

# SECURING THE IN-VEHICLE NETWORK

BY

ANDREEA-INA RADU

A thesis submitted to  
University of Birmingham  
for the degree of  
DOCTOR OF PHILOSOPHY

School of Computer Science  
College of Engineering and Physical Sciences  
University of Birmingham  
October 2019

UNIVERSITY OF  
BIRMINGHAM

**University of Birmingham Research Archive**

**e-theses repository**

This unpublished thesis/dissertation is copyright of the author and/or third parties. The intellectual property rights of the author or third parties in respect of this work are as defined by The Copyright Designs and Patents Act 1988 or as modified by any successor legislation.

Any use made of information contained in this thesis/dissertation must be in accordance with that legislation and must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the permission of the copyright holder.



# Abstract

Recent research into automotive security has shown that once a single electronic vehicle component is compromised, it is possible to take control of the vehicle. These components, called Electronic Control Units, are embedded systems which manage a significant part of the functionality of a modern car. They communicate with each other via the in-vehicle network, known as the Controller Area Network, which is the most widely used automotive bus.

In this thesis, we introduce a series of novel proposals to improve the security of both the Controller Area Network bus and the Electronic Control Units.

The Controller Area Network suffers from a number of shortfalls, one of which is the lack of source authentication. We propose a protocol that mitigates this fundamental shortcoming in the Controller Area Network bus design, and protects against a number of high profile media attacks that have been published. We derive a set of desirable security and compatibility properties which an authentication protocol for the Controller Area Network bus should possess. We evaluate our protocol, along with other proposed protocols in the literature, with respect to the defined properties. Our systematic analysis of the protocols allows the automotive industry to make an informed choice regarding the adoption suitability of these solutions.



However, it is not only the communication of Electronic Control Units that needs to be secure, but the firmware running on them as well. The growing number of Electronic Control Units in a vehicle, together with their increasing complexity, prompts the need for automated tools to test their security. Part of the challenge in designing such a tool is the diversity of Electronic Control Unit architectures. To this end, this thesis presents a methodology for extracting the Control Flow Graph from the Electronic Control Unit firmware. The Control Flow Graph is a platform independent representation of the firmware control flow, allowing us to abstract from the underlying architecture. We present a fuzzer for Electronic Control Unit firmware fuzz-testing via Controller Area Network. The extracted Control Flow Graph is tagged with static data used in instructions which influence the control flow of the firmware. It is then used to create a set of input seeds for the fuzzer, and in altering the inputs during the fuzzing process. This approach represents a step towards an efficient fuzzing methodology for Electronic Control Units. To our knowledge, this is the first proposal that uses static analysis to guide the fuzzing of Electronic Control Units.

*This research was funded by Jaguar Land Rover and the Engineering and Physical Sciences Research Council (Industrial CASE Award 14220107) as part of the project Lightweight Cryptography for Next Generation Vehicle Electrical Architecture.*

# Acknowledgements

My research and this thesis would not have been possible without the support of my PhD supervisor, Flavio Garcia, who is an amazing mentor and role model. Thank you for all your guidance, and for all the coffee that kept me awake during these past five years.

I would like to thank my examiners, Casten Maple and David Oswald for their time reviewing this manuscript, and my thesis group members, Mark Ryan and Dave Parker, for their appraisal of my work, throughout my studies.

My time at the University of Birmingham has been enriched by numerous people I would like to give thanks, to name a few: Richard Thomas, Mihai Ordean, Tom Chothia, David Oswald, David Galindo, Edu Marin, Chris Hicks, Volker Sorge, Eike Ritter, George Vasilakis, Zitai Chen, Shehnila Zardari, Tom Goodman, Lauren Rawlins and Gurchetan Grewal, the latter who convinced me to embark on my PhD journey; and a special thanks to my office colleagues, who have always found time for exquisite office banter: Sam Thomas, Jan van den Herrewegen, Chris McMahon-Stone, Nabi Omidvar, Kit Murdock and Lauri Laaksonen.

I would like to thank my partner, Daniel Clark, for his love, patience, and confidence in me. Also, for his endless appetite for debate and analytical discussions, it has surely sharpened my critical self.

To my parents, Carmen and Teodor, you have provided me with all the love, support and strength I could have ever wished for. Thank you for always believing in me and encouraging me to forge my own path.

## Abbreviations and Acronyms

<b>ABS</b>	Anti-lock Braking System
<b>ADAS</b>	Advanced Driver-Assistance Systems
<b>ADC</b>	Analog-to-Digital Converter
<b>AEC</b>	Authentication Error Channel
<b>API</b>	Application Program Interface
<b>AUTOSAR</b>	AUTomotive Open System ARchitecture
<b>BAP</b>	Binary Analysis Platform
<b>BCM</b>	Body Control Module
<b>BIL</b>	Binary Independent Language
<b>CAN</b>	Controller Area Network
<b>CFG</b>	Control Flow Graph
<b>CFL</b>	Control Flow List
<b>CMAC</b>	Cipher-based Message Authentication Code
<b>CPU</b>	Central Processing Unit
<b>CRC</b>	Cyclic Redundancy Check
<b>CRYIF</b>	Crypto Interface
<b>CRYPTO</b>	Crypto Driver
<b>CSM</b>	Crypto Service Manager
<b>CSMA/CD+AMP</b>	Carrier Sense Multiple Access with Collision Detection and Arbitration Message Priority

<b>DLC</b>	Data Length Code
<b>DMR</b>	Direct Memory Reference
<b>DTC</b>	Data Trouble Code
<b>DoS</b>	Denial of Service
<b>EBS</b>	Electronically-Controlled Braking System
<b>ECM</b>	Engine Control Module
<b>ECU</b>	Electronic Control Unit
<b>FPGA</b>	Field-Programmable Gate Array
<b>FS</b>	Flow Status
<b>FV</b>	Freshness Value
<b>GMAC</b>	Galois Message Authentication Code
<b>GPIO</b>	General Purpose Input/Output
<b>GPS</b>	Global Positioning System
<b>JSON</b>	JavaScript Object Notation
<b>JTAG</b>	Joint Test Action Group
<b>HMAC</b>	Hash-based Message Authentication Code
<b>HSM</b>	Hardware Security Module
<b>HVAC</b>	Heating, Ventilation and Air Conditioning
<b>IDA</b>	Interactive DisAssembler
<b>IDS</b>	Intrusion Detection System
<b>IO</b>	Input/Output
<b>IPAS</b>	Intelligence Park Assist System
<b>IPC</b>	Instrument Panel Cluster
<b>IR</b>	Intermediate Representation
<b>ISN</b>	Interrupt Source Node
<b>LIN</b>	Local Interconnect Network
<b>LR</b>	Link Register
<b>MAC</b>	Message Authentication Code
<b>MCU</b>	Microcontroller Unit
<b>MIPS</b>	Microprocessor without Interlocked Pipelined Stages

<b>MOST</b>	Media Oriented Systems Transport
<b>MPH</b>	Miles Per Hour
<b>NRZ</b>	Non Return Zero
<b>NvM</b>	Non-volatile Random-Access Memory
<b>OBD-II</b>	On-Board Diagnostics
<b>OS</b>	Operating System
<b>PAM</b>	Parking Assist Module
<b>PCB</b>	Printed Circuit Board
<b>PCI</b>	Protocol Control Information
<b>PDF</b>	Portable Document Format
<b>PDU</b>	Protocol Data Unit
<b>PPC</b>	PowerPC
<b>PPT</b>	Probabilistic Polynomial Time
<b>PSCM</b>	Power Steering Control Module
<b>RAM</b>	Random Access Memory
<b>REC</b>	Receive Error Counter
<b>RFID</b>	Radio-Frequency Identification
<b>ROM</b>	Read Only Memory
<b>RPM</b>	Rotations Per Minute
<b>SAE</b>	Society of Automotive Engineers
<b>SecOC</b>	Secure Onboard Communication
<b>SID</b>	Service Identifier
<b>SN</b>	Sequence Number
<b>SPE</b>	Signal Processing Engine
<b>SPI</b>	Serial Peripheral Interface
<b>SWD</b>	Serial Wire Debug
<b>TEC</b>	Transmit Error Counter
<b>TEE</b>	Trusted Execution Environment
<b>TPMS</b>	Tyre Pressure Monitoring System
<b>UART</b>	Universal Asynchronous Receiver Transmitter

<b>UDS</b>	Unified Diagnostics Services
<b>V2X</b>	Vehicle-to-Everything
<b>VLE</b>	Variable Length Encoding
<b>WMA</b>	Windows Media Audio

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contributions and Thesis Overview . . . . .	4
<b>I</b>	<b>Preliminaries</b>	<b>9</b>
<b>2</b>	<b>Background</b>	<b>11</b>
2.1	Controller Area Network . . . . .	11
2.1.1	CAN Bus Protocol (ISO 11898) . . . . .	13
2.1.2	CAN-TP Protocol (ISO 15765-2) . . . . .	17
2.1.3	UDS (ISO 14229) . . . . .	19
2.2	AUTOSAR Classic Platform Standards . . . . .	21
2.2.1	Secure Onboard Communication . . . . .	22
2.2.2	Crypto Stack . . . . .	24
2.2.3	Key Manager . . . . .	25
2.2.4	Cryptographic Algorithms Recommendations . . . . .	26
2.3	Summary . . . . .	27
<b>3</b>	<b>Vehicle Security Threats</b>	<b>29</b>
3.1	From Hypothesising Security Risks... . . . .	30
3.2	To Assessing ECUs Under Local Access... . . . .	30
3.3	Then Gaining Access Remotely... . . . .	35
3.4	To the Ultimate Remote Pwn! . . . . .	38
3.5	Summary . . . . .	40





<b>II</b>	<b>CAN Bus Authentication</b>	<b>41</b>
<b>4</b>	<b>A Lightweight Authentication Protocol for CAN</b>	<b>43</b>
4.1	Motivation . . . . .	43
4.2	Contribution . . . . .	44
4.3	Building Blocks for Authentication Protocols . . . . .	45
4.3.1	Security Notions . . . . .	45
4.3.2	Message Authentication Codes . . . . .	47
4.3.3	Adversarial Model . . . . .	48
4.4	LEIA: A Lightweight Authentication Protocol for CAN . . . . .	49
4.4.1	Session Key Generation . . . . .	50
4.4.2	Sending Authenticated Messages . . . . .	51
4.4.3	Resynchronisation . . . . .	52
4.5	Dealing with the Shortcomings of CAN . . . . .	54
4.6	Performance Evaluation . . . . .	58
4.6.1	Implementation . . . . .	58
4.6.2	Latency analysis . . . . .	60
4.7	Security Analysis . . . . .	63
4.7.1	Security Proof . . . . .	63
4.7.2	Security and Safety Trade-off Discussion . . . . .	64
4.8	AUTOSAR compliance . . . . .	66
4.8.1	AUTOSAR 4.2 . . . . .	66
4.8.2	AUTOSAR 4.4 . . . . .	67
4.9	Summary . . . . .	68
<b>5</b>	<b>CAN Authentication Protocols: Review and Evaluation</b>	<b>71</b>
5.1	Motivation . . . . .	72
5.2	Contribution . . . . .	72
5.3	Desirable Properties for CAN Authentication Protocols . . . . .	73
5.3.1	Security properties . . . . .	73
5.3.2	Compatibility properties . . . . .	74
5.4	Overview and Eval. for OoB CAN Bus Auth. Protocols . . . . .	76
5.5	Overview and Eval. of Auth. Protocols over Traditional CAN Bus . . . .	79
5.6	Performance Evaluation . . . . .	86
5.7	Summary . . . . .	92



<b>III</b>	<b>Towards Large-Scale Fuzzing of Automotive Electronic Components</b>	<b>95</b>
<b>6</b>	<b>Extracting the Control Flow Graph from ECU Firmware</b>	<b>97</b>
6.1	Related Work . . . . .	97
6.1.1	Analysing (Embedded) Firmware . . . . .	101
6.1.2	ECU Firmware Analysis . . . . .	103
6.1.3	Tools and Frameworks . . . . .	105
6.2	Motivation and Challenges . . . . .	107
6.3	Contribution . . . . .	110
6.4	Targets . . . . .	110
6.5	Register Documentation . . . . .	111
6.6	Disassembling Electronic Control Unit Firmware . . . . .	114
6.6.1	Disassembling ARM . . . . .	115
6.6.2	Disassembling PowerPC . . . . .	118
6.6.3	Disassembling Infineon TriCore . . . . .	120
6.7	Control Flow Graph Extraction . . . . .	121
6.7.1	CFG Extraction from ARM Firmware . . . . .	121
6.7.2	CFG Extraction from PowerPC Firmware . . . . .	122
6.7.3	CFG Extraction from Infineon TriCore Firmware . . . . .	122
6.8	Summary . . . . .	124
<b>7</b>	<b>Fuzzing Electronic Control Units</b>	<b>125</b>
7.1	Related Work . . . . .	125
7.1.1	ECU Fuzzing . . . . .	127
7.1.2	Tools and Frameworks . . . . .	129
7.2	Motivation and Challenges . . . . .	130
7.3	Contribution . . . . .	131
7.4	Tools and Setup . . . . .	132
7.5	Data Extraction . . . . .	133
7.5.1	Control Flow Graph Tagging . . . . .	133
7.5.2	Forming Input Seeds for the Fuzzer . . . . .	134
7.6	Fuzzer Design . . . . .	137
7.6.1	Fuzzer Prerequisites . . . . .	137
7.6.2	Input Transformation . . . . .	138
7.6.3	Additional Features . . . . .	140

7.7	Crash Detection . . . . .	142
7.8	Evaluation and Results . . . . .	143
7.9	Summary . . . . .	145



## **IV Concluding Remarks 147**

### **8 Directions for Future Research 149**

8.1	Key Management for Secure CAN Communication . . . . .	149
8.2	The Future of Automotive Networks and Architectures . . . . .	150
8.3	Security in Autonomous Vehicles . . . . .	154
8.4	Automotive Fuzzing . . . . .	156

### **9 Conclusion 159**

## **Appendices 161**

### **A Antonyms 163**

## **List of References 165**

# CHAPTER 1

## INTRODUCTION

The automotive industry has recently undergone a significant digital transformation, which has given rise to serious security threats [59, 34, 93]. The increasing number of wireless interfaces available in today's cars exposes them to new attack vectors. Modern cars have dozens and sometimes even over a hundred Electronic Control Units (ECUs). Whilst more technology is being introduced in modern vehicles, transforming them into smart, connected cars, the underlying security infrastructure has struggled to keep up with the pace of these changes.

While vehicle owners think about the condition of their car, if it is fit to be driven, if it has enough oil or fuel, they are less likely to consider whether their vehicle is *secure*, from the point of view of cyber security. Even standards and ratings of vehicle have only just realised that automotive cyber security is crucial. Most countries or areas of the world have a New Car Assessment Programme (NCAP), which assesses the *safety* of new vehicles and provide ratings for occupants protection, pedestrian protection or driver assistance technologies. The automotive industry has mostly been focused on detection of states or actions that would impact driver safety or impair a vehicle's

ability to run as expected, for example notifications for service, clogged filters, skidding or uneven traction. Failure-recovery protocols have been implemented to guard against vehicles becoming unresponsive and uncontrollable in the case that a critical functionality stops working. Anti-theft solutions have also been of interest as a general means of preventing badly intended individuals from gaining access to a vehicle. Until recently, the implications of vehicles relying on electronic components which run software were not well understood. The earliest attempt at providing some guidelines for the automotive industry in relation to cyber security, is the SAE J3061 Guidebook [37]. The document describes a framework to allow automotive stakeholders to understand cyber security concepts, such as threats, vulnerabilities, attacker models, and consider them in the development cycle of a vehicle. A new ISO standard is currently under development, ISO/SAE 21434 [86], scheduled to be released in 2020. The standard is expected to outline guidelines for risk management based approaches to cyber security, focusing on software development and cyber security testing.

The Controller Area Network (CAN), standardised in [75], is the most commonly used automotive bus nowadays. Its purpose is to connect the ECUs of a car, and allow them to communicate without a source or destination address. As the in-vehicle network has traditionally been considered a trusted environment, and there were no wireless interfaces, resilience against cyber-attacks has not been of prime concern. The CAN bus is a broadcast network, whereby any message sent can be read by all connected ECUs. By design, it does not provide security features, such as authenticity (the source or destination of a message is unknown), or confidentiality (messages are not encrypted, therefore they can be eavesdropped) [158]. Most attacks presented in the literature could be prevented if authentication was present on the network, or at least their impact would be localised and mitigated.

Existing attacks rely on the fact that messages can be sent on the CAN network by a malicious attacker or a compromised ECU, and they are accepted by other ECUs as if

they were legitimate. The lack of source authentication is an enabler for all these types of attacks. While vehicles are designed to tolerate random failures, they cannot currently cope with malicious cyber-attacks. The lock-down of components is not a viable solution, both from a legislative point of view (e.g. right-to-repair legislation) or from the economic point of view of the manufacturer. When dealing with the security of the in-vehicle network, current research in the field is focused on two main solutions, one being introducing authentication protocols on the network (and we will delve further into this in Part II), or using Intrusion Detection Systems (IDSes). Maple *et al.* offer a review of Intrusion Detection Systems in [5], investigating 42 published works of such systems.

The use of complex electronic systems within cars has become more prevalent, and the technological trend is moving towards driverless, fully autonomous vehicles (levels 4 or 5, according to SAE J3016 [146]). These cars will communicate with each other and with road-side infrastructure. In light of these advancements, research into the state of security in current vehicles is essential. Findings from such research are often alarming, and have led to a number of recalls, such as the case of Fiat Chrysler which had to recall 1.4 million vehicles, after researchers showed they could remotely kill the engine of a car [59].

The security of common embedded devices we have surrounded ourselves with (e.g. routers, smartphones) has received notable attention from researchers. However, ECU security is an area which has yet to reach maturity. The large number of ECUs in a modern vehicle means the manufacturers do not have the resources to completely produce the components themselves. They rely on first tier suppliers, which are companies that work directly with manufacturers, and outsource a large part of their ECU production to them. In the case where the firmware is outsourced, the manufacturers do not receive the source code, just the firmware image. Therefore, they cannot easily verify the code is free of bugs or vulnerabilities. The supply chain for ECUs is complex,

and can go deeper than first tier suppliers, with tier 2 suppliers being companies that create parts which end up in a vehicle, but do not directly interface with manufacturers (e.g. chip manufacturers), and tier 3 suppliers being providers of “raw” materials (e.g. plastic). Creating a tool which manufacturers can use to test the ECUs, using a grey box and non-invasive approach, would allow them to gain some security guarantees about the firmware of the electronic components they outsource.

## 1.1 Contributions and Thesis Overview

Within this thesis we explore the security of the CAN bus, which allows ECUs to communicate with each other, and the security of the ECUs themselves. We present below the overview of the thesis, and the contributions this thesis brings to the field of automotive cyber security.

---

### Part I: Preliminaries

#### Chapter 2: Background

In this chapter we discuss the required background knowledge for understanding the internal vehicle network, its components and how they communicate with each other. We present the standards which specify the behaviour of the most used in-vehicle network, the CAN bus, and what approaches the industry is taking for standardisation of ECU software development.

#### Chapter 3: Vehicle Security Threats

In this chapter, we review offensive research into vehicle security. We aim to explore how a vehicle can be compromised by a cyber attacker, what are possible entry vectors and which are the components or protocols that are vulnerable. Later in the thesis we propose solutions for weaknesses which we highlight throughout this chapter.

---

## Part II: CAN Bus Authentication

### Chapter 4: Lightweight Authentication Protocol for CAN

As we will see from the Background and Related Work chapters, the lack of authentication for the messages sent by ECUs on the CAN bus appears throughout the literature as a weakness which enables attackers to take control of a vehicle. Therefore, we present **a new design for an authentication protocol for the CAN bus**. The protocol, LEIA, is a software-only solution, and does not require any additional hardware components, therefore not increasing the manufacturing cost of a vehicle. The protocol is designed to satisfy the stringent resource requirements of low/medium level Microcontroller Units (MCUs), the chips most often used in ECUs and is flexible in terms of frequency of message authentication. This allows manufacturers to decide how often messages should be authenticated, for each CAN id, based on their own security assessment. The protocol is also backwards compatible with the existing CAN configuration, requiring minimal changes at application layer. LEIA features a *resynchronisation* procedure to ensure counters, used to prevent replay attacks, can reach a new shared state, in case their values become desynchronised. It is also compatible with the AUTOSAR standards, discussed in Chapter 2, therefore being ready to be implemented by manufacturers. We use provable security techniques to reduce the security of LEIA to that of the Message Authentication Code algorithm chosen when implementing it.

The chapter is based on the following publication: *LEIA: A Lightweight Authentication Protocol for CAN*, by the author and Flavio D. Garcia [140], presented at *The European Symposium on Research in Computer Security (ESORICS'16)*.



## Chapter 5: CAN Authentication Protocols: Review and Evaluation

In this chapter, we present a **systematic analysis of CAN bus authentication protocols** proposed in the literature. We evaluate them with respect to a set of desirable security and compatibility properties we define. The properties are based on the specifications of the standards presented in Chapter 2 and the lessons learnt from understanding the vulnerabilities presented in Chapter 3, therefore incorporating the views of both academia and industry. We also discuss their performance, giving particular attention to the overhead and latency these protocols add. These play an important role, especially in the case of ECUs used for safety-critical functionality.

