, the protocols used by a diagnostic tester (client) and an ECU (server) are broken into the following layers, in accordance with Table 2.11.

Layer	Description	Standard for ISO
8	Diagnostic Application Layer	User
7	Application Layer	ISO 14229-1; ISO 15765-3
6	Presentation Layer	Not applicable
5	Session Layer	ISO 15765-3
4	Transport Layer	ISO 15765-2
3	Network Layer	ISO 15765-2
2	Data Link Layer	ISO 11898-1
1	Physical Layer	ISO 11898-2; ISO 11898-3

Table 2.11: Open Systems Interconnection (OSI) Basic Reference Model in diagnostic

In the automotive industry, Application Layer services are usually referred to as diagnostic services. The Application Layer services are used in client-server based systems (see Figure 2.8) to perform functions such as test, inspection, monitoring or diagnosis of on-board vehicle servers. The client, usually referred to as external test equipment, uses the Application Layer services to request diagnostic functions to be performed in one or more servers. The server, usually a function that is part of an ECU, uses the Application Layer services to send response data, provided by the requested diagnostic service, back to the client. The client is usually an off-board tester, but can in some systems also be an on-board tester. The usage of Application Layer services is independent from the client being an off-board or on-board tester. It is possible to have more than one client in the same vehicle system.
The Application Layer protocol shall always be a confirmed message transmission, meaning that for each service request sent from the client,

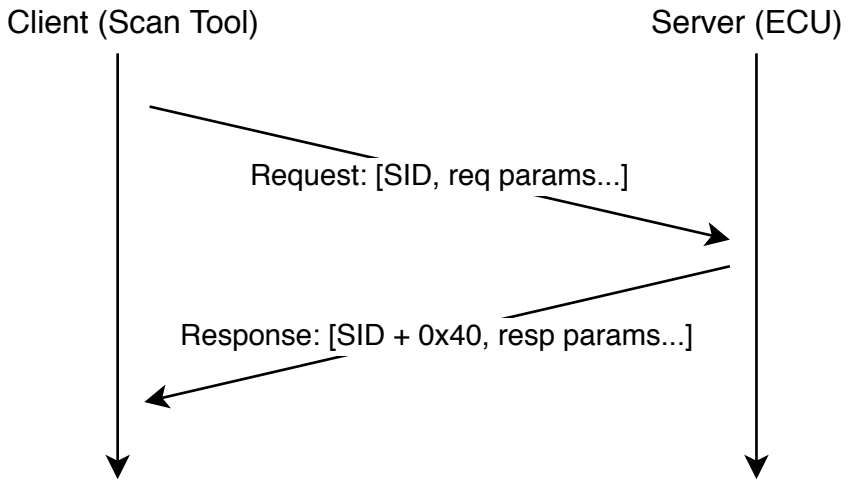


Figure 2.8: UDS client-server communication

there shall be one or more corresponding responses sent from the server. If the request can be successfully completed then a positive response is sent, other ways a negative response should be sent by the server.
The standard supports two different addressing methods, the same seen for ISO-TP protocol: physical addressing and functional addressing. Physical addressing shall always be a dedicated message to a server implemented in one ECU. When physical addressing is used, the communication is a point-to-point communication between the client and the server. When functional addressing is used, the communication is a broadcast communication from the client to a server implemented in one or more ECUs.
For service requests, TA represents the server identifier for the server that shall perform the requested diagnostic service. The server shall support its list of diagnostic services regardless of addressing mode (physical or functional addressing type).
Negative response messages with negative response codes of SNS (serviceNotSupported), SNSIAS (serviceNotSupportedIn- ActiveSession), SFNS (sub-functionNotSupported), SFNSIAS (sub-functionNotSupportedInActive-Session), and ROOR (requestOutOfRange) shall not be transmitted when functional addressing was used for the request message.
UDS specifies that a CAN data frame shall always contain 8 bytes so if a request message does not fill eight bytes the remaining bytes are filled with 0x55. Response messages are filled with 0xAA instead of 0x55.

2.1.3.2 *Diagnostic Services*

Diagnostic services have a common message format and each one is identified by a unique Service IDentifier (SID). Each service defines a Request Message, a Positive Response Message, and a Negative Response Message.
The Request Message has the SID as first byte, plus additional service-defined parameters.
The Positive Response Message has the SID with bit 6 set to one (this means adding 0x40 to the original SID) as first byte, plus the service-defined response parameters.
The Negative Response Message is usually a three-byte message: it has the Negative Response SID (0x7F) as first byte, the original SID as second byte, and a NegativeResponseCode as third byte. The UDS standard partly defines the NegativeResponseCodes, but there is room left for manufacturer-specific extensions.
The SID values for each service request and respective response are assigned according to ranges defined in Table 2.12. Each request service shall be assigned a unique SID value. Each positive response service shall be assigned a corresponding unique SID value.

SID range	Service type	Where defined
0x10 - 0x3E	Service requests	ISO 14229-1
0x50 - 0x7E	Service positive responses	ISO 14229-1
0x7F	Negative response service identifier	ISO 14229-1
0xBA - 0xBE	Service requests	Defined by system supplier
0xFA - 0xFE	Service positive responses	Defined by system supplier
0x83 - 0x88	Service requests	ISO 14229-1
0xC3 - 0xC8	Service positive responses	ISO 14229-1

Table 2.12: Services SID range definition

The standards on automotive diagnostics define many different services for many purposes. Unfortunately, most services leave a large amount of room for manufacturer-specific variants and extensions. We provide in next sections a detailed overview of the main services employed during this work and how they are structured.

SERVICE DIAGNOSTICSESSIONCONTROL

A diagnostic session enables a specific set of diagnostic services and/or

functionality in the server. This service provides the server with the capability to report Data Link Layer specific parameter values valid for the enabled diagnostic session (e.g. timing parameter values). The exact set of services and/or functionality enabled in each diagnostic session is left open by the standard and should be defined by the supplier. There shall always be exactly one diagnostic session active in a server. A server shall always start the default diagnostic session when powered up. If no other diagnostic session is started, then the default diagnostic session shall be running as long as the server is powered. If the client has requested a diagnostic session, which is already running, then the server shall send a positive response message and behave as showed in Figure 2.9 that describes the server internal behavior when transitioning between sessions. The set of diagnostic services and diagnostic functionality in a non- default diagnostic session (excluding the programmingSession) is a superset of the functionality provided in the defaultSession, which means that the diagnostic functionality of the defaultSession is also available when switching to any non-default diagnostic session. A session can enable vehicle manufacturer specific services and functions, which are not specified by the standard.

The data field of a DiagnosticSessionControl request is composed as

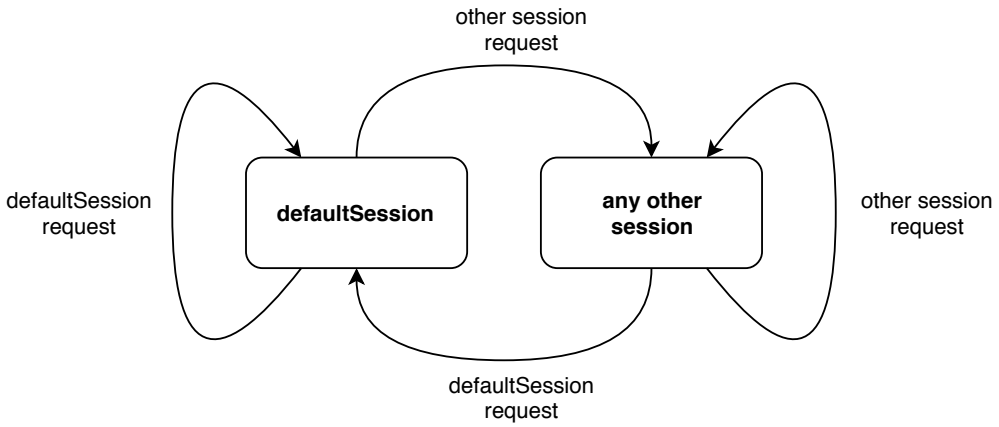


Figure 2.9: Finite state machine of diagnostic session transitions

showed in Table 2.13.

Data byte	Parameter name	Byte value	
#1	DiagnosticSessionControl Request SID	0x10	Mandatory
#2	diagnosticSessionType	0x00 - 0xFF	Mandatory

Table 2.13: DiagnosticSessionControl request definition

The possible values for parameter diagnosticSessionType are summarized in Table 2.14.

Value	Description
0x00	Reserved by the standard
0x01	**defaultSession**
0x02	**programmingSession**
0x03	**extendedDiagnosticSession**
0x04	**safetySystemDiagnosticSession**
0x05 - 0x3F	Reserved by the standard
0x40 - 0x5F	**vehicleManufacturerSpecific**
0x60 - 0x7E	**systemSupplierSpecific**
0x7F	Reserved by the standard

Table 2.14: diagnosticSessionType parameter definition

The data field of a DiagnosticSessionControl response is composed as showed in Table 2.15.

Data byte	Parameter name	Byte value	
#1	DiagnosticSessionControl Response SID	0x50	Mandatory
#2	diagnosticSessionType	0x00 - 0xFF	Mandatory
#3 - #4	P2Server_max	0x0000 - 0xFFFF	Mandatory
#5 - #6	P2*Server_max	0x0000 - 0xFFFF	Mandatory

Table 2.15: DiagnosticSessionControl response definition

P2Server_max and P2*Server_max session parameter records are respectively the default timing supported by the server for the activated diagnostic session and the enhanced timing supported by the server for the activated diagnostic session. First one has resolution equal to 1-ms, minimum value of 0-ms and maximum value of 65,535-ms. Second one has resolution equal to 10-ms, minimum value of 0-ms and maximum value of 65,5350-ms. Practically both indicates the time to wait for a response after a successive request. P2*Server_max must be considered if a negative response with negativeResponseCode equal to 0x78 (which identifies the requestCorrectlyReceived-ResponsePending negative response type) has been received after the request.
The following Negative Response Code (NRC) shall be implemented for this service. The circumstances under which each response code would

occur are documented in Table 2.16. The listed negative responses shall be used if the error scenario applies to the server.

NRC	Description
0x12	**sub-functionNotSupported** This NRC shall be sent if the diagnosticSessionType is not supported.
0x13	**incorrectMessageLengthOrInvalidFormat** This NRC shall be sent if the length of the message is wrong.
0x22	**conditionsNotCorrect** This NRC shall be returned if the criteria for the request DiagnosticSessionControl are not met.

Table 2.16: DiagnosticSessionControl negative response codes supported

SERVICE READDATABYIDENTIFIER

The ReadDataByIdentifier service allows the client to request data record values from the server identified by one or more Data IDentifier (DID). The client request message contains one or more two byte dataIdentifier values that identify data records maintained by the server. The format and definition of the dataRecord contained in the response is vehicle manufacturer or system supplier specific. The server may limit the number of dataIdentifiers that can be simultaneously requested as agreed upon by the vehicle manufacturer and system supplier.
The data field of a ReadDataByIdentifier request is composed as showed in Table 2.17.

Data byte	Parameter name	Byte value	
#1	ReadDataByIdentifier Request SID	0x22	Mandatory
#2 - #3	dataIdentifier	0x0000 - 0xFFFF	Mandatory
...	...	...	...
#n-1 - #n	dataIdentifier#n	0x0000 - 0xFFFF	Optional

Table 2.17: ReadDataByIdentifier request definition

The data field of a ReadDataByIdentifier response is composed as showed in Table 2.18.

Data byte	Parameter name	Byte value	
#1	ReadDataByIdentifier Response SID	0x62	Mandatory
#2 - #3	dataIdentifier	0x0000 - 0xFFFF	Mandatory
#4 - #(k-1)+4	data#1, ..., data#k	foreach data#i 0x00 - 0xFF	Mandatory
...	...	...	...
#n-(o-1)-2 - #n-(o-1)-2	dataIdentifier#n	0x0000 - 0xFFFF	Mandatory
#n-(o-1) - #n	data#1, ..., data#o	foreach data#i 0x00 - 0xFF	Mandatory

Table 2.18: ReadDataByIdentifier response definition

The following NRC shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 2.19. The listed negative responses shall be used if the error scenario applies to the server.

NRC	Description
0x13	**incorrectMessageLengthOrInvalidFormat** This NRC shall be sent if the length of the request message is invalid or the client exceeded the maximum number of dataIdentifiers allowed to be requested at a time.
0x14	**responseTooLong** This NRC shall be sent if the total length of the response message exceeds the limit of the underlying transport protocol (e.g., when multiple DIDs are requested in a single request).
0x22	**conditionsNotCorrect** This NRC shall be returned if the criteria for the request DiagnosticSessionControl are not met.
0x31	**requestOutOfRange** This NRC shall be sent if none of the requested dataIdentifier values are supported in the current session.
0x33	**securityAccessDenied** This NRC shall be sent if at least one of the dataIdentifiers is secured and the server is not in an unlocked state.

Table 2.19: ReadDataByIdentifier negative response codes supported

SERVICE ECURESET

The ECUReset service is used by the client to request a server reset. This service requests the server to effectively perform a server reset based on the content of the resetType parameter value embedded in the ECUReset request message. The ECUReset positive response message shall be sent before the reset is executed in the server. After a successful server reset the server shall activate the defaultSession.
ISO 14229-1 does not define the behavior of the ECU from the time following the positive response message to the ECU reset request until the reset has successfully completed. It is recommended that during this time the ECU does not accept any request messages and send any response messages.
The data field of a ECUReset request is composed as showed in Table 2.20.

Data byte	Parameter name	Byte value	
#1	ECUReset Request SID	0x11	Mandatory
#2	resetType	0x00 - 0xFF	Mandatory

Table 2.20: ECUReset request definition

The possible values for parameter resetType are summarized in Table 2.21.

Value	Description
0x00	Reserved by the standard
0x01	**hardReset** Simulates a shutdown of the power supply.
0x02	**keyOffOnReset** Simulates the drain and turn on the ignition with the key.
0x03	**softReset** Immediately restart the application program if applicable.
0x04	**enableRapidPowerShutDown**
0x05	**disableRapidPowerShutDown**
0x06 - 0x3F	Reserved by the standard
0x40 - 0x5F	**vehicleManufacturerSpecific**
0x60 - 0x7E	**systemSupplierSpecific**
0x7F	Reserved by the standard

Table 2.21: resetType parameter definition

The data field of a ECUReset response is composed as showed in Table 2.22.

Data byte	Parameter name	Byte value	
#1	ECUReset Response SID	0x51	Mandatory
#2	resetType	0x00 - 0xFF	Mandatory
#3	powerDownTime	0x00 - 0xFF	Mandatory

Table 2.22: ECUReset response definition

The parameter powerDownTime indicates to the client the minimum time of the stand-by-sequence the server will remain in the power down sequence. The resolution of this parameter is one (1) second per count. The following values are valid:

- 0x00 - 0xFE: 0 - 254 seconds powerDownTime;
- 0xFF: indicates a failure or time not available.

The following NRC shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 2.23. The listed negative responses shall be used if the error scenario applies to the server.

NRC	Description
0x12	**sub-functionNotSupported** This NRC shall be sent if the diagnosticSessionType is not supported.
0x13	**incorrectMessageLengthOrInvalidFormat** This NRC shall be sent if the length of the message is wrong.
0x22	**conditionsNotCorrect** This NRC shall be returned if the criteria for the request DiagnosticSessionControl are not met.
0x33	**securityAccessDenied** This NRC shall be sent if the requested reset is secured and the server is not in an unlocked state.

Table 2.23: ECUReset negative response codes supported

SERVICE SECURITYACCESS

The purpose of this service is to provide a means to access data and/or diagnostic services, which have restricted access for security, emissions, or safety reasons. Diagnostic services for downloading/uploading routines or data into a server and reading specific memory locations from a server are situations where security access may be required. Improper routines or data downloaded into a server could potentially damage the electronics or other vehicle components or risk the vehicle's compliance to emission, safety, or security standards. The security concept uses a seed and key relationship. A typical example of the use of this service is as showed in Figure 2.10.

The client shall request the server to "unlock" by sending the service

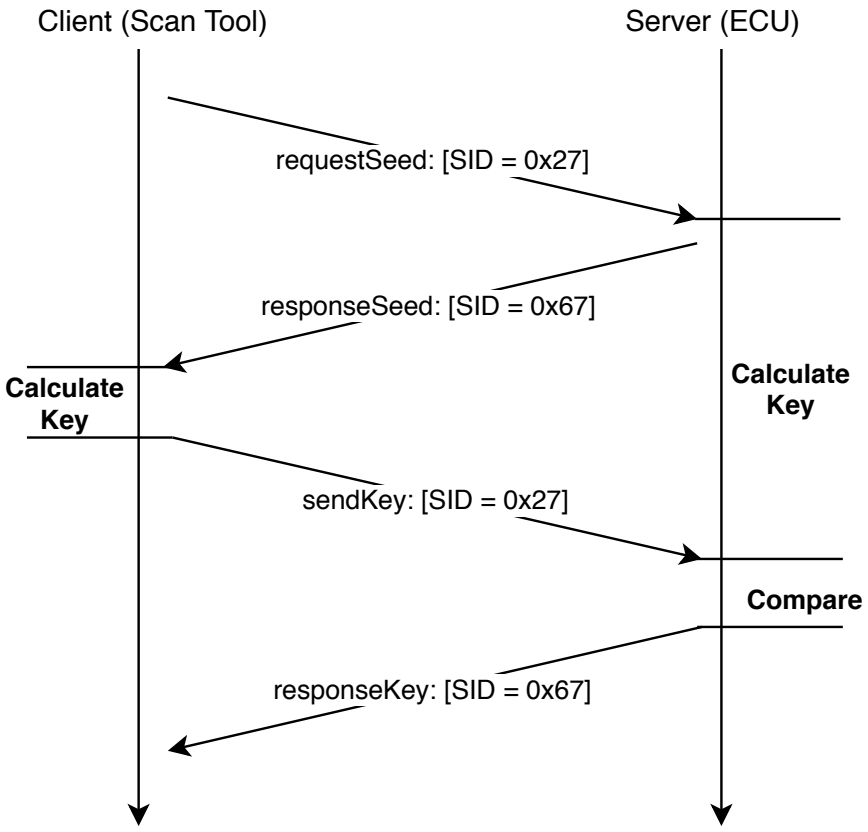


Figure 2.10: Messages exchanged during SecurityAccess procedure

SecurityAccess 'requestSeed' message. The server shall respond by sending a "seed" using the service SecurityAccess 'requestSeed' positive response message. The client shall then respond by returning a "key" number back to the server using the appropriate service SecurityAccess 'sendKey' request message. The server shall compare this "key" to one internally stored/calculated. If the two numbers match, then the server shall

enable ("unlock") the client's access to specific services/data and indicate that with the service SecurityAccess 'sendKey' positive response message. If the two numbers do not match, this shall be considered a false access attempt. An invalid key requires the client to start over from the beginning with a SecurityAccess 'requestSeed' message. Attempts to access security shall not prevent normal vehicle communications or other diagnostic communication.
Since this service is not directly employed by what we've implemented we do not enter in the details of the messages structure, which can be examined in ISO 14229-1.

SERVICE COMMUNICATIONCONTROL

The purpose of this service is to switch on/off the transmission and/or the reception of certain messages of a server. The data field of a CommunicationControl request is composed as showed in Table 2.24.

Data byte	Parameter name	Byte value	
#1	CommunicationControl Request SID	0x28	Mandatory
#2	controlType	0x00 - 0xFF	Mandatory
#3	communicationType	0x00 - 0xFF	Mandatory
#4 - #5	nodeIdentificationNumber	0x0000 - 0xFFFF	Optional

Table 2.24: CommunicationControl request definition

The parameter controlType contains information on how the server shall modify the communication type referenced in the communicationType parameter. The possible values for parameter controlType are summarized in Table 2.25.

Value	Description
0x00	**enableRxAndTx** The reception and transmission of messages shall be enabled.
0x01	**enableRxAndDisableTx** The reception of messages shall be enabled and the transmission shall be disabled.
0x02	**disableRxAndEnableTx** The reception of messages shall be disabled and the transmission shall be enabled.
0x03	**disableRxAndTx** The reception and transmission of messages shall be disabled.
0x04	**enableRxAndDisableTxWithEnhancedAddressInformation** The addressed bus master shall switch the related sub-bus segment to the diagnostic-only scheduling mode.
0x05	**enableRxAndTxWithEnhancedAddressInformation** The addressed bus master shall switch the related sub-bus segment to the application scheduling mode.
0x06 - 0x3F	Reserved by the standard
0x40 - 0x5F	**vehicleManufacturerSpecific**
0x60 - 0x7E	**systemSupplierSpecific**
0x7F	Reserved by the standard

Table 2.25: controlType parameter definition

Parameter communicationType is used to reference the kind of communication to be controlled. The communicationType parameter is a bit-code value, which allows controlling multiple communication types at the same time. Parameter nodeIdentificationNumber s used to identify a node on a sub-network somewhere in the vehicle, which cannot be addressed using the addressing methods of the lower OSI layers 1 to 6. This parameter is only present, if the sub-function parameter controlType is set to 0x04 or 0x05.
The data field of a CommunicationControl response is composed as showed in Table 2.26.

Data byte	Parameter name	Byte value	
#1	ECUReset Response SID	0x68	Mandatory
#2	controlType	0x00 - 0xFF	Mandatory

Table 2.26: CommunicationControl response definition

The following NRC shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 2.27. The listed negative responses shall be used if the error scenario applies to the server.

NRC	Description
0x12	**sub-functionNotSupported** This NRC shall be sent if the diagnosticSessionType is not supported.
0x13	**incorrectMessageLengthOrInvalidFormat** This NRC shall be sent if the length of the message is wrong.
0x22	**conditionsNotCorrect** Used when the server is in a critical normal mode activity and therefore cannot disable/enable the requested communication type.
0x31	**requestOutOfRange** The server shall use this response code, if it detects an error in the communicationType or nodeIdentificationNumber parameter.

Table 2.27: CommunicationControl negative response codes supported

SERVICE READMEMORYBYADDRESS

The ReadMemoryByAddress service allows the client to request memory data from the server via provided starting address and size of memory to be read. The ReadMemoryByAddress request message is used to request memory data from the server identified by the parameter memoryAddress and memorySize. The number of bytes used for the memoryAddress and memorySize parameter is defined by addressAndLengthFormatIdentifier. It is also possible to use a fixed addressAndLengthFormatIdentifier and unused bytes within the memoryAddress or memorySize parameter are padded with the value 0x00 in the higher range address locations. In case of overlapping memory areas, it is possible to use an additional memoryAddress byte as a memory identifier. The server sends data record values via the ReadMemoryByAddress positive response message. The format and definition of the dataRecord parameter shall be vehicle manufacturer specific. The dataRecord parameter may include internal data and system status information if supported by the server.

Since this service is not directly employed by what we've implemented

we do not enter in the details of the messages structure and parameter definition, which can be examined in ISO 14229-1.

SERVICE TESTERPRESENT
This service is used to indicate to a server (or servers) that a client is still connected to the vehicle and that certain diagnostic services and/or communications that have been previously activated have to remain active. This service is used to keep one or multiple servers in a diagnostic session other than the defaultSession. This can either be done by transmitting the TesterPresent request message periodically or in case of the absence of other diagnostic services to prevent the server(s) from automatically returning to the defaultSession.
Since this service is not directly employed by what we've implemented we do not enter in the details of the messages structure and parameter definition, which can be examined in ISO 14229-1.

2.1.3.3 *NegativeResponse*

When an UDS service request cannot be successfully completed the server shall response with a NegativeResponse. These responses have always SID equal to 0x7F, reports the SID of the original request in byte two and as third byte reports the Negative Response Code (NRC), which indicates the failure cause. The general structure of a NegativeResponse is the one showed in Table 2.28.

Data byte	Parameter name	Byte value	
#1	NegativeResponse SID	0x7F	Mandatory
#2	Original Request SID	0x00 - 0xFF	Mandatory
#3	NRC	0x00 - 0xFF	Mandatory

Table 2.28: NegativeResponse definition

Each diagnostic service specifies applicable negative response codes. The diagnostic service implementation in the server may also utilize additional and applicable negative response codes specified in this as defined by the vehicle manufacturer.

2.1.3.4 *FunctionalGroupIdentifiers*

The FunctionalGroupIdentifier specifies different functional system groups. The identifier is used to distinguish commands sent by the test equip-

ment between different functional system groups within an electrical architecture which consists of many different servers.
The main purpose is to be able to query information specific to a functional system group, indifferently to the ECU which is able to respond. An ECU may be part of several functional system groups. The functional group identifiers defined by the standard are reported in Table 2.29.

Byte value	Description
0x00 - 0x32	Reserved by the standard SID
0x33	**Emissions-system group** This value identifies the Emissions system in a server.
0x34 - 0xCF	Reserved by the standard SID
0xD0	**Safety-system group** This value identifies the Safety system in a server.
0xD1 - 0xDF	**Legislative system group** This range of values is reserved for legislative required group identifiers by this document for future definition.
0xE0 - 0xFD	Reserved by the standard SID
0xFE	**VOBD system** This value identifies the VOBD system device. Depending on the VOBD strategy which is implemented, only a gateway, a dedicated VOBD ECU or any other ECU which has the VOBD function implemented (e.g. engine controller) may respond.
0xFF	**All functional system groups** This value identifies all functional system groups as listed in this table in a server.

Table 2.29: FunctionalGroupIdentifiers definition

2.2 ECU DISCOVERY AND VULNERABILITY ASSESSMENT

One of the first time-consuming activities to be performed when approaching to make a security analysis of an unknown automobile is to realize which systems it implements, how these are implemented and which ECU is responsible for performing each functionality of the automobile. All these analyses are meant to individuate what and how many are the weakness of the systems that an attacker could possibly exploit. The literature often presents techniques that employs little automated

tools and a lot of manual reverse engineering to perform those tasks, for example 'THE CAR HACKER'S HANDBOOK' book [39] devolves two chapters to present the reverse engineering techniques meant to individuate the ECUs and the UDS servers available on the automobile CAN bus.
Researches published in order to present a vulnerability or a criticality about automotive often starts by presenting an overview of the automobile available systems and employed ECUs [15]. Most of the times the presentation of the methods employed in order to reconstruct that information is missing from the published paper.
Many times, the vulnerabilities discovered by researchers exploits structural weakens of the protocols used in the automotive systems, for example the denial of service attack showed by Andrea Palanca [40]. In those cases, is hard, or even impossible, to fix the weakens during an automobile lifetime. Even if the vulnerabilities are related to applicative software (i.e. implementation of UDS services), like the one identified by us and described in Section 3.7.1, the diffusion of updates to fix them is not so simple. The Over The Air (OTA) firmware upgrade is making progress [41] but it is still confined to a niche of models [42]. To date, in most situations, it will require a recall campaign of the involved models [43]–[45]. On top of that the majority of the discovered vulnerabilities exploits generic weakness of the employed protocols, which most probably are spread across different manufacturer besides of the ones of the tested automobiles for the purpose of demonstrating the research results.
That said, it is very probable that even if a vulnerability is known it remains in the involved systems for a long time after disclosure, hence the necessity to have a vulnerability test framework which allows in a structured manner to develop test for known vulnerabilities. In this way would be quick and easy to get a complete overview of the known vulnerabilities presents in ad automotive systems and test for their presence when approaching a security analysis of a new vehicle.
The increasing availability of features and safety systems in modern vehicles, especially ones belonging to luxury segment, imply high complexity of modern vehicles technologies and the presence of many ECUs. Modern cars are equipped with an average of 50 ECUs, number which leap to 150 for luxury automobiles [46] and falls to 15 or less for low end market vehicles. In such a scenario becomes increasingly difficult to manually, or with the aid of partial and semi-automatic tools, carry out the task aforementioned.

2.3 STATE OF ART

The currently availability of tools which response to the stated problem is very limited. The most popular tools at the moment are:

- Caring Caribou: it is presented by its authors as a friendly, zero-knowledge, car security exploration tool [47];
- CANToolz: it is presented by its authors as a framework for analysing CAN networks and devices [48].

To realize the currently offered possibilities we tried and tested the Caring Caribou tool. It offers a module called DCM which can perform brute-force of the 11-bit arbitration IDs to discover the available UDS services. In order to do this, it sends the UDS DiagnosticSessionControl service request, with defaultSession set as diagnosticSessionType parameter, on each arbitration ID present in a range of specified values and reports any response (positive or negative). The 'THE CAR HACKER'S HANDBOOK' book [39] provides an example of use of this tool when first trying to understand the systems present on an automobile. The tool does not support any kind of discovery procedure for 29-bit arbitration IDs.
We performed the provided scanning procedure on the Audi A3 Sportback automobile and obtained a scanning time of 4 minutes and 21 seconds. The output shows fourteen pairs of discovered CAN arbitration IDs, with no other information about the belonging ECU. Since the most limiting parameter in this test is the CAN bus bit rate we can safely assume that the performances of the USB-to-CAN device and the employed PC are negligible for evaluating the result in terms of time.
The time employed for performing the brute-force scanning procedure for 11-bit CAN arbitration IDs can be reputed as acceptable. Unfortunately, the tool does not provide a similar procedure for 29b arbitration IDs and if we try to run the same procedure with the range of values admitted by 29 bits (0x00000000 to 0x1FFFFFFF) the tool gets stuck.
Since the only relevant difference between the two types of arbitration ID is the length, we can realistically estimate the time needed to brute-force a 29-bit arbitration ID by establishing a relation, based on the arbitration ID length, with the time needed for brute-force a 11-bit arbitration ID. In this way, we estimate that the brute-force operation for a 29-bit arbitration ID would take approximately 2 years, 61 days and 21 hours. This is no more acceptable for a first approach to the security analysis of an automobile.

CANToolz tool states itself to be an attempt to unify most off the tricks, tools and other things that one would need to do CAN analysis in one unique place. For what regarding the ECU discovery process it provides an example [49] which from the point of view of CAN communications seams to perform almost the same concrete operations described for Caring Caribou. Any way it performs for sure a brute-force procedure since the range of arbitration IDs can be specified. Unfortunately, it does not provide support for the specific USB-to-CAN devices that we have available and we could not test it.
All seen tools rely on brute-force in order to discover the communication CAN IDs of the ECUs present on the CAN bus, while none of them offers any sort of information or automatic vulnerability assessment on the ECUs present on the automobile.
As we have showed the brute-force approach may be practicable if 11-bit CAN arbitration IDs are employed but it takes definitely too much if the target automobile employees 29-bit CAN arbitration IDs. On top of that any kind of brute-force procedure is easily detectable by a local observer node (since the CAN bus is intrinsically broadcast), ore a gateway through which the messages of the OBD-II port could be forced to pass, noticing the presence on the bus of multiple CAN frames with same Data field but sent with multiple, usually progressive, arbitration IDs. Those actors could make ineffective any effort in performing an ECU discovery procedure by brute-force.

2.4 CMAP

This thesis presents a novel technique which allows to list the ECUs present on an automobile CAN bus. For each discovered ECU, we provide a set of information useful to establish the exact purpose of the ECU in the automobile systems. After that we propose a framework which facilitates the development and putting in practice of tests for known vulnerabilities. The framework gives the ability to apply to each discovered ECU the available set of tests to reveal the presence of known vulnerabilities.
This leads to the implementation of cmap, an automatic tool which allows to have a quick, yet complete, panorama of the systems present on the automobile CAN network and to do vulnerability assessment for each of them, facilitating the process of finding and measuring the severity of vulnerabilities. These significantly reduce the time and effort for

the first approach, when performing a security analyse of a new automobile.
The problems discussed regarding the currently available solutions are overcame by our tool since it does not rely on brute-force to discover the ECUs present on the bus. Its activity cannot be distinguished from that of a normal diagnostic tester since it does not generate abnormal traffic, neither in quantity nor in content. It also proposes an integrated structured framework to implement and subsequently apply tests for known vulnerabilities.
The major difficulties encountered during this work were related to find a common model for performing discovery of the ECUs on the CAN bus without rely on brute-force. This was challenging due to following causes:

- often the information about diagnostic protocols and their implementations by different manufacturer is proprietary and not accessible;
- the little information that we could find was often fragmented and incomplete;
- different car manufacturer implements proprietary diagnostic protocols, not sticking to the standard ones;
- the availability of automobiles dedicated to testing purposes was limited.

3

APPROACH

3.1 INTRODUCTION

This chapter describes why we chose UDS diagnostic protocol and how the protocol specifications and common methods of implementation are exploited in order to take advantage for the development of our tool. Afterwards we present how we have structured the vulnerability test framework and analyse the vulnerability for which we have implemented a test (to reveal it if present) based on the framework itself.

3.2 WHY UDS PROTOCOL

UDS protocol aims to be spread across multiple manufacturers since it is an international standard, not a proprietary one. It is derived from ISO 15765-3 and KWP2000 and has been designed to become the successor of the latter.
UDS is part of the AUTomotive Open System ARchitecture (AUTOSAR) specifications [50]. AUTOSAR is a worldwide development partnership of vehicle manufacturers, suppliers, service providers and companies from the automotive electronics, semiconductor and software industry [51]. Between the AUTOSAR partners we find names such BMW, BOSH, Ford, General Motors, Toyota, Volkswagen, Continental, Daimler [52]. AUTOSAR main goals are to increase scalability and flexibility, re-use of software, improve containment of product and process complexity and risk. Being considered a standard by a partnership which comprises the major automotive manufacturers is definitely an indicator of the validity of the UDS protocol.
UDS has been developed and supported to become a new standard in the automotive field, which is leading the manufacturer toward its development in newer ECUs [8], [53]–[55].
The most significant alternative protocols available are:

- VPW (Variable pulse width): proprietary protocol used by General Motors;
- PWM (Pulse-width modulation): proprietary protocol used by Ford.

- ISO 9141 [56]. Very old, definitely outdated;
- KWP2000 [57]. Still present among different manufacturers, especially in the low-end market and city cars. Since it is the base for UDS protocol it will be soon replaced by this last one;
- OBD-II the amount of diagnostic information is limited to the emission related one;

We have discarded VPW and PWM protocols since they are proprietary solution, adopted only by a limited number of manufacturers and which is not probably going to increase since the proprietary nature of the protocol itself, while UDS is an international standard and could potentially be adopted by any manufacturer.
ISO 9141 is very old, its specification dates to 1989, and is based on K-Line bus technology[1] which is becoming obsolescent [58] in favour of CAN [59], which indeed is the bus technology on which UDS works.
We didn't chose KWP2000 even if is still spread across different manufacturer because UDS is its natural evolution and presents several advantages and improvements compared to its predecessor [60], we report some of them:

- UDS provides event-driven and periodic services;
- involves 2-byte DIDs and allows higher number of IDs to be exchanged.
- the specified data identifiers are more comprehensive;
- does not include a separate service to identify the ECU,like KWP2000 does, rather this functionality is included within the ReadDataByIdentifier service.

On top of that the KWP2000 protocol supports only CAN and K-Line bus systems. The UDS protocol is designed to be independent of the underlying vehicle network as it supports a range of bus systems. In fact, there is a part of the standard ISO 14229 for the most common automotive bus technologies:

1. *ISO 14229-4:2012 Road vehicles – Unified diagnostic services (UDS) – Part 4: Unified diagnostic services on FlexRay implementation* [61]

1 The K-Line is a single-wire connection and thus a serial interface, which is directed via a data strand and has a ground connection. It is practically a bidirectional single-wire bus serving data transfer between elements of a network in automobile technology

2. *ISO 14229-5:2013 Road vehicles – Unified diagnostic services (UDS) – Part 5: Unified diagnostic services on Internet Protocol implementation* [62]

3. *ISO 14229-7:2015 Road vehicles – Unified diagnostic services (UDS) – Part 7: UDS on local interconnect network* [63]

UDS is the underlying layer of OBD-II, the last one is just a subset of the former one. While OBD-II has a limited set of services, UDS is the diagnostic protocol that provides advanced features like calibration, access to memory areas of the ECU and even flashing firmware for ECU reprogramming.
By now this communication protocol is used in almost all new ECUs made by Tier 1 suppliers of Original Equipment Manufacturer (OEM) [38]. It can provide the information for emission related systems, which are mandatory in the European Union [64].
All these considerations lead us to analyse UDS protocol and to choose it as the underlining protocol on which to base our tool.

3.3 FUNCTIONAL ADDRESSING FOR RECEIVING ARBITRATION ID

In relation to the specification we focused on for the developing of our tool, functional addressing feature provided by UDS protocol is a crucial point since allows broadcast communication between a client and multiple UDS servers. Usually this functionality is not actually considered. Sometimes it is just used in order to send the TesterPresent service request by the diagnostic tester client to all the ECUs, in order to indicate that it is still connected and to maintain active the requested diagnostic session. The fact that this functionality is largely ignored is justified by the fact that normally all the diagnostic testers rely on databases in [65] format, which is a proprietary format that describes the data over a CAN bus, including the arbitration which should be used for diagnostic communication with each ECU. Therefore, the tester already knows all the arbitration IDs to use with each ECU when performing its tasks.
Exploiting this functionality, we are able to get multiple ECUs responding to a single request made on the functional address, from those responses we can infer the actual CAN arbitration ID used by each server and then, applying different methods depending of the addressing format employed, we compute the arbitrary ID used for transmitting messages to the server. The knowledge of the CAN arbitration IDs is the first and essential information to known in order to be able to establish

a communication with each ECU for gathering information and perform the tests for the known vulnerabilities.

3.4 11-BIT CAN IDS - TRANSMISSION ARBITRATION ID

When 11-bit CAN arbitration IDs are employed for diagnostic communications, the ISO-TP addressing method used is the Normal addressing. Unfortunately, this addressing method does not directly specifies neither the functional addresses neither the mapping between the AI field and the CAN arbitration ID. In this section, we describe the approach adopted in order to get the transmission ID for the ECUs on the CAN bus.
UDS uses a specific address previously used by ISO 15765-4 [66] for emission related systems which is 0x7DF. Since the emission related system is based on UDS protocol the functional address is not strictly limited only to those inherent services, but can be also used for all the services made available by the UDS protocol implementation.
Studying and analyzing different manufacturer UDS implementations on real cars systems we discovered that another popular functional address is 0x700. This is the base arbitration ID of the diagnostic range of 11b CAN IDs, as showed in Table 4.1. To trigger a response from the ECUs on the CAN bus we make an UDS DiagnosticSessionControl request, of type diagnosticSession, using each functional address identified. For clarifying what an UDS service request made on a functional address means and how the standard definitions are concretized in CAN frames, we report one of the manual tests we have done on the Audi A3 Sportback. This is the car that we have mostly exploited for the tests.
We manually send a CAN frame[2] for UDS DiagnosticSessionControl service request on the functional address 0x700, the full CAN frame is showed in Listing 3.1.

```
ID=0x700,DLC=0x8,Data=[0x02,0x10,0x01,0x55,0x55,0x55,0x55,0x55]
```

Listing 3.1: Manual DiagnosticSessionControl request on functional address

The first byte of the Data segment is the PCI information, it states that we are sending a SingleFrame (first four bits are 0) which data length of 2 bytes (last four bits decimal equivalent is 2). Second byte is the SID of the UDS service being requested, which in case of DiagnosticSessionControl

2 For the details of what hardware device we employed to connect to the car CAN bus see Section 5.1. For the detail of how to employ it with SocketCAN see Appendix A.2

is precisely 0x10. The third byte is the session being requested, in our example corresponds to 0x01 which is the defaultSession. The rest of the Data bytes contained in the frame are simply filled with the padding value.
Since our frame employs a functional address as arbitration ID, we get multiple UDS response frames, from different ECU servers with different arbitration IDs. All the CAN frames obtained as response are showed in Listing 3.2.

```
ID=0x77E,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x00,0xC8,0xAA]
ID=0x7B4,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x7B5,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x7E8,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0x55]
ID=0x7E9,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x7B0,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x77C,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x776,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x778,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x774,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x77F,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x77A,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x7DD,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
ID=0x77D,DLC=0x8,Data=[0x06,0x50,0x01,0x00,0x32,0x01,0xF4,0xAA]
```

Listing 3.2: DiagnosticSessionControl responses on functional address request

All UDS response are positive, note that second byte Data field is equal to the requested SID + 0x40 which indeed means a positive response. At this point we know that on the car CAN bus fourteen different ECUs implement and UDS server and each one of these servers was successfully able to activate the diagnostic session of type defaultSession. The bytes #4,#5 and #6,#7 indicates respectively the values of parameters P2Server_max and P2*Server_max. For example, ECU responding with arbitration ID 0x77E communicates the values of 50ms for P2Server_max and 2000ms for P2*Server_max.
We did not found any reasonable explanation for the different padding value used by the ECU responding on address 0x7E8. This is not an issue since our tool does not make any check on the padding value of the responses, it just extracts the number of data byte specified as data length in the PCI byte.
Since the ISO-TP protocol limits the use of functional addressing to frames of type SF we cannot use the functional address for requests which implies longer responses, composed by FF and CF frames. In or-

der to obtain the physical address needed for transmitting to each ECU we start from the arbitration ID used for the response and apply a simple heuristic based on conventions and observed behaviors. We try to guess the transmission arbitration ID by adding or subtracting to the response arbitration ID standard offset values. The offset values employed were collected by direct observation in the analysed car networks or seen in other papers which have done sundry tests on other car manufacturers. These are: 0x6A, 0x40, 0x16, 0x8, 0x1. In order to validate each guessed transmission arbitration ID, we make an UDS request of service ReadDataByIdentifier, using the DID 0xF19E. If we obtain a positive response means that we got the correct transmission physical arbitration ID for communicating whit that ECU and we can assume the data contained in the response as the ECU name, interpreting the hexadecimal values as ASCII characters. Even with a NegativeResponse we confirm the transmission arbitration ID, since there is no difference in the arbitration IDs employed. But in this case, we do not get the ECU name.
As last action, we move the used offset value at the top of the list, since usually the majority of the ECUs present on a CAN bus employs the same offset between transmission and response arbitration IDs. In this way, we should need less attempts in order to discover the transmission arbitration IDs of next ECUs.
We employ the DID 0xF19E for the ReadDataByIdentifier request since the Annex C.1 of ISO 14229-1 standard specifies that this DID should represent the 'ODXFileDataIdentifier'. This value is used to reference the ODX (Open Diagnostic Data Exchange) file of the server to be used to interpret and scale the server data. We think it can be reasonably assumed as the ECU name, since it should be unique among different ECUs because it serves to identify the file which tells how to interpret the ECU data.