---

## Part III: Towards Large-Scale Fuzzing of Automotive Electronic Components

This part of the thesis is based on the following publication: *Grey-box Analysis and Fuzzing of Automotive Electronic Components via Control-Flow Graph Extraction*, by the author and Flavio D. Garcia [141], presented at *The 4th ACM Computer Science in Cars Symposium (CSCS '20)*.

## Chapter 6: Extracting the Control Flow Graph from ECU Firmware

The hardware that ECUs are built on is diverse, with multiple architectures being used, and the number of tools available for analysing ECU firmware is highly limited. In this chapter we discuss the reasons why we believe having an automated method for analysing ECU firmware is important and we present a **methodology for ECU firmware analysis**, which allows us to abstract the architecture-dependent firmware and represent it through its Control Flow Graph (CFG). The CFG represents the execution paths of the firmware, and can be enriched with other relevant information, such as static data used in flow-influencing instructions. The CFG can be used in developing other tools, which are not dependent on a specific architecture, therefore removing

one of the challenges of analysing ECU firmware. We also provide an overview of the scripts we developed in order to aid in decreasing the manual labour required for this task.

## Chapter 7: Fuzzing Electronic Control Units

This chapter presents the design of **an ECU fuzzer**, which communicates with the ECU via the CAN bus and uses the previously extracted CFG. We annotate it with information about static data comparisons that affect the control flow of the firmware and what branch instruction was used to determine the flow of the execution path. This information is used to create initial seeds for the fuzzer. It is also used to adapt the input messages in order to cover hard to reach execution paths. We debate **what a crash means, within the context of an ECU**, and how we can detect it. We **evaluate the fuzzer** on three ECUs, from different manufacturers. The tool was able to crash two of the ECUs. Our fuzzer is the first approach that uses static analysis when forming CAN messages.

---

## Part IV: Concluding Remarks

### Chapter 8: Directions for Future Research

We discuss possible future research directions within the area of vehicle security which pertain to the thesis subject: key management, future automotive networks, highlighting the upcoming move towards Ethernet, autonomous vehicles, in the context of securing sensors input, spoofing prevention and adversary-resilient machine learning, and automotive fuzzing.

### Chapter 9: Conclusion

We conclude the thesis by looking at how the research presented herein contributes to the improvement of cyber security within the automotive field, and discuss possible directions for future work.



---

# PART I

## *Preliminaries*

---



## CHAPTER 2

# BACKGROUND

This chapter provides the required background for understanding how the ECUs in a vehicle communicate with each other, elaborating on the existing standards for the CAN bus and what directions the automotive industry is taking with respect to software development for ECUs, through the AUTomotive Open System ARchitecture (AUTOSAR) standards.

### 2.1 Controller Area Network

The large number of ECUs in the vehicle means that point-to-point communication is not efficient and therefore the exchange of messages is done via a bus. The most popular and widely used automotive bus is CAN. Each ECU is equipped with a CAN controller which implements the CAN protocol, and a CAN transceiver, which translates the bus signals to the appropriate logical values.

The CAN is a serial bus designed for the automotive industry, with applications in other contexts as well, such as the marine industry [71], aviation [9, 10] or industrial

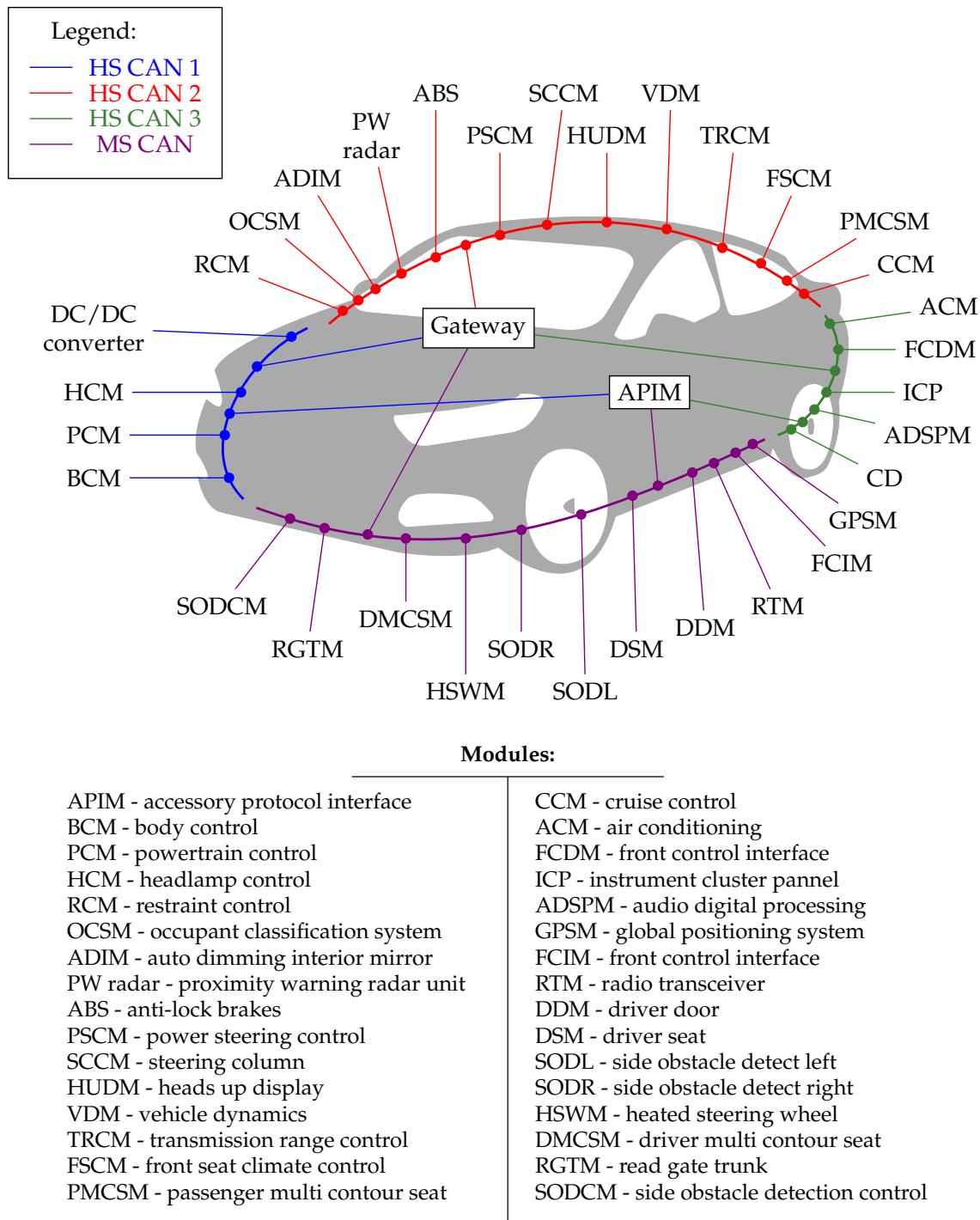


Figure 2.1: Example of CAN bus networks architecture from a Ford Fusion, 2014 (based on [114]).

control systems [35, 72]. Its data transmission rates are limited, but it can achieve up to 1Mbit/s over short distances (up to 40 meters). Typically, vehicles are equipped with a high speed CAN which connects safety-critical nodes which require real-time communication, and several low/medium speed CAN buses (Figure 2.1). A Gateway ECU acts as a bridge between all the CAN buses of the vehicle.

In the current context of increased digitalisation of car components, together with the trend of increasing the number of said components, the CAN bus falls short, by design, in providing any protection against a cyber attacker. Designed in the 1980s, its major shortfall relies on the assumption that the network is a trusted environment. At the time, inter-connectivity and external facing interfaces were not at the forefront of potential issues. The CAN bus aims to provide real-time, fault-tolerant communication between ECUs in a simple, cost-efficient manner. However, it suffers from the following issues [180, 69, 158]:

- No confidentiality: any node on the network can eavesdrop;
- No authenticity: the sources of a message is unknown, as it only contains the identifier (id) of the destination node;
- As any node can broadcast, it is easy to carry out a denial of service attack;
- No integrity: a fake message or an altered message cannot be detected;
- No non-repudiation: a network node cannot prove it has sent or received a certain message.

### 2.1.1 CAN Bus Protocol (ISO 11898)

The CAN bus was officially released in 1986, at the Society of Automotive Engineers (SAE) conference and is standardised in ISO 11898-1 [84], describing the data layer and physical layer in accordance with the ISO reference model ISO/IEC 7598-1 [73], and in SAE J2284 [68]. The current CAN versions are 2.0A and 2.0B; version 2.0A uses 11 bits to represent the data identifier, whereas version 2.0B uses 29 bits (and it is called an



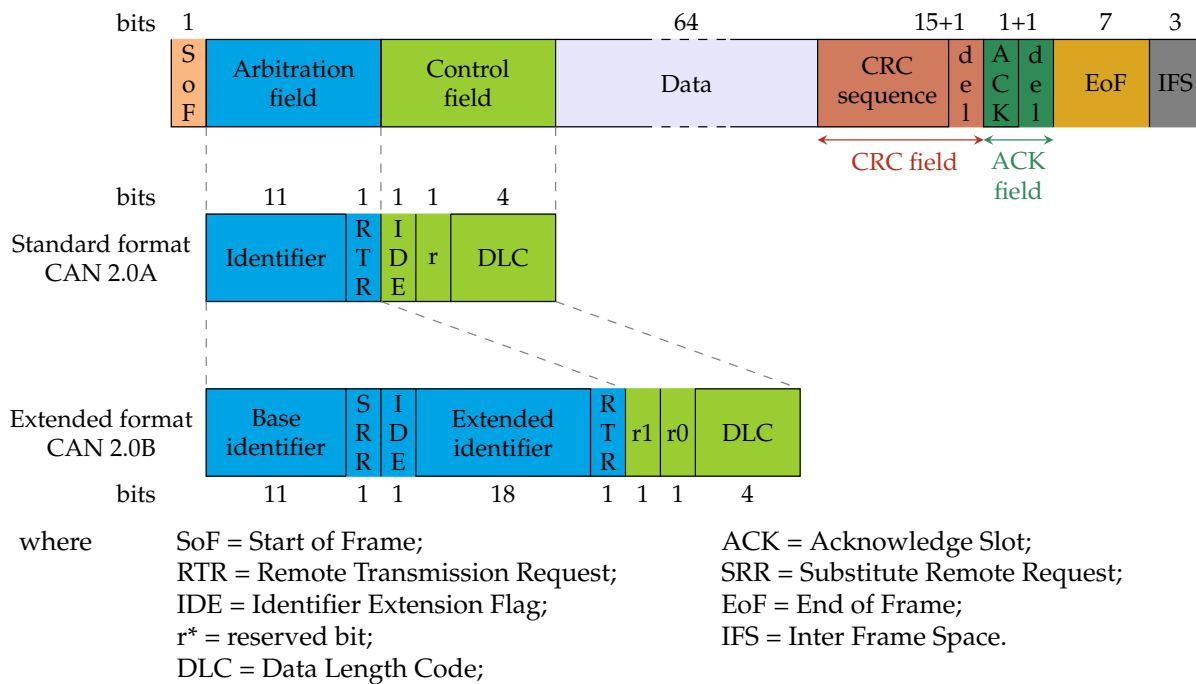


Figure 2.2: CAN data frame structure.

Extended Identifier). The two versions are compatible and can co-exist in the same network [96]. In a nutshell, the identifiers are used as an indication of the contents of a message transmitted, and nodes on the network decide whether to disregard or act upon the data of a message based on the identifier associated to it.

Information on the CAN bus is sent in *frames*, or Protocol Data Units (PDUs). The structure of a data frame is shown in Figure 2.2. Up to 8 bytes of data can be sent in one data frame, with the Data Length Code (DLC) field specifying the number of bytes of the message. As the DLC field has 4 bits, it can take any value up to 15, inclusive, but values greater than 8 are automatically considered to be 8. Error frames, remote frames and overload frames are also used for signalling faults, triggering data re-transmission and synchronisation, and they do not transport any message data. The bit stream of a frame is coded according to the Non Return Zero (NRZ) method. For synchronisation purposes, the Start of Frame (SoF), arbitration, control and data fields, and the Cyclic Redundancy Check (CRC) sequence are coded according to the bit stuffing method.

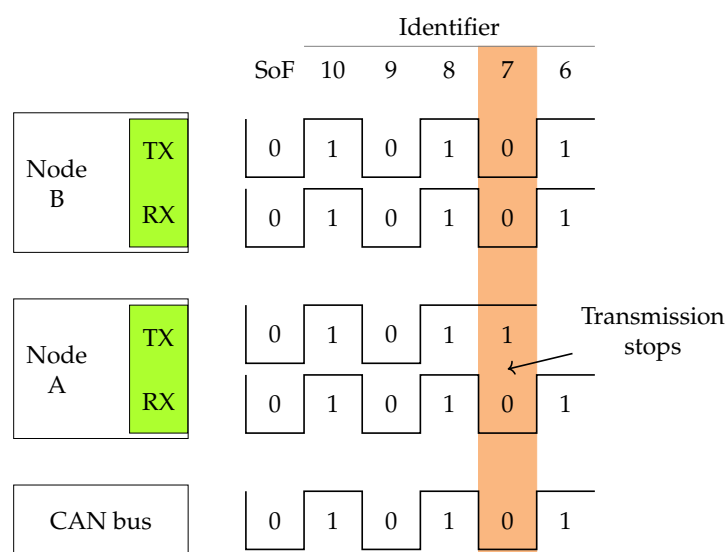


Figure 2.3: CAN bus arbitration between two nodes.

This means, for every five consecutive bits of the same value, a *stuff bit* of complementary value is transmitted on the bus (i.e. for a sequence of five 1s, a 0 is sent, and for a sequence of 0s, a 1 is sent). The End of Frame (EoF) field is not coded according to the bit stuffing method, and neither is the entirety of error frames and overload frames.

Nodes which want to send messages on the CAN bus at the same time have to negotiate which one will transmit first (Figure 2.3). The CAN bus uses the Carrier Sense Multiple Access with Collision Detection and Arbitration Message Priority (CSMA/CD+AMP) method. Recessive and dominant bits are represented by a logical one and a logical zero, respectively, with the dominant bit overwriting the recessive bit, if transmitted at the same time. Two nodes may start transmitting at the same time, with the first bit sent being the SoF. The value of the arbitration field is used in deciding which node gets the priority as follows: the two ECUs send the bits of their arbitration field and read back the logical value on the bus; if a recessive bit has been sent but a dominant bit has been read, the node considers this a *bit-error* and will stop transmitting. This means that messages which have an identifier with a lower numerical value will have priority.

Furthermore, frame correctness is acknowledged by the other nodes on the bus. The transmitting node will send a recessive bit as the ACK bit of the ACK field, and monitoring nodes will overwrite this with a dominant bit, if a syntactically correct frame has been observed so far. At least one other node needs to be online on the bus for the acknowledgement process to be successful.

Failures and errors on the bus are monitored by the CAN transceivers of all connected nodes and faulty units will be logically disconnected from the bus, such that they cannot transmit or receive any frames. A node can be in one of three states, with respect to its error status: error-active, error-passive or bus-off. Two error counters per node are used in order to determine the error level: *Receive Error Counter (REC)* and *Transmit Error Counter (TEC)*. The error handling is implemented at hardware level, in the CAN Integrated Circuit.

A node in the error-active state has both REC and TEC at most 127, this being the normal operation mode of a CAN node. If at least one of the counters is greater than 127, and TEC is less than 256, the node is in the error-passive state. An error-passive node can send and transmit messages normally, but it can only send a *passive error flag* (consisting of 6 recessive consecutive bits, without bit stuffing). This means the node cannot hinder the traffic generated by the other CAN nodes. An error-passive node can transition back to error-active once both counters are less than or equal to 127. Finally, a node goes into bus-off state if TEC is greater than 255, and cannot send or receive any frames or send dominant bits on the bus. The error handling mechanism is intrinsic to the CAN controller, and the counters are constantly updated by it, as communication takes place on the network.

The rules for the error counters are as follows:

1. If a receiving node detects an error (except for when transmitting an *active error flag* or an *overload flag*, each consisting of 6 dominant bits, without bit stuffing), the REC is incremented by 1;

2. If a receiver samples a dominant bit after sending an error flag, the REC is incremented by 8;
3. If a transmitter node sends an error flag, TEC is incremented by 8;
4. If a transmitter node samples a bit error when sending an active error flag or an overload flag, TEC is incremented by 8;
5. If a receiver detects a bit error when sending an active error flag or an overload flag, REC is incremented by 8;
6. If a node detects 14 consecutive dominant bits after an active error flag or an overload flag, or it detects 8 consecutive dominant bits after a passive error flag, it will increment REC by 8 if it is a receiver, or TEC by 8 if it is a transmitter; the counters will also be incremented as they correspond, by a value of 8, for each sequence of additional 8 consecutive dominant bits detected;
7. If a frame has been successfully transmitted, TEC is decremented by 1;
8. If a frame is successfully received, REC is decremented by 1 if it is in error-active, and it is set to a value between 119 and 127 if it is in error-passive.

The mechanism for error handling ensures that communication failures at the level of the CAN bus protocol are dealt with in an appropriate manner, but it does not account for communication which is maliciously sent, with the aim of breaking the protocol. Therefore, it is important to note that the fault confinement process can be abused by an active attacker.

### 2.1.2 CAN-TP Protocol (ISO 15765-2)

The transport protocol and network layer services for CAN are defined in ISO 15765-2 [85]. Due to the small payload of the CAN data frame, the standard sets out how a data payload longer than 8 bytes will be split across frames. The network layer communication protocol is intended for ECU to ECU communication, or between ECU and external test equipment. The network layer services support the Unified Diagnostics

Services (UDS) standard, as specified in [82] and various application-layer implementations, mainly aimed at diagnostic services (emission related on-board diagnostics [77], world-wide harmonised diagnostics [79], end of life activation of on-board pyrotechnic devices (e.g. airbag) [76]).

Under the transport layer protocol, messages with up to  $2^{32} - 1$  bytes of data can be transmitted. Messages with up to 7 bytes of data are sent unsegmented, and these frames are known as *single frames*. If a message requires segmentation, it will be split across multiple CAN data frames. The *first frame* will contain the total length of the data to be transmitted, in bytes, in the Protocol Control Information (PCI) field, and the first 6 bytes of the message. Subsequent *consecutive frames* will carry the rest of the data, split into chunks of 7 bytes. Sequence Numbers (SNs) are used to signal the order of the frames, for reassembly purposes on the receiving node. The first frame does not have an explicit SN, it is assumed to be 0. The first consecutive frame will have an SN with the value of 1, and each additional consecutive frame will increment the SN by 1. The SN can take values up to 15, after which it wraps up to 0 and carries on. Finally, *flow control frames* are used by the receiving node to send information on whether the sender node can go ahead with the data transmission (Flow Status (FS) parameter) and about its acceptable transmission parameters: block size (the maximum number of PDUs a node can transmit before being required to await authorisation to continue transmission) and minimum separation time (the minimum acceptable time delay between two transmitted PDUs). The FS parameter informs the sender node if it can continue sending the data, if it needs to wait, or if an overflow occurred, in which case the transmission is aborted. An overflow is signalled after the first frame of a transmission is received, if the length of the data to be transmitted exceeds the buffer size of the receiver node. Figure 2.4 shows all four types of CAN transport layer frames and their structure.

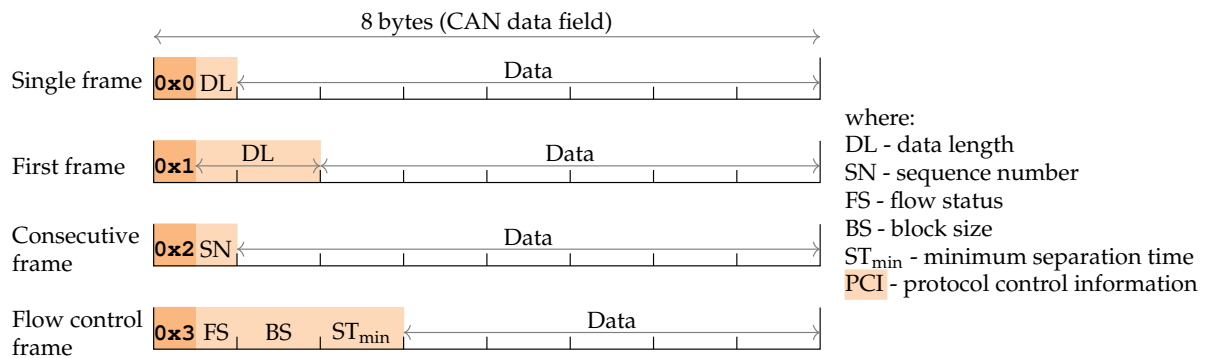


Figure 2.4: CAN transport layer frame types.

### 2.1.3 UDS (ISO 14229)

The Unified Diagnostics Services (UDS) are defined in [82] and specify the ‘*data link independent requirements of diagnostic services, which allow a diagnostic tester (client) to control diagnostic functions in an on-vehicle Electronic Control Unit (ECU, server)*’ (Part 1 [80]), the requirements of session layer services (Part 2 [81]) and the implementation of UDS over CAN (UDSonCAN, Part 3 [83]).

Communication occurs in a request-response manner. The request specifies the service required, by Service Identifier (SID), a sub-function (optional), parameters and data length. The response contains information whether the request was successful via a positive response message, or unsuccessful, by employing a range of negative response codes. The SID encodes the service type in the 6th bit. For requests, the bit is set to 0, and for positive responses, the bit is set to 1.

The UDS are split into six groups, based on the functional unit they belong to. An overview of each unit is given below, mentioning the services it contains.

### Diagnostic and Communication Management functional unit

SID	Name	SF <sup>a</sup>	DS <sup>b</sup>	Description
0x10	Diagnostic session control	✓	✓	By default, the control unit is in the <i>default session</i> . The <i>programming session</i> is used for firmware uploads. The <i>extended diagnostic session</i> is used for advanced diagnostic functionality (e.g. sensor calibration). Safety-critical diagnostic functions (e.g. airbag tests) require the <i>safety system diagnostic session</i> .
0x11	ECU reset	✓	✓	Power supply shutdown via <i>hard reset</i> , drain and ignition turn on via <i>key off on reset</i> and <i>soft reset</i> for program units and storage structures initialisation.
0x27	Security access	✓	✓	Enables access to security-critical services, via a challenge-response protocol.
0x28	Communication control	✓	✓	Switch off sending and/or receiving of messages.
0x3E	Tester Present	✓	✓	Keep current session active.
0x83	Access timing parameter	✓	✓	Set timeout for communication between client and server.
0x84	Secured data transmission	✓	✓	Transmission of data that is protected against attacks from third parties.
0x85	Control DTC setting	✓	✓	Toggle detection of any/all errors.
0x86	Response on event	✓	✓	Toggle response transmission on specific event.
0x87	Link control	✓	✓	Gain bus bandwidth for diagnostic session (change baudrate).

### Data Transmission functional unit

SID	Name	SF <sup>a</sup>	DS <sup>b</sup>	Description
0x22	Read data by identifier	✗	✓	Request data record values from the server, based on some data identifier (e.g. sensor reading).
0x23	Read memory by address	✗	✓	Request memory data based on starting address and data size.
0x24	Read scaling data by identifier	✗	✓	Read scaling data stored in the server using data identifier.
0x2A	Read data by periodic identifier	✗	✓	Request periodic transmission of data records.
0x2C	Dynamically define data identifier	✓	✓	Dynamically define a data identifier that can be read at a later time (by a source identifier or by a memory address).
0x2E	Write data by identifier	✓	✓	Write data at internal location defined by the data identifier.
0x3D	Write memory by address	✓	✓	Write data at a specified memory address.

<sup>a</sup> Sub-Function;

<sup>b</sup> Default Session

### Stored Data Transmission functional unit

SID	Name	SF <sup>a</sup>	DS <sup>b</sup>	Description
0x14	Clear diagnostic information	✗	✓	Clear Data Trouble Codes (DTCs).
0x19	Read DTC information	✓	✓	Request info on DTCs by status, time of occurrence, severity, function group, memory, etc.

**InputOutput Control functional unit**

SID	Name	SF <sup>a</sup>	DS <sup>b</sup>	Description
0x2F	Input output control by identifier	✓	✗	Gives control to a client over the internal/external signals (e.g. RPM, pedal position); used for simple input substitution/output control.

**Routine functional unit**

SID	Name	SF <sup>a</sup>	DS <sup>b</sup>	Description
0x31	Routine control	✓	✓	Execute a defined sequence of steps and obtain any relevant results (e.g. erasing memory, resetting or learning adaptive data, running a self-test).

**Upload Download functional unit**

SID	Name	SF <sup>a</sup>	DS <sup>b</sup>	Description
0x34	Request download	✗	✗	Download data onto the server, at the specified memory address.
0x35	Request upload	✗	✗	Upload data from the server to the client, from the specified memory address.
0x36	Transfer data	✗	✗	Transfer data between client and server (direction specified by one of the previous commands).
0x37	Request transfer exit	✗	✗	Request data transfer termination.

<sup>a</sup> Sub-Function;<sup>b</sup> Default Session

The CAN transport protocol, together with UDS, enables powerful functionality, for diagnostics purposes. However, this functionality can be taken advantage of by a knowledgeable adversary. It has been shown in [113] that attacks via diagnostics are possible, such as rendering the brakes not functional, killing the engine, or even re-flashing an ECU with modified firmware.

## 2.2 AUTOSAR Classic Platform Standards

The AUTomotive Open System ARchitecture (AUTOSAR), founded in 2003, is a world-wide partnership between the parties involved in the automotive industry (vehicle manufacturers, suppliers, electronics and semiconductors manufacturers, and software providers) [159]. The goal of AUTOSAR is to standardise basic functionality and interfaces, with the purpose of abstracting from hardware-specific implementations and



improving software/firmware development for ECUs. The standard has been gaining popularity among manufacturers and suppliers, with a number of companies creating software compatible with it ([128, 52, 51, 110, 111]), or tools to ensure software development respects the guidelines ([106]). AUTOSAR uses a layered software architecture, depicted in Figure 2.5.

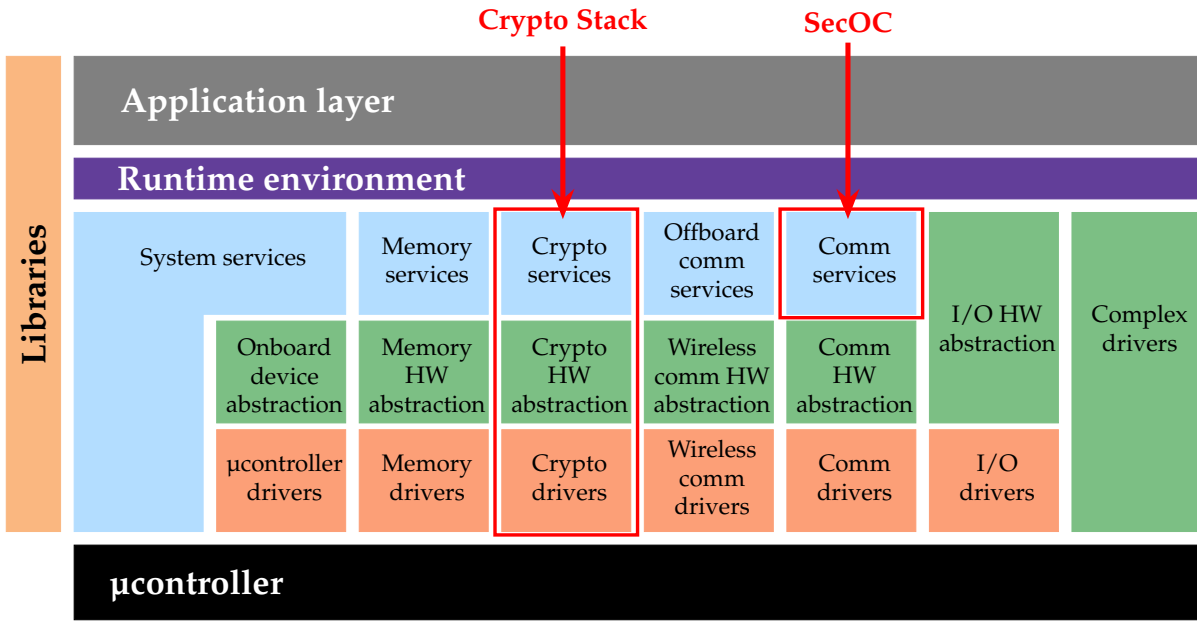


Figure 2.5: AUTOSAR architecture software layers overview.

Of interest to this thesis are the *Classic Platform* specifications [13], which are aimed at embedded real-time ECUs, within vehicles that do not heavily rely on automation or driverless feature. More precisely, the specifications for *Secure Onboard Communication* (SecOC), for the *Crypto Stack* and for the *Key Manager* are discussed below.

At the time of writing, AUTOSAR Classic Platform 4.4 is the latest standard (released October 2018).

### 2.2.1 Secure Onboard Communication

SecOC [18] is part of the Communication Services and its functional aims are to provide resource-efficient authentication and integrity for sensitive data, at PDU level,

ensuring protection against malicious manipulation of messages and replay attacks. It describes the protocols by using symmetric key cryptography, and provides details for the scenario where public key cryptography is used. SecOC sets out the software requirement specification for how CAN messages should be authenticated and how freshness, used in preventing replay attacks, should be handled, both for regular CAN frames, as well as CAN-TP messages.

With respect to authentication, a sender ECU and (all) receiver ECU need to have a shared secret (key) between them. An ECU which wants to send an authenticated message needs to compute an *Authenticator* (e.g. Message Authentication Code (MAC)). The Authenticator is generated based on a key, the CAN id of the message, the payload and an Freshness Value (FV). An ECU receiving an authenticated message needs to check the FV (which is the element that protects against replay attacks), and needs to verify the authentication information that belongs to the message. The Authenticator may be truncated to the most significant bits, however MAC sizes of at least 64 bits are recommended. Similarly, a recommendation for key size is provided, which is at least 128 bits. The standards also state that *authentication build* counters and *authentication verify attempt* counters need to be maintained, such that no ECU is stuck in trying to authenticate/verify one message.

According to the specification, freshness can be provided via counters or via timestamps. In the case of counters, there will be one counter corresponding to each CAN id to authenticate. The counter *must* be incremented prior to the message being verified, on the receiver side. With respect to timestamps, a global synchronised time source can be used, with the possibility of defining an acceptance window. A *Freshness Manager* deals with providing the FV required for computing the Authenticator.

If asymmetric key cryptography is preferred, instead of a shared secret, a key pair is to be used (a private key and a public key). The private key is used to sign the message by the sender, and the public key is used to verify the message, by the receiver.

Signatures cannot be truncated.

### 2.2.2 Crypto Stack

The Crypto Stack provides access to cryptographic functionality, either supplied by a software library or a hardware module, to the various applications, services and systems an ECU runs. The *Crypto Drivers (CRYPTO)*, *Crypto Interface (CRYIF)* and *Crypto Service Manager (CSM)* are the components of the Crypto Stack. Each component maps over one of the three layers of AUTOSAR (drivers, hardware abstraction, services), as described below.

The **Crypto Drivers** are part of the microcontroller abstraction layer, and hold the actual implementations of the available cryptographic primitives. They also deal with key storage, key configuration and key management. Keys may be stored in either cryptographic hardware, or in Non-volatile Random-Access Memory (NvM). A Crypto Driver has one or more Crypto Driver Objects. The Crypto Driver Object is '*the endpoint of a crypto channel*' [14]. An ECU may have multiple Crypto Drivers, in the case it uses implementations from different vendors (e.g. it could have a hardware module and a software library, both implementing the same primitives). The Crypto Drivers are called by the Crypto Interface, compute the operation on the requested primitive, and return the result to the interface.

The **Crypto Interface** is the intermediate layer between the Crypto Drivers and the CSM, representing the interface to the services of the Crypto Drivers [15]. CRYIF maintains a map from the cryptographic operations the CSM could request, to the Crypto Driver services that could supply said operation. The interface may manage multiple Crypto Drivers, providing seamless access to the underlying hardware or software solutions.

The **Crypto Service Manager** facilitates the concurrent access of software modules to cryptographic functionality, by interfacing with the Crypto Interface. The CSM represents the highest abstraction layer of the cryptographic services available on an ECU. The software modules requesting access to crypto operations are oblivious of their specific implementation, handled by the Crypto Driver. The CSM may provide a queuing mechanism, with job prioritisation, for software accessing the same cryptographic primitives. CSM provides services for hash computation, MAC generation/verification, signature generation/verification, symmetric and asymmetric encryption/decryption, random number generation, secure counters and key management [16]. Each service can be configured and initialised according to the required algorithms, as long as they are supported by the Crypto Driver. The module also provides a service for key derivation and a service for key exchange.

### 2.2.3 Key Manager

The Key Manager deals with both symmetric keys and certificates, and is part of the crypto services layer. It works together with CSM to allow software modules to access, update or create cryptographic key material [17]. In the case of authenticated communication, as described by the SecOC, the Key Manager is responsible for supplying the adequate keys to the CSM, which will request the appropriate crypto job (e.g. verify MAC). With respect to certificates, these are stored by the Key Manager, either as *root*, or as part of a hierarchical chain. Certificates can be verified on demand or at runtime, and the CSM can request full certificates or elements of a certificate, in order to perform crypto jobs. The manager provides a service for destroying all data used as cryptographic key material that is held in RAM, `KeyM_Deinit`.

Algorithm	(Key) Length	Comments
<b>Symmetric key</b>		
AES	128, 256 bits	modes: ECB, CBC, CTR, GCM, OFB, CFB, XTS
PRESENT	128 bits	modes: ECB, CBC, CTR, GCM, OFB, CFB, XTS
ChaCha23/ChaCha20	256 bits	
<b>Asymmetric key</b>		
RSA	1024, 2048, 3072, 4096 bits	padding: PKCS#1 v2.2
Curve25519/Ed25519	32 bytes	
<b>Hash</b>		
SHA-2	224, 256, 384, 512 bits	
SHA-3	224, 256, 384, 512 bits	
BLAKE	224, 256, 384, 512 bits	
RIPEMD-160	160 bits	

Table 2.1: AUTOSAR recommendations for symmetric and public key, and hash algorithms.

### 2.2.4 Cryptographic Algorithms Recommendations

AUTOSAR also specifies which cryptographic algorithms should be supported, as well as key length suggestions (where appropriate) for both symmetric and asymmetric key cryptography, and hash lengths. The recommendations are presented in Table 2.1. For MAC algorithms, they recommend using Cipher-based Message Authentication Code (CMAC), Galois Message Authentication Code (GMAC) or Hash-based Message Authentication Code (HMAC). A deterministic random number generator and a true random number generator should also be supported [14].

The ECB mode recommended by AUTOSAR is well known for its weakness: the lack of diffusion. Two encryptions of the same plaintext in ECB mode will result in identical ciphertexts. Therefore, an attacker would be able to determine the same message was sent twice. In the context of CAN communication, where some messages only take a limited amount of values, this would possibly allow an attacker to map a ciphertext to an action executed by an ECU, without needing to know the plaintext of

the message. Similarly, CBC and CTR are susceptible to bit flipping attacks, whereby changing the value of a bit in the ciphertext results in a different plaintext when decrypting it. This would allow an attacker to e.g. turn on the left indicator when in reality the right one was activated by the driver.

## 2.3 Summary

In this chapter we have reviewed the standards surrounding CAN bus communication, in order to understand how ECUs communicate with each other on the in-vehicle network; we have also reviewed key parts of the AUTOSAR industry standard, which are directly related to ECU communication.



## CHAPTER 3

# VEHICLE SECURITY THREATS

This chapter explores literature related to vehicle security, in particular looking at offensive research. Understanding *how* cars can be hacked aids in elaborating solutions which could prevent or limit the damage an attacker can do.

While transitioning from mostly mechanical systems to complex systems with digital components, manufacturers have overlooked the possibility of a cyber-attacker in their designs. We present works which have as focus the security of the in-vehicle CAN bus. We explore whether vulnerabilities in components connected to it can be leveraged into developing attacks which allow an adversary to (fully) control a vehicle. However, we would like to point out that the field of automotive security is extensive, and research which assesses the security of other components with high risk implications exists. For example, the KeeLoq block cipher, used by various car manufacturers in anti-theft mechanisms, was first attacked by Bogdanov in [27]. Later, this attack was improved in [40, 74, 89]. Verdult *et al.* proposed an attack against the Megamos Crypto [173] and Hitag2 [174] vehicle immobilisers. These attacks allow an adversary to start the vehicle without the car key.



### 3.1 From Hypothesising Security Risks...

One of the early works on the security of automotive bus systems belongs to Wolf, Weimerskirch, and Paar [179], in which they discuss the importance of securing the buses existent in a car (Local Interconnect Network (LIN), CAN, FlexRay, Media Oriented Systems Transport (MOST)), as well as the gateways which allow information to flow from one network to another. The authors remark that safety was the most crucial factor in automotive design. However, vehicles have powerful diagnostics functionality which is unprotected and, can therefore be easily abused, and the trend of introducing more connectivity, for convenience, opens up new attack vectors. They also draw attention to the fact that one compromised bus system can affect the whole in-vehicle communication network, due to the interconnection of all the buses. Denial of Service (DoS) is highlighted as a possible CAN attack, as well as the ability to disconnect every CAN controller on the network. The authors recommend controller authentication, communication encryption and gateway firewalls as solutions to achieve secure bus communication within vehicles.

Larson and Nilsson argue in [95] for the need to employ the defence-in-depth design style, focusing on prevention of unauthorised access with the vehicle and within it, detection of unauthorised access attempts, deflection systems, such as honeypots, active countermeasure protections, such as intrusion detection systems, and recovery abilities, in order to allow systems to return to a stable state after an attack.

### 3.2 To Assessing ECUs Under Local Access...

The work of Hoppe, Kiltz and Dittmann [69] is among the first in which the security of the CAN bus is experimentally tested. Their research analyses the security of the electric window lift, warning lights and airbag control system, all of which they suc-

cessfully attack. The attacks they showcase range from minor nuisances to a driver, to major distractions or life-threatening (e.g. if the airbag were to deploy suddenly, without warning, while the vehicle is being driven in normal conditions). They also identify the underlying problems which allowed them to carry out the attacks. The lack of authentication, integrity or confidentiality on the CAN bus are repeatedly highlighted as issues throughout the existing literature. The authors extend their work in [70] to show an attack on a gateway ECU as well, being able to break the network isolation which the gateway should enforce.

Koscher *et al.* [93] provide an extensive description of attack vectors under the assumption that an attacker *has access* to the in-vehicle network, either via *physical access* (e.g. through the On-Board Diagnostics (OBD-II) port or via malicious aftermarket components), or through the various wireless interfaces modern vehicles have.

The experiments were conducted in a laboratory setting, on a test bench and on the road, proving their viability and effects in a real-world scenario. The aim was to observe the communication on the CAN bus and identify valid and relevant packets for different ECUs. The authors also partially reverse-engineered the telematics unit to understand its functionality. This was an important step as the telematics unit acts as a bridge between the low and high CAN buses. Their research shows it is possible to control ECUs across a range of trust boundaries, e.g. low trust components such as radio, Instrument Panel Cluster (IPC), Heating, Ventilation and Air Conditioning (HVAC), Body Control Module (BCM) (which controls door locks, interior and exterior lights, horn, windows, wipers, ignition), as well as safety-critical functionality of the engine and brakes. They launched a generic DoS attack which disabled communication on the CAN bus and froze the instrument panel cluster. The attacks can also be combined, for example showing falsified speed in the dashboard requires both intercepting the actual packets and transmitting purposefully crafted speed update packets. The telematics unit was an important attack vector, especially due to its position on both buses.

By reprogramming the unit, an ECU from the low speed CAN could deliver packets on the high speed bus. This implies that by compromising any ECU, irrelevant of its placement, the attacker can gain complete control over the entire in-vehicle network, and therefore affect safety-critical components. Also, vulnerabilities in the telematics allowed malicious code to be uploaded to the ECU Random Access Memory (RAM). The code sent messages on the CAN network, then force the unit to reboot and the evidence of its existence is wiped.

Miller and Valasek [113] provide an extensive and thorough analysis of the ECUs in two vehicles: a Ford Escape and a Toyota Prius, both from 2010. They experimentally investigate each ECU the vehicles have, showing that each can be controlled, to some degree. Table 3.1 summarises their findings, with the mention that all the attacks were carried out over the CAN bus, and involved sniffing CAN traffic, reverse engineering CAN packet payloads and the ability to send CAN messages crafted by the researchers. Their attack entry vector was the OBD-II port for all but one tests.

As explained in Subsection 2.1.3, some Unified Diagnostics Services require an elevated level of access, achieved through challenge-response authentication, via the *SecurityAccess* service. While investigating this service for the Ford ECUs the authors noted that for the Parking Assist Module (PAM) module, the challenge nonce is always the same. Therefore, sniffing a genuine response once allows an attacker to replay it and successfully authenticate. The other ECUs had properly programmed challenge-response protocols. They then set out to reverse engineer the Ford Integrated Diagnostic Software tool, and noticed that all the keys, for all ECUs, can be recovered from it, as the tool has the capability of performing diagnostics on a vast range of modules. They also reverse engineered the code responsible for computing the response for the challenges. On the Toyota, the *SecurityAccess* service appeared to only be required if re-flashing an ECU. The challenge-response protocol generates a new seed every time the car is restarted, or after a number of invalid attempts, thus making brute forcing

Target	Car state	Outcome(s)
<b>FORD — Normal CAN packets</b>		
Driver door	Any	Dashboard indicates driver door ajar
Speedometer	Any	Speed and Rotations Per Minute (RPM) shown on combination meter can be set to any arbitrary value
Odometer	Any	Odometer can increase at a faster pace than distance actually travelled
Steering	Any	DoS causes Power Steering Control Module (PSCM) to shut-down, steering wheel is difficult to move; driver cannot take sharp turns
Steering	<5 MPH	Make steering wheel turn more/less (only small offsets work)
<b>Diagnostic CAN packets</b>		
Brakes	Stopped	Proprietary diagnostic services which allow for the brakes to be engaged
Brakes	<5 MPH	Ignore physical brakes input, disabling them
Engine	Any	Kill any/all cylinders, therefore killing the engine; even after the attack is stopped, the engine continues to be affected
<b>TOYOTA — Normal CAN packets</b>		
Speedometer	Any	Speed shown on combination meter can be set to any value
Brakes	Cruise Control	The vehicle is prevented from accelerating, eventually coming to a halt
Acceleration	Any	CAN injection tool must be directly connected to CAN bus going to Power Management ECU; gasoline internal combustion engine must be engaged, then disengaged; vehicle continues accelerating even when pedal not pressed
Steering	<4 MPH	Vehicle must believe it is in reverse gear; take control of the servo packet of the Intelligence Park Assist System (IPAS), therefore controlling the steering wheel
Steering	Any	Vehicle must believe it is in reverse gear & CAN must be flooded with fake speed messages; take control of the servo packet of the IPAS, therefore controlling the steering wheel
Steering	Any	Lane Keep Assist is misused to allow steering wheel movements of up to 5°.
<b>Diagnostic CAN packets</b>		
Brakes	Stationary in park	Control individual solenoids <sup>a</sup> in the Anti-lock Braking System (ABS) and Electronically-Controlled Braking System (EBS)
Engine	Stationary in park	Kill fuel to any/all cylinders, therefore killing the engine
Lights	Any	Switch must be in auto state; turn headlights on/off
Horn	Any	Turn horn on/off; horn on state persists for a while, even after car is turned off
Seat belt	Any	Pre-tighten driver/passenger seat belts
Doors	Any	Lock or unlock any/all doors; if doors are locked, they can still be opened from the inside
Fuel gauge	Any	Spoof fuel gauge to show any desired level, regardless of how much fuel the vehicle actually has

<sup>a</sup> A cylindrical coil of wire acting as a magnet when carrying electric current  
Miles Per Hour (MPH)

Table 3.1: Summary of Charlie Miller and Chris Valasek findings, Adventures in Automotive Networks and Control Units [113].

the key difficult. The Toyota service tool Toyota Calibration Update Wizard was reverse engineered. The code which computes the response was identified, which lead to the discovery of the keys for the Engine Control Module (ECM), Power Management System and ABS.