3.5 29-BIT CAN IDS - TRANSMISSION ARBITRATION ID

When 11-bit CAN arbitration IDs are employed for diagnostic communications, the ISO-TP addressing method used is the Normal Fixed addressing. This addressing method defines exactly how AI fields are composed in order to obtain the arbitration ID, distinguishing between physical and functional addressing.
In order to compute the functional address where to make the request we use the FunctionalGroupIdentifiers presented in Section 2.1.3.4. Since we want to address the request to the major number of possible ECUs

we use as TA the value 0xFF, which represents all the functional groups. The standard ISO 15765-4 specifies the SA used by the external test equipment to be 0xF1. Applying the normal fixed addressing schema, relative to functional addresses, presented in Table 2.9 we map the AI field into the arbitration ID, obtaining that the functional address to use is 0x18DBFFF1.
As in the previous scenario we make an UDS DiagnosticSessionControl request, of type diagnosticSession, on the functional address identified. In this case, we don't need to use any heuristic in order to compute the transmission arbitration ID from the arbitration ID of the response. For each ECU which responded we can deterministically compute it by extracting the SA and TA fields from the arbitration ID of the response. In order to extract them we follow the normal fixed addressing mapping for physical addresses, presented in Table 2.8. Once we get the values for SA and TA we switch them and apply the same mapping. In this way, we obtain the physical address for transmission of the ECU being considered.
Since in this scenario there is no need to validate the transmission arbitration address, being it deterministically computed, we simply add the previously mentioned DID to the list of those used to gather information about each ECU.

With the presented techniques, we discover the arbitration IDs to use for the CAN messages, without the aid of DBC files [65] and without appeal to brute-force. We are now able to communicate with the ECU UDS servers present on the car network. We can obtain more info and apply vulnerability test on each ECU previously discovered.

3.6 INFO GATHERING

Once we are able to communicate with a discovered ECU we start querying it in order to obtain all the possible information about it. At this purpose, we employ once again the UDS ReadDataByIdentifier service, making requests for DIDs of which meaning is known. In Table 3.1 we present the employed DIDs, with the meaning specified by Annex C.1 of ISO 14229-1 standard.

Byte value	Data Identifier Description
0xF190	**VIN** This value shall be used to reference the VIN number.
0xF191	**vehicleManufacturerECUHardwareNumber** This value shall be used by reading services to reference the vehicle manufacturer specific ECU (server) hardware number.
0xF192	**systemSupplierECUHardwareNumber** This value shall be used to reference the system supplier specific ECU (server) hardware number.
0xF193	**systemSupplierECUHardwareVersionNumber** This value shall be used to reference the system supplier specific ECU (server) hardware version number.
0xF194	**systemSupplierECUSoftwareNumber** This value shall be used to reference the system supplier specific ECU (server) software number.
0xF195	**systemSupplierECUSoftwareVersionNumber** This value shall be used to reference the system supplier specific ECU (server) software version number.
0xF197	**systemNameOrEngineType** This value shall be used to reference the system name or engine type.
0xF19F	**Entity** This value shall be used to reference the entity data identifier for a secured data transmission.
0xFF00	**UDSVersion** This value shall be used to reference the UDS version implemented in the server.
0xF18D	**supportedFunctionalUnits** This value shall be used to request the functional units implemented in a server.

Table 3.1: Information gathering employed DIDs definition

Behind the illustrated DIDs we also give the possibility of adding to the information gathering process the following sets of DIDs, but for which we don't know the exact meaning since them definition is proprietary:

- 0xF1F0 - 0xF1FF identificationOptionSystemSupplierSpecific. This range of values should be used for system supplier specific server/vehicle system identification options;
- 0xF100 - 0xF17F identificationOptionVehicleManufacturerSpecific. This range of values should be used for vehicle manufacturer specific server/vehicle identification options;
- All the other DIDs admitted by the standard, which means any possible value in the format 0xXXXX where X is a hexadecimal digit, and not specified yet.

We chose to leave the decision of using those other sets of DIDs to the user in order to give him the possibility of preserving performances in terms of time employed for the scan or the completeness of the provided information.
Since the proprietary specific limitation about the number of DIDs presents in one request of ReadDataByIdentifier service, we will make each request including only one DID each time. In this way, we are sure to never incur in the limitation if this exists, although at the price of a slowdown in the whole procedure.

3.7 VULNERABILITY TEST

We realize an environment that facilitates the development and execution of tests for known vulnerabilities. This is done by implementing a series of helper classes and methods. Those allow the test implementer to just focus in translating a known vulnerability in a test case, without having to worry too much about technical aspects of the communication. Those can be taken for granted and will be implemented by the framework itself. On top of that we propose a base structure which each test should follow. This will help in writing effective tests and will lower the probability of permanently alter the behavior of the system after the test was performed. Since in this work we deeply studied and analysed the UDS protocol we have further refined the base structure of a test deployed in this context, in order to provide specialized class regarding UDS scenario.
Following we present a novel vulnerability in UDS implementations which can be exploited in order to perform other attacks easier. As proof-of-concept of our framework we've implemented and performed a specific test for discovering it in a real car network. The details of the

implementation and the results of the test are provided respectively in Section 4.3.2 and Section 5.3.

3.7.1 *Vulnerability example: Unsecured UDS Critical Services*

Because there is no enforced addressing or message source identification, it is easy for components to both sniff the CAN network as well as masquerade as other ECUs and send CAN packets. In previous work, [13] and [16] have shown that this property allows trivial replay attacks, activating a range of automotive functions. One of the problems when performing a CAN attack by packets injections is the real ECU which keeps sending the legitimate packets. As discussed in [67], there are several techniques in order to bypass this drawback. In the most trivial cases is sufficient to send the hijacked packet at a much higher frequency than the legitimated one in order to overhang it. In other case this could trigger protection mechanisms in the receiving ECUs, which seeing very different and fluctuating values for the same packet, may decide to completely ignore it and vanish the effects of the attack. A more effective method is to completely stop the legitimate ECU to send messages. This could be achieved, for example, by making an UDS request for the service RequestUpload which puts the ECU in boot mode, in order to upload a new firmware. In this mode, the ECU automatically stop sending any message but the action could be irreversible: once the ECU is in this state, sometimes, it can be restored only by uploading a valid firmware. Another, less invasive and reversible, service that can be used is CommunicationControl, which has the ability to tell the ECU to stop sending any message of certain kind and normal messages are one of the possibilities. Sending a request for service CommunicationControl is possible to achieve the same result of sending a request for service RequestUpload but the action is completely reversible by sending a request for service ECUReset, which restores the ECU to normal behavior.
Since the access to those services could became a vulnerability for performing other kind of attacks, it should be secured. Currently UDS protocol provides the SecurityAccess service which should be the enabling trigger for those kinds of other services. In case a critical service is requested without performing first the SecurityAccess procedure, the ECU should respond with a negative response of type securityAccessDenied and not executing the procedure normally expected for the request

made.[3]
Based on the developed vulnerability test framework, we implemented the actual test in order to assert if the most security-critical services are protected by the security access mechanism on all the ECUs discovered in the automobile network, as long as those services exist on the examined ECU.
The execution of the test on the Audi A3 Sportback showed that none of the discovered ECUs was correctly requesting to the external tester to perform a SecurityAccess request when a request for a critical service was received.

3 The effectiveness of this security access control has been analysed and strongly criticized by other papers, for example [16] . Is not in this work scope to discuss it. To date is the only security access mechanism provided by UDS protocol.

4

IMPLEMENTATION DETAILS

4.1 INTRODUCTION

To prove the concepts explained in APPROACH chapter and to release a concrete result of this work we designed and realized the **cmap** tool. The software architecture is divided principally in two different frameworks: discovery and vulnerability test.
We based our tool on Python as programming language. We've made this choice because it is an open source cross platform programming language. Also, multiple tools are based on Python and it is a common programming language, which may enable and ease the community to use and eventually contribute to extend our tool further.
We started from the pyvit library as a first implementation of the UDS and ISO-TP protocols, since the specific protocols details needed for our purposes were not completely implemented, we further extended it and we implemented in depth the specifications of the protocols. In order to communicate with the physical CAN bus, using a common USB 2.0 port, we used the CANtact device [68], but the cmap tool implementation was not limited only to this device. The CANtact provides a serial interface which is easy to access using pyserial library [69]. The details about the build process of the CANtact are reported in Section 5.1.
In this chapter, we will describe the modifications done to Evenchick's library, the architecture and functionalities of the cmap tool as a result of this work.

4.2 PYVIT

The pyvit library is proposed as an interface to interact with automotive CAN networks. It provides classes in order to facilitate three main aspects: different hardware support; CAN packets representation and manipulation; interface to some automotive diagnostic protocols. The base structure is the one showed in Figure 4.1. We had to deeply understand the library structure and how it works in order to bring the modifications needed for our work without distort the original logic designed by

the author, so that other applications already using the library will still be compatible.

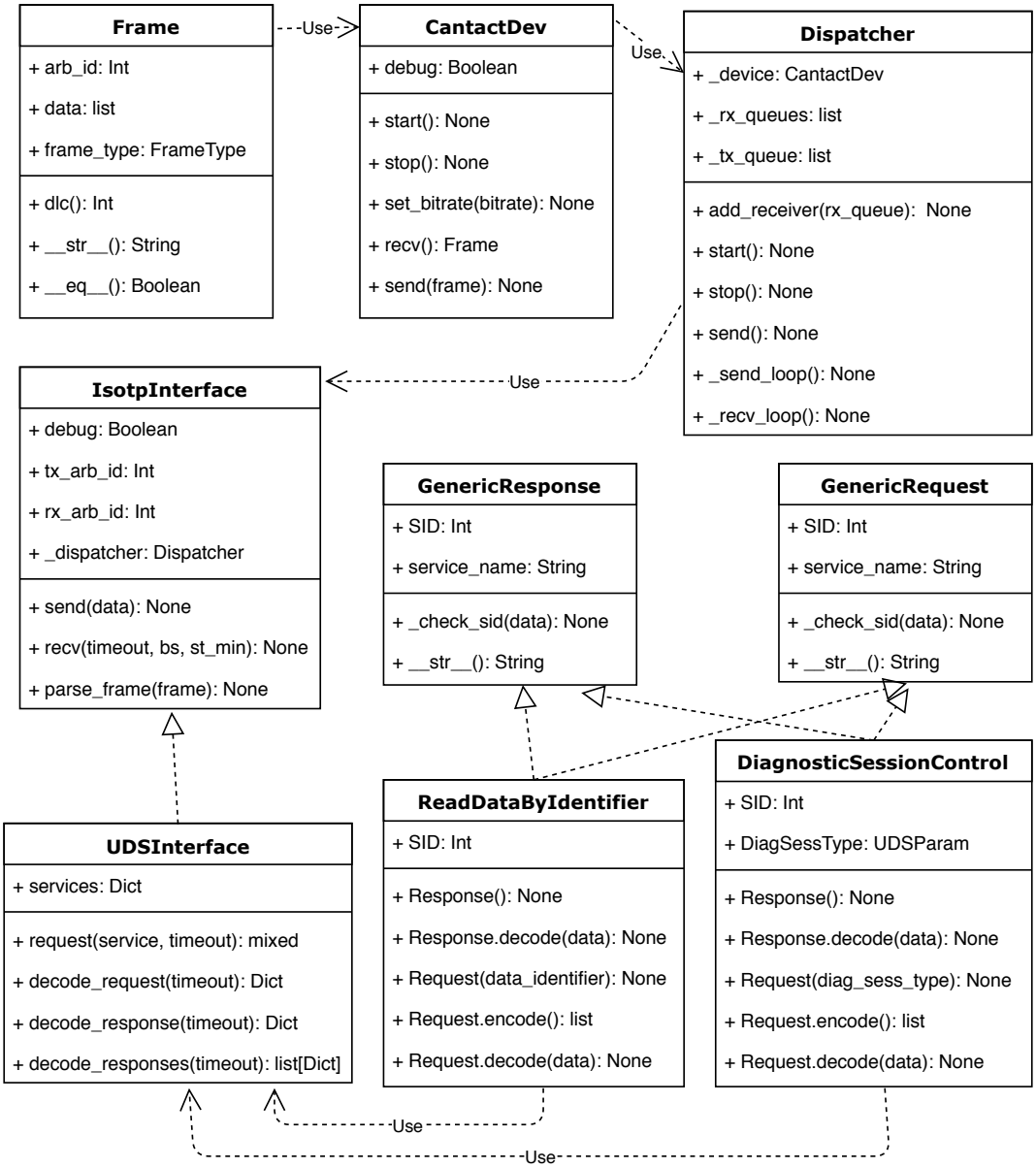


Figure 4.1: as a class, here we can see an example for DiagnosticSessionControl and ReadDataByIdentifier

4.2.1 *The Hardware Interface*

The devices supported by the library are:

1. CANtact device [68];

2. any device compatible with SocketCAN[1] [70];
3. PCAN-USB device [71].

Even if not in a formalized way, since there is no interface or abstract class generally employed, the author suggests a standard method of implementing the class used to interact with a device so that the actual hardware used does not influence the rest of the library code or the programs based on it. Each class should provide essentially 6 mandatory methods for interacting with the physical device. Below a description of how each one should be implemented and an example about the CantactDev class which is the one handling the CANtact serial device:

__INIT__ method, which is the class constructor, has to open a communication channel with the device itself. For the CANTact it opens a serial connection with the device employing the pyserial library;

START method has to perform all those actions meant to initialize the device before the first use. For the CANtact it takes care of sending the 'O' character which the device reads as a command to start the CAN serial communication;

STOP method has to perform all those actions meant to stop the communication with the CAN. For the CANtact it takes care of sending the 'C' character which the device reads as a command to stop the CAN serial communication;

SET_BITRATE method has to perform all those actions in order to set the CAN bus speed. For the CANtact it takes care of sending the 'SX' characters, where X is a number indicating the CAN bus speed according to Appendix A.1, which the device reads as a command to set the indicated CAN bus speed;

RECV method has to read from the device a complete CAN packet and return it as an instance of the class Frame described in Section 4.2.2. For the CANtact it reads characters from the serial channel until it reaches a \t [2] character, then the string presenting a complete CAN packet is analysed and used to instantiate an object of class Frame for representing the packet. The method returns this object;

1 SocketCAN is a set of open source CAN drivers and a networking stack of the Linux kernel.
2 Codification representing a newline character.

SEND method which has to send an instance of the Frame class to the device. For the CANtact it takes a Frame object and constructs a string according to the packed characteristics to be sent to the device.

Furthermore, the class may have other two methods in order to allow the usage of filters on the CAN ID, direct in the device. Those methods have a reason to exist as long as the functionality is supported by the device firmware.

SET_FILTER_ID method which sets in the device the CAN ID passed as parameter as the base of the filter. For the CANtact it sends to the device a string formed by the character 'F' followed by the characters of the ID in hexadecimal notation;

SET_FILTER_ID method which sets in the device the mask passed as parameter, which is used with the specified ID in order to apply the filter. For the CANtact it sends to the device a string formed by the character 'K' followed by the characters of the mask in hexadecimal notation

Most commonly the behavior of a filter is that the device will transmit a packet to the PC only if the bitwise AND between the base ID and the mask is equal to the bitwise AND between the ID of the received CAN packet and the mask.
In order to avoid the blocking of the whole program when waiting for receiving a message from the CAN bus, usually the device instance is not directly used to communicate. It is preferable to use the class Dispatcher, provided by the library. It is an auxiliary class used to delegate the interaction with the device to two different processes respect to the main one: first one for sending packets and second one for receiving packets. The sending process runs simply an infinite loop which calls the send method of the device for each frame found in the transmission queue (attribute _tx_queue in the class). The receiving process also runs an infinite loop which takes any packet returned by the recv method of the device instance and puts it in all the registered queues for receiving. An external user of the class first has to add a receiving queue to the Dispatcher object, from where he will get the received frames. Calling the method start of the Dispatcher the external processes will be forked and the communication can start. In order to send out a frame the user can call the method send, passing as parameter the desired frame, which will take care of adding it to transmission queue.

4.2.2 *CAN Packet Representation*

Each CAN packet is represented by the class Frame. It has attributes for memorizing the frame type, the CAN ID type, the arbitration ID itself and the data record. The DLC is automatically computed when needed, based on the length of the list containing the data record. In order to facilitate the output of a CAN packet the __str__ method is implemented to obtain a human readable string in a standard format, for example: ID=0x7DF, DLC=[3], Data=[02,10,01].

4.2.3 *Application Layer Interfaces*

The library provides an implementation of the UDS [35] and ISO-TP [34] protocols. It also provides the class ObdInterface meant to facilitate the interrogation of classic OBD PIDs [72] but this isn't actually the implementation of a protocol and since it isn't of interest for our work we will focus on the implementation of first two mentioned protocols.
For what regards the UDS protocol each service is modeled as a class. It has an attribute SID set to the code of the intended service. Each class in turn has two inner classes [73] called Request and Response, both inheriting respectively from the generalizations GenericRequest and GenericResponse. We can see an example of this structure in Figure 4.1 for services DiagnosticSessionControl and ReadDataByIdentifier.
GenericRequest and GenericResponse classes are both a specialization of class Dict [74], in this way is easy to represent each different data parameter of the services as a key-value pair of the object itself. The class GenericRequest has the method _check_sid which takes as parameter a data list, ready to be sent for requesting a service, and checks that the SID present in it is the same as the SID specified for the service being requested. The class GenericResponse has the method _check_nrc which takes as parameter a data list corresponding to a received response and in case it is a negative response it raises an exception of type NegativeResponseException, otherwise it checks the received SID and if this is not equal to the one requested plus 0x40, which means a positive response, it raises a ValueError exception since the response data does not make sense according to ISO 14229-1 specifications. The class Request, through his method encode, has the job of encoding the SID value and the values of keys representing the data parameters of the service request in a list of hexadecimal data, ready to be sent with the aid of the implementation of ISO-TP protocol. With the method decode it performs the dual oper-

ation of the method encode, or rather when it has to interpret a request it maps each element of the data list to the corresponding key representing a data parameter of the service identified by the SID present in the request.
The implementation of the ISO-TP is all enclosed in the class IsotpInterface. When instantiated it takes as parameters a Dispatcher object and the CAN arbitration IDs among which the communication should occur. The most important methods of the class are send, recv and parse_frame.

SEND takes as parameter a list of data bytes to be sent, its maximum length should be 4095, as specified by ISO-TP protocol. Depending if the data length is less or equal to 7 bytes or not it decides if it's dealing with a single frame or with a multiple frame transmission. In the first case, it just adds in front of the data the PCI information and instantiate a Frame object which is sent with the aid of the dispatcher, while in the second case the communication process is more articulated as we've explained in the protocol description. It splits the data in 7 bytes blocks and after sending the first frame of the sequence it waits for the control frame from the other endpoint of the communication. Then from the payload of the received control frame it extracts the parameters indicating the minimum amount of time to wait between two consecutive frame transmissions and the number of frames to transmit before waiting for another control frame. Once those parameters are known the sending of successive frames proceed accordingly, until the data is completely transmitted. For each sent frame it keeps track of the sequence number and before sending it computes the PCI information and prepends them to the data.