For the last part of their research, Miller and Valasek set out to modify the firmware, of the Ford PAM. The firmware was extracted through a debug interface available on the ECU Printed Circuit Board (PCB) and the CodeWarrior debugger from NXP<sup>1</sup>. The authors reverse engineered the code responsible for CAN communication, the CAN-TP communication and the UDS services handling. They also found the functions that implement the challenge-response protocol for the *SecurityAccess* service. They note that the challenge is initially computed correctly, but it is then overwritten with the value 0x11 0x22 0x33. Miller and Valasek speculate this was done for testing purposes, but the developers forgot to remove the overwrite before the firmware went into production. Therefore, the authentication is broken by using the same fixed challenge. By sniffing the communication between the Ford diagnostic tool and the ECU, the researchers were able to understand how to upload code to the PAM and have it executed. By using the *RequestDownload* service, the tool uploads binary blobs to the ECU, at a fixed address. The code they upload is prefixed with a 4 byte signature, followed by 4 offsets into the uploaded code. After the transfer is complete, the *RoutineControl* service is used to start a routine, which inspects the uploaded code. If the code starts with the magic signature, it will be executed from the first offset provided. Therefore, the authors created a proof of concept attack by using the previously gained knowledge about how the CAN communication is handled in the firmware, and wrote a snippet of code which sends a pre-established message, then reads a message from the bus, slightly modifies it, and sends it back.

---

<sup>1</sup>CodeWarrior Debugger Manual

([https://www.nxp.com/docs/en/reference-manual/Engine\\_PPCRM.pdf](https://www.nxp.com/docs/en/reference-manual/Engine_PPCRM.pdf))

### 3.3 Then Gaining Access Remotely...

The work of Koscher *et al.* raised a key issues: *how would an attacker get access to the vehicle*. The assumption that prior physical access to the car was criticised as an unrealistic scenario. Therefore, Checkoway *et al.* explore the external attack surface of vehicles in [34]. Besides the internal networks (e.g. CAN buses), modern vehicles have a variety of wireless interfaces: keyless entry, telematics, anti-theft systems, entertainment systems or vehicle to vehicle networks, all use wireless communication and add value and functionality to the overall system. However, they can also be used as access points by a malicious adversary in launching remote attacks. Checkoway *et al.* classify the wireless attacks under three categories, based on the entry points: indirect physical access, short-range wireless access or long-range wireless access.

*Indirect physical attacks* are achieved through the OBD-II port or the entertainment module. The OBD-II port provides direct access to the internal network of the car and, therefore, complete control can be acquired. The OBD-II is used by service providers and repair shops for routine maintenance and upgrades. By compromising the diagnostics tools (e.g. PassThru devices) or the laptops connected to these devices, an attacker can manipulate the data sent to the vehicle. Taking the attack scenario one step further, the authors compromise a PassThru device and install malicious software which, in turn, installs malware on the telematics unit. They then modify the malicious code to act like a worm, actively seeking other PassThru devices on the same Wi-Fi network, and infecting them. Vulnerabilities in the media player read file functions and in the Windows Media Audio (WMA) parser allowed the researchers to engineer a WMA audio file that would play on a PC but would send random CAN packets when used in the car. As discussed previously, any controlled ECU can affect the car in its entirety.

*Short-range wireless attacks* take advantage of networks operating over short distances, such as Bluetooth, remote keyless entry, Radio-Frequency Identification (RFID),

Wi-Fi or Tyre Pressure Monitoring System (TPMS). Bluetooth is the most common way of connecting the mobile phone with the vehicle and uses the car's telematics unit. Vulnerabilities in the feature integration code allowed the researchers to execute arbitrary code on the telematics unit. Compromising a phone which was previously paired with the car's Bluetooth is one way of exploiting this weakness. Without a paired device, the effort of the attacker is considerably greater, as they need to sniff the car's Bluetooth MAC (Media Access Control) address and brute-force the PIN needed for pairing.

The *long-range wireless attacks* were carried over the cellular network. Modern vehicles incorporate this feature for Internet-based features (via 3G data channel) or for safety purposes (via voice channel). This functionality is fulfilled by the telematics unit, which runs a program (Gateway) to deal with both voice and data cellular communication. An in-band tone-based signalling protocol<sup>2</sup> is used for switching between voice and data mode. By reverse engineering the AqLink software responsible for converting between analogue waveforms and digital bits, contained in the Gateway program, incompatibilities were found between the packet size and expected input size by the command protocol, thus allowing for a buffer overflow exploit. However, the attack is prevented by a caller authentication protocol which closes the connection within 12 seconds, if it does not receive a valid response (sending the buffer overflow payload required about 14 seconds). The caller authentication protocol was not securely implemented either. The challenge was produced by a random number generator which was initialised with the same seed each time the telematics unit rebooted, thus enabling a replay attack. Furthermore, for approximately 1 in 256 authentication attempts, incorrect responses, which have been carefully crafted, were accepted as valid. Putting all the vulnerabilities together, the authors were able to create an attack by which they repeatedly called the vehicle until they successfully authenticated, they changed the

---

<sup>2</sup>In-band signalling refers to control information sent over the same physical channel whereas tone-based signalling refers to a 'steady or pulsating periodic sound' used to indicate a specific condition should be entered or is happening [176]

timeout to 60 seconds, and closed the connection. They then re-called the vehicle, exploited the buffer overflow vulnerability, and forced the telematics unit to download and run additional malicious code from the Internet. The attack could also be delivered by encoding the exploit payload in an audio format, calling the car, and playing the audio over telephone.

Miller and Valasek also provide a survey of the remote automotive attack surfaces in [116]. They investigated 20 vehicles, from 11 manufacturers, released between 2006 and 2015. They were the first to provide a review of automotive network architectures. They note that a remote attack may have up to three stages, depending on the aims of the attacker. First, an ECU with wireless functionality needs to be compromised. This could be the target of the attack in some situations, e.g. the telematics unit is compromised, and the goal is audio exfiltration. However, if the aim is controlling some safety-critical ECU (which, typically, is connected to a different bus), the next step is taking control of the gateway, which is responsible for forwarding messages to the various networks connected in the car. Finally, the attacker needs to have a very good understanding of the CAN messages used by the vehicles and any safety features that might be built-in to the target ECU. Therefore, a large effort towards reverse engineering the CAN traffic is required; also, this will be manufacturer, or even vehicle specific, therefore it is not easily scalable. The researchers identify the following features/ECUs as possible remote attack vectors, in order of likelihood of success of code exploitation: passive anti-theft system, TPMS, remote keyless entry, Radio Data System, Bluetooth, telematics/cellular/Wi-Fi, Internet/apps. The last category, Internet and apps, is especially dangerous as it would open up vehicles to web browser exploits, malicious apps or internet service exploitation, all of which have well established compromise methodologies, which are also known to attackers.



## 3.4 To the Ultimate Remote Pwn!

Miller and Valasek set out to remotely compromise an unaltered vehicle in [115]. Drawing on their previous research, they choose a 2014 Jeep Cherokee as target. The car has a number of possible wireless entry points for an attacker, such as remote keyless entry, TPMS, Bluetooth, Radio Data System, cellular, Wi-Fi hotspot and Internet-based apps. After a preliminary investigation, the UConnect system is chosen as focus for the task. The system provides Internet, radio, cellular communication and apps, having all *infotainment* features in one module, which is conveniently connected to *both* vehicle CAN buses. Most of the functionality of the ECU is fulfilled by a Texas Instruments OMAP-DM3730 (32-bit ARM chip). However, the CAN communication is handled by a second chip, a 32-bit Renesas 850V . The two chips can communicate via an Serial Peripheral Interface (SPI) interface. UConnect runs QNX<sup>3</sup>, an Unix-like real-time Operating System (OS), and an image of the system could be downloaded from the Internet. Therefore, the researchers were able to obtain the filesystem, with all binary and configuration files.

As the vehicle has the ability to create a Wi-Fi hotspot (paid subscription service), the authors decided to examine it further. Wi-Fi security is a well established research area, and the methodologies for both attacking and securing such networks are well established and documented. The Wi-Fi hotspot had a 12 character lowercase and uppercase alphabetical preset password, which did not present obvious biases. The password generation algorithm was identified in the `WifiSvc` binary, and reverse engineered. The password was generated during the first ever boot of the ECU, and is a function of the epoch time. This could be abused by an attacker, who could generate a list of possible passwords, by taking into account the year the car was manufactured.

---

<sup>3</sup>QNX in automotive

(<https://blackberry.qnx.com/en/solutions/industries/automotive/index>)

However, the service responsible for setting the time had a condition that, if the time could not be retrieved, it was set to 1st of January 2013 GMT, 00:00:00. If the time could not be retrieved when the unit first booted, the time would be incorrect, therefore the search space for possible passwords is greatly reduced. After testing this hypothesis, it was found that the Wi-Fi password on the Jeep corresponded to the time 00:00:32 1st of January 2013. If this behaviour was prevalent for all other vehicles using this system, their passwords would be easily crackable. By performing port scanning on the Wi-Fi hotspot, nine ports were found to be open. One of the most interesting ones was an unauthenticated D-Bus<sup>4</sup> daemon (on port 6667). The service allows for a number of direct interactions with the head unit, such as adjusting radio volume or retrieving Global Positioning System (GPS) data. One other service, `NavTrailService`, provides an `execute` method which allows for shell command execution.

Cellular connectivity was provided by Sprint, and it seemed the IP address allocated to the vehicle on the internal Sprint network was limited to 21.0.0.0/8 or 25.0.0.0/8, and the D-Bus service was exposed. Furthermore, Sprint did not use device isolation on their network, therefore any Sprint device could communicate with any other Sprint device. Scanning the aforementioned address ranges for devices with port 6667 open, the authors were able to identify 2695 vehicles with an unauthenticated D-Bus service, which could be queried (Fiat Chrysler recalled 1.4 million vehicles due to this research [23]).

The final step was being able to send CAN messages, which meant reverse engineering the firmware of the V850 chip and finding a way of transmitting the data from the OMAP chip to the V850. They identified how the CAN read and write functionality works, and also investigated how communication over SPI is handled, as this was used as a channel to convey commands and messages between the two chips. The

---

<sup>4</sup>D-Bus is a software bus, used for inter-process communication, and for remote procedure calls; it allows multiple processes running on the same host to communicate with each other ([Wikipedia D-bus](https://en.wikipedia.org/wiki/D-Bus) <https://en.wikipedia.org/wiki/D-Bus>)

firmware was modified such that a specific command sent over SPI jumps to attacker code in the firmware, from where arbitrary CAN messages can be sent, and the vehicle can be controlled, following the same processes as explained in Subsection 3.2.

To recap, the authors could remotely address the vehicle, from any Sprint device, and they had a way of executing shell commands on it via the exposed D-Bus daemon. This access allowed them to re-flash the firmware on the V850 chip, introducing the ability to send messages on both CAN networks, from input taken from the SPI. On the OMAP chip, any preferred method of interacting with the SPI could have been used, in this case Lua<sup>5</sup> scripts were chosen.



#### How to Freak Out a Journalist 101

The authors demonstrated the attacks live, on the road, for Wired Magazine [59], showing they could completely control the vehicle over the Internet: the car's dashboard functions, steering, brakes, heating system, radio, windshield wipers, the car's digital display, engine and transmission.

## 3.5 Summary

In this chapter we have reviewed related research to automotive security, to firmware analysis and fuzzing. Understanding literature related to vehicle threats allows us to identify what enabled the attacks to be carried out, and the lack of authentication on the CAN bus appears throughout the works presented here.

---

<sup>5</sup>Lua scripting language (<https://www.lua.org/about.html>)

---

## PART II

### *CAN Bus Authentication*

---



## CHAPTER 4

# A LIGHTWEIGHT AUTHENTICATION PROTOCOL FOR CAN

This chapter describes LEIA, which was the first AUTOSAR compliant, lightweight authentication protocol for CAN in the literature, at the time it was published. The protocol respects the requirements laid out to become a standard in the automotive industry, as described in the Secure Onboard Communication Module Specification of the AUTOSAR standards.

### 4.1 Motivation

As seen in Chapter 3, most of the attacks presented in the literature have as basis the ability of an attacker to send arbitrary CAN messages, which the other ECUs on the network receive and act upon. This is enabled by the fact that there is no source authentication within the CAN messages and no way for a receiver ECU to *verify* the message it received does, indeed, come from the intended party. The protocol presented in this

chapter aims to limit and mitigate this fundamental flaw in the CAN bus protocol, by introducing source authentication.

## 4.2 Contribution

LEIA does not require additional hardware components or substantial implementation costs thus is less expensive than previously proposed solutions, while providing higher security levels. The protocol has been designed by taking into consideration real-world requirements and limitations of the CAN bus such as limited bandwidth, short data frames and publisher-subscriber broadcast architecture where newly arrived messages may overwrite older ones in the receiver's buffer [166].

Furthermore, LEIA is fully backwards compatible with existing CAN configuration, and is designed such that it can be flexibly implemented, providing different security vs bandwidth or computational overhead trade-offs.

LEIA has been implemented on a resource constrained MCU, the Atmel AVR AT-Mega328p (Arduino UNO) and has shown suitable performance, the latency introduced being of only 3ms, therefore respecting the industry standard requirement of less than 5ms for safety-critical functionality.

Finally, the protocol is proven to provide secure authentication under the unforgeability assumption of the MAC scheme under chosen plaintext attacks. Since the same MAC scheme is used also for key diversification, there is the additional requirement that the produced MAC values are indistinguishable from the output of the key generation function.

### 4.3 Building Blocks for Authentication Protocols

This section discusses the security definitions for an authentication protocol, as well as the security definitions for the underlying primitives. Our focus is on a symmetric key solution, therefore the notions for MAC are given.

#### 4.3.1 Security Notions

An authentication protocol is an interactive cryptographic protocol executed between a prover  $\mathcal{P}$  and a verifier  $\mathcal{V}$ . In an initial phase, both parties run a  $\text{setup}(\eta)$  function, where  $\eta$  is the security parameter, which produces a shared secret  $s$  and, potentially, public parameters  $ns$ . After an execution of the protocol,  $\mathcal{V}$  outputs the identity of the prover,  $\text{id}$ , and the message  $\text{data}$ . An authentication protocol has completeness error  $\alpha$  if for all secrets  $s$  generated by  $\text{setup}(\eta)$ , the honestly executed protocol rejects the identity and message with a probability at most  $\alpha$ .

The main focus of the authentication protocols in this thesis is to be secure against active attacks. These allow the adversary  $\mathcal{A}$  to interact with the honest prover a polynomial amount of times. Then,  $\mathcal{A}$  interacts with the verifier only, and wins if the verifier returns `accept`. The adversary interacts with  $\mathcal{V}$  only once. An authentication protocol is  $(t, Q, \eta)$ -secure against active adversaries if every Probabilistic Polynomial Time (PPT) adversary  $\mathcal{A}$ , running at most  $t$  times and making  $Q$  queries to the honest prover, has probability at most  $\varepsilon$  to win the above game.

Firstly, some notation needs to be introduced. Let  $\mathbb{F}_2 = \{0, 1\}$  be the field of two elements (or the set of Booleans).  $\mathbb{F}_2^l$  denotes a bitstring of length  $l$  and  $\mathbb{F}_2^*$  is a bitstring of arbitrary length.  $\parallel$  stands for the concatenation of two bitstrings.

**Execution environment.** Let  $n$  be the number of identifiers in the system, and  $\mathcal{I} = \{\text{id}_0, \dots, \text{id}_{n-1}\}$  be the set of all identifiers. Let  $\mathcal{P} = \{P_0, \dots, P_{n-1}\}$  be the set of all



protocol participants, where participant  $P_i$  knows the secret parameter  $s_i$  and public parameters  $ns$ .

**Definition 4.1 (Protocol setup).** Let the function  $\text{setup} : \eta \rightarrow (s, ns)$  be the initialisation procedure of the protocol parties, where  $\eta$  is the security parameter and  $(s, ns)$  is a tuple formed by the secret parameter  $s$  and the public parameters  $ns$ .

**Definition 4.2 (Authentication oracles).** Let  $\Pi = \{\pi(s_i) \mid s_i \in s\}$  be a set of oracles such that  $\pi(s_i)$  emulates party  $P_i$  of the authentication protocol.

**Definition 4.3 (Protocol output).** Let  $\text{output} : P \rightarrow \mathcal{I} \times \mathbb{F}_2^*$  be the protocol output function of a protocol participant  $P_i$  and outputs a tuple  $(\text{id}_j, \text{data})$  corresponding to the last successful protocol instance of  $P_i$ , where  $\text{id}_j \in \mathcal{I}$  is the identity of the sender and  $\text{data}$  is the message that was sent.

The security notions for symmetric key authentication protocols are introduced below. Most of it is standard, and most of the definitions proposed here are adapted from [170].

**Definition 4.4 (Matching conversations [170]).** A *matching conversation* is a successful execution of the authentication protocol, between two parties.

The authentication game  $\text{Auth}_\Pi(\eta, \mathcal{A})$  is presented and formally defined below. The public and secret parameters are generated by calling the  $\text{setup}(\eta)$  function. Then adversary  $\mathcal{A}$  interacts with the oracles  $\pi(s_i)$  which emulate the protocol participants which respond according to the protocol description. At some point the adversary  $\mathcal{A}$  terminates.  $\mathcal{A}$  wins if there is a party  $P_i$  which has accepted, and thus outputs,  $(\text{id}_j, \text{data})$  while  $P_i$  and  $P_j$  did not have any matching conversation.

By  $\text{Adv}_{\text{MAC}}^{\text{Auth}}(\eta, \mathcal{A})$  the advantage of the adversary  $\mathcal{A}$  in breaking the authentication protocol is expressed.

**Experiment  $\text{Auth}_\Pi(\eta, \mathcal{A})$** 

$$ns, s \leftarrow \text{setup}(\eta)$$

$$\mathcal{A}^{\Pi(ns, s)}(\eta, ns)$$

**winif**  $\exists i, j, data : \text{output}(P_i) = (id_j, data)$  is the output of a party  $P_i$  and, parties  $P_i$  and  $P_j$  did not have any *matching conversation*.

**Definition 4.5 (Authentication Protocol Security).** An authentication protocol is said to be *secure* if for all PPT adversaries  $\mathcal{A}$ , the probability that  $\mathcal{A}$  wins the game  $\text{Auth}_\Pi(\eta, \mathcal{A})$  is a negligible function of  $\eta$ :

$$\text{Adv}_{\text{MAC}}^{\text{Auth}}(\eta, \mathcal{A}) \leq \varepsilon(\eta)$$

### 4.3.2 Message Authentication Codes

A MAC is a set of three algorithms  $\{\text{KG}, \text{MAC}, \text{Verify}\}$ , with associated key space  $\mathcal{K}$ , message space  $\mathcal{M}$  and MAC space  $\Phi$ .

The standard security notion for a MAC is *unforgeability under a chosen message attack* (UF-CMA). The secret key  $K$  is generated by calling the key generation algorithm KG of the MAC. Then, adversary  $\mathcal{B}$  makes up to  $Q$  queries to the  $\text{MAC}(K, \cdot)$  and  $\text{Verify}(K, \cdot, \cdot)$  algorithms. At some point,  $\mathcal{B}$  terminates and outputs a tuple  $(\mathbf{m}, \phi)$ , where  $\mathbf{m} \in \mathcal{M}$  is a message and  $\phi \in \Phi$  is a MAC. Adversary  $\mathcal{B}$  wins if it did not query  $\text{MAC}(K, \mathbf{m})$  and  $\phi$  verifies for message  $\mathbf{m}$ , under the secret key  $K$ .

$\text{Adv}_{\text{MAC}}^{\text{uf-cma}}(\eta, \mathcal{B}, Q)$  denotes the advantage of the adversary  $\mathcal{B}$  in forging a messaged under a chosen message attack for MAC, on the security parameter  $\eta$ .

**Experiment UF-CMA<sub>MAC</sub>( $\eta, \mathcal{B}, Q$ )**

$$K \leftarrow \text{KG}(1^\eta)$$

Invoke  $\mathcal{B}^{\text{MAC}(K, \cdot), \text{Verify}(K, \cdot, \cdot)}$  which can make up to  $Q$  queries to  $\text{MAC}(K, \cdot)$  and  $\text{Verify}(K, \cdot, \cdot)$ .

$$(\mathbf{m}, \phi) \leftarrow \mathcal{B}^{\text{MAC}(K, \cdot), \text{Verify}(K, \cdot, \cdot)}$$

**winif**

1.  $\text{Verify}(K, \mathbf{m}, \phi) = \text{accept}$
2.  $\mathcal{B}$  did not already request  $\text{MAC}(K, \mathbf{m})$

**Definition 4.6 (UF-CMA Security).** A message authentication code algorithm MAC is  $(t, Q, \eta)$ -secure against UF-CMA adversaries if for any adversary  $\mathcal{B}$  running in time  $t$  the experiment above, the following is true:

$$\text{Adv}_{\text{MAC}}^{\text{uf-cma}}(\eta, \mathcal{B}, Q) \leq \varepsilon(\eta)$$

**Assumption 4.1 (MAC indistinguishability from random).** The output of the MAC algorithm is computationally *indistinguishable from random* and, the output of the key generation (KG) function the output of the MAC function have the same distribution.

### 4.3.3 Adversarial Model

The adversary considered is a Dolev-Yao adversary [50], who controls the network. In particular, they can passively monitor the network, read all data passing through the CAN and send messages with any id. They can also send error frames to destroy current data or remote frames. However, in practice, the CAN error handling limits the attacker's capabilities in this respect.

## 4.4 LEIA: A Lightweight Authentication Protocol for CAN

This section outlines the design of LEIA, with a detailed description of each function of the authentication protocol.

The CAN bus uses a publish-and-subscribe architecture model, where one ECU can broadcast a message with a certain identifier ( $id_i$ ). The identifier is not a way to identify the source or destination of a message. Therefore, the protocol provides unidirectional authentication, with a method of signalling if any of the subscribed ECUs have gone out of sync or authentication has failed.

Each protocol participant which needs to authenticate data, will need to store a tuple  $\langle id_i, K_{id_i}, e_{id_i}, K_{id_i}^e, c_{id_i} \rangle$  per relevant CAN identifier, where:

- the identifier  $id_i$  is a CAN ID;
- the key  $K_{id_i}$  is a 128-bit long term symmetric key that is used to derive the session key;
- the epoch  $e_{id_i}$  is a 56-bit counter; the value is incremented at every vehicle start-up or when the counter  $c_{id_i}$  overflows; participates in the generation of the session key;
- the session key  $K_{id_i}^e$  is a 128-bit key used for generating the MAC; re-generating the session key when the epoch  $e_{id_i}$  changes ensures that only a small amount of data is authenticated under the same key; also, if the session key becomes compromised, the attacker can compute valid MACs only until the epoch changes (limited time);
- the counter  $c_{id_i}$  is a 16-bit counter included in the MAC and is sent within the data frame containing the MAC, in order to provide freshness.

The long term keys and epochs are assumed to be stored in tamper-resistant mem-

ory. Updating the set of keys (e.g. if adding or replacing a node in the network) should require direct physical access to the involved nodes and, therefore, could only be done by an authorised repairs shop. How exactly this can be achieved is beyond the scope of this chapter, as it is only concerned with the design of a protocol that would authenticate the communication between ECUs.

We describe below the functions of the protocol, for a pair of nodes: sender  $S$ , which is the broadcaster of messages with the identifier  $\text{id}_i$ , and receiver  $R$ , which is the node subscribed to messages broadcast on the identifier  $\text{id}_i$ .

The authentication protocol LEIA has an associated key space  $\mathcal{K} \in \mathbb{F}_2^{128}$ , message space  $\mathcal{M} \in \mathbb{F}_2^*$  and MAC space  $\Phi \in \mathbb{F}_2^{64}$ .

**Protocol setup** The function  $\text{setup}: \eta \rightarrow \langle s, ns \rangle$  is the initialisation procedure of the ECUs, where  $\eta$  is the security parameter and  $\langle s, ns \rangle$  is a tuple formed by the secret parameter  $s$  and the public parameters  $ns$ . The secret parameter  $s = \langle K_{\text{id}_0}, \dots, K_{\text{id}_{n-1}} \rangle$  is computed by running the key generation algorithm  $\text{KG}(1^\eta)$  of the MAC for each identity  $\text{id}_i$ , with  $K_{\text{id}_i} \in \mathcal{K}$ . The public parameters are  $ns = \langle \langle c_{\text{id}_0}, e_{\text{id}_0} \rangle, \dots, \langle c_{\text{id}_{n-1}}, e_{\text{id}_{n-1}} \rangle \rangle$ , where  $c_{\text{id}_i} \in \mathbb{F}_2^{16}$  is the counter and  $e_{\text{id}_i} \in \mathbb{F}_2^{56}$  is the epoch. Both the counter and epoch are initialised to zero, for each identity  $\text{id}_i$ . The session key generation function is then called for each identity  $\text{id}_i$ , in order to generate the session key  $K_{\text{id}_i}^e$ .

#### 4.4.1 Session Key Generation

Let  $\text{session\_key\_gen}: \mathcal{K} \times \mathbb{F}_2^{56} \rightarrow \mathcal{K}$  be the session key generation function. This function (Figure 4.1) takes as input a long term symmetric key  $K_{\text{id}_i}$  and an epoch  $e_{\text{id}_i}$ , both associated with an identity  $\text{id}_i$ , and outputs the session key  $K_{\text{id}_i}^e$  computed as follows:

1. increment epoch:  $e_{\text{id}_i} \leftarrow e_{\text{id}_i} + 1$
2. apply MAC algorithm, having as parameters the long term symmetric key  $K_{\text{id}_i}$

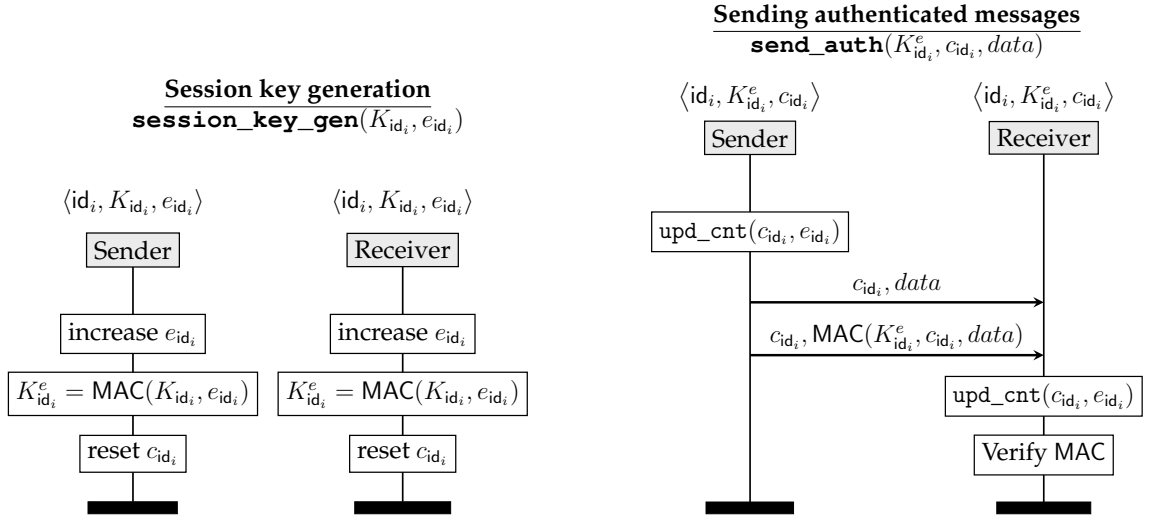


Figure 4.1: Session key generation between sender  $S$  and receiver  $R$  for message with identifier  $id_i$ .

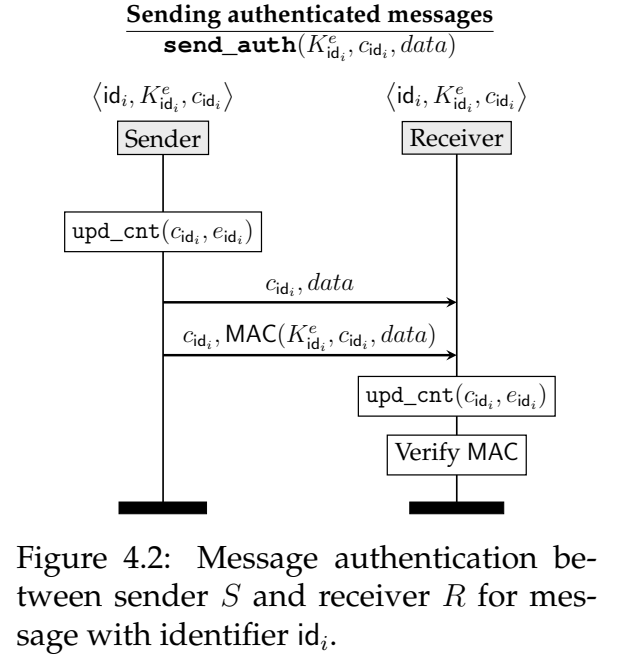


Figure 4.2: Message authentication between sender  $S$  and receiver  $R$  for message with identifier  $id_i$ .

and epoch  $e_{id_i}$ :

$$K_{id_i}^e \leftarrow \text{MAC}(K_{id_i}, e_{id_i})$$

3. reset counter to zero:  $c_{id_i} \leftarrow 0$

A session key is generated whenever the ECU is powered on or when the epoch counter  $e_{id_i}$  changes, due to the overflow of counter  $c_{id_i}$ .

#### 4.4.2 Sending Authenticated Messages

Let  $\text{send\_auth}: \mathcal{K} \times \mathbb{F}_2^{16} \times \mathcal{M} \rightarrow \mathcal{M} \times \Phi$  be the function for sending authenticated messages. In order to send an authenticated message (Figure 4.2), the sender first needs to update the counter  $c_{id_i}$ . If  $c_{id_i}$  overflows, then the epoch  $e_{id_i}$  is incremented and  $c_{id_i}$  is reset to 0 (Algorithm 4.1). It then calls the MAC algorithm which takes as input the session key  $K_{id_i}^e$ , the counter  $c_{id_i}$  and the message  $data$ , and produces as output a MAC  $\phi \in \Phi$  computed as:

$$\phi = \text{MAC}(K_{id_i}^e, c_{id_i}, data)$$

Line 3 in Algorithm 4.1 would never happen in practice, due to the length of the epoch counter. If we assume an attacker is able to reboot the ECU once per millisecond to forcefully increment the epoch counter, they would need 2 million years to roll it over.

The sender then transmits the counter, data and MAC. After reading the values, the receiver updates the counters and verifies the MAC.

```

Function upd_cnt ( $c_{id_i}, e_{id_i}, K_{id}$ ):
  Input:  $c_{id_i}, e_{id_i}, K_{id}$  * freshness counter, epoch and long term symmetric key *
  Result:  $c_{id_i}$  and  $e_{id_i}$  are incremented accordingly
1  if  $c_{id_i} = 0xFFFF$  then
2    if  $e_{id_i} = 0xFFFFFFFFFFFFFFFF$  then
3       $e_{id_i} \leftarrow 0x0000000000000000$  * both counters have maxed out; roll to 0 *
                                     * this would never happen in practice *
4    else
5       $e_{id_i} \leftarrow e_{id_i} + 1$  * increment the epoch *
6    end if
7     $c_{id_i} \leftarrow 0x0000$  * reset the counter *
8    session_key_gen ( $K_{id_i}, e_{id_i}$ ) * new epoch  $\Rightarrow$  regenerate the session key *
9  else
10    $c_{id_i} \leftarrow c_{id_i} + 1$  * increment the counter *
11 end if
end

```

**Algorithm 4.1:** Counter (and epoch) update algorithm.

### 4.4.3 Resynchronisation

Let  $\text{resync}: \mathcal{K} \times \mathbb{F}_2^{56} \times \mathbb{F}_2^{16} \times \mathcal{M} \rightarrow \mathbb{F}_2^{56} \times \Phi$  be the resynchronisation function. If a MAC cannot be verified, the receiver sends an AUTH\_FAIL signal to the sender. When an AUTH\_FAIL message is read, the sender  $S$  broadcasts a message containing its current epoch value, a MAC of the epoch and counter  $c_{id_i}$ , then proceeds with normal data transmission (Figure 4.3). This will help the receiver nodes resynchronise their epoch and counter.

The receiver  $R$  will update  $e_{id_i}$  and  $c_{id_i}$  only if the values are higher (or equal, for the

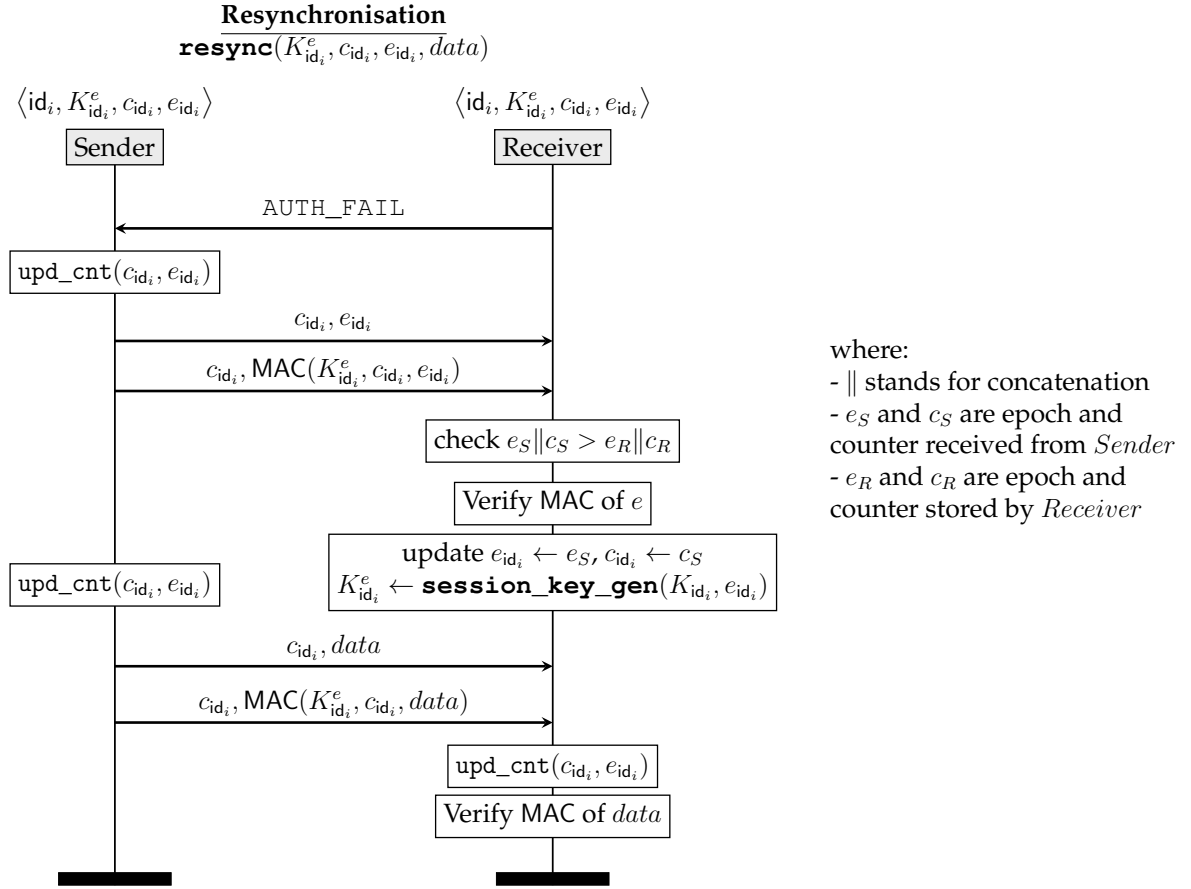


Figure 4.3: Message authentication failure and resynchronisation procedure, between sender  $S$  and receiver  $R$  for message with identifier  $id_i$ .

epoch) than the stored ones. If the new counter is lower than the receiver's counter, it means there is an attacker attempting a replay attack, therefore the data is discarded and the counter not incremented. The most common cause for a MAC to fail verification, in the context of the CAN bus communication, is the de-synchronisation of counter  $c_{id_i}$  and epoch  $e_{id_i}$  values. Not all nodes join the network at the same time, therefore the counters will be outdated and the receiver will need to request the current values from the sender.

A complete protocol outline is given in Figure 4.4.



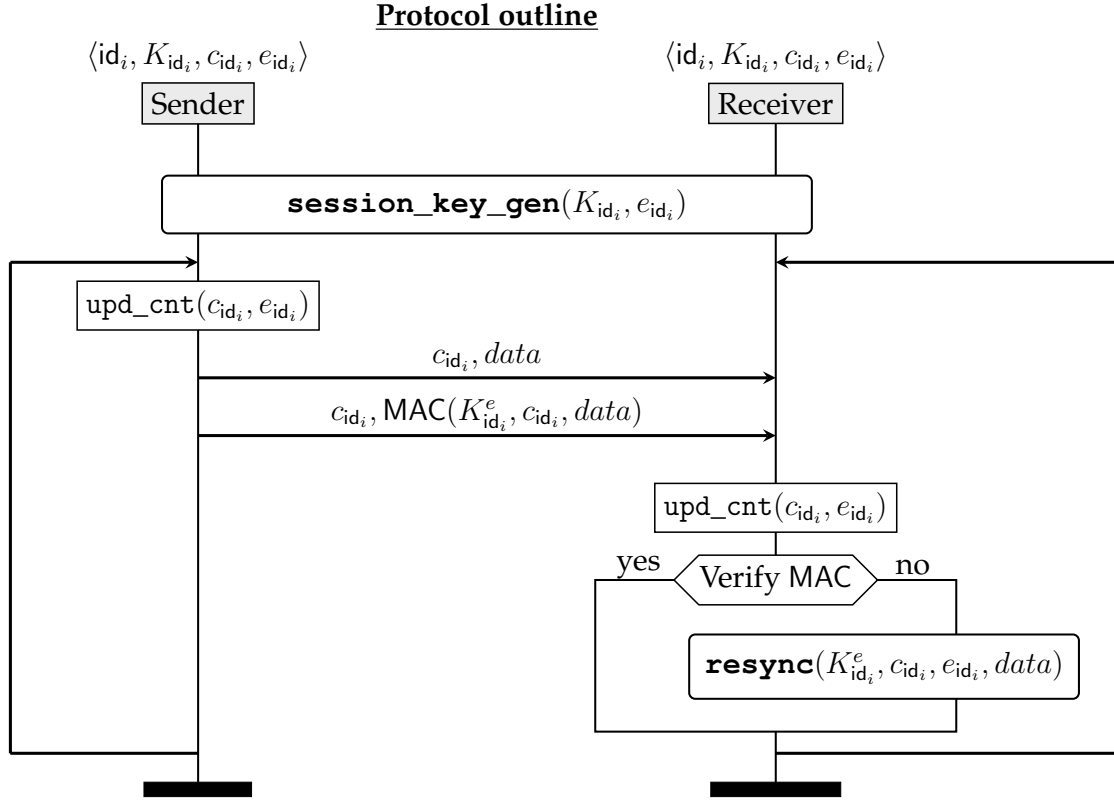


Figure 4.4: Communication between sender  $S$  and receiver  $R$  for message with identifier  $\text{id}_i$  – LEIA protocol outline.

## 4.5 Dealing with the Shortcomings of CAN

As some of the ECUs are involved in safety-critical functions such as acceleration and ABS, latency is of prime concern. Any solution aiming at providing extra security features, such as authentication, cannot introduce significant latency (more than  $5\text{ms}$  [184]). To this end, lightweight cryptography is best suited. Furthermore, many ECUs have limited memory available, therefore the implementation of the protocol should be compact as well. For this reason, the solution uses a MAC algorithm for two different purposes: authenticating data and deriving session keys.

In order to compensate for the modest security provided by lightweight cryptographic primitives, the long term secret key is not used directly. Instead, session keys are generated, which are used to authenticate the messages exchanged. A session key

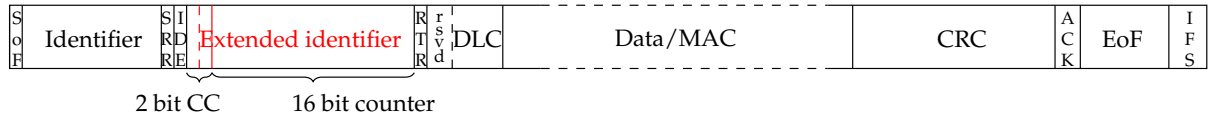


Figure 4.5: Extended Data Frame CAN 2.0B (29-bit identifier) – placement of command code (CC) and counter within Extended Identifier field.

is used to authenticate at most  $2^{16}$  messages, after which a new session key is derived, as the freshness counter  $c_{id}$  is 16 bits, the epoch counter  $e_{id}$  is incremented whenever the freshness counter overflows, and a new epoch counter triggers a new session key. This limits the amount of key-dependent data an attacker has access to. In case a session key is compromised, an attacker can use it either until  $2^{16}$  messages have been authenticated, or until the vehicle is restarted, whichever comes first.