RECV takes as parameters the maximum time to wait for a frame, the number of blocks to receive before sending a control frame and the minimum amount of time that must pass between two consecutive frames arrival. As first action, it sets on the device a filter in order to pass only the frames with the arbitration ID equal to the one passed as parameter as being the one used by the other endpoint. The filter is reset as last action before exiting the method and returning the data received. After setting the communication parameters passed in the class attributes, for each received frame it calls the function parse_frame until this does not return a value different from None, which means data was received completely, or the timeout expires.

PARSE_FRAME method is in charge of reconstructing the data from the sequence of received frames and return it when the transmission is completed, it takes as parameter a received frame. For each call extracts the PCI info from the passed frame and treats it consequently: when a first frame is recognized it initialize the needed class attributes and memorize the data length to be received, they will be used to know when the transmission is completed; for each consecutive frame, it checks the sequence number contained in the PCI and if it corresponds appends the data to the previous ones. Each time it is called the method checks if it has to send a control frame based on the parameters of the communication present in the class attributes and set by recv method. Received control frames are ignored.

UDSInterface is the class that brings together the complete process of communication in a request-response sequence when an UDS service is called. In the original implementation, it is designed as a specialization of the IsotpInterface class, so it takes as parameters all the ones of the IsotpInterface which are automatically passed to the parent constructor. The method request is the principal actor of this class, it takes two parameters: an instance of the class Request of a service class and a timeout parameter. The first action of the method is to encode the request passed and to send the list representing the data with the aid of the send method present in the parent class. The timeout parameter is passed downward to the method recv, of the parent class, in order to know how much it should wait for a response, which is returned to the caller as soon as a complete response is received. If timeout expires the None value is returned.

4.2.4 *pyvit Implementations*

4.2.4.1 *UDSInterface uses IsotpInterface*

One of the first changes we've made was removing the inheritance of class UDSInterface from the class IsotpInterface in favor of a usage relation, as showed in Figure 4.2. We have made this change in order to be able to further specialize the class IsotpInterface to manage the different addressing methods, as we will see in next section. On top of that the inheritance between UDSInterface and IsotpInterface is conceptually wrong since they represent different levels of the OSI Basic Reference Model. UDSInterface is the implementation of application level protocol

while IsotpInterface is the implementation of transport level protocol. There are other parts of UDS standard which define the usage of UDS on different transport layer protocols, which have nothing to do with ISO 15765-2. We added the attribute transport_layer to the UDSInterface, so that a user of the class is able to set any desired transport layer. In the class code, we replaced every usage of the old parent method's send and recv with those provided by the transport layer instance present in the transport_layer attribute. To maintain the compatibility with previous versions we held the original parameters of the constructor and instantiated as default transport layer an object of IsotpInterface class.

4.2.4.2 *ISOTPAdressing*

The implementation of the ISO-TP different addressing methods, specified in Section 2.1.2.3, is the most significant contribution given by this work to the pyvit library. We designed the structure in order to have the class IsotpAddressing which inherits from IsotpInterface and specializes it with the features and attributes common to all the addressing methods. In particular, it defines the attributes for encapsulating the addressing information associated with the data to be exchanged:

- N_SA: network source address. Could be a one byte value;
- N_AE: network address extension. Could be a one byte value;
- N_TA: network target address. Could be a one byte value;
- MType: message type. The possible values, diagnostic and remote diagnostic, are represented by the class Mtype;
- N_TAtype: network target address type. The possible options, physical and functional, are represented by the class N_TAtype.

This class also defines two abstract methods: compute_tx_arb_id; compute_rx_arb_id; since those methods are abstract the IsotpAddressing class should never be instantiated itself. For each addressing method we developed a dedicated class which in turn inherits from IsotpAddressing and implements his abstract methods, in Figure 4.2 we report the final class structure obtained. The implementation of those methods provides the mapping procedure of the addressing information fields in the actual arbitration ID according to the rules of the specific addressing method described in Section 2.1.2. The developed classes are:

- IsotpMixedAddressing

- IsotpExtendedAddressing
- IsotpNormalFixedAddressing
- IsotpNormalAddressing

As specified in the standard, classes IsotpMixedAddressing and IsotpExtendedAddressing utilize the first byte of the data record to transmit respectively the N_AE and N_TA addressing fields, consequently the space for actual data is reduced from 7 bytes to 6 bytes[3]. In class IsotpInterface we transformed the hard-coded limit of 7 bytes in the value of a class attribute which has this as default value. In this way, we are able to take in account the variability of the data length of first frame, according to the addressing method employed, by redefining the introduced attribute in the aforementioned classes.

4.2.4.3 *Advance Frame Filtering and Multiple Responses*

The original class IsotpInterface was designed in order to perform the communication between client and UDS server knowing a priori the arbitration ID for transmission and the one from which to expect the responses. The constructor of the class expects both as parameters. Since we developed a discovery tool to find out what are the UDS servers on the network we could not know from which arbitration ID we would get the responses, actually discovering them is one of the purposes of our tool. On top of that, in case of functional addressing, the protocol itself contemplates multiple responses from different IDs. Considering this, it is evident that filtering the incoming frames only on a determined arbitration ID could not always work. In order to overcome this major limitation, we modified the class and removed the requirement of a value for the response arbitration ID. This modification was not sufficient since it would mean to accept every message seen on the CAN bus as a response to our communication. To filter only the messages actually being part of our communication, even without knowing the response arbitration ID, we introduced a new method to be applied on incoming frames. It checks if the arbitration ID of the received frame is not in a blacklist and executes an externally specified callable function with the frame as parameter. Both these conditions have to be true in order to let the frame pass and consider it being part of the communication.

BLACKLIST is by default empty, it has to be populated by the user of IsotpInterface class. It is mostly useful when there are many mes-

3 consider that 1 byte is already occupied by the PCI information

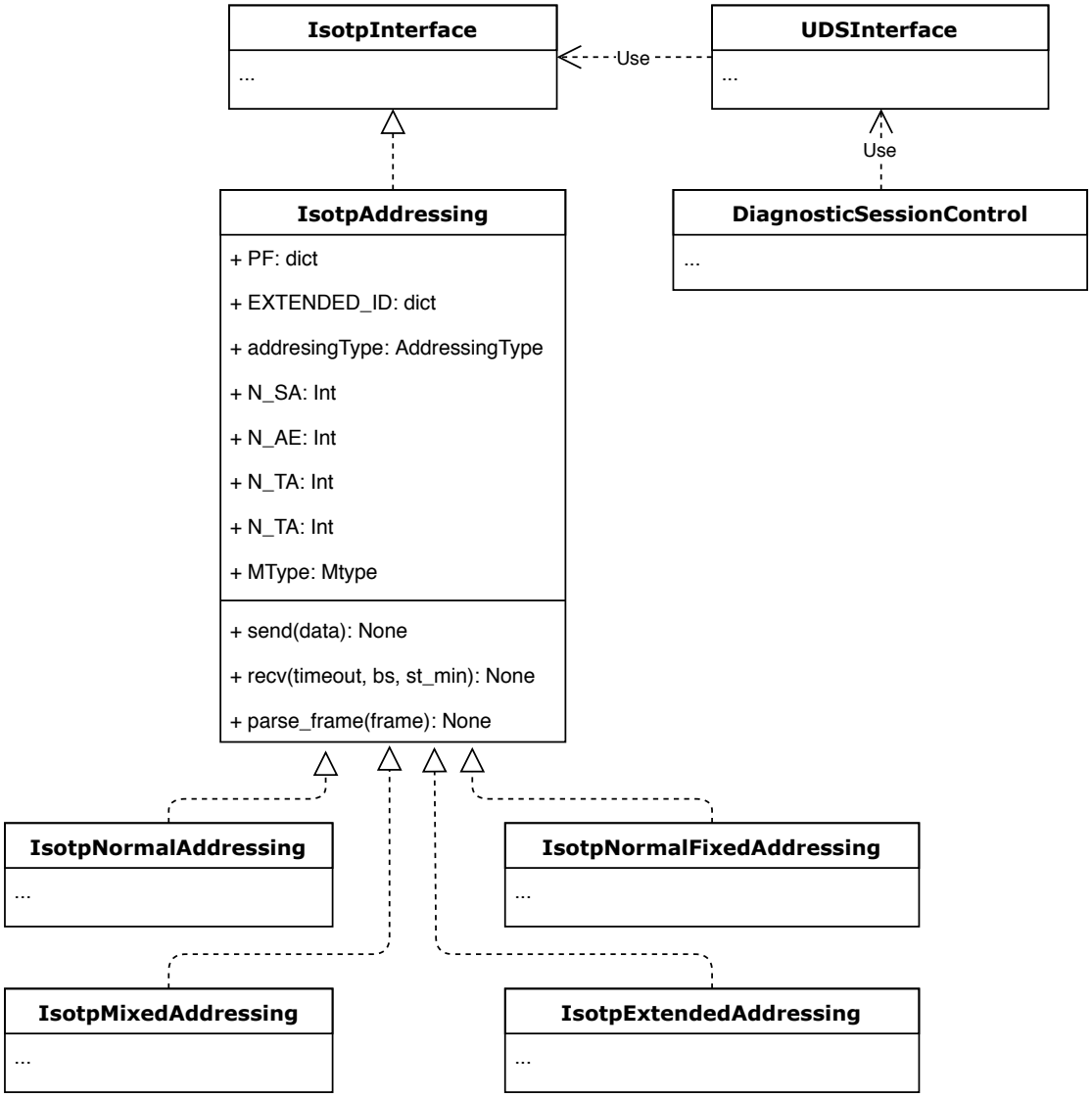


Figure 4.2: Class structure of pyvit library after the new implementations

sages present on the Controller Area Network (CAN) bus. The user of the class is supposed to register the arbitration IDs of the messages seen on the bus for a certain amount of time before the actual communication. Then those IDs should be used to populate the blacklist. This is possible since the arbitration IDs used for UDS communication are not used for normal message.

EXTERNAL CALLABLE FUNCTION used to filter frames. It is assumed to be set by the user of the class according to his needs. In the UDS communication scenario one of the most useful implementations is the one to check if the received frame either has a SID equal to the one requested plus 0x40, so it is a positive response, or if it is a negative response for the requested SID.

In order to be able to manage multiple responses from different UDS servers we also modified the request method of class UDSInterface. Now the method checks if the addressing target address is of type functional and if this is the case calls the custom method decode_responses which, instead of terminating as soon as a complete response is received, continues to listen for responses until the functional timeout[4] expires

4.2.4.4 *Minor Modifications*

Beyond the main modifications described until now we've done other minor fixes and implementations. Among these the following deserves to be cited:

DIFFERENT CLASSES FOR DIFFERENT NEGATIVE RESPONSES in order to rise an ad-hoc exception for each UDS negative response code. This simplifies dealing with them in the rest of the code and improves the code readability itself. In order to deal with the analysis of a negative response in a single place and to establish which specific exception has to be raised, we implemented the factory pattern [75] in the NegativeResponseException class.

SUPPORT OF EXTENDED IDS IN ISOTPINTERFACE in order to be able to deal with CAN buses where messages with 29b arbitration IDs are employed.

4 the functional timeout is the maximum amount of time which a client has to wait in order to be sure that all the possible UDS servers responded

4.3 CMAP

The cmap tool is structurally divided in three parts: the main flow of the program, the discovery framework and the vulnerability test framework. Each of these sections has always to deal with the info regarding one or multiple ECUs, hence for representing each ECU involved in the process we implemented the class ECU.
The class ECU is designed as a specialization of Dict class, in this way it is easy to store different information gathered at runtime about each ECU as a key-value pairs. The information about ECU name and communication arbitration IDs are also mapped to tree class attributes as well as being key-value pairs. We opted for this implementation since these are the key information for each identified ECU and in this way ti is easier to use them in the rest of the code. In order to make transparent this duality to the external users we implemented custom getters and setters for those attributes, which take care to deal with the values in both memorization parts: the class attribute and the key-value pair; resulting in a transparent behavior for the user of the class.
In the main flow of the program we define and manage all the options supported, which are illustrated in Appendix B. Then, according to which parameters are passed by the user, the tool executes these steps:

1. instantiates a communication channel with the CAN bus, through pyvit library functions and employing the physical interface indicated as parameter;

2. computes, if requested, a blacklist of CAN IDs seen on the bus, with the aid of helper function 'compute_blacklist', scanning the bus for the indicated amount of time;

3. optionally loads from file a list of ECUs resulted from a precedent computation, in this way next steps will integrate this list rather than compute one starting by zero;

4. creates the discovery instance with the characteristics specified as parameters and runs the discover procedure. Its output is a list of identified ECUs, where each one is represented by class ECU and contains all the info that discovery framework could extract;

5. if the test of known vulnerability is requested calls the helper function which in order creates an instance of each available test and applies it to each ECU discovered by previous step. The helper

function stores the result of each test in the object representing the tested ECU;

6. outputs the whole result to console, accordingly to the verbosity level requested by the user;

7. saves the whole result in JSON format to the file eventually specified by the user.

The core functionalities are those provided by discovery and vulnerability test packages, we describe them in depth in next two sections.

4.3.1 *Discovery*

We designed the discovery package starting from a general representation of what features a discovery methodology should provide. After that we implemented the actual classes and methods meant to put in practice the approach steps described previous chapter. In this way should be easier in the future to extend the framework with new discovery techniques, which potentially could be based on totally different approaches, but lead to the same result. This concept is implemented as the abstract class DiscoveryMethod which provides the signature of following attributes and methods:

ECUS is a list of objects of type ECU where each one represents a discovered control unit whit all the information found about it

DISCOVER method has to provide the capabilities of scanning the network and detect the presence of the greatest number of ECUs possible

IDENTIFYECUS method has to provide the capabilities to identify the ECUs discovered by previous step. It has to find a name for each one and establish with certainty the arbitration IDs for the UDS communication with each ECU. After this step, it calls the protected method _gather_info

_GATHER_INFO protected method has the task to extract information regarding ECUs. Its implementation should consider by default a base level of effort employed, with particular attention in preserving performances and low execution time, and only in case the method full_info was previously called should perform all its best, regardless performances and execution time, to extract the greatest

amount of information possible from each ECU. More effort means more time spent in the scanning, hence this design was done in order to allow the user to choose whether to give more importance to speed performances or to information completeness, depending of the situation.

The first implemented discovery methodology is enclosed in the class DMUDSFunctionalReq, extending class DiscoveryMethod. It is based on IsotpNormalAddressing class as transport layer and UDSInterface as application layer. It implements the techniques described in Section 3.4 regarding the 11-bit arbitration IDs. The class constructor takes as parameters: a Dispatcher object, the verbosity level[5] and the functional timeout to use. It calls the protected function _uds_init which takes care of instantiating the transport layer of type IsotpNormalAddressing, passing to it, as the function to filter the received frames, the method rx_filter_func. This function applies with an AND logic the following criteria:

1. frame data length should be at least 2, since each valid response frame has to contain at least the PCI byte and one byte of data;
2. then according to the info contained in the PCI byte one of the following criteria is employed:
   a) if it is dealing with a *single frame* it accepts a frame representing a positive response or a negative response related to the UDS request made. This means accepting frames which have: second byte equal to requested SID + 0x40 in case of positive response; second byte equal to 0x7F and third byte equal to requested SID;
   b) if it is dealing with a *first frame* it accepts only a frame representing a positive response, since negative ones are not contemplated in this context. After accepting the frame, we store the response arbitration ID in the transport layer since consecutive frame will use the same one;
   c) if it is dealing with a *consecutive frame* it accepts only if the arbitration ID is the same as the one of already received first frame.

5 For a level grater or equal to 2 we report the general steps of the discovery process; for a level grater or equal to 3 we report also the frames exchanged by the transport layer during the communication; for a level grater or equal to 4 we report all the frames seen on the CAN bus by the device. Normally levels 3 and 4 should be employed only for debug purposes.

The functional addresses 0x700 and 0x7DF to be used as arbitration IDs for the requests are stored in the class attribute 'functional_addresses' as a list. The discovery methods described earlier for the general representation are implemented as follows:

DISCOVER method creates an instance of the UDS service diagnostic session control trough the class DiagnosticSessionControl of pyvit library. It employs the default session diagnostic session type. It is the one with lowest functionalities available but is also the fundamental one so it needs to be implemented in each ECU which supports UDS protocol, which is what it matters in this phase in order to get responses from the largest number of ECUs possible. With the aid of UDSInterface instance, created during the constructor call, it sends the request on the network for each functional address. After each request, it waits for the responses and parses each one in order to add an entry to the ECUs list;

IDENTIFYECUS method applies the heuristics showed in Section 3.4 to identify the arbitration ID which should be used to make a physical communication with each detected ECU. The guessed arbitration ID is checked by making an UDS service request through the class ReadDataByIdentifier, using the data identifier 0xF19E. We keep applying the heuristic and making request on a new guessed arbitration ID until we obtain a valid response coming with the arbitration ID that the ECU already used in order to respond to the functional request made by the discover procedure. Once we obtain a positive response we memorize the arbitration ID used for the request and the actual response of the read data by identifier service, which is assumed as the ECU name.

_GATHER_INFO method, once the arbitration IDs for the communication are known, implements the process to obtain the desired amount of information regarding each ECU. It creates an object of class ReadDataByIdentifier and makes one separate request for each DID summarized in Table 3.1. If previously the method full_info was called it also makes a request for the manufacturer and supplier specific DIDs described in Section 3.6. After each request if no response is obtained the DID is skipped[6], otherwise both

6 The case of no response is unlikely, practically impossible since we have already confirmed that on the used address there is a server responding. More commonly we will get a negative response of type serviceNotSupported or requestOutOfRange respectively if the request data by identifier service or the requested DID are not supported.

positive and negative responses are registered. In the first case the information obtained is registered in the relative ECU object under the key 'info', in the second case the negative response is registered under the key 'info_NRC' so that the user can further analyse why there was no information obtained from that DID.

The second implemented discovery methodology is enclosed in the class DMUDSFunctionalReqNormalFixed, extending class DMUDSFunctionalReq. It is based on IsotpNormalFixedAddressing class as transport layer and UDSInterface as application layer. It implements the techniques described in Section 3.5 regarding the 29-bit arbitration IDs. This discovery methodology implementation inherits multiple aspects of the one of DMUDSFunctionalReq class, but with a few differences which allow to manage 29b arbitration IDs and to exploit the advantages that this addressing type offers in respect to IsotpNormalAddressing. The constructor behaves similar to the one of DMUDSFunctionalReq, the main differences are the transport layer class used, which in this case is IsotpNormalFixedAddressing, and the functional address which is dynamically computed by consider as source address 0xF1 and as target address the functional address 0xFF.
For what regards the discovery methods described earlier for the general representation the differences respect to DMUDSFunctionalReq class are in the methods discover and identifyECUs:

DISCOVER method changes since in the case of normal fixed addressing the transmission arbitration ID is deterministically computable from the arbitration ID of the response. We do it in the parsing sub function of discover method employing the following steps for each response:

1. extract the source address by making a bitwise 'AND' between the arbitration ID of the received frame and the mask 0xFF. In this way, we obtain the last byte which, accordingly to the addressing method, represents the source address or rather for us the target address to use when transmitting a frame;
2. instantiate an object of type IsotpNormalFixedAddressing using the value obtained in previous step as target address and the value 0xF1 as source address;
3. call the method compute_tx_arb_id on the IsotpNormalFixedAddressing object in order to obtain the arbitration ID to use in transmission for that ECU.

IDENTIFYECUS method changes since we already have both arbitration IDs needed for a physical communication, therefor we don't need to apply any heuristic in this case for computing the transmission arbitration ID. We can start directly with the information gathering, which is performed in the same way as for DMUDSFunctionalReq.

For testing and debugging purposes we have two other discovery methodologies: DMUDSAllDiagAddr and DMUDSAllAddr. Both inherit from the class DMUDSFunctionalReq and they just redefine the list of functional addresses to be used with all the arbitration IDs between different ranges. In the first class, we consider all the arbitration IDs between 0x700 and 0x7FF, commonly used for diagnostic purposes accordingly to Table 4.1. In the second class, we consider all the arbitration IDs between 0x100 and 0x6FF, usually employed for normal messages.

Usage field	**ID Min**	**ID Max**
On event	0x000	0x0FF
Periodic and on event	0x100	0x1FF
If active or periodic and if active	0x200	0x2FF
Periodic	0x300	0x3FF
Network management	0x400	0x4FF
Unknown	0x500	0x5FF
Development	0x600	0x6FF
Diagnostic	0x700	0x7FF

Table 4.1: Division by field of utilization of the 11b arbitration ID range of values [76]

Figure 4.4 represents the complete architecture of the described discovery framework.

4.3.2 *Vulnerability Testing*

With the vulnerability test framework, we provide a set of classes and methods in order to facilitate the implementation of standard tests that should reveal the presence of known vulnerabilities in an ECU application layer. In the following two sections of this chapter we describe how the framework is designed and an example of an implemented test.

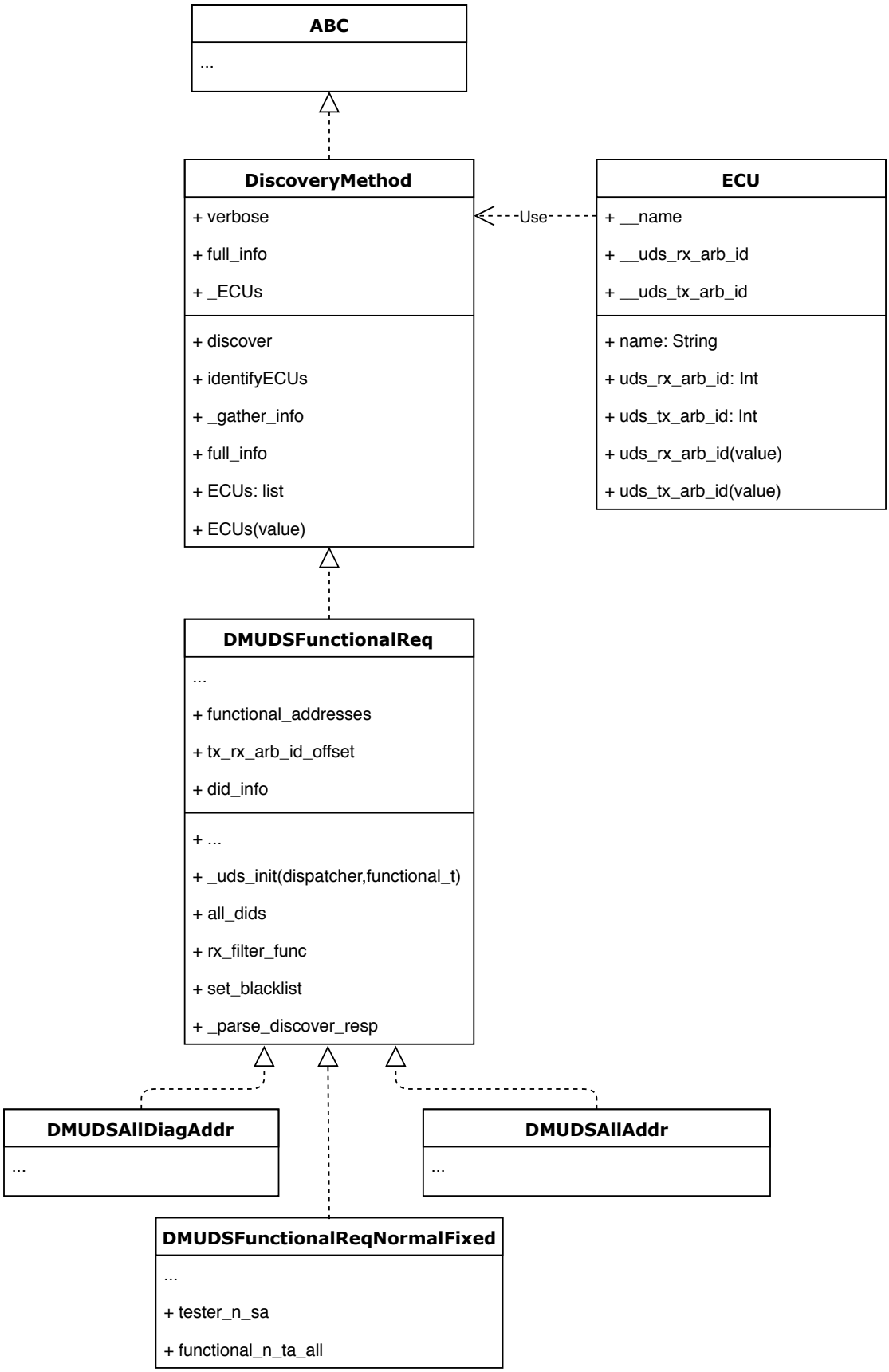


Figure 4.3: Class structure and methods of the discovery framework

4.3.2.1 *General Architecture*

We designed the vulnerability test package starting from a representation of how a generic test should be structured and defining 4 possible severity levels for a vulnerability. The severity levels are defined through the enumerative class VulnerabilitySeverity as being: critical; high; medium; low. The abstract class VulnerabilityTest represents the structure of a generic test. Its design was made thinking about application layer vulnerabilities hence the transport layer is taken for granted and is an input of the class constructor as well as the object of class ECU representing the actual ECU on which the test has to be performed. The class provides the signature, and in some cases an implementation, of following attributes and methods:

NAME attribute to indicate the vulnerability denomination;

SEVERITY attribute to indicate a severity level among those provided by class VulnerabilitySeverity;

DESCRIPTION attribute to provide a description of the vulnerability for which the test is performed;

PASSED attribute representing the result of the test. It should always be initialized to False value and became True only if the test does not reveal the presence of the vulnerability;

TEST method is already implemented and its task is to call sequentially the methods: _set_requirements; _do_test; _reset in order to execute the test;

_SET_REQUIREMENTS method should be implemented in order to perform all those preliminary steps needed to have the ECU in the right conditions in order to perform the actual test for the vulnerability;

_DO_TEST method should implement all the steps to execute the intended test in order to assert if the vulnerability is or not present in the examined ECU;

_RESET method should be implemented in order to perform all those steps needed to bring back the ECU to the normal state, normally the one being before the test;

OTHERINFO method should provide additional information about how the test was performed and eventually which problems occurred.

Those should help the user understand why the test has failed or passed.

For the specific context of UDS application layer we have implemented the class UDSVulnerabilityTest. It extends the general VulnerabilityTest class and further specialize it providing a more specific implementation of some of the methods described earlier:

TEST method calls the parent test method and catches all exceptions deriving from class NegativeResponseException which are not handled earlier for the purpose of the test itself. The receipt of an exception at this level implies the failure of the test, so the vulnerability is considered as being present. This is a conservative choice in order to not assert that the vulnerability is not present while it could be not detected for a flaw in the test implementation itself. All exceptions are added to a list which is then added to the test result in order to facilitate the analysis of why the test has failed;

_RESET method provides a basic technique to bring the ECU in the default status. It makes the ECUReset UDS service request, specifying the reset type being 'keyOffOnReset'. Accordingly to the standard specifications, this type of reset should be the same as turning off and on the car by the key operation.

In order to allow the test implementations to properly manage unattended behaviors we have defined two exceptions: VulnerabilityTestRequirementsException and VulnerabilityTestResetException. First one should be raised when, for any reason, is impossible to fulfill the required steps prior to proceed with the actual test. Second one should be raised when, for any reason, is impossible to perform the reset steps needed to bring back the ECU to a normal state, after the test was performed.

4.3.2.2 *Test Implementation*

We realized a specific test in order to assert if the critical services provided by UDS are protected by SecurityAccess service request. As we have seen in Section 3.7.1, access at those services may lead to a vulnerability which makes easier to perform other packet injection attacks. We implemented the actual test in the class VTUDSSecurityAccess, which inherits from the class UDSVulnerabilityTest. The UDS services which will be tested are stored in the 'criticalServices' class attribute. As a proof of concept we included only the CommunicationControl service but also

services like: RequestUpload; ReadMemoryByAddress; WriteDataByAddress; WriteDataByIdentifier; may be included as part of this category. The class overrides, with a specific implementation related to the actual test, the following methods of the parent class:

_SET_REQUIREMENTS method makes a UDS request of service DiagnosticSessionControl, in order to enable the desired session type on the target ECU. If any kind of exception is raised or no valid response is received the method in turn raises the exception VulnerabilityTestRequirementsException. The session type to be used is passed as parameter since we don't know a priori which is the appropriate one, it could also be a proprietary one. During the test, in specific conditions, this method is called again in order to establish a different session type on the ECU.

_DO_TEST method takes every UDS critical service from the list 'criticalServices' and uses them to make a request to the target ECU. At this point the possible scenarios are 4:

1. it gets a negative response saying that the service or its subfunction is not supported in the active diagnostic session, this is traduced in receiving an exception of class ServiceNotSupportedInActiveSessionException or SubFunctionNotSupportedInActiveSessionException. We call the _set_requirements method specifying another diagnostic session type, after that we make the same service request again. If we keep falling in this case we keep setting different diagnostic session types on the ECU and make the service request again, until we either fall into another case or the available diagnostic session types are all tested. In this last case, we have made the conservative choice of considering the test for the specific service failed. This is motivated by the fact that the UDS standard specifies that the earlier mentioned negative responses should be given only if the service is supported in at least one other diagnostic session type. Unless something unexpected happens, we should find this session type because we seek for them all.
2. it gets a negative response saying that the requested service is not supported by the ECU. This is translated in the exception ServiceNotSupportedException. Since the service is not supported, obviously, there is no need for asking the security access procedure so we consider the test passed for the specific service;

3. it gets a negative response saying that security access is needed, this is traduced in the exception SecurityAccessDeniedException. This is the appropriate behavior that we want so we consider the test for the specific service as passed;
4. any other exception or response leads to consider the test for the specific service as failed.

We consider the whole test as passed only if all the critical services examined passed it. During the process, we collect all the services that did not pass the test in a list. For the other scenarios described we memorize a message.

OTHERINFO method, if there is any service which failed the test, adds to the list of messages another one specifying all the UDS services that did not passed the test. It returns the list of messages.

Figure 4.4 represents the complete architecture of the described vulnerability framework.

The severity level assigned to this vulnerability is 'Medium', since it does not directly allow an attack but its presence can simplify the conditions for other attacks. We suggest the use of "Common Vulnerability Scoring System" [77] guidelines in order to establish the severity level of a vulnerability.

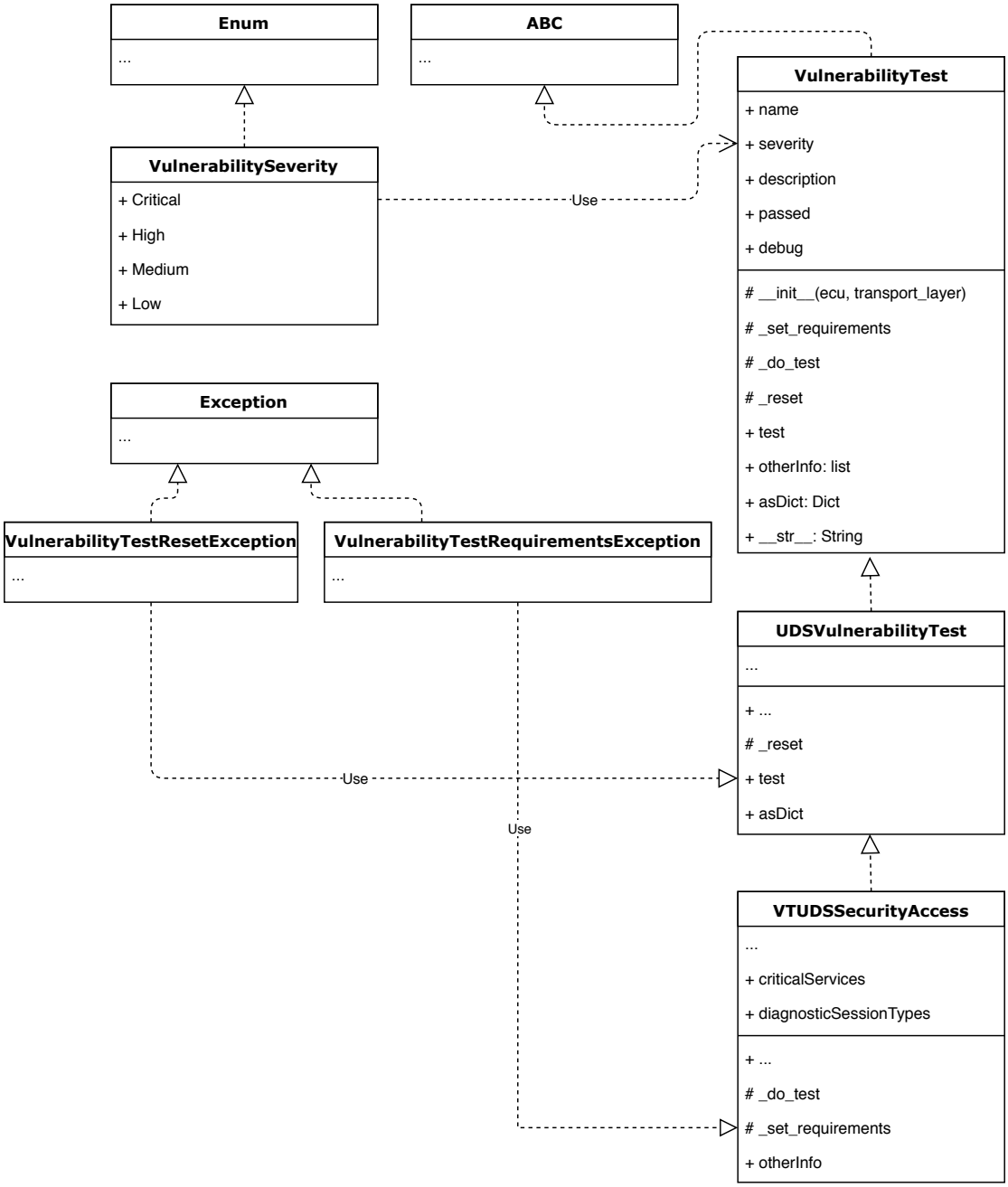


Figure 4.4: Class structure and methods of the vulnerability framework

5

EXPERIMENTAL VALIDATION

This chapter reports the setup needed in order to perform experiments on real cars, the results of the discovery and the vulnerability test experiment, the problems encountered while we were conducting these experiments and how we dealt with them.
After the analyses and implementation phases we tested our tool on real cars in order to prove that:

- the discovery framework is able to get a panorama of the ECUs present on different automobiles, with many details about each ECU;
- the test implemented with the vulnerability test framework can actually reveal the presence of the vulnerability.

One of the main problems we've encountered during the test phase was the lack of an official comparison term to evaluate our results. We didn't have precise knowledge of which ECUs there were on each car, neither how many. This due to the fact that all the documentation regarding technical aspects of cars is proprietary and often it is unmanageable to have precise knowledge of it. We provide more details about how we reasonably validated the results in each experiment sections.
Since we were using personal cars for safety reasons and due to high cost of the required equipment (i.e. cars) special careful was needed. Afresh for lack of technical documentation about the cars and them ECUs, we were not hundred percent sure of what effects we could get on the cars while running our tests. Anyway, the field of intervention is limited to diagnostic functionalities and the used UDS services were carefully studied before trying them on the real cars. From what reported in the standard all the UDS services employed should not have any permanent effect, they only request information or produce temporary changes. Hence the probability of a permanent damage to the vehicle was very limited. In order to avoid any possible risk no tests were carried out with the car on the move.

5.1 EXPERIMENTAL SETUP

CANtact

We employed the CANTact hardware device [68] in order to be able to communicate with the CAN bus of the cars. The CANtact is a completely open source low cost USB-to-CAN converter scriptable via Python. Since both the firmware [78] and hardware [79] are open source projects we have built one.
We bought all the needed components on-line. We slightly modified the circuit scheme in order to make the obtained PCB feet the enclosing box with the integrated J1962 connector showed in Figure 5.1 (a).
We realized the device starting from a blank photosensitive PCB (Figure 5.1 (b)) and the electrical scheme printed on tracing paper. Then we impressed the scheme on the PCB by overlapping the tracing paper to the PCB itself and exposing them to UV light for about 1 minute. The parts of the photosensitive film not covered by the printed scheme have been exposed to the UV light so they will be easily removed by steeping the board in a developer solution. Now we have the PCB's copper layer covered by the photosensitive film only in the parts representing the electrical scheme, hence we can steep the board in a solution based on muriatic acid (Figure 5.1 (c)). This last step will etch away the surplus of copper, leaving only the electrical scheme on the PCB (Figure 5.1 (d)). We soldered the electronics components on the obtained PCB and finally mounted it in the enclosing box, obtaining the device showed in Figure 5.2.

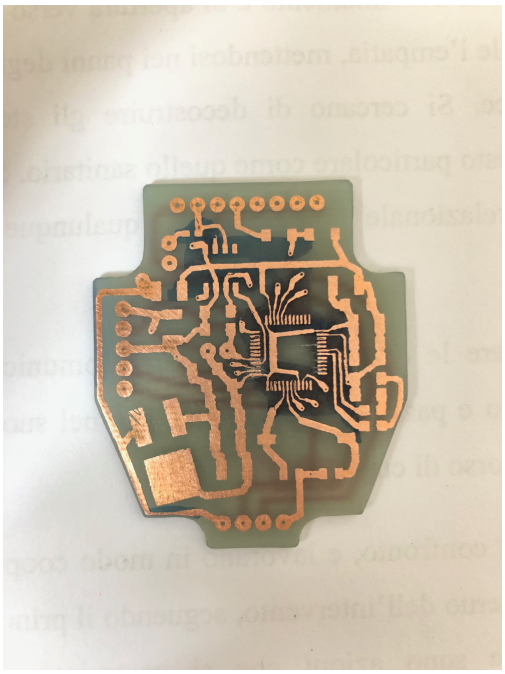
Firmware flashing and different tests

The flashing procedure of the official CANtact firmware [78] can be done with any STM32 Discovery board as a programmer, it's also possible to use other tools that support SWD. We have used the STM32F4DISCOVERY [80] showed in Figure 5.3.
After the first firmware was flashed, the flashing procedure for new firmwares can be done directly through USB, by connecting the boot pins on the CANtact, without the need of the STM32 Discovery board anymore.
While we were using the CANtact in real environments (i.e. on cars) we noticed a strange behavior: on the same car, in the same conditions, sometimes we were not receiving the expected response on a sent CAN frame, while the majority of times the response were correctly received.

(a) *Enclosing box with integrated J1962 connector*

(b) *Blank PCB*

(c) *PCB Etching Process*



(d) *PCB with the printed scheme*

Figure 5.1: CANtact realization phases

Figure 5.2: Final CANtact device

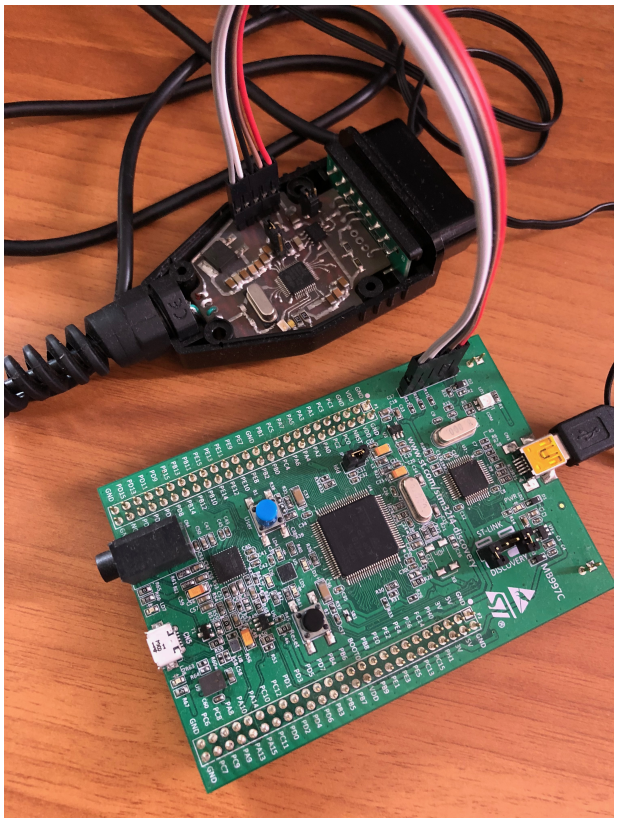


Figure 5.3: STM32F4DISCOVERY connected to the CANtact

We observed such a behavior was happening more frequently on very loaded CAN buses, while was almost absent on unloaded CAN buses. We hypothesized it was a problem of the firmware, which is not able to support such a work load as the one implied by a very busy CAN bus. In order to find the best configuration possible and to get rid of this problem we tried and tested other firmwares. Usually those firmwares were different forks of the original one so the problem kept showing up. At the end, we tried the CandleLightFirmware [81] which is written by zero and it is claimed to be more reliable at high bitrates. We connected the CANtact to the PC and we configured it launching the commands showed in Listing 5.1. With this firmware the CANtact worked correctly and did not stuck again.

```
# 500000 states the CAN bitrate, can be setted accordingly to each
    specific case
sudo ip link set can0 type can bitrate 500000
sudo ifconfig can0 up
# increase the number of frames allowed per kernel transmission
    queue for the queuing discipline
sudo ifconfig can0 txqueuelen 1000
```

Listing 5.1: CANtact CandleLightFirmware PC setup commands

The main drawback of CandleLightFirmware firmware was that it implements the gs_usb protocol, which works out-of-the-box only on Linux kernel with SocketCAN. Since our tool can run independently from the operating system this was not a problem for us.