LEIA makes use of the extended identifier data frames. It uses the Extended Identifier 18-bit field in order to send the 16-bit counter and a 2-bit command code, as explained below (Figure 4.5). The 29-bit identifier data frames co-exist with the 11-bit data frames without interfering with the arbitration process of CAN, as the priority of a message is decided based on the 11-bit Identifier field.

The following transmission channels over CAN are defined:

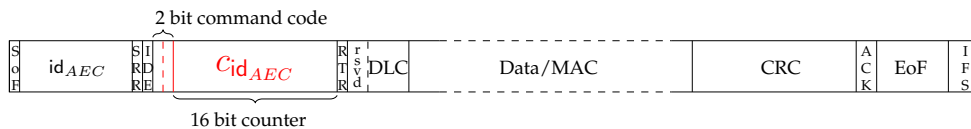
**Data Channel** All ids which are used to transmit data and signals constitute the data channel. The data is transmitted within the payload field of the frame. The counter  $c_{id_i}$  which is used to generate the MAC is placed in the extended identifier field. The two leftmost bits are the *command code* 00, and signal that data is being transmitted in the frame.

**Authentication Channel** All ids which are used to transmit MACs make up the Authentication Channel. The MACs are transmitted on a different identifier than the data. This id should be a fixed offset from the base id on which the data is sent. It should be as close as possible to the base id, in order to avoid scheduling issues caused by arbi-

tration. In the example,  $id_{MAC} = id_{data} + 1$ . This will avoid messages with the same identifier being overwritten in the CAN controller *buffer*. The counter is placed in the extended identifier field. The two leftmost bits, which represent the *command code*, are defined as follows:

- 00:** the data frame contains application data;
- 01:** the data frame contains a MAC of data;
- 10:** the data frame contains an epoch value  $e_{id_i}$ ;
- 11:** the data frame contains a MAC of an epoch  $e_{id_i}$ .

**Authentication Error Channel (AEC)** Each node connected to CAN has an Authentication Error Channel (AEC). This is used for resynchronisation purposes. The AUTH\_FAIL signal is sent on the ECU. Nodes which are broadcasters of messages with  $id_i$  become subscribers of the ECU of the nodes listening to  $id_i$ . The AUTH\_FAIL signal is defined as a set of two messages. The first data frame contains the id of the message which failed MAC verification ( $id_{failed}$ ), concatenated with the lower 53 bits of the ECU epoch counter ( $lsb_{53}(e_{id_{AEC}})$ ). Sending the epoch within the data frame ensures the receiving nodes can verify they have the correct values, and a resynchronisation procedure for the ECU is not needed. The second message contains the MAC of the previous one, as shown in Figure 4.6. Sending an AUTH\_FAIL signal is considered a rare event, therefore overwriting messages within the buffer are not of concern, in contrast to data transmission. Thus, the same identifier ( $id_{AEC}$ ) can be used for both message types.



where:

$$\begin{aligned} \text{data} &= id_{failed} || lsb_{53}(e_{id_{AEC}}) \\ \text{MAC} &= MAC(K_{id_{AEC}}^e, Cid_{AEC}, id_{failed}, lsb_{53}(e_{id_{AEC}})) \end{aligned}$$

Figure 4.6: Data frame structure for AUTH\_FAIL signal.

<i>Identifier</i>	Node A	Node B	Node C	Node D		<i>Identifier</i>	Node A	Node B	Node C	Node D
0x004	S		R		→	0x004	S		R	
0x010		R	R	S		0x005	S		R	
0x015		S		R		0x010		R	R	S
						0x011		R	R	S
						0x015		S		R
						0x016		S		R
						0x7FD		S		R
						0x7FE	R		S	R
						0x7FF		R		S

Table 4.1: Extended communication matrix example. ‘S’ stands for Sender and ‘R’ for Receiver.

Table 4.1 shows a small example of an extended communication matrix. The *identifiers* highlighted are the additional identifiers introduced by LEIA. Identifiers 0x005, 0x011 and 0x016 correspond to the *Authentication Channel*, while identifiers 0x7FD, 0x7FE and 0x7FF correspond to the *Authentication Error Channel*.

In the case an attacker fully compromises and takes control of an ECU, for the ids the node broadcasts or listens on, the attacker will unavoidably be able to generate valid MACs, but not for any other id. This is not a problem of the protocol but an inherent limitation of using symmetric key cryptography. An attacker can collect some AUTH\_FAIL answers from the sender, knowing one of the receiver nodes is offline. When the receiver node joins the network and sends the AUTH\_FAIL signal, as it does not have the correct counter and epoch values, the attacker sends a stored answer. The receiver will accept the message, provided the stored counter and epoch are lower than the received ones. However, due to the design of CAN, the initial AUTH\_FAIL signal is also received by the sender node, which will send the correct epoch and counter values. The attacker can destroy these frames, but *S* will broadcast them again, due to the error handling mechanism of CAN. After a number of destroyed frames, the CAN flags the attacker as error passive, meaning it cannot destroy other frames. Therefore, the correct message of *S* will be transmitted and the receiver node will be able to update its values accordingly. Communication then resumes under the protocol.

## 4.6 Performance Evaluation

The following section presents the implementation details for our protocol, as well as a worst case latency analysis.

### 4.6.1 Implementation

LEIA has been implemented on Arduino UNO boards<sup>1</sup>, with Atmel AVR ATmega328p microcontrollers, clock frequency 16 MHz, 32 Kb flash and 2 Kb RAM, fitted with MCP2515 CAN controllers. The boards were chosen for their constrained resources, in terms of memory and computation power. The ECUs in today's vehicles have more powerful MCUs, therefore by choosing the Arduinos, an estimate of performance in a worst-case scenario can be provided. For example, the MCU of a BCM we will research in later chapters has a clock frequency of 64 MHz, 518 Kb flash and 48 Kb RAM. Keccak sponge 200 with a capacity of 64 was chosen as the underlying MAC algorithm. The protocol was tested on a network consisting of four nodes, each sending its own messages, as detailed in Figure 4.7. The boards were connected to a computer and report via serial when they send a message messages or when they successfully verify a message. The computer keeps track of this and calculates the latency of the protocol. *Node D* reads all CAN messages and forwards them to the computer; the node's purpose was mainly to verify that the other nodes behave as expected. The bus speed was set at 500Kb/s, which is the most common bus speed used in vehicles.

In the benchmark implementation, every LEIA authenticated CAN message has a MAC sent alongside it. However, the protocol allows for a trade-off between computational resources and bandwidth usage versus security. The configuration of the network, and the decision of which CAN ids to authenticate and how often should be taken by a manufacturer after assessing the security requirements of each ECU and

---

<sup>1</sup>Arduino UNO Wikipedia ([https://en.wikipedia.org/wiki/Arduino\\_Uno](https://en.wikipedia.org/wiki/Arduino_Uno))

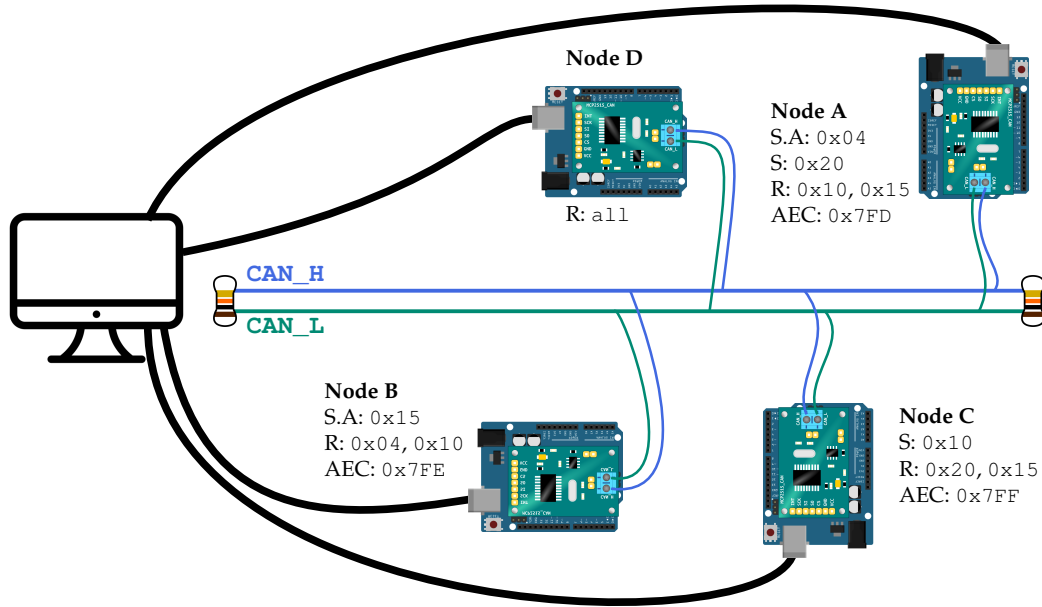


Figure 4.7: Test network architecture for LEIA implementation benchmarking; S.A ids are sent authenticated.

whether it fulfils a safety critical role. The CAN bus has a static configuration. Due to this, LEIA can be implemented in two ways, depending on the functionality of the ECU. As described above, each message sent is authenticated. If applied to all ECUs, this doubles the communication overhead. However, for nodes not involved in safety-critical functions, the protocol can be implemented such that one MAC is sent after  $n$  messages, where  $n$  can be chosen based on the node's security requirements. For CAN-TP messages split across multiple frames, the data payload can be authenticated before being split into multiple frames, and therefore only one MAC needs to be sent.

With respect to performance, the MAC computation takes on average 738  $\mu$ s for every message sent, and the time between receiving a frame and successfully verifying it is 2.96 ms. At a speed of 100 km/h, the induced latency means a travelling distance of 8.2 cm.

### 4.6.2 Latency analysis

We perform a worst case latency analysis for our protocol, based on the works of Tindell [164, 166, 165]. Tindell defines the *worst case response time* ( $R_m$ ) of a given message  $m$  as the maximum time between a scheduler queuing message  $m$  in the transmit buffer of the ECU, and the latest time the message arrives at a receiver ECU [166]. Mathematically, the worst case response time is defined as:

$$R_m = J_m + w_m + C_m$$

where:

- $J_m$  is the queuing jitter of message  $m$  (latest time the message can be queued),
- $w_m$  represents the worst case queueing delay of message  $m$ , given by messages with a higher priority taking transmission precedence and lower priority messages that might already be in transmission on the bus,
- $C_m$  is the longest possible time that sending a message on the bus could take (accounting for extra stuff bits).

The longest time to send a message can be calculated as:

$$C_m = \left( \left\lfloor \frac{34 + 8 * s_m}{5} \right\rfloor + 47 + 8 * s_m \right) * \tau_{bit}$$

where:

- $s_m$  is the size of the message  $m$  (in bytes),
- $\tau_{bit}$  is the time required to send one bit on the bus.

The queuing delay of a message  $m$  can be calculated as:

$$w_m = B_m + \sum_{\forall j \in hp(m)} \left\lceil \frac{w_m + J_j + \tau_{bit}}{T_j} \right\rceil * C_j$$

where:

- $B_m$  is the maximum time message  $m$  can be delayed by lower priority messages,
- $hp(m)$  is the set of messages with higher priority than  $m$ ,
- $T_j$  and  $J_j$  are the period and jitter respectively of a given message  $j$ .

The maximum delay time by lower priority messages can be calculated from the set of lower priority messages  $lp(m)$  as:

$$B_m = \max_{\forall k \in lp(m)} (C_k)$$

Based on the SAE dataset [145] used in [164], we calculate  $R_m$  for the scenario in which each transmitted message is authenticated. We use a CAN bus speed of 500Kb/s, as this is the most common found in vehicles currently. This gives us a bit transmission time of  $\tau_{bit} = 0.002ms$ . We perform the analysis on the aggregated messages system proposed by Tindell, in which some short, sporadic signals are grouped together at the sender and sent as part of one periodic message.

Table 4.2 presents the results of our analysis. Messages with the label ‘A’ carry the MAC on a different CAN id and are all 8 bytes in size. The fourth column, ‘Max latency’ is the maximum acceptable latency for the given message. The last column represents the total worst case response time. It is the sum of the  $R$  for the message containing the data, the worst case response time  $R$  for the message containing the MAC and the time required to compute a MAC (738  $\mu s$ ), as the receiver ECU would need to verify it before being able to consume the data. The table shows there is no worst case response time for any authenticated message transmission that exceeds the maximum latency acceptable for that message. We can therefore conclude that LEIA would be suitable for use in a vehicle, as it does not cause any delays that might negatively impact safety.



Signal	Size (bytes)	J (ms)	T (ms)	Max latency (ms)	R (ms)	R Total (ms)
14	1	0.1	50	5	0.386	
14A	8	0.1	50	5	0.646	1.770
8, 9	2	0.1	5	5	0.792	
8, 9 A	8	0.1	6	5	1.052	2.582
7	1	0.1	5	5	1.178	
7A	8	0.1	5	5	1.438	3.354
43, 49	2	0.1	5	5	1.584	
43, 49A	8	0.1	5	5	1.844	4.166
11	1	0.1	5	5	1.970	
11A	8	0.1	5	5	2.230	4.938
32, 41	2	0.1	5	5	2.376	
32, 41A	8	0.1	5	5	2.636	5.750
31, 34, 35, 37, 38, 39, 40, 44, 46, 48, 53	6	0.2	10	10	2.858	
31, 34, 35, 37, 38, 39, 40, 44, 46, 48, 53A	8	0.2	10	10	3.118	6.714
23, 24, 25, 28	1	0.2	10	10	3.244	
23, 24, 25, 28A	8	0.2	10	10	3.504	7.486
15, 16, 17, 19, 20, 22, 26, 27	2	0.2	10	10	3.650	
15, 16, 17, 19, 20, 22, 26, 27A	8	0.2	10	10	3.910	8.298
41, 43, 45, 47, 49, 50, 51, 52	3	0.2	10	10	4.074	
41, 43, 45, 47, 49, 50, 51, 52A	8	0.2	10	10	4.334	9.146
18	1	0.2	50	20	4.460	
18A	8	0.2	50	20	4.720	9.918
1, 2, 4, 6	4	0.3	100	100	4.904	
1, 2, 4, 6A	8	0.3	100	100	5.164	10.806
12	1	0.3	100	100	5.290	
12A	8	0.3	100	100	5.550	11.578
10	1	0.2	100	100	5.676	
10A	8	0.2	100	100	5.936	12.350
3, 5, 13	3	0.4	1000	1000	6.100	
3, 5, 13A	8	0.4	1000	1000	6.360	13.198
21	1	0.3	1000	1000	6.486	
21A	8	0.3	1000	1000	6.746	13.970
33, 36	1	0.3	1000	1000	6.872	
33, 36A	8	0.3	1000	1000	7.132	14.742

Table 4.2: Worst case response time for the SAE dataset used by Tindell [164], if every message was authenticated using LEIA.

## 4.7 Security Analysis

This section analyses the security of LEIA under the unforgeability assumption of the MAC scheme under chosen message attacks.

### 4.7.1 Security Proof

While various tools exist for formally proving the security of a protocol, e.g. Tamarin [21], StatVerif [144], we believe our protocol is fairly straightforward, relying on the properties mentioned in 4.3, and therefore a manual games proof is equally suitable.

**Theorem 1** *The LEIA authentication protocol is secure with respect to Definition 4.5 (see Section 4.3.1).* □

**PROOF** Assume that there is an adversary  $\mathcal{A}$  that breaks the  $\text{Auth}_{\Pi}(\eta, \mathcal{A})$  security of the authentication protocol LEIA. Then, an adversary  $\mathcal{B}$  is built, that breaks the  $(t, Q, \eta)$ -security of the  $\text{UF-CMA}_{\text{MAC}}$  scheme.

At the beginning, the adversary  $\mathcal{B}$  randomly picks one target identifier  $\text{id}^*$  and a target epoch  $e^*$ . Then,  $\mathcal{B}$  runs the protocol setup function for each identity  $\text{id}_i$ .

The adversary  $\mathcal{B}$  executes  $\mathcal{A}$ . For this,  $\mathcal{B}$  needs to emulate oracles  $\pi(K_{\text{id}_i})$ . Emulating party  $P_i$  means generating the session key, and keeping track of the counters  $c_{\text{id}_i}$  and epochs  $e_{\text{id}_i}$ , as specified in the protocol description. The session key for an identity is regenerated every time the associated epoch is incremented. The adversary  $\mathcal{A}$  has access to the oracles in  $\Pi$ .

When transitioning from  $e^* - 1$  to  $e^*$ , for identity  $\text{id}^*$ ,  $\mathcal{B}$  will not use the MAC algorithm, as described in the protocol, to generate the session key  $K_{\text{id}^*}^{e^*}$ . Instead, whenever a MAC needs to be computed under the key  $K_{\text{id}^*}^{e^*}$ , the adversary will use the  $\text{MAC}(\cdot, \cdot)$  oracle from the  $\text{UF-CMA}_{\text{MAC}}$  game. Note that due to Assumption 4.1, this will be indistinguishable from the case of using the key generation algorithm  $\text{KG}(\cdot)$ . For all other

cases, it will compute it herself, by running the MAC algorithm.

At some point,  $\mathcal{A}$  terminates. With non-negligible probability ( $> \varepsilon(\eta)$ , where  $\eta$  is the length of the long term key  $K_{id}$ ), there must exist a  $P_i$  which outputs an identity  $id_j$  and a message  $\mathbf{m}$ , without having a matching conversation between  $P_i$  and  $P_j$ . In order for  $P_i$  to produce this output, it means  $\mathcal{A}$  has sent a message  $\mathbf{m} = (c||data)$  and a MAC  $\phi = \text{MAC}(K_{id}^e, \mathbf{m})$  which  $P_i$  has verified, and therefore this must be a valid MAC.

If  $id_j = id^*$  and  $e = e^*$ , the adversary  $\mathcal{B}$  will output  $(\mathbf{m}, \phi)$ ; otherwise, it will output a tuple of random strings. As the identity  $id^*$  and epoch  $e^*$  are chosen at random before the  $\text{setup}(\eta)$  phase, the probability that  $\mathcal{A}$  also attacks  $id^*$  and  $e^*$  is:

$$\mathcal{P}(K_{id^*}^{e^*} = K_{id_j}^e) = \frac{1}{n} \cdot \frac{1}{2^{56}}$$

and recall that  $n$  is the number of identifiers in the system.

In order to win the **UF-CMA**<sub>MAC</sub> game, the adversary needs:

1.  $\text{Verify}(K_{id^*}^{e^*}, \mathbf{m}, \phi) = \text{accept}$ ;
2. the MAC  $\phi$  was never queried to the MAC oracle.

Condition 1. holds because  $\phi$  is a valid MAC, as it was verified by party  $P_i$ .

Condition 2. holds because  $P_i$  and  $P_j$  do not have a matching conversation. ■

### 4.7.2 Security and Safety Trade-off Discussion

As explained in 4.6, LEIA can be implemented by either authenticating every CAN message for a given CAN id, or by setting a *frequency* of authenticating messages from a specific CAN id. The two methods can co-exist on the network. This flexibility comes with its own challenges. While it allows manufacturers to choose a most suitable trade-off between security and bandwidth for their vehicles, determining which messages should fall under which category and, indeed, what frequency is suitable for the second method is a complex process with high stakes. An example of network configura-

tion could be that every message from a safety-critical ECUs would be authenticated (e.g. brakes, parking, ECM), while messages that have a lesser impact on the safety, but would cause the driver to be confused or distracted, would be authenticated at a given frequency (e.g. IPC, HVAC). A safety policy would need to be determined, in the case the authenticated messages do not pass verification, even after the re-synchronisation procedure. For frequency authenticated messages, a possible policy is warning the driver of the inconsistency, but still consuming the messages. For safety-critical messages, it is a more delicate matter. For example, if the messages in question are related to the brakes, a decision should be made on whether to act on the received messages even if they fail authentication, such that the functionality of the vehicle is still respected, or ignore them. The latter case could pose physical safety issues, as the driver could genuinely be pressing the brakes and the vehicle would not respond, creating a dangerous situation. A different approach would be to create a fallback system, whereby if safety critical messages cannot be authenticated, the functionality of the vehicle would not be affected overall, but there would be caps on e.g. the maximum speed the vehicle could be driven at, and the driver would be warned the vehicle has entered a 'safe mode'.

The CAN is an architecture which is highly susceptible to DoS attacks. LEIA is not a solution that tackles this issue, as it cannot be solved at application level. It would require hardware-level changes, e.g. the introduction of a monitor node that would detect anomalous transmission rates and is able to disconnect the misbehaving node for a given time. Nonetheless, such solutions raise the question of what happens if a safety-critical ECU is compromised and used for a DoS attack. Disconnecting it could have serious repercussions for the functionality of the vehicle. LEIA does not influence the ease of carrying out a DoS attack, as the Authentication Channel ids have a lower priority than the corresponding plaintext message ids. In fact, the simplest way of carrying out a DoS attack on the CAN bus, irrespective of whether the network

uses LEIA, is to flood the network with messages sent on the highest priority id ( $0 \times 0$ ), at the highest possible transmission rate. Nonetheless, DoS attacks do not affect the security of LEIA. We emphasize that all other proposed authentication protocols from the literature are susceptible to DoS attacks and do not protect against attackers taking full control over an ECU.

## 4.8 AUTOSAR compliance

At the time of publishing LEIA, the standard for AUTOSAR was version 4.2. This section elaborates how the protocol satisfied the requirements of AUTOSAR 4.2 and describes what changes are needed to make it AUTOSAR 4.4 compliant, which is the standards' version at the time of writing.