PC configuration and connection

In order to run the cmap tool we've used a MacBook Pro 8,2 with macOS 10.13.4 operating system, python3 and VirtualBox 5.2.12 with Xubuntu 16.04 installed in a virtual environment. The Xubuntu installation has also python3 and in addition it provides the can-utils package, which includes some useful tools like candump and cansniffer. They functionalities and usage are better detailed in Appendix A.2.
All the tests need to connect the PC to the automobile OBD-II diagnostic connector [82], through the use of the CANtact device. The OBD-II connector, showed in Figure 5.4, specification provides for two standardized hardware interfaces, called type A and type B. Both are female, 16-pin (2x8), D-shaped connectors, and both have a groove between the two rows of pins, but type B's groove is interrupted in the middle. This prevents the insertion of a type A male plug into a type B female socket while allowing a type B male plug to be inserted into a type A female socket. The type A connector is used for vehicles that use 12V supply voltage, whereas type B is used for 24V vehicles and it is required to mark the front of the D-shaped area in blue color. All the tested cars present a type A connector. By the current legislation, the OBD-II connector is required to be within 0.61m of the steering wheel and accessible without the need of any tool [83].
The port pin-out (Figure 5.5) generally differs in function of the manufacturer, but always has on pin 6 the CAN-High wire and on pin 14 the CAN-Low wire.
Once the CANtact device is connected to the USB port of the PC on macOS you can access it directly by reading and writing to /dev/cu.usbmodem####. The #### is an alphanumeric value and will change depending on which USB port the tool is connected to. On Linux, depending on your Linux distribution, CANtact will either appear as /dev/ttyACM# or /dev/ttyUSB#. The # is a numeric value and will depend on how many devices are connected.

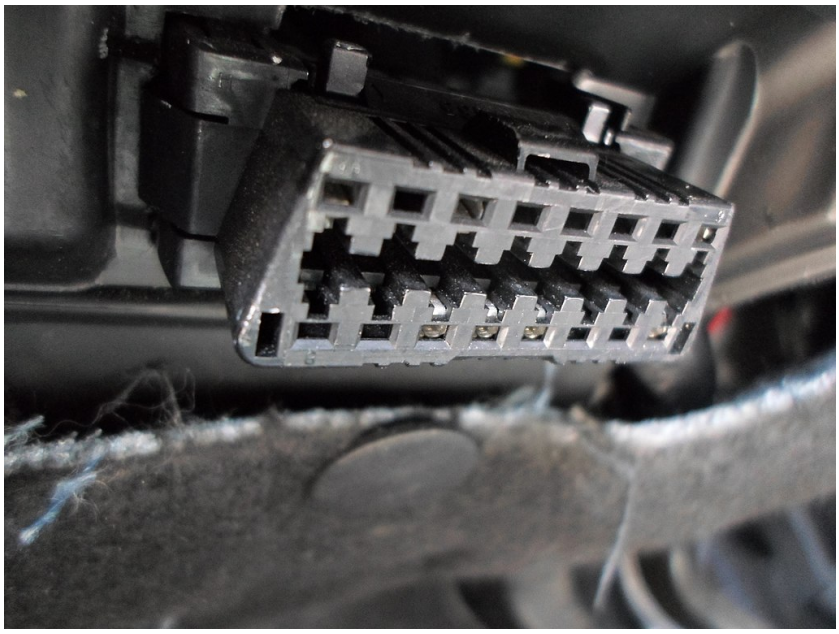
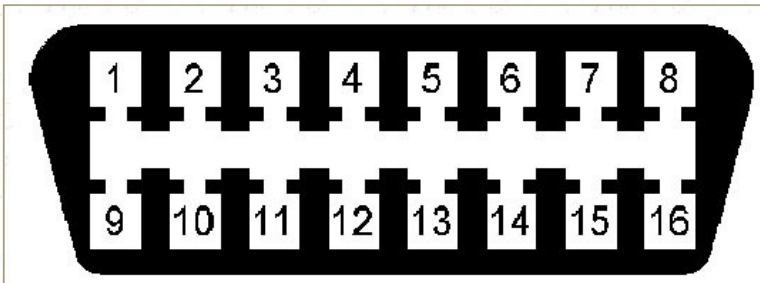


Figure 5.4: OBD-II type A connector



PIN	DESCRIPTION	PIN	DESCRIPTION
1	Vendor Option	9	Vendor Option
2	J1850 Bus +	10	j1850 BUS
3	Vendor Option	11	Vendor Option
4	Chassis Ground	12	Vendor Option
5	Signal Ground	13	Vendor Option
6	CAN (J-2234) High	14	CAN (J-2234) Low
7	ISO 9141-2 K-Line	15	ISO 9141-2 Low
8	Vendor Option	16	Battery Power

Figure 5.5: OBD-II pinout

Once the PC is successfully connected to the CAN bus (Figure 5.6) we can start with the tests.

Figure 5.6: Test setup

5.2 DISCOVERY EXPERIMENT

The Discovery experiment consists in executing the cmap tool in order to perform the scanning of the CAN bus network of an automobile applying the techniques that we have described until now. The goal is to see how many ECUs it is able to find and what information it provides for each discovered ECU.
In order to run the cmap tool for performing a discovery scanning we execute as base the command showed in Listing 5.2.

```
$ python3 cmap.py -d cantact -i "/dev/cu.usbmodemFD121" -I -o
   scan_res_output.json -vvv -f 0.5
```

Listing 5.2: Launch command for discovery experiment of the cmap tool

The full list of options and possible parameters can be consulted in Appendix B. Following the explanation of the options used in command showed in Listing 5.2. Depending on each particular case we may add or remove some options, the reason for it and what different commands are used for will be explained at need

- -d allows to specify the device type, in our case indicates the CANtact;
- -i allows to specify the interface where to connect to the device which on our macOS system results to be /dev/cu.usbmodemFD121;
- -I specifies that in addition to the standard DIDs we are also interested in the additional ranges of DIDs described in Section 3.6;
- -o allows to specify an optional output file where to store the discovery results, in our case scan_res_output.json. Anyway, the output will be printed in the console;
- -v allows to specify the verbosity level by incrementing the number of v in the option, in our case this is set to 3 which means seeing the description of each performed step and the exchanged frames until transport layer protocol. The maximum level is 4 which prints out all the frames that the device transmits to the PC;
- -f allows to specify the functional timeout, which means the minimum amount of time in seconds to wait after a functional request for ECUs responding. In our tests, we observed that a value of 0.5s was always sufficient in order to catch all the responses. With lower values sometimes happened that we've lost some responses.

5.2.1 *Audi A3 Sportback*

The car on which we have tested our tool for all the duration of the development is the Audi A3 Sportback, model 8V motorization 2.0 TDI, deployed in November 2014. In Table 5.1 we report the technical specifications of the automobile, which is showed in Figure 5.7.

Description	Value
Fuel type	Diesel
Fuel System	Direct Injection
Engine Alignment	Transverse
Engine Position	Front
Engine size	1,968 cm3
Number of valves	16 Valves
Aspiration	Turbo (TGV) + Intercooler
Maximum power	150 PS or 110 kW @ 3,500-4,000 RPM
Maximum torque	340 Nm @ 1,750-3,000 RPM
Traction	FWD
Transmission Gearbox	S-Tronic 6 speed Automatic
Top Speed	218 km/h
Acceleration 0 to 100 km/h	8.2 s

Table 5.1: Audi A3 Sportback technical specifications

.

Figure 5.7: Audi A3 Sportback 8V

The cmap tool returned a list of fourteen ECUs discovered on the automobile. For each one it was able to individuate the CAN arbitration IDs in order to communicate and to extract information. The full list of

discovered ECUs is presented in Table 5.2 while some of them will be discussed in detail below.

ECU Name	CAN TX arbitration ID	CAN RX arbitration ID	Vehicle Manufacturer ECU Hardware Number
EV_ECM20TDI01104L906021AD	0x7E0	0x7E8	04L907309A
EV_TCMDQ250021	0x7E1	0x7E9	02E927770AQ
EV_Brake1UDSContiMK100IPB	0x713	0x77D	5Q0907379G
EV_AirCondiFrontVaAU37X	0x746	0x7B0	8V0820043C
EV_DCUDriveSideEWMAXKLO	0x74A	0x7B4	5Q0959593B
EV_DCUPasseSideEWMAXKLO	0x74B	0x7B5	5Q0959592B
EV_SMLSKLOMQB	0x70C	0x776	5Q0953549C
EV_BCMCONTI	0x70E	0x778	5Q0937084AG
EV_EPHVA14AU3700000	0x70A	0x774	5Q0919283
EV_GatewLear	0x710	0x77A	5Q0907530E
EV_SteerAssisMQB	0x712	0x77C	5Q0909144P
EV_MUStd4CPASE	0x773	0x7DD	8V0035864B
EV_DashBoardVDDMQBAB	0x714	0x77E	8V0920860H
EV_AirbaVW20SMEVW37X	0x715	0x77F	5Q0959655N

Table 5.2: Audi A3 Sportback discovered ECUs

As we can observe by the resulting communication IDs of each ECU the offset between them is 0x6A. For example, the EV_AirCondiFrontVaAU37X responded with the arbitration ID is 0x7B0 while the transmission arbitration ID was 0x746, that results in an offset of 0x6A if we subtract the second value from the first one. The ECUs normally used for emission related system are an exception to this rule. This because the standard ISO 15765-4 [66] fixes their values to well defined ones. The transmission arbitration ID is always in the range 0x7E0 - 0x7E7 and the response arbitration ID is always in the range 0x7E8 - 0x7EF, while the offset between the communication arbitration ID of the same ECU is always 0x8. Furthermore, the standard strongly recommends that communication arbitration IDs 0x7E0/0x7E8 should be for Engine Control Module (ECM), while 0x7E1/0x7E9 should be for Transmission Control Module (TCM). In our case those ECUs are respectively the EV_ECM20TDI01104L906021AD and the EV_TCMDQ250021. Seven ECUs out of fourteen contain the information about the

Figure 5.8: Audi A3 Sportback air conditioning ECU

Vehicle Identification Number (VIN), this is always mapped to the DID 0xF190 as specified by ISO 14229-1 standard and coincide with the one engraved on the vehicle. The ECUs that contain this information are: EV_ECM20TDI01104L906021AD; EV_TCMDQ250021; EV_DashBoardVDDMQBAB; EV_AirbaVW20SMEVW37X; EV_BCMCONTI; EV_GatewLear; EV_MUStd4CPASE.
If we further look at the information extracted by each ECU we can find other identification codes as well as hardware and firmware version indication. In order to confirm the functionality of an ECU we have done some researches relative to the obtained codes. Taking as example the ECU EV_AirCondiFrontVaAU37X and searching the Vehicle Manufacturer ECU Hardware Number 8V0820043C on-line, as the name suggests, we confirm that it is the module which allows to control the settings of the air conditioning, showed in Figure 5.8.
The other meaning full information that we found about this ECU are:

- systemSupplierECUHardwareNumber, on DID 0xF192, has value of 80626-173;
- systemSupplierECUHardwareVersionNumber, on DID 0xF193, has value of 0011;
- systemSupplierECUSoftwareNumber, on DID 0xF194, has value of 05553-098;
- systemSupplierECUSoftwareVersionNumber, on DID 0xF195, has value of 0014;
- systemNameOrEngineType, on DID 0xF197, has value of 'AC Automat';
- calibrationDate, on DID 0xF19B, has value of '14 5 26', which is coherent with the year of production of the automobile.

In the information obtained interrogating the ECU we also found many other codes and acronyms at which we were not able to assign a precise meaning since it is proprietary information:

- identificationOptionVehicleManufacturerSpecific, on DID 0xF179, has value of 0070;
- identificationOptionVehicleManufacturerSpecific, on DID 0xF17C, has value of PP1-03013.05.1400010070. The value seams to contain a date embedded 13.05.14, could be the date of production of the ECU or something similar;
- identificationOptionVehicleManufacturerSpecific, on DID 0xF17E, has value of 20026000;
- identificationOptionVehicleManufacturerSpecific, on DID 0xF17F, has value of PP1;
- identificationOptionVehicleManufacturerSpecific, on DID 0xF18A, has value of 'Preh GmbH, Schweinfurter Str. 5 - 9, D-97616 Bad Neustadt a. d. Saale'. Seems to be the company that produces the ECU;
- identificationOptionVehicleManufacturerSpecific, on DID 0xF1A2, has value of 008020;
- identificationOptionVehicleManufacturerSpecific, on DID 0xF1A3, has value of H13;

Another example is the ECU EV_BCMCONTI which description identifies it as being the Body Control Module. Is the ECU which takes care of the locking/unlocking system of the doors and the recognition of the authorized remote controls. The information found about it are similar to the ones presented for the EV_AirCondiFrontVaAU37X ECU. The main difference seems to be about the systemSupplierECUSoftwareVersionNumber, which in this case results to be more verbose: 'E3_0.9 27.02.14 FBL02.71 PPE2.01 MLFB5WK50891E HWMH18B PAR'. It contains the date 27.02.14, supposedly the date of installation or compilation, and some other codes to which we were not able to assign a meaning.
The information found about the other ECUs discovered are equivalent to the ones presented until now.
In order to confirm our results, we found a car workshop which inspected the vehicle with the Bosch KTS 570 diagnostic tool [84]. The list

of ECUs showed as available by the professional diagnostic tool is practically the same as the one obtained by our tool. The principal differences are in the ECU names, since the tool provided all the names in a more human readable form and even in Italian. Not having enough information on the Bosch KTS 570 diagnostic tool, what we suppose is that it uses an internal DBC database file which contains all the descriptions for vehicle model. Due to time constraints and urgent commitments there was not availability from the car workshop to inspect in detail each ECU and to compare eventually provided codes and information with the ones that we extracted with our tool.

5.2.2 *Golf VII Alltrack*

We performed the same test on the Golf VII 4motion Alltrack 2015, showed in Figure 5.9. In Table 5.3 we report the technical specifications of the automobile.

Description	**Value**
Fuel type	Diesel
Fuel System	Direct Injection
Engine Alignment	Transverse
Engine Position	Front
Engine size	1,968 cm3
Number of valves	16 Valves
Aspiration	Turbo (TGV) + Intercooler
Maximum power	184 PS or 135 kW @ 3,500-4,000 RPM
Maximum torque	380 Nm @ 1,750-3,000 RPM
Traction	AWD
Transmission Gearbox	DSG 6 speed Automatic
Top Speed	219 km/h
Acceleration 0 to 100 km/h	7.8 s

Table 5.3: Golf VII 4motion Alltrack technical specifications

.
The cmap tool returned a list of sixteen ECUs discovered on the automobile. For each one it was able to individuate the CAN arbitration IDs in order to communicate and to extract information. Since the information gathered for each ECU are equivalent to the one discussed in the test

Figure 5.9: Golf VII 4motion Alltrack

on the Audi A3 Sportback presented in Section 5.2.1, we just present the full list of discovered ECUs for this automobile, in Table 5.4.

ECU Name	CAN TX arbitration ID	CAN RX arbitration ID	Vehicle Manufacturer ECU Hardware Number
EV_DashBoardVDDMQBAB	0x714	0x77E	5G1920741A
EV_SMLSVALEOMQB	0x776	0x70C	5Q0953569A
EV_AirbaVW20SMEVW37X	0x715	0x77F	5Q0959655D
EV_ECM20TDI01104L906026BT	0x7E0	0x7E8	04L907309R
EV_EPHVA18AU3700000	0x70A	0x774	5Q0919294A
EV_GatewConti	0x710	0x77A	5Q0907530M
EV_TCMDQ250021	0x7E1	0x7E9	02E927770AQ
EV_SteerAssisMQB	0x712	0x77C	5Q0909144R
EV_MUStd4CTSAT	0x773	0x7DD	3Q0035846
EV_DCUDriveSideEWMAXCONT	0x74A	0x7B4	5Q4959593B
EV_ACClimaBHBVW37X	0x746	0x7B0	5G0907044BC
EV_Brake1UDSContiMK100IPB	0x713	0x77D	3Q0907379F
EV_DCUPasseSideEWMAXCONT	0x74B	0x7B5	5Q4959592B
EV_HeadlRegulVWAFSMQB	0x754	0x7BE	7P6907357A
EV_AllWheelContrHA1VW37X	0x70F	0x779	0CQ525130
EV_BodyContrModul1UDSBosc	0x70E	0x778	5Q0937086AK

Table 5.4: Golf VII 4motion Alltrack discovered ECUs

We contacted an official VW car workshop to ask them information about the number of ECUs present on this model. They daily use VAS diagnostic test tool, which is the Volkswagen official diagnostic solution. They informed us that the Golf VII 4motion Alltrack has 16 ECUs, so our tool seems to reveal them all.

5.2.3 *Seat Ibiza I-Tech*

We performed the same test also on the Seat Ibiza I-Tech from year 2015, showed in Figure 5.10. In Table 5.5 we report the technical specifications of the automobile.

Description	Value
Fuel type	Petrol
Fuel System	Indirect Injection
Engine Alignment	Transverse
Engine Position	Front
Engine size	1,198 cm3
Number of valves	12 Valves
Aspiration	N/A
Maximum power	69 PS or 51 kW @ 5,400 RPM
Maximum torque	112 Nm @ 3,000 RPM
Traction	FWD
Transmission Gearbox	5 speed Manual
Top Speed	163 km/h
Acceleration 0 to 100 km/h	13.9 s

Table 5.5: Seat Ibiza I-Tech technical specifications

.

Figure 5.10: Seat Ibiza I-Tech

The cmap tool returned a list of six ECUs discovered on the automobile. For each one it was able to individuate the CAN arbitration IDs in order to communicate and to extract information. The full list of discovered ECUs is presented in Table 5.6.

ECU Name	CAN TX arbitration ID	CAN RX arbitration ID	Vehicle Manufacturer ECU Hardware Number
EV_LWSKLOVW25X	0x751	0x7BB	6R0959654
EV_ECM12MPI02103E906019AL	0x7E0	0x7E8	03E906019E
EV_ImmoUDSMM9RM10	0x711	0x77B	6J0920807K
EV_Brake1ESP90iBOSCH	0x713	0x77D	6R0907379AS
EV_KombiUDSMM9RM10	0x714	0x77E	6J0920807K
EV_AirbaVW10BPAVW250	0x715	0x77F	6R0959655K

Table 5.6: Seat Ibiza I-Tech discovered ECUs

For this vehicle model, we had no other comparison terms in order to validate our results, being a third party automobile we could not take it to the car workshop in order to use the Bosch KTS 570 diagnostic tool and unfortunately we had no response from the official Seat car workshop with information regarding the automobile. Anyway, the results seem to be in line with the one of the other two automobiles. The lower number of ECUs is justified by the fact that this is a lower end model in the market and consequently it has less features. For example, the Seat Ibiza automobile has manual climate control unlike the VW Golf and the Audi A3 automobiles which have the automated once, this implies that there is no need for a dedicated ECU. Another example is the EV_AllWheelContrHA1VW37X present in the VW Golf, which is in charge of controlling the 4motion, and which is missing both from the Seat Ibiza and from the Audi A3 since none of them have 4x4 traction.

5.2.4 *Mercedes-Benz E220*

We performed the test also on the Mercedes-Benz E220 from year 2008, showed in Figure 5.11. In Table 5.7 we report the technical specifications of the automobile.

Description	Value
Fuel type	Diesel
Fuel System	Common Rail
Engine Alignment	Longitudinal
Engine Position	Front
Engine size	2148 cm3
Number of valves	16 Valves
Aspiration	TGV + Intercooler
Maximum power	170 PS or 125 kW @ 2,800 RPM
Maximum torque	400 Nm @ 2,000 RPM
Traction	RWD
Transmission Gearbox	5 speed Automatic
Top Speed	227 km/h
Acceleration 0 to 100 km/h	8.4 s

Table 5.7: Mercedes-Benz E220 technical specifications

.