### 4.8.1 AUTOSAR 4.2

Regarding freshness, the specification 4.2 [12] states both sending and receiving sides need to maintain a Freshness Value (e.g. counter, timestamp). In LEIA, this is achieved by the 16-bit counters  $c_{id_i}$ , placed in the extended identifier field of a data frame. AUTOSAR recommends the use of 128-bit keys, which LEIA respects though  $K_{id}$ . It also states that, depending on the authentication algorithm chosen, the Message Authentication Code can be truncated, with a minimum recommended length of 64-bit. As described in the protocol, 64-bit MACs are used, which fit in the 8-byte payload field of a data frame. Furthermore, the standard requires the MAC to be calculated based on the id, data and complete FV. In LEIA, the MAC is computed based on the session key  $K_{id_i}^e$ , which is uniquely associated with an identifier  $id_i$ , the counter  $c_{id_i}$  and the data to be transmitted. Regarding MAC verification failure, SecOC requires the receiver to attempt to verify for a number of times (defined by the parameter `SecOCFreshnessCounterSync Attempts`), after which the data is dropped. LEIA

uses the `resync` procedure, in order to keep the protocol in sync, and avoid a possible internal denial of service attack due to the de-synchronisation of counters.

### 4.8.2 AUTOSAR 4.4

AUTOSAR 4.4 [13] contains clarifications regarding the Freshness Value (FV). It states the length of the value cannot exceed 64 bits. If a counter is used, this should be created by concatenating information from four separate counters as depicted in Figure 4.8. The counters are:

**Trip counter:** is incremented by 1 when the ECU starts, if it resets or when ignition changes status from off to on (max 24 bits);

**Reset counter:** is incremented by 1 at regular time intervals, specified by the variable `ResetCyle` (max 24 bits);

**Message counter:** is incremented by 1 for each message transmitted (max 48 bits);

**Reset flag:** updated with the reset flag; it contains the length of the reset flag (max 2 bits).



Figure 4.8: AUTOSAR freshness counter composition from trip, reset and message counters, and reset flag.

The trip counter used in AUTOSAR corresponds to the epoch counter  $e_{id}$  of LEIA, and the message counter corresponds to counter  $c_{id}$ . The standard also specifies that the FV may be truncated, sending the least significant bits out with the authenticated message. For LEIA, the counter would need to be truncated to the least significant 16 bits. Therefore, instead of sending  $c_{id}$  in the extended identified field of a CAN frame, the protocol would transmit the truncated FV. It also employs a *Synchronisation Mes-*

*sage*, which is sent periodically and contains the values of the trip and reset counters. The Synchronisation Message should be authenticated. The trip counter should be stored in Non-volatile Random-Access Memory (NvM) or, ideally, in secure flash.

Using the shorter trip counter instead of the epoch may be seen as a risk. However, it would take an attacker 194 days to roll over the counter, assuming the ECU reset operation takes 1 second. And even in this case, the attacker can only replay previously sniffed messages, a task which would require very precise timing. In a normal operation scenario, if the vehicle was started 10 times/day, it would take 4596 years for the trip counter to roll over, well beyond the expected lifetime of a car.

The Synchronisation Message AUTOSAR employs resembles the `resync` procedure of LEIA, and one could argue the re-synchronisation is now redundant. However, we explain why it is still needed. If the Synchronisation Message is sent too often, it would add significant bandwidth consumption (it would double the communication overhead) to what is already a fairly congested CAN bus. However, sending the message rarely, would mean that de-synchronised nodes will have a large time window in which they cannot trust messages and, potentially, discard them. For safety-critical ECUs, this is not an acceptable outcome. Therefore, we believe that the two solutions deployed together will give the best security vs overhead balance. The counters will be kept synchronised via the Synchronisation Message, and in the case an ECU becomes de-synchronised, it can quickly update its state via `resync`.

## 4.9 Summary

We have proposed LEIA, a lightweight authentication protocol for CAN, that allows ECUs to authenticate each other, therefore preventing a number of attacks presented in the literature. The protocol is proven to be secure under the unforgeability assumption of the MAC scheme under a chosen message attack. LEIA has been designed to run

under the stringent time and bandwidth constraints of automotive applications, and is backwards compatible with existing CAN configuration. The proof of concept implementation proves that LEIA's performance is adequate even for ECUs which fulfil safety-critical functionality. LEIA is the first AUTOSAR compliant lightweight authentication protocol available in the literature. Also, the protocol achieves higher security levels than previously proposed solutions, without the need of additional hardware components or substantial implementation costs. Finally, the real-world requirements and constraints of the CAN bus are taken into consideration, providing a discussion on how to mitigate and overcome them. The properties of LEIA make it suitable for deployment in automotive applications as it strikes the right balance between practicality, cost, latency and security.





## CHAPTER 5

# CAN AUTHENTICATION PROTOCOLS: REVIEW AND EVALUATION

The insecure communication of the CAN bus is often mentioned in the literature as a fundamental flaw that has facilitated attacks, such as the ones we described in Chapter 3, with serious consequences to the safety of a vehicle. Several protocols that improve the security of the CAN bus have been proposed. While a very small number of them explore a design where only an encryption layer is added on the network [55, 88, 117], most works have focused on ensuring that messages are authenticated, with a few of them adding confidentiality as well. In this chapter we will review and evaluate CAN bus authentication protocols, considering recommendations found in both academic research and industry standards.

## 5.1 Motivation

CAN bus authentication has received significant interest from the research community. The topic is paramount to securing the in-vehicle communication of ECUs and, with so many solutions presented in the literature, we believe it is an opportune moment to reflect on them. Reviewing the designs of the protocols, and evaluating them based on a set of common properties allows the reader to form an opinion on their security guarantees, as well as make informed considerations regarding the practicalities of implementing them.

## 5.2 Contribution

We present a survey of authentication protocols designed for the CAN bus, which have been proposed in the literature. Based on research concerned with CAN bus exploitation, the CAN communication standards (ISO 11898) and the industry standard AUTOSAR, we derive a set of desirable properties for a CAN bus authentication protocol. We evaluate the surveyed protocols with respect to the identified properties. Finally, we discuss the reported performance of the protocols, giving particular consideration to the increase in latency and overhead the protocols bring. These two metrics are critical. Greatly increasing the number of frames on the CAN bus can lead to delays for messages with lower priorities. The latency incurred by computing the authentication data and verifying it can lead to delays in processing the message and acting on its contents, which may be unacceptable for safety-critical ECUs.

## 5.3 Desirable Properties for CAN Authentication Protocols

Given the design flaws of the CAN bus mentioned in Section 2.1, it is clear that a solution to stop, or at least mitigate the lapses must be developed. We identify a set of properties an authentication protocol for the CAN bus should have, based on the literature presented in Chapter 3, the CAN bus standards (Section 2.1) and the AUTOSAR guidelines (Section 2.2). We split the properties into two groups: *security properties* and *compatibility properties*. The former refers to the security of the communication on the bus; the latter reflects the ability of the protocols to be implemented on CAN infrastructure that is currently in place in vehicles.

### Security properties

- P1.1:** Sender authenticity;
- P1.2:** Freshness;
- P1.3:** Key uniqueness.

### Compatibility properties

- P2.1:** Backwards compatibility;
- P2.2:** No additional hardware;
- P2.3:** Flexibility.

#### 5.3.1 Security properties

**P1.1 Sender authenticity:** proving the identity of the CAN message source. It should be possible to verify that the message came from *one* specific ECU on the network, and no other ECU can pretend to be any other node on the network. The property is identified as a key requirement in [179, 69, 91, 19]. Without it, an attacker could, for example, disable the brakes of a vehicle, through a cellular-enabled device attached to the OBD-II port [185].

**P1.2 Freshness:** ensuring the content of the message has not been altered or replayed. The protocol should protect against unauthorised modification of data. The works of

[108, 28, 44] highlight freshness as a requirement for securing CAN bus communication. In its absence an attacker could, for example, remove the airbag module and replay its messages on the CAN bus, such that the vehicle believes it is installed and operating correctly, and therefore does not give the driver any warnings; in case of an accident, the airbag would not deploy, as it does not exist [69].

Attacks carried out by injecting CAN messages into the network will be prevented if properties **P1.1** and **P1.2** are satisfied.

**P1.3 Key uniqueness:** ensuring the sets of keys used are unique to the component. If the same keys are used, it means that if an attacker is able to extract them from the firmware of an ECU, or they were recovered through cryptanalysis (e.g. RKE [58] or UDS security keys [46]), they can compromise other components in the vehicle (if per-vehicle keys are used), or other vehicles (if per-model keys are used).

### 5.3.2 Compatibility properties

**P2.1 Backwards compatibility:** the introduction of authentication should have no effect on legacy components, and these should be able to operate oblivious of the authentication layer introduced. This will facilitate incremental adoption of the protocol in existing vehicles, with the incentive of manufacturers being able to stagger the upgrades over a long period of time, not requiring substantial upgrade costs at one time, but allowing for the most safety-critical communications to be secured as soon as possible.

**P2.2 No additional hardware:** no additional hardware shall be required to enable authentication. The hardware can be in the form of other ECUs that need to be introduced on the network, or secure storage components. The addition of extra hardware is a costly venture for manufacturers, therefore it will play a significant role in their

decision of adopting a protocol.

**P2.3 Flexibility:** the protocol should allow for manufacturers to decide which messages need to be authenticated, as not all CAN messages are of safety-critical importance. Therefore, its design should be flexible enough to allow for periodic message authentication (every  $n^{th}$  message, where  $n$  is a parameter chosen by, e.g., the manufacturer) or bulk message authentication. The flexibility property also signals a protocol's ability to keep communication overheads to a minimum, of significance especially as the CAN bus is an already congested network.

## Security parameters and optional considerations

Security parameters play an important role in understanding the security of a protocol. If they are not properly chosen, the protocol might be easily bypassed, even though its design is not flawed. For this reason, we also report on the chosen key size and MAC size of the surveyed authentication protocols. We also report on whether a protocol is AUTOSAR compliant, as this may influence a manufacturer's decision on adopting it, and discuss whether the protocol could be adapted to fit the standards, if it is not already compliant.

**Key size:** the key size directly impacts the security of a protocol. If a key is short enough to be brute-forced in reasonable time, the underlying algorithm will fail to fulfil its purpose. NIST recommends key sizes of at least 128 bits for cipher-based MACs and key sizes of at least 112 bits for hash-based MACs [123].

**MAC size:** there are two conflicting factors to consider – the fact that a CAN frame can carry a limited payload of 8 bytes, as well as standards recommendations for MAC lengths. By NIST standards, MAC lengths of less than 64 bits should not be used [4],

as they are deemed insecure. Short MACs weaken the security of the protocol in that they may allow an attacker to “guess” (brute-force) the correct MAC for a message they want to send [138].

**AUTOSAR compatibility:** the standards are gaining more popularity, with many suppliers and manufacturers implementing them. Therefore, a protocol which already respects the guidelines set out in the standards might be more appealing to a manufacturer. We also discuss the possibility for a protocol to be adapted such that it becomes AUTOSAR compliant, outlining changes that would need to be made, or explaining why the protocol could not be adapted.

## 5.4 Overview and Evaluation of Out-of-Band CAN Bus Authentication Protocols

In the following section, two protocols for lightweight authentication are considered. Both solutions make use of the CAN+ protocol, an improvement of the existing CAN protocol. CAN+ is presented first, then CANAuth and LiBrA-CAN are discussed.

The CAN+ protocol was introduced by Ziermann *et al.* in [190]. It takes advantage of the fact that, for a bit sent on the CAN bus (as specified in the protocol standard), extra bits can be sent in-between the transmission and sampling points. This allows CAN+ to operate alongside CAN-conforming nodes without disturbing their communication. The authors have identified a *grey zone* (Figure 5.1) in the transmission of a CAN bit, delimited by the synchronisation and sampling zones.

The grey zone is used to insert additional information at higher data rates. During this time the bus can take any value without disturbing the CAN-conforming communication. The efficiency of the new protocol has been tested by implementing CAN+ on an Field-Programmable Gate Array (FPGA) (as CAN-controller) with a newly built

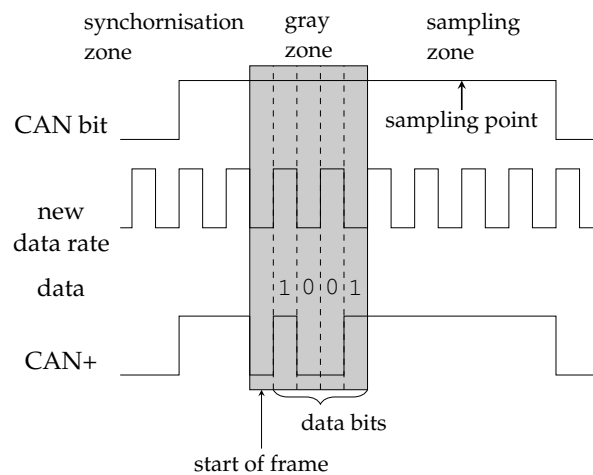


Figure 5.1: Transmission of a data bit for normal CAN and CAN+ protocol.

CAN-transceiver which supports data rates up to 60 Mbit/s (no off-the-shelf products support these). CANcaseXL was used as the Controller Area Network. A usable grey zone of 15% to 40% of the length of a bit transmission interval has been observed. This means that for one CAN bit transmitted, 16 CAN+ bits can be inserted.

Both the main drawback and main strength of the CAN+ protocol is the CAN-transceiver. On the one hand, by being backwards compatible with CAN-conforming nodes, selected ECUs can be included in the CAN+ network and only they would need to receive the upgraded transceivers. The rest of the CAN-conforming nodes on the network would operate normally, oblivious to the CAN+ protocol. While the ultimate goal would be for all nodes to be able to communicate through the CAN+ protocol, this can be done progressively, reducing the immediate costs. CAN+ could be used to add extra functionality to the network, such as authentication. However, vehicles from the previous generations would not benefit from the message authentication scheme unless their transceivers would be changed. Various schemes can be used for solving this issue as well, one being letting the car owners decide if they want this functionality and offering the upgrade when they come to service the vehicle, at a price.

**CANAuth** is a backwards compatible lightweight authentication protocol proposed by Van Herrewege, Singelee, and Verbauwhede in [169]. It guarantees message au-



thentication and replay attack resistance. CANAuth makes use of the CAN+ protocol. The authors state that '*groups of related messages*' should be authenticated using the same key, and the ECU that transmits the messages is responsible for initiating a key-exchange protocol with the receiver nodes. The pre-shared key, a 24 bit counter and a random nonce are used to derive session keys, which are used to authenticate messages. For message authentication, a 32 bit counter is used, in order to prevent replay attacks. The key establishment protocol requires each node to have one or more pre-shared 128 bit keys, and the MAC should be truncated to 80 bits. Due to hard real time constraints in the environment where the CAN bus is used, the authors recommend using a fast MAC function, such as HMAC.

The major assumption of the CANAuth protocol is that the adversary cannot mount invasive attacks on the nodes of the network (ECUs), e.g. obtaining access to tamper-proof storage (where the counters are stored). However, such attacks are known to be possible due to the attacks presented by Koscher *et al.* [93] and Checkoway *et al.* [34].

Groza *et al.* introduce the **LiBrA-CAN** protocol in [60]. It uses MAC mixing and key splitting, achieving authentication for groups of nodes, rather than for individual nodes. The authors group nodes based on their manufacturer, considering they should be trustworthy to each other. The ECU with the most computational power is considered master and the rest are slaves. The master node manages the key distribution in its sub-network. One group can use the same key for authentication. Mixed MACs (LMMACs) are used to integrate multiple MACs in one, through a system of linear equations. A LMMAC passes verification if all component MACs are correct. A number of truncated MACs are sent within one data frame. Replay attacks are prevented by the use of a counter, with a suggested length of 20–40 bits. There is some flexibility left to the manufacturer, in terms of the sizes of the groups of ECUs. The drawbacks of LiBrA-CAN are the increase in traffic on the CAN bus, due to the authentication messages and that all nodes in the system should be aware of it. The protocol can be

implemented over CAN or over CAN+, taking advantage of the out-of-channel communication the latter provides. The benefit is that only CAN+ nodes see the LiBrA-CAN data and a compromised CAN-conforming node would not be able to attack the protocol.

The out-of-band authentication protocols we presented do not respect the *additional hardware* property, or the *AUTOSAR compatibility* property, as they rely on CAN+. By using CAN+ they are *backwards-compatible*, as CAN+ ECUs can exist alongside CAN-conforming ECUs.

We summarise the evaluation of CANAuth and LiBrA-CAN with respect to the other properties in the first part of Table 5.1.

## 5.5 Overview and Evaluation of Authentication Protocols Over Traditional CAN Bus

The second part of Table 5.1 presents a summary of the evaluation of the proposed authentication protocols over traditional CAN, with respect to the security and compatibility properties defined in Section 5.3.

Schweppe *et al.* present an authentication protocol (**c2xCAN**<sup>1</sup>) aimed at securing the communication of sender ECUs with their receivers, in [149]. Their solution assumes a Hardware Security Module (HSM) is used for the storage of shared key material. A *key master* ECU is responsible for key management. Every ECU shares two keys with the key master, one for authenticating to the key master, and one for encrypting other keys. A sender ECU generates session keys and sends it to the appropriate receiver ECUs, via the key master. A session key has a 48 hours lifetime. The communication between ECUs and the key master is authenticated. The protocol can (optionally) use

---

<sup>1</sup>The protocol was not named; based on the title of the article, *Car2x communication: securing the last meter*, we shall refer to the protocol as *c2xCAN*.

Protocol	Sender auth. <sup>a</sup>	Freshness	Key uniqueness	Backwards compatibility	No extra hardware	Flexibility	Key size (bits)	MAC size (bits)	AUTOSAR compatibility	Additional remarks
Out-of-band CAN bus authentication protocols										
CANAuth	E2E	counters	session keys	yes	CAN+ transceiver	not mentioned, but possible	128	80	no, non-standard CAN	
LiBrA-CAN	E2G	counters	session keys	yes	CAN+ transceiver	no	64/128	config. dependent	no, non-standard CAN	recommends insecure MD5
Authentication protocols over traditional CAN bus										
c2xCAN	E2G	optional timestamps	per ECU group + session keys, max 48hrs life	no, uses modified CAN-TP	key master with HSM	not mentioned, but possible	256	32, 64, 96, 128	no, incompatible due to E2G config	
LCAP	E2E	none, MAC not computed on data	session keys	no, encryption layer	none required	no	16	16-bit "magic number"	no, incompatible due to handshakes & MAC computation	data is encrypted, but recommends insecure RC4
MaCAN	E2E, E2G	timestamps	per ECU pair + session keys, max 48hrs life	no, new structure defined in data field of CAN frame	time server, key server	yes, frequency auth	256	32	no, non-trivial in E2E config, incompatible in E2G config	
csCAN	E2E, E2G	counters	per ECU pair	yes	none required	allows data auth for subset of ECUs	n/a	n/a	yes, assumes AUTOSAR compliant ECUs	
CaCAN	E2M	counters	session keys	yes	monitor node	no, as all comm. monitored	256	8	incompatible, it is a monitoring system	
VeCure	E2G, E2E within group	counters	key-pair	yes	none required	choice of allocating ECUs in groups	128	32	no, incompatible due to E2G config	lowest trust group does not authenticate msgs
spCAN	E2E	counters	session keys	no, changes to CRC + encryption layer	key server	no	128	32	no, changes to standard CAN	data is encrypted
VatiCAN	E2E	timestamps	key-pair	yes	global nonce server	yes	128	64	not mentioned, but yes	
LEIA	E2E	counters	session keys per CAN Id	yes	none required	yes	128	64	yes	
TOUCAN	E2E	none	key-pair	no, encryption layer	none required	no, due to encryption layer	128	24	yes	data is encrypted

<sup>a</sup> E2E – ECU to ECU, a sender node authenticates itself to a receiver node; E2G – ECU to group, a sender node authenticates itself to a group of ECUs; E2M – ECU to monitor, a sender node authenticates itself to a monitoring node.

Table 5.1: Comparison of CAN authentication protocols with respect to security and compatibility properties.

timestamps, which requires time synchronisation between the ECUs. It is assumed at least one of the ECUs in a pair of communicating nodes has a real time clock. The authors propose a modified version of the UDS protocol, which would allow them to use the frame fragmentation feature for sending the data and the MAC together, as one Protocol Data Unit (PDU). They suggest using MACs of at least 32-bits, with other acceptable values being 64, 96 and 128 bits [148]. The protocol is not AUTOSAR compliant.