Figure 5.11: Mercedes-Benz E220

We observed, with the aid of candump tool, that for this automobile on the OBD-II port we can see the whole flow of CAN frames exchanged on the bus for normal operations. In order to avoid interferences and consequently false responses identification we decided to employ the blacklist

feature of the tool. To activate it is sufficient to add the option -k [X] to the launch command, where X states the number of seconds that the tool should sniff the CAN bus and add the IDs of the seen frames to the ones to be ignored when waiting for a response. Using a value of 5 seconds for X leads to a blacklist of 28 arbitration IDs. By further increasing the value of X the length of the blacklist remains unchanged. By using smaller values for X the length of the blacklist decreases significantly, for example assuming X equal to 4 seconds we obtain a blacklist of 16 elements. We consider that for this automobile a value of 5 seconds is the lowest required timer to receive all IDs.
On this automobile, the tool wasn't able to perform as well as on the other three cars. At the first attempt, it didn't discover any ECU. We further analyzed the vehicle CAN bus structure in order to better understand why. We discovered that it is composed by three distinct CAN buses:

- CAN C - Interior bus with a bit rate of 83,3 kbps
- CAN B - Engine bus with a bit rate of 500 kbps
- CAN D - Diagnostic bus with a bit rate of 500kbps

Those three buses are linked together by the ECU Central Gateway Module (CGM), which therefore is connected to all the buses and is the only one connected to the CAN D diagnostic bus. The OBD-II port on which we connected the CANtact device is also positioned on this bus so all the messages to and from the external device mandatorily passed through the CGM ECU. In Figure 5.12 we report a representation of the described topology.
We hypothesized that the CGM was performing some kind of filtering on the packets send by our tool, not letting them to reach the ECU on the CAN C and CAN B buses.
To be sure about our supposition, we started sniffing the packets seen with the CANtact device connected to the OBD-II port, we observed that even if the port is not directly connected to a main CAN bus the CGM ECU permit to CAN frame packets to pass in CAN D bus so it was possible to see them flowing from the diagnostic port. Our goal was to identify a specific packet (or a set of packets) corresponding to an action performed in a controlled manner by us, then to try to send it back on the OBD-II port. If we had observed the same behavior as the one triggered by the manual action means that the CGM module was not performing any kind of filtering. This conclusion is due to the fact that if the CGM

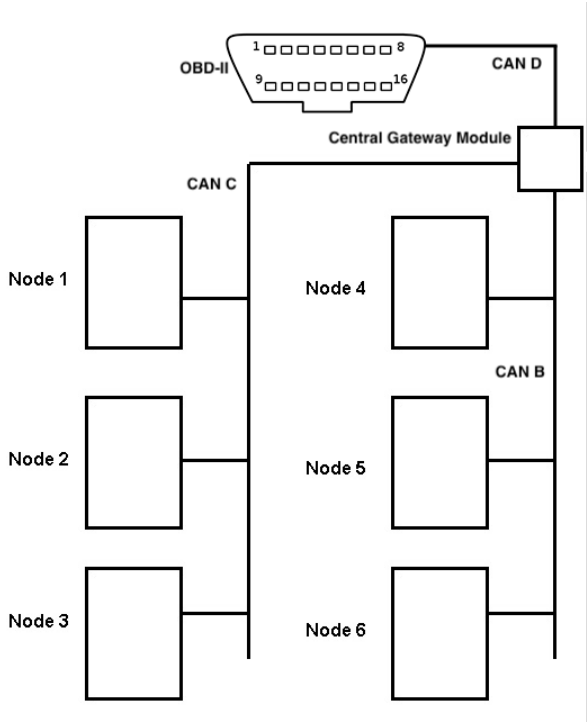


Figure 5.12: Mercedes-Benz E220 CAN bus topology

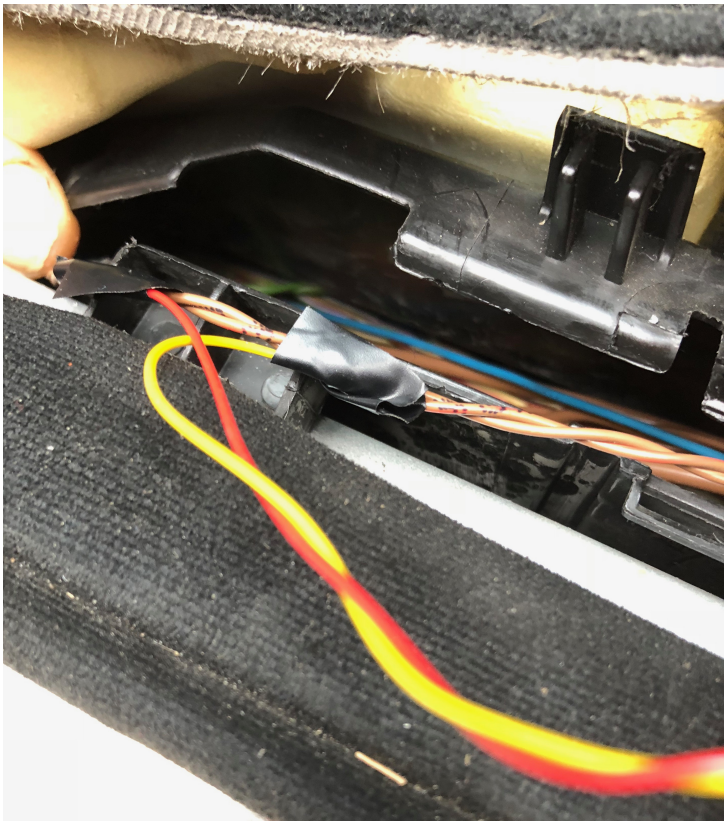


Figure 5.13: CAN B bus connection after CGM

was applying a filtering policy the first packets to filter would be the ones injected from outside and regarding normal operations while the automobile is running. If instead, as we hypothesized, the same behavior was not observed, very probably the packet has been filtered by the CGM. To confirm this hypothesis, we have searched for and founded a connection to the CAN B bus after the CGM. In this way, we could inject the same package without the need to pass through the CGM and see if the action is performed as expected. We found such a connection in the base of the driver's door, see Figure 5.13. With the aid of the candump and cansniffer tools we analyzed the frames seen from the diagnostic port. We decided to try isolating the frames corresponding to the push of the button 'next radio station', from the steering wheel controls. We chose this action principally for three reasons:

- it is an action which implies communication between two different parts of the automobile, with high probability the communication is performed on a CAN bus;
- it is an action which implicates only components related to non-critical systems internal to the automobile. Most probably the CAN frames will be sent on the CAN B bus;
- it is an extemporaneous action, commanded by the user. It should imply well-defined CAN frames which are relatively easy to isolate.

In order to isolate the CAN frames used for transmitting the command to the car radio we observed the output of cansniffer, first without doing anything and then when the manual action was performed, in order to see the new packets transmitted. After a few attempts we isolated the sequence of frames showed in Listing 5.3.

```
ID=0x1ca, DLC=0x4, Data=[0x03,0x01,0x00,0x00]
ID=0x1ca, DLC=0x4, Data=[0x03,0x01,0x00,0x00]
ID=0x1ca, DLC=0x4, Data=[0x03,0x01,0x00,0x00]
ID=0x1ca, DLC=0x4, Data=[0x03,0x00,0x00,0x00]
ID=0x1ca, DLC=0x4, Data=[0x03,0x00,0x00,0x00]
ID=0x1ca, DLC=0x4, Data=[0x00,0x00,0x00,0x00]
```

Listing 5.3: CAN frames sequence when 'next radio station' button is pressed

Once we isolated the sequence we tried to send it back, in the same order and respecting the timing intervals showed by candump, through the OBD-II port using cansend tool. Nothing happened. We detached the CANtact from the OBD-II port and attached the wires of the CAN B

bus directly to the pin 6 (CAN High wire) and pin 14 (CAN Low wire) of the device. We injected again the same sequence and the radio station changed immediately, without touching any physical button of the car. This experiment confirmed our thoughts about the CGM, it was actually performing filtering on the CAN frames sent from the OBD-II port.
With the CANtact device still attached directly to the CAN B bus, we executed again our tool but another time it was not able to identify any ECU. At this point we tried to perform a brute-force scanning, sending the DiagnosticSessionControl request on all the 11b CAN IDs. This can be done by adding the option "-A" to the command which runs the cmap tool. While the brute-force scanning was going on we start seeing some dashboard alert lights turning on. At this point we decided to stop the test in order to avoid abnormal consequences to the automobile's systems. After a key on/off operation all the alert lights turned off, meaning that all the systems returned to a normal behavior.
We performed a posteriori analysis of the CAN frames exchanged between our tool and the ECUs presents on the CAN B bus during the brute-force scanning procedure. We have seen that some of them actually responded to the DiagnosticSessionControl request but with a sort of NegativeResponse, not actually conform to the standard specifications.

```
ID=0x667,DLC=0x8,Data=[0x02,0x10,0x01,0x55,0x55,0x55,0x55,0x55]
ID=0x4E7,DLC=0x8,Data=[0x03,0x7F,0x10,0x12,0x91,0xFF,0x50,0xDD]

ID=0x6A5,DLC=0x8,Data=[0x02,0x10,0x01,0x55,0x55,0x55,0x55,0x55]
ID=0x4E5,DLC=0x8,Data=[0x03,0x7F,0x10,0x12,0x70,0x98,0xD7,0xC2]

ID=0x4E0,DLC=0x8,Data=[0x02,0x10,0x01,0x55,0x55,0x55,0x55,0x55]
ID=0x5FF,DLC=0x8,Data=[0x03,0x7F,0x10,0x21,0x02,0x20,0x00,0x00]

ID=0x641,DLC=0x8,Data=[0x02,0x10,0x01,0x55,0x55,0x55,0x55,0x55]
ID=0x681,DLC=0x8,Data=[0x03,0x7F,0x10,0xA0,0x00,0x00,0x00,0x00]

ID=0x645,DLC=0x8,Data=[0x02,0x10,0x01,0x55,0x55,0x55,0x55,0x55]
ID=0x685,DLC=0x8,Data=[0x03,0x7F,0x10,0xA0,0x00,0x00,0x00,0x00]

ID=0x648,DLC=0x8,Data=[0x02,0x10,0x01,0x55,0x55,0x55,0x55,0x55]
ID=0x688,DLC=0x8,Data=[0x03,0x7F,0x10,0xA0,0x00,0x00,0x00,0x00]

ID=0x64B,DLC=0x8,Data=[0x02,0x10,0x01,0x55,0x55,0x55,0x55,0x55]
ID=0x68C,DLC=0x8,Data=[0x03,0x7F,0x10,0xA0,0x00,0x00,0x00,0x00]
```

Listing 5.4: Examples of CAN frames exchange on Mercedes-Benz E220

Listing 5.4 reports some examples of the CAN frames obtained in response to the DiagnosticSessionControl request made by our tool. Looking to the first three bytes of the Data field it seems a normal NegativeResponse. Going further we notice that:

- the fourth byte, which should represent the NRC, in some cases has a value of 0xA0. This NRC value in the standard is part of the range 0x94 - 0xEF, which is reported as being reserved for future definition, so not even like manufacturer or supplier specific. It should not be used yet;
- the first tree examples has defined values after the fourth byte, thing that is not contemplated by the standard;
- the padding value used for last five examples is not the usual 0xAA, specified by the standard for response messages and used by the other two automobiles, but a 0x00 value.

When the cmap tool was launched in normal mode, trying to exploit the functional addressing characteristics, no results were obtained. When we launched the cmap tool with the brute-force option at least we obtained some kind of feedback. This observation lead us to think that the problem was not the CGM performing some kind of filter but the absence of the functional addressing feature in the protocol implementation. Another point to this hypothesis is that even if the CGM is acting a filtering policy the diagnostic messages which we use should be allowed to pass through, since this is the normal behavior when a diagnostic is performed. In order to prove this point, we launched again the brute-force scanning, but this time with the CANtact device connected to the OBD-II port. Also this time, while the scanner was running, the alert lights of the dashboard turned on and, to avoid possible damages to the vehicle, we opted to stop the scan. Any way the goal of this test was reached. Even if the CGM was performing filtering on frames coming from OBD-II port, it was not filtering the diagnostic packets.
The conclusion that we can draw from what aforementioned and given the year in which the vehicle was produced is that most probably the Mercedes-Benz E220 implements some sort of proprietary diagnostic protocol, which has some common characteristics with the UDS protocol but is not the same, reason for which our tool was not able to provide the expected results on this automobile and caused unattended behaviors.

5.3 VULNERABILITY TEST EXPERIMENT

The vulnerability test experiment consists in executing the cmap tool in order to perform the available tests meant to identify the known vulnerabilities on an automobile CAN bus. In our case, it will perform the implemented test relative to the vulnerability described in Section 3.7.1. In order to run the cmap tool for performing the test of the known vulnerability we executed the command showed in Listing 5.5.

```
$ python3 cmap.py -d cantact -i "/dev/cu.usbmodemFD121" -o
   scan_res_output.json -vvv -f 0.5 -t
```

Listing 5.5: Launch command for vulnerability test experiment with the cmap tool

The differences with the command showed in Section 5.2 for the discovery test are:

- the absence of the option -I, since in this case we are not interested in obtaining as much as possible information about each ECU;
- the introduction of the option -t. This option tells the software to execute the implemented test for the known vulnerabilities.

We performed the vulnerability test only on the Audi A3 Sportback. We didn't execute the vulnerability test on other cars since we don't have a precise knowledge of how each manufacturer has implemented the UDS services. For example, we could had got that the service CommunicationControl was not under SecurityAccess but ECUReset instead was protected by SecurityAccess. In this scenario, we would have been able to successfully disable the standard messages of an ECU using the CommunicationControl request but we would have not been able to take it back to normal behavior by using ECUReset request. Even if we are firmly convinced that such a potential damage could be corrected with the aid of a manufacturer specific tool, we preferred to not take any risk on automobiles made available by third parties.

5.3.1 *Audi A3 Sportback*

The execution of the test embedded in the known vulnerability framework on the Audi A3 Sportback failed. This means that the vulnerability described is actually present in the ECUs of this automobile. To witness

this fact, we reported in Figure 5.14 snapshots of the dashboard alert lights on while the test was running. This behavior is due to the fact that the ECU responsible for the respective functions stopped sending the normal messages. After the ECUReset request on each ECU all the systems returned to the normal behavior and the dashboard alert lights turned off.

As we can see from the dashboard snapshots security relevant systems present functioning anomalies, meaning that the relative ECUs stopped communicating as normally they do. While the test was running for some instants the headlights system and interior lights stopped. This should be the time elapsed between the ComunicationControl request and the ECUReset request for the ECU in-charge of controlling those systems.

We executed the test twice: first time with engine on and second time with engine off; in order to analyse the output of the cmap tool in different conditions and to spot out the differences in some ECUs behavior to the CommunicationControl request. When the test was executed with the engine on six ECUs refused to execute the CommunicationControl request:

- EV_ECM20TDI01104L906021AD responded with a NegativeResponse with NRC corresponding to engineIsRunning;
- EV_TCMDQ250021 responded with a NegativeResponse with NRC corresponding to engineIsRunning
- EV_MUStd4CPASE responded with a NegativeResponse with NRC corresponding to conditionsNotCorrect
- EV_AirCondiFrontVaAU37X responded with a NegativeResponse with NRC corresponding to rpmTooHigh
- EV_GatewLear responded with a NegativeResponse with NRC corresponding to rpmTooHigh
- EV_DashBoardVDDMQBAB responded with a NegativeResponse with NRC corresponding to rpmTooHigh

All the remaining ten ECUs correctly interrupted normal messages after the CommunicationControl request was send.

When the test was executed with the engine off all the ECUs, included the one that in the previous scenario responded with a NegativeResponse, correctly interrupted normal messages after the CommunicationControl request was send.

(a)

(b)

Figure 5.14: Audi A3 Sportback dashboard snapshots while performing the test

In no way, any ECU responded with a NegativeResponse with NRC corresponding to securityAccessDenied, as it would have been necessary to pass the test.

6

LIMITATIONS AND FUTURE WORK

Our work is mainly focused on the UDS implementation over CAN bus, although UDS protocol is also defined for other protocols which specify the OSI Model from the transport layer to the physical layer: ISO 14229-4 [61] defines UDS over FlexRay [85] [86]; ISO 14229-5 [62] defines UDS over DoIP [87] and ISO 14229-7 defines UDS over Local Interconnect Network (LIN) [88]. We had no access to automobiles with such characteristics to analyze implementations of UDS over other bus systems, but in our tool implementation the application layer and transport layer are well defined and divided. With adequate testing automobiles, the techniques presented and implemented in this thesis for UDS over CAN could be extended to the other technologies mentioned.
Our discovery technique for what regards CAN frames with 11-bit arbitration ID is based on conventions and observed behaviors, consequently it is not absolutely certain. This limitation will be naturally overcame through time due to the gradual migration of diagnostic implementations to the use of CAN frames with 29-bit arbitration IDs, for which we propose a deterministic approach. Unfortunately to date we were not able to find any automobile which employs this type of diagnostic implementation.
The concept of broadcast addressing and features which allows to obtain information directly from the ECUs are not exclusive characteristics of the UDS diagnostic protocol. Indeed, there are other diagnostic protocols, proprietary or not, which could have similar characteristics. Although they have not been tested during the realization of this work, they could specify similar features which could be exploited in order to obtain other discovery methods, not relying neither on brute-force nor on DBC files. An example of such a diagnostic protocol is KWP2000 [57]. It is one of the protocols on which UDS specifications are based, it might have closely related features.
Many researchers are currently focusing on discovering new vulnerabilities in automotive systems. It would be advisable to integrate the vulnerability test framework with specific tests regarding the new vulnerabilities discovered.

7

CONCLUSIONS

This thesis has presented and analyzed a technique for performing ECU discovery on CAN networks, without relying neither on brute-force nor on existing databases for specific manufacturers or automobile models, but exploiting the until now underestimated functional addressing feature of UDS diagnostic protocol. For each discovered ECU we provided a methodology for advanced information gathering, in order to identify the functionality performed by the ECU in the automotive system.
We proposed a structured framework in order to implement tests for known vulnerabilities and perform automated vulnerability assessment. As the UDS is an applicative layer protocol the presented work is valuable to other kind of automotive networks (as DoIP or LIN), for which the UDS standard specifies the transport and network layers.
The research has been focusing on automotive CAN networks. We traduced our approach in the implementation of the cmap tool, which encloses the discovery and vulnerability test frameworks. While developing our tool we extended and improved the pyvit open-source library.
A novel vulnerability in the UDS implementations has been analyzed and a specific test was implemented with the aid of the vulnerability test framework as an experimental proof-of-concept of the validity of the framework itself.
The discovery test was performed on four modern unaltered vehicles, proving the effectiveness and efficiency of the tool on real automotive systems while the vulnerability assessment test was performed on a single vehicle proving its correctness and showing the temporary effects on the automobile itself.
The discovery framework structure lends well to support integration with other discovery methods. Future research may study other techniques in order to identify the ECUs present on the automotive network and integrate the discovery framework itself. The vulnerability framework can be constantly updated with test for new vulnerabilities discovered by researchers.
Our hope is to provide valuable and concrete contribution to lower the initial effort in performing automotive security analyses, lowering the entry barriers for performing automotive security research and facilitating researchers work of improving the tomorrow vehicle's security.

BIBLIOGRAPHY

[1] Eurostat. (April 2018). Freight transport statistics, [Online]. Available: http://ec.europa.eu/eurostat/statistics-explained/index.php?title=Freight_transport_statistics_-_modal_split (cit. on p. 1).

[2] Bureau of Transportation Statistics. (2017). Number of U.S. Aircraft, Vehicles, Vessels, and Other Conveyances, [Online]. Available: https://www.bts.gov/content/number-us-aircraft-vehicles-vessels-and-other-conveyances (cit. on p. 1).

[3] World Health Organization. (November 2015). Registered vehicles data by country, [Online]. Available: https://www.bts.gov/content/number-us-aircraft-vehicles-vessels-and-other-conveyances (cit. on p. 1).

[4] Matthew Nitch Smith. (April 2016). The number of cars worldwide is set to double by 2040, [Online]. Available: https://www.weforum.org/agenda/2016/04/the-number-of-cars-worldwide-is-set-to-double-by-2040 (cit. on p. 1).

[5] Wikipedia. (May 2018). Motor vehicle fatality rate in U.S. by year, [Online]. Available: https://en.wikipedia.org/wiki/Motor_vehicle_fatality_rate_in_U.S._by_year (cit. on p. 1).

[6] flexautomotive. (September 2015). CAN bus (Controller Area Network), [Online]. Available: http://www.flexautomotive.net/EMCFLEXBLOG/post/2015/09/08/can-bus-for-controller-area-network (cit. on p. 2).

[7] Robert Bosch GmbH. (2016). KTS 590, [Online]. Available: https://it-ww.bosch-automotive.com/it/products_workshopworld/testing_equipment_products/ecu_diagnostics/kts560_1/kts560_4 (cit. on p. 4).

[8] Helmut, Frank and Schmidts, Uwe. (March 2007). Vehicle Diagnostics – The whole Story, [Online]. Available: https://vector.com/portal/medien/cmc/press/PDG/Diagnostics_Congress_ElektronikAutomotive_200703_PressArticle_EN.pdf (cit. on pp. 3, 45).

[9] Muneeswaran, A., "Automotive Diagnostics Communication Protocols Analysis- KWP2000, CAN, and UDS," February 2015 (cit. on p. 3).

[10] Checkoway, Stephen, McCoy, Damon, Kantor, Brian, Anderson, Danny, Shacham, Hovav, Savage, Stefan, Koscher, Karl, Czeskis, Alexei, Roesner, Franziska, and Kohno, Tadayoshi, "Comprehensive Experimental Analyses of Automotive Attack Surfaces," August 2011. [Online]. Available: http://www.autosec.org/pubs/cars-usenixsec2011.pdf (cit. on p. 4).

[11] Foster, Ian, Prudhomme, Andrew, Koscher, Karl, and Savageo, Stefan, "Fast and Vulnerable: A Story of Telematic Failures," August 2015. [Online]. Available: http://www.autosec.org/pubs/woot-foster.pdf (cit. on p. 4).

[12] ——, "Fast and Vulnerable: A Story of Telematic Failures," August 2015. [Online]. Available: http://www.autosec.org/pubs/woot-foster.pdf (cit. on p. 4).

[13] Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S., "Experimental security analysis of a modern automobile," May 2010. [Online]. Available: http://www.autosec.org/pubs/cars-oakland2010.pdf (cit. on pp. 4, 54).

[14] Miller, Charlie and Valasek, Chris, "A Survey of Remote Automotive Attack Surfaces," Aug 2014 (cit. on p. 4).

[15] ——, "Remote Exploitation of an Unaltered Passenger Vehicle," Aug 2015 (cit. on pp. 4, 40).

[16] ——, "Adventures in automotive networks and control units," Aug 2013 (cit. on pp. 4, 54, 55).

[17] United States Government Accountability Office, "Vehicle cybersecurity - dot and industry have efforts under way, but dot needs to define its role in responding to a realworld attack," March 2016. [Online]. Available: https://www.gao.gov/assets/680/676064.pdf (cit. on p. 5).

[18] BBC, "Fbi warns on risks of car hacking," March 2016. [Online]. Available: http://www.bbc.com/news/technology-35841571 (cit. on p. 5).

[19] Markey, Sen. and Edward, J. (2017). SPY Car Act of 2017, [Online]. Available: https://www.congress.gov/bill/115th-congress/senate-bill/680?q=%7B%22search%22%3A%5B%22spy+car+act%22%5D%7D&r=1 (cit. on p. 5).

[20] Kowall, Mike, Horn, Ken, Schmidt, Wayne A., and Warren, Rebekah. (2016). Senate Bill 0927 (2016), [Online]. Available: http://legislature.mi.gov/(S(3ts1aofktaijn1uuqhxnzufr))/mileg.aspx?page=getObject&objectName=2016-SB-0927 (cit. on p. 5).

[21] nmap.org. (March 2007). nmap, [Online]. Available: https://nmap.org (cit. on p. 5).

[22] OCTech. (2017). How do i know whether my vehicle is obd-ii compliant? [Online]. Available: https://obdsoftware.desk.com/customer/en/portal/articles/2909558-how-do-i-know-whether-my-vehicle-is-obd-ii-compliant (cit. on p. 6).

[23] GmbH, Robert Bosch. (1991). CAN Specification Version 2.0, [Online]. Available: http://esd.cs.ucr.edu/webres/can20.pdf (cit. on p. 10).

[24] Robert Bosch GmbH. (2018). Robert Bosch GmbH, [Online]. Available: https://www.bosch.com/corporate-information/ (cit. on p. 10).

[25] SAE International. (2018). SAE International, [Online]. Available: https://www.sae.org/about/ (cit. on p. 10).

[26] Wikipedia. (June 2018). CAN Bus, [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus (cit. on p. 10).

[27] International Organization for Standardization, *ISO 11898:1993 Road vehicles – Interchange of digital information – Controller area network (CAN) for high-speed communication*, 1993 (cit. on p. 10).

[28] ——, *ISO 11898-1:2015 Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*, 2015 (cit. on pp. 10, 21).

[29] ——, *ISO 11898-2:2016 Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*, 2016 (cit. on p. 13).

[30] Texas Instruments. (August 2012). Industry's first integrated CAN transceiver microcontroller solution, [Online]. Available: https://www.nxp.com/docs/en/brochure/75017050.pdf (cit. on p. 14).

[31] Kvaser. (August 2012). Microcontrollers with CAN, [Online]. Available: https://www.kvaser.com/about-can/can-controllers-transceivers/microcontrollers-with-can/ (cit. on p. 14).

[32] NXP Semiconductors N.V. (2011). Introduction to the Controller Area Network (CAN), [Online]. Available: http://www.ti.com/lit/an/sloa101b/sloa101b.pdf (cit. on p. 15).

[33] SAE International, *Single Wire CAN Network for Vehicle Applications*, 2000 (cit. on p. 15).

[34] International Organization for Standardization, *ISO 15765-2 Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services*, 2016 (cit. on pp. 16, 61).

[35] ——, *ISO 14229-1 Road vehicles – Unified diagnostic services (UDS) – Part 1: Specification and requirements*, 2013 (cit. on pp. 24, 61).

[36] Wikipedia. (May 2018). Keyword Protocol 2000, [Online]. Available: https://en.wikipedia.org/wiki/Keyword_Protocol_2000 (cit. on p. 25).

[37] International Organization for Standardization, *ISO 15765-3:2004 Road vehicles – Diagnostics on Controller Area Networks (CAN) – Part 3: Implementation of unified diagnostic services (UDS on CAN)*, 2004 (cit. on p. 25).

[38] Wikipedia. (May 2018). Unified Diagnostic Services, [Online]. Available: https://en.wikipedia.org/wiki/Unified_Diagnostic_Services (cit. on pp. 25, 47).

[39] Smith, Craig, *THE CAR HACKER'S HANDBOOK*, Chun, Laurel, Ed. William Pollock, 2016 (cit. on pp. 40, 41).

[40] Palanca, Andrea. (September 2016). A stealth, selective, link-layer denial-of-service attack against automotive networks, [Online]. Available: https://www.politesi.polimi.it/handle/10589/126393 (cit. on p. 40).

[41] Verma, Abhi. (March 2018). Securing automotive software over the air updates, [Online]. Available: https://excelfore.com/blog/securing-automotive-software-air-updates/ (cit. on p. 40).

[42] ResearchAndMarkets.com. (March 2018). Global Automotive Over-the-air (OTA) Updates Market 2018-2022: Market to Grow at a CAGR of 58.15%, [Online]. Available: https://www.businesswire.com/news/home/20180314005455/en/Global-Automotive-Over-the-air-OTA-Updates-Market-2018-2022 (cit. on p. 40).

[43] Gaines, Lynda. (June 2014). 5 Automotive Embedded Software Recalls and Updates we've seen in 2014, [Online]. Available: https://www.vectorcast.com/blog/2014/06/5-automotive-embedded-software-recalls-and-updates-weve-seen-2014-0 (cit. on p. 40).

[44] dw.com. (February 2018). BMW to recall 12,000 cars over faulty emissions software, [Online]. Available: http://www.dw.com/en/bmw-to-recall-12000-cars-over-faulty-emissions-software/a-42721910 (cit. on p. 40).

[45] BBC, "Fiat chrysler recalls 1.4 million cars after jeep hack," July 2015. [Online]. Available: http://www.bbc.com/news/technology-33650491 (cit. on p. 40).

[46] Embitel. (July 2017). ECU is a Three Letter Answer for all the Innovative Features in Your Car: Know How the Story Unfolded, [Online]. Available: https://www.embitel.com/blog/embedded-blog/automotive-control-units-development-innovations-mechanical-to-electronics (cit. on p. 40).

[47] CaringCaribou. (2018). Caring Caribou, [Online]. Available: https://github.com/CaringCaribou/caringcaribou (cit. on p. 41).

[48] CANToolz. (2017). CANToolz, [Online]. Available: https://github.com/CANToolz/CANToolz (cit. on p. 41).

[49] ——, (August 2017). UDS Scan, [Online]. Available: https://github.com/CANToolz/CANToolz/blob/master/examples/uds_scan.py (cit. on p. 42).

[50] AUTOSAR, *Requirements on Diagnostic*, March 2017. [Online]. Available: http://esd.cs.ucr.edu/webres/can20.pdf (cit. on p. 45).

[51] ——, (2018). AUTOSAR, [Online]. Available: https://www.autosar.org (cit. on p. 45).

[52] ——, (2018). AUTOSAR Core Partners, [Online]. Available: https://www.autosar.org/about/current-partners/ (cit. on p. 45).

[53] Ross Tech. (2014), [Online]. Available: http://forums.ross-tech.com/showthread.php?1905-UDS-Vs-non-UDS (cit. on p. 45).

[54] KPIT. (2016), [Online]. Available: https://www.kpit.com/campaign/decoding-vehicle-diagnostics-with-kpit (cit. on p. 45).

[55] Softing. (2015), [Online]. Available: https://blog.softing.com/blog/automotive-electronics/diagnostics-odx-otx-uds-and-other-market-standards/ (cit. on p. 45).

[56] International Organization for Standardization, *ISO 9141:1989 Road vehicles – Diagnostic systems – Requirements for interchange of digital information*, 1989 (cit. on p. 46).

[57] ——, *ISO 14230-3:1999 Road vehicles – Diagnostic systems – Keyword Protocol 2000 – Part 3: Application layer*, 1999 (cit. on pp. 46, 109).

[58] Kunbus. (July 2016). THE K-LINE, [Online]. Available: https://www.kunbus.com/k-line.html (cit. on p. 46).

[59] Softing. (August 2016). K-Line - ISO 9141E, [Online]. Available: https://automotive.softing.com/en/standards/bus-systems/k-line-iso-9141.html (cit. on p. 46).

[60] Embitel. (March 2018). KWP 2000 and UDS Protocols for Vehicle Diagnostics: An Analysis and Comparison, [Online]. Available: https://www.embitel.com/blog/embedded-blog/kwp-2000-and-uds-protocols-for-vehicle-diagnostics-an-analysis-and-comparison (cit. on p. 46).

[61] International Organization for Standardization, *ISO 14229-4:2012 Road vehicles – Unified diagnostic services (UDS) – Part 4: Unified diagnostic services on FlexRay implementation*, 2012. [Online]. Available: https://www.iso.org/standard/55285.html (cit. on pp. 46, 109).

[62] ——, *ISO 14229-5:2013 Road vehicles – Unified diagnostic services (UDS) – Part 5: Unified diagnostic services on Internet Protocol implementation*, 2013 (cit. on pp. 47, 109).

[63] ——, *ISO 14229-7:2015 Road vehicles – Unified diagnostic services (UDS) – Part 7: UDS on local interconnect network*, 2015 (cit. on p. 47).

[64] European parliament and Council. (1998). Direttiva 98/69/CE, [Online]. Available: https://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31998L0069:IT:HTML (cit. on p. 47).

[65] Preet. (October 2017). DBC Format, [Online]. Available: http://socialledge.com/sjsu/index.php/DBC_Format (cit. on pp. 47, 51).

[66] International Organization for Standardization, *ISO 15765-4:2016 Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 4: Requirements for emissions-related systems*, 2016 (cit. on pp. 48, 91).

[67] Miller, Charlie and Valasek, Chris, "CAN Message Injection," 2016 (cit. on p. 54).

[68] Evenchick, Erik. (2018). CANtact, [Online]. Available: http://linklayer.github.io/cantact/ (cit. on pp. 57, 58, 82).

[69] Liechti, Chris. (2018). pyserial, [Online]. Available: https://pythonhosted.org/pyserial/ (cit. on p. 57).

[70] Volkswagen Research. (2012). SocketCAN, [Online]. Available: https://www.kernel.org/doc/Documentation/networking/can.txt (cit. on p. 59).

[71] PEAK-System. (2018). PCAN-USB, [Online]. Available: https://www.peak-system.com/PCAN-USB.199.0.html?&L=1 (cit. on p. 59).

[72] Wikipedia. (2018). OBDII PIDs, [Online]. Available: https://en.wikipedia.org/wiki/OBD-II_PIDs (cit. on p. 61).

[73] Pythonspot. (2017). Inner classes, [Online]. Available: https://pythonspot.com/inner-classes/ (cit. on p. 61).

[74] Python Software Foundation. (2018). Dict, [Online]. Available: https://docs.python.org/3/library/stdtypes.html#dict (cit. on p. 61).

[75] Wikipedia. (June 2018). Factory method pattern, [Online]. Available: https://en.wikipedia.org/wiki/Factory_method_pattern (cit. on p. 67).

[76] Liu, Jianhao and Yan, Minrui. (March 2017). A visualization tool for evaluating can-bus cybersecurity, [Online]. Available: https://www.slideshare.net/CanSecWest/minrui-yanjianhaoliu-a-visualization-tool-for-evaluating-canbus-cybersecurity/15 (cit. on p. 73).

[77] FIRST. (2017). Common Vulnerability Scoring System SIG, [Online]. Available: https://www.first.org/cvss/ (cit. on p. 78).

[78] Evenchick, Erik. (2015). CANtact, [Online]. Available: https://github.com/linklayer/cantact-fw (cit. on p. 82).

[79] ——, (2015). CANtact, [Online]. Available: https://github.com/linklayer/cantact-hw (cit. on p. 82).

[80] STMicroelectronics. (2017). STM32F4DISCOVERY, [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/user_manual/70/fe/4a/3f/e7/e1/4f/7d/DM00039084.pdf/files/DM00039084.pdf/jcr:content/translations/en.DM00039084.pdf (cit. on p. 82).

[81] Denkmair, Hubert. (2017). CandleLightFirmware, [Online]. Available: https://github.com/HubertD/candleLight_fw (cit. on p. 85).

[82] SAE International, *SAE J1962 20020 Diagnostic Connector*, July 2017 (cit. on p. 86).

[83] Wikipedia. (June 2018). On-board diagnostics, [Online]. Available: https://en.wikipedia.org/wiki/On-board_diagnostics (cit. on p. 86).

[84] Robert Bosch GmbH. (2014). KTS 570, [Online]. Available: http://se-ww.bosch-automotive.com/en/products_workshopworld_4/testing_equipment_products_4/ecu_diagnostics_4/kts_22/kts_570_1 (cit. on p. 93).

[85] International Organization for Standardization, *ISO 17458-2:2013 Road vehicles – FlexRay communications system – Part 2: Data link layer specification*, 2013 (cit. on p. 109).

[86] ——, *ISO 17458-4:2013 Road vehicles – FlexRay communications system – Part 4: Electrical physical layer specification*, 2013 (cit. on p. 109).

[87] ——, *ISO 13400-2:2012 Road vehicles - Diagnostic communication over Internet Protocol (DoIP) – Part 2: Transport protocol and network layer services*, 2012 (cit. on p. 109).

[88] ——, *ISO 17987-2:2016 Road vehicles – Local Interconnect Network (LIN) – Part 2: Transport protocol and network layer services*, 2016 (cit. on p. 109).

[89] Haddon, Robert and Verari Systems Inc. (April 2018). slcand, [Online]. Available: https://github.com/linux-can/can-utils/blob/master/slcand.c (cit. on p. 121).

[90] linux-can. (2018). can-utils, [Online]. Available: https://github.com/linux-can/can-utils (cit. on p. 122).

A

APPENDIX

A.1 THE SERIAL BUS SPEED SETTING

For a serial device, the CAN bus speed can be set sending the characters present in Table A.1, according to the bus speed desired

Characters	Bus Speed
S0	10kbps
S1	20kbps
S2	50kbps
S3	100kbps
S4	125kbps
S5	250kbps
S6	500kbps
S7	750kbps
S8	1Mbps

Table A.1: Serial bus speed specification

In case of using the device through slcand [1] [89] the bus sped can be set with option -sX, where X follows the same numeration as bus speed indication as reported in Table A.1

A.2 USE THE CANTACT WITH SOCKETCAN

To use CANtact with SocketCAN, we need to run slcand. For example:

```
sudo slcand -o -c -s6 /dev/ttyACM0 can0
```

Listing A.1: slcand example command for CANtact

This command creates a new device called cano that is connected to the CANtact at /dev/ttyACMo. It will open the device when starting (-o), close the device when finished (-c), and set the speed mode to 6 (-s6). After running this command, you will need to enable the interface:

1 slcand is a user space daemon for serial line CAN interface driver SLCAN

```
sudo ifconfig can0 up
```

Listing A.2: CAN interface enabling command

Once a device is enabled, there are several utilities, part of the can-utils [90] package, that can be used.

candump

candump displays messages on the bus in real-time. To show all traffic in real time on device cano, run:

```
candump can0
```

Listing A.3: candump command example

The displayed messages can be filtered using a mask and identifier. Two filter types are available:

- [can_id]:[can_mask] matches when [received_can_id] & [can_mask] == [can_id] & [mask]
- [can_id]~[can_mask] matches when [received_can_id] & [can_mask] != [can_id] & [mask]

Examples:

```
# Only show messages with ID 0x123 on can0:
candump can0,0x123:0x7FF
# Only show messages with ID 0x123 or ID 0x456 on can0:
candump can0,0x123:0x7FF,0x456:0x7FF
```

Listing A.4: candump filtering command examples

cansend

cansend sends a single CAN frame onto the bus. You will have to specify a device, an identifier and data bytes to send. For example:

```
cansend can0 123#1122334455667788
```

Listing A.5: cansend command example

will send a message on interface cano with identifier 0x123 and data bytes [0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88]. Note that this tool always assumes values are given in hexadecimal.

cangen

cangen can generate random CAN data, which can be useful for testing. For more information, run cangen for detailed usage information.

cansniffer

cansniffer displays frames that are currently on the bus, but filters out frames with data that is not changing. This is very useful for reverse engineering CAN bus systems. For more information, run cansniffer for detailed usage information.

B

APPENDIX

CMAP HELP MENU

In Listing B.1 is reported the cmap command structure to use in order to lunch the tool. Table B.1 reports the available options as showed by the help menu of the cmap tool. Each software option is explained and in some cases, provide tips and examples of use.

```
usage: cmap.py [-h] [-k BLACK_LIST] [-o OUTPUT_FILE] [-l LOAD_FILE]
   [-a] [-A]
   [-I] [-M] [-B] [-d {socketcan,cantact,logplayer,obdlinksx}]
   [-x] [-i INTERFACE] [-b BITRATE] [-v] [-f FUNCTIONAL_TIMEOUT]
   [-t]

Find out the ECUs in the car and test them!
```

Listing B.1: The cmap tool command structure

Option	Description
-h, –help	show this help message and exit
-k BLACK_LIST, –black-list BLACK_LIST	read all the CAN packets on the bus and store them IDs in a blacklist in order to ignore them in successive communications
-o OUTPUT_FILE, –output-file OUTPUT_FILE	output to file in JSON format
-l LOAD_FILE, –load-file LOAD_FILE	load ECUs in file resulting from a previous scan
-a, –all-diagnostic-addr	scan all the 11b diagnostic addresses (0x700 -> 0x7FF)
-A, –all-addr	scan all the 11b addresses
-I, –full-info	try to find a lot more info about each ECU, the scan process is slower
-M, –all-info	try to find a lot more info about each ECU, the scan process is slower
-B, –can-b	use CAN B, or rather use 29b CAN IDs
-d {socketcan, cantact, logplayer, obdlinksx}, –device {socketcan, cantact, logplayer, obdlinksx}	specify the kind of device to connect to. 'obdlinksx' works only when used with CAN IDs of 11b
-x, –use-filter	when making discovery request filter on first digit of the CAN ID, for example if request is on 0x7DF only responses with 0x7XX will be accepted
-i INTERFACE, –interface INTERFACE	parameter in order to connect to the device, e.g.: for cantact '/dev/cu.usbmodemFD121', for socketcan 'can0' or for logplayer path to log file
-b BITRATE, –bitrate BITRATE	bitrate of the CAN bus
-v, –verbose	level of information to output during processing. 2 shows the descriptions of the steps. 3 also prints the exchanged frames until transport layer protocol. The maximum level is 4 which prints out all the frames that the device transmits to the PC
-f FUNCTIONAL_TIMEOUT, –functional-timeout FUNCTIONAL_TIMEOUT	time in seconds to wait for responses after an UDS functional request. Could be sufficient 100ms, for being sure to catch all the responses use values >= 0.5s. When scanning multiple addresses lower is the value faster will be the scan.
-t, –vuln-test	perform the available set of vulnerability tests on founded ECUs. If this option is used in combination with -l option the discovery procedure is skipped and the tests are performed only on the ECUs loaded from the file

Table B.1: The cmap tool options, can be obtained with the command python3 cmap.py -h