**LCAP** was proposed by Hazem *et al.* in [64]. Is a lightweight broadcast authentication protocol, which also provides confidentiality for the data transmitted. The authors propose the use of a 2 byte *magic number*, which is appended to the message. However, the magic number is not computed on the plaintext of the message being sent. The magic number is inspired by the key derivation function of the TESLA protocol [133], which uses a *chain* of keys. Hazem *et al.* use an HMAC as a one-way function with a key size of 16 bits. Briefly, the way the magic number is computed is as follows:

1. let  $K_H$  be the HMAC key
2. let  $n$  be a pre-established chain length
3. the sender ECU picks a random number,  $r_n$ , which will be the *last* element in the chain
4. the sender then computes  $r_0$ , the *first* element of the chain:

$$r_0 = \text{HMAC}^n(K_H, r_n)$$

5. the sender sends  $r_0$  to the receiver(s)
6. each element in the chain can be computed from  $r_n$  as

$$r_i = \text{HMAC}^{n-i}(K_H, r_n)$$

Authenticated messages will include the magic numbers  $r_1, \dots, r_n$ , which can be veri-

fied by the receiver as

$$r_i = HMAC(K_H, r_{i+1})$$

(e.g. if  $r_0 = HMAC(K_H, r_1)$ , then  $r_1$  is part of the chain)

Handshakes are used in order to establish the secure channel, for session key distribution and to keep the nodes synchronised. This requires a significant number of CAN message identifiers be added to the network (five new ids for each sender-receiver pair), and therefore, due to the number of new CAN ids that need to be introduced in the network configuration, LCAP requires a large address space. Setting up communication channels between senders and receivers, and keeping the communication synchronised, requires a large number of messages to be exchanged. LCAP has a significant communication overhead. No additional hardware is required by the protocol, but LCAP is not AUTOSAR compliant due to its many handshakes. As the data is encrypted, the protocol is not backwards compatible, and it assumes all nodes implement it. The authors use RC4 in their implementation of the protocol, however this cipher is no longer considered secure [136]. Noureldeen *et al.* describe a replay attack on LCAP in [125]. The attack leverages the protocol's Channel Setup Request message, to which a paired ECU will respond with 5 messages. By recording CAN traffic, replaying it message by message, and observing if the message triggers a response of 5 consecutive messages on the same id, the authors can determine if the replayed message was a Channel Setup Request. Once they know the message, they can repeatedly send it with short periodicity. As the Channel Setup Request and its responses are assigned low CAN ids, this means they take priority to other CAN communication, leading to 100% bus load and a DoS.

**MaCAN** is an authenticated protocol described by Hartkopp, Reuber, and Schilling in [63]. It is designed specifically for the CAN bus and takes into account the network's constraints, such as message length and available resources. MaCAN authenticates at

most 4 bytes of data with a 4-byte MAC. Timestamps are used as a source of freshness, and this requires a *time server* to be added onto the network. The time server broadcasts a timestamp at regular intervals. A *key server* deals with key management in the network. It shares a symmetric long term key with each security-enabled ECU. The key server mediates the establishment of keys between two nodes that want to communicate securely. In the case multiple nodes need to be able to verify the authenticity of a message, the authors propose using group keys. MaCAN allows for the frequency of authenticating a message to be set by a special request message. It is not backwards-compatible, as it redefines the CAN data frame to add a destination field. MaCAN is not AUTOSAR compliant and adapting it would be a non-trivial task. Bruni *et al.* give a formal analysis of the MaCAN protocol in [30]. They formally prove the secrecy of both long term keys and session keys used by the protocol. However, they found an attack which leaves one node unauthenticated. They proposed a fix for the protocol.

Lin *et al.* discuss securing the communication on the CAN bus in a number of articles ([103, 104, 105]) and propose an authentication protocol (**csCAN**<sup>2</sup>) with a minimal communication overhead. Their solution is based on symmetric key cryptography, with each pair of nodes that needs to authenticate data sharing a secret key. They do not suggest a length for the MAC, but they do mention counters should be used for replay attack prevention. The protocol allows an ECU to authenticate a message only for a subset of receiver ECUs. From the description, we understand that multiple MACs will need to be sent, one for each receiver ECU. This is not detailed in the paper. No additional hardware is required by the protocol. They also suggest a version of the protocol which uses groups of *receivers*, whereby nodes belonging to the same group share the same secret key. This allows for a lower communication overhead, but weakens the security of the protocol, as direct attacks are possible within the trust group.

---

<sup>2</sup>The protocol was not named; based on the title of the first article, *Cyber-Security for the Controller Area Network (CAN) Communication Protocol*, we shall refer to the protocol as *csCAN*.

The groups protocol assumes the ECUs are AUTOSAR compliant.

**CaCAN** has been introduced in [94] by Kurachi *et al.*; their approach is to use a *monitor node*, which authenticates the other nodes in the network. It detects and destroys unauthorised data frames by overwriting them with an error frame in real time. Challenge-response authentication is used in order to establish a secure channel, and session keys are derived by hashing the *program code* and a nonce, received from the monitor node. A 32-bit counter is used to prevent replay attacks. A MAC length of 8 bits is argued to be enough, due to the usage of session keys. The protocol is backwards compatible in the configuration where the MAC is sent in a separate CAN frame. As all ECUs have their communication monitored, there is no room for flexibility. The monitor node requires a modified CAN controller, and is an extra node that needs to be added on the network. Also, as is the general case with centralised authorities, if the monitor node is compromised or removed, the entire network is compromised as well. The protocol is not AUTOSAR compliant, as it is a monitoring system.

**VeCure** is proposed by Wang and Sawhney in [177]. Their solution relies on trust groups. ECUs are grouped based on how difficult it is to compromise them. ECUs without any external interfaces are considered highly-trusted. ECUs with wireless interfaces (e.g. telematics) or nodes with direct access to the CAN network (e.g. OBD-II), are considered low-trust. All nodes in a group share a key. In the case of multiple tiers of trust, the nodes of a group also know the keys of other, lower-trust groups. VeCure allows for a flexible implementation, in the sense that there is freedom in choosing the number of trust groups, but it cannot tailor the frequency a message is authenticated. Session counters and message counters are used for protecting against replay attacks. The length of the MAC is set to 32-bits. ECUs in the lowest trust group do not use authenticated communication, and therefore do not possess any key material. It is unclear how this would prevent e.g. the OBD-II to masquerade as the telematics unit, as it is suggested by the authors the two nodes should belong to the same low-trust group.

ECUs in other groups would not be able to tell the difference between distinct ECUs in the low-trust group, due to the lack of source identifier and authentication. VeCure is backwards compatible, as the data and MAC are sent in separate data frames. The protocol is not AUTOSAR compliant and it does not require additional hardware.

Woo, Jo and Lee propose a protocol (**spCAN**<sup>3</sup>) which provides both encryption and authentication in [181]. Their design uses AES-128 and a keyed HMAC. A gateway ECU is responsible for distributing session keys to the other ECUs on vehicle startup. The session key derivation protocol is fully explained in the paper, and shows how a sender ECU obtains its encryption and authentication keys. However, it is not specified how a receiver ECU obtains those keys as well, such that it can decrypt and verify messages. The protocol makes use of the extended CAN frames, and splits a 32-bit MAC in two 16-bit chunks, one sent in the extended id field and one in the CRC field. Due to the placement of the MAC in the frame and due to the encryption layer, the protocol is not backwards compatible and does not allow for flexibility in its implementation. Counters are used to prevent replay attacks. spCAN is not AUTOSAR compliant.

**VatiCAN** [127] is a CAN authentication protocol proposed by Nürnberger and Rossow. It is backwards compatible with the CAN bus standard. VatiCAN provides sender and content authenticity by using 64-bit HMACs, based on KECCAK. Keys need to be unique per ECU, but the authors mention groups of ECUs can be formed, in order to save memory. VatiCAN can be implemented in a flexible way, whereby only some messages are authenticated. The protocol requires one additional ECU to be introduced on the network, a *global nonce generator*. One major assumption of the protocol is that the nonce generator is protected by a hardware-assisted spoofing prevention mechanism, and an attacker cannot impersonate and inject arbitrary nonces. This is proven to be a weakness by Bulck, Mühlberg and Piessens in [168], where they

---

<sup>3</sup>The protocol was not named; based on the title of the article, *A practical wireless attack on the connected car and security protocol for in-vehicle CAN*, we shall refer to the protocol as spCAN.



describe a birthday attack on the nonce generator. While not mentioned in the paper, VatiCAN is AUTOSAR compliant.

Radu and Garcia present **LEIA** in [140]. They propose an authentication protocol which is AUTOSAR compliant, and is backwards compatible with the CAN bus standard. LEIA provides authentication by using a 64-bit MAC, and uses a 16-bit counter for protection against replay attacks. The key derivation for session keys, is based on long term keys associated with a particular CAN id and on epoch counters. LEIA is a software-only solution, therefore no additional hardware needs to be introduced onto the in-vehicle network. The protocol can be implemented in a flexible way, where either all messages can be authenticated or a per CAN id authenticating frequency can be configured.

The most recent proposal, **TOUCAN**, comes from Bella *et al.* [24]. The protocol provides both integrity and confidentiality. It is designed to respect the AUTOSAR guidelines, and it fits one of the profiles defined by the standard (profile 2 — 24Bit-CMAC-No-FV). TOUCAN uses a 24-bit MAC (Chaskey [118] is recommended) for authentication, and AES-128 for encryption. However, the design does not use a counter, and therefore it is susceptible to replay attacks. The AUTOSAR standard mentions this profile should only be used ‘*if no synchronized freshness value is established*’ [18]. No additional hardware is required by the protocol. TOUCAN is not backwards-compatible, due to the encryption layer. It is not flexible either, as the MAC is appended to the data, and sent within the same CAN frame. Key material is assumed to be on the ECU already, and there are no further details.

## 5.6 Performance Evaluation

*Overhead* and *latency* play a crucial role in any design for CAN authentication. Due to the sensitivity of the communication to time delays, a proven secure protocol can

be rendered unusable if these two metrics are deemed unacceptable. Due to the discrepancy in hardware used for implementations, a direct comparison cannot be made among all the schemes, therefore the metrics are reported per individual protocol.

**Overhead:** we consider how many CAN data frames are required to send, if we want to transmit an authenticated payload of 8 bytes. While data frames can carry less than 8 bytes, from our own experiments<sup>4</sup> and from available datasets [99, 43, 42], we know that most CAN frames carry payloads of 8 bytes. Some protocols fit MACs in the data field of a frame, and therefore we choose to present the comparison with respect to a fixed payload.

**Latency:** some ECUs fulfil safety-critical functionality (e.g. brakes), and therefore the delay in them being able to process and take action on the data within CAN messages is crucial. We report on the additional time needed to successfully verify a CAN frame; in some cases, the authors only benchmarked the cryptographic algorithm used in the verification, and therefore we can only present this.

The authors of **c2xCAN** analyse the performance of their key distribution protocol by modelling it with the DIPLODOCUS UML profile [8] and the TTool simulation engine [92]. Their tests reveal that it takes *9ms* for the key distribution to take place, on a bus speed of 500Kb/s. It requires 9 CAN frames to be sent. For data transmission, they modelled the simulation in a MATLAB toolkit [32]. They report latency times for a number of MAC sizes, as presented in Table 5.2. Their model assumes 0.4ms are needed to send a frame and 0.2ms are needed to process a received frame, without further reference to the source of these numbers. The overhead of the protocol varies by the implementation and MAC length chosen. In the best case scenario, 32 bits of data are authenticated with a 32-bit MAC, which means the communication doubles.

**LCAP** was implemented on a Freescale Starter-TRAK TRK-MPC5604B board, clock

---

<sup>4</sup>We had access to a Ford Focus, for another part of the research presented in this thesis (Part III). We sniffed the CAN traffic, and from the trace, we could clearly see most messages had a length of 8 bytes.

MAC size	Avg latency
32	1.8 ms
64	2.1 ms
128	3.0 ms
256	4.1 ms
512	7.3 ms

Table 5.2: Latency of c2xCAN authenticated messages, based on MAC size.

Hash algorithm	1st run	Subsequent runs
MD5	144 $\mu$ s	86 $\mu$ s
SHA1	278 $\mu$ s	154 $\mu$ s
SHA224	539 $\mu$ s	285 $\mu$ s
SHA256	540 $\mu$ s	285 $\mu$ s

Table 5.3: LCAP performance of different hashing algorithms, used in the *magic number* computation — key size was 16 bits.

frequency 64 MHz, 512 Kb code flash and 64 Kb data flash. A number of hash functions are benchmarked, any of which can be used in the magic number computation (Table 5.3). They also benchmarked using the RC4 cipher for the encryption layer, which took an average of 160  $\mu$ s, for a 64 bit input. LCAP has a significant overhead with respect to the number of new CAN ids it requires. The authors note that  $5 \times n_s \times n_r$  new CAN ids are needed, where  $n_s$  is the number of sender ECUs and  $n_r$  is the number of receiver ECUs. The protocol also has a significant communication overhead. Its *Channel setup* phase requires 6 CAN frames to be transmitted, *Message setup* requires  $n_{id}$  frames, where  $n_{id}$  is the number of CAN ids used in the vehicle, *Data exchange* needs one CAN frame, *Chain refresh* also requires one frame, but it is done periodically, and there is no further information of a suitable time window, *Soft sync* needs  $2 + n_{s-r}$  frames, where  $n_{s-r}$  is the number of CAN ids used by one pair of sender-receiver ECUs, and *Hard sync* requires  $6 + n_{s-r}$  CAN frames.

**MaCAN** has not been implemented, but it would double the communication, for sending an 8-byte payload, as it authenticates 32-bits of data with a 32 bit MAC.

The authors of **csCAN** do not provide information on the size of the MAC, and therefore we cannot draw conclusions on how the protocol influences the bus load. The protocol does not appear to have been implemented, but the authors compute

$n$	$P$	$Q$		
		$10^{-1}$	$10^{-4}$	$10^{-7}$
1	$10^{-3}$	12.05ms	12.09ms	12.12ms
	$10^{-6}$	12.45ms	12.49ms	12.55ms
	$10^{-9}$	12.93ms	12.96ms	n/a
3	$10^{-1}$	12.12ms	12.19ms	12.22ms
	$10^{-2}$	12.49ms	12.55ms	12.65ms
	$10^{-3}$	12.93ms	12.96ms	n/a

Table 5.4: Latency for an authenticated message, using the csCAN protocol.  $n$  is the number of receivers for a message,  $P$  is the probability of a successful attack, and  $Q$  is the probability that a counter is desynchronised.

the latency of the authenticated messages, for a bus speed of 500Kb/s, based on the methodology elaborated in [49, Chapter 3]. Their evaluation depends on the probability of a successful attack ( $P$ ), the probability that a counter is not synchronised ( $Q$ ), and the number of receivers a message has ( $n$ ). For a number of receivers (for each CAN message) of one ( $n = 1$ ), they report the findings for a probability of a successful attack of  $10^{-3}$ ,  $10^{-6}$  and  $10^{-10}$ . If the number of receivers is three ( $n = 3$ ), the authors state that it is infeasible to use the protocol for a probability of attack smaller than  $10^{-3}$ . The results are summarised in Table 5.4.

**CaCAN** was implemented on an Altera FPGA development board, clock frequency 50 MHz, 512 Kb flash memory and 20 Kb RAM. The results show that the average time required to compute the authentication data for one data frame is 2.12  $\mu$ s (input will be 52 bits of data + 32 bit counter + 256 bit key<sup>5</sup> + 11 bits CAN id). As the protocol relies on the monitor node to destroy frames, it means that at a bus speed of 1Mb/s, the monitor has verified the message within the transmission of 3 more bits (the CRC would be in transmission), and is therefore within an appropriate time window to be able to destroy the frame, if it cannot verify it.

The **VeCure** protocol has two phases, when authenticating data: an offline phase,

<sup>5</sup>The authors mention that the authentication key is derived by hashing the program code and a nonce, by using the SHA256 algorithm; while it is not explicitly stated the authentication key is 256 bits, the output of SHA256 is, and it is reasonable to assume it is used in its entirety

which is calculated based on the ECU id, session counter, overflow counter, MAC counter and a key, and an online phase, which is computed on the data that needs to be authenticated and the result of the offline phase. The authors argue this design allows for a lower latency, as the offline phase can be pre-computed. VeCure has been implemented on a Freescale EVB9S12XEP100 board, 50 MHz clock frequency, 1Mb flash and 64Kb RAM. SHA-3 was used as the algorithm for the offline phase. The authors measured the time for each phase. They report the offline phase takes approximately 931  $\mu$ s, and the online phase needs 50  $\mu$ s. In terms of overhead, the communication doubles, as the data is sent in one frame and the MAC in a subsequent frame.

**VatiCAN** was implemented on Atmel AVR ATmega328p microcontrollers, clock frequency 16 MHz, 32 Kb flash and 2 Kb RAM. The CAN bus speed was set at 500Kb/s. The HMAC computation takes 935 $\mu$ s<sup>6</sup>, and the delay introduced between receiving a frame and successfully verifying it is 3.3 ms. The protocol doubles the communication overhead, as the data and MAC are sent in two separate frames (though this is worst case scenario, as VatiCAN allows for periodic authentication as well).

**LEIA** has been implemented on Atmel AVR ATmega328p microcontrollers, clock frequency 16 MHz, 32 Kb flash and 2 Kb RAM. The CAN bus speed was set to 500Kb/s. The MAC computation takes 738  $\mu$ s, and the delay introduced between receiving a frame and successfully verifying it is 2.96 ms. At a speed of 100 km/h, the induced latency means a travelling distance of 8.2 cm. In the worst case scenario, the protocol doubles the communication overhead, as one extra frame needs to be sent, with the MAC. The protocol allows for flexibility in implementation, being able to choose periodic authentication, and not all nodes need to be aware of the protocol.

---

<sup>6</sup>It is reported in [127] that the MAC computation takes 2950 $\mu$ s. Compared to LEIA's performance, we found the large difference in timing odd, as they both use Keccak. We ran timing tests on the code supplied by the authors at [VatiCAN Open Source Lighthouse Project](http://www.automotive-security.net/vatican/) (<http://www.automotive-security.net/vatican/>) and obtained more comparable readings. It is possible that the authors have, since publishing, changed the crypto library, opting for one optimised for the AVR microcontroller.

Protocol	MAC ( $\mu$ s)	Latency ( $\mu$ s)
VatiCAN <sup>6</sup>	935	3300
LEIA	738	2960

Table 5.5: VatiCAN and LEIA performance benchmark on Atmel AVR.

Table 5.5 presents the comparison of VatiCAN and LEIA with respect to MAC computation and latency introduced by the protocols. Such a direct comparison is possible as both implementations were done on the same hardware (Arduino Uno) and use the same cryptographic library.

Both VatiCAN and LEIA have also been implemented in [168], within the framework VulCAN, a generic design for efficient vehicle message authentication on top of Sancus, an open-source embedded protected module architecture. Sancus [124] can extend the CPU with a cryptographic core, therefore providing hardware-level authenticated encryption, key derivation and secure key storage. Both protocols have been implemented in two variants: unprotected, where only the protocol is part of the trusted code base, and protected, where the CAN driver is part of the trusted code base. Table 5.6 presents the results of the experiments. LEIA's increased timings are due to the usage of session keys, which add overhead for key generation, raising a trade-off between security and reaction time.

Scenario	Cycles	Time	Overhead
Sancus+VatiCAN (unprotected)	15570	0.78 ms	91%
Sancus+VatiCAN (protected)	16036	0.80 ms	97%
Sancus+LEIA (unprotected)	18770	0.94 ms	131%
Sancus+LEIA (protected)	19211	0.96 ms	136%

Table 5.6: Overhead to send an (authenticated) CAN message with and without Sancus encryption and software protection [168].

Table 5.7 presents a comparison of communication overheads in the case of transmitting 8 bytes of data, authenticated under the respective schemes. Additional overheads are also presented, according to the specifications of each protocol. The recom-

mended MAC algorithms are also presented, alongside the number of calls to cryptographic functions each protocol requires, for data authentication and any other setup requirements.

## 5.7 Summary

We have evaluated and reviewed a number of CAN authentication protocols which have been proposed in the literature. We have identified a set of desirable properties such a protocol should have, both from a security and a compatibility perspective. When deriving these properties we took into account both related research and industry standards. The aim of this evaluation is to provide a clearer overview of protocols available, what their security goals are, and what are the strengths and weaknesses of each. From our evaluation, the LEIA protocol appears to be the most promising candidate for securing the CAN bus communication, as it fulfils all the properties defined.

Protocol	Frames per 8 bytes auth. data	Other overheads (in frames)	MAC Algorithm	MAC input size (bytes)	Calls to crypto. functions, per ECU (to MAC functions, unless otherwise specified)
c2xCAN	2	key distribution: $9 + 9 * n_r$	HMAC or CMAC	n/a <sup>b</sup>	n/a <sup>b</sup>
LCAP	1 <sup>a</sup>	channel setup: 6; msg setup: $n_{ids}$ ; soft sync: $2 + n_{rids}$ ; hard sync: $6 + n_{rids}$	HMAC	hash dependent (8/20/28/32)	data exchange: 1 enc.; chain generation: $\lambda * n_{id}$ HMACs; channel setup: $\sum_{id \in id_t} n_{rid}$ enc.; chain refresh: 1 enc.; soft resync: $n_{rid}$ enc.; hard resync: $1 + n_{rid}$ enc.
MaCAN	2	key distribution: $6 + 6 * (n_r + 1)$	AES-CMAC	42	data auth: 1; session key: 1 decryption, 1 CMAC
CaCAN	1 <sup>a</sup>	key exchange: 3	HMAC SHA-256	45	data auth: 1 HMAC; session key: 1 hash;
VeCure	2	none	HMAC SHA-3	37	data auth: 1;
spCAN	2	key distribution: 3; key update: 2	HMAC	42	data auth: 1 enc., 1 HMAC; key distribution: $3 * id_t$ key update: 1 enc., 1 HMAC;
VatiCAN	2	none	HMAC Keccak	30	data auth: 1
LEIA	2	resync: 3	HMAC Keccak	26	data auth: 1; session key: 1 resync: up to 2
Toucan	2	none	Chaskey	21	data auth: 1 MAC, 1 enc.

where  $n_r$  is the number of *receivers* (ECUs or groups) for an ECU initiating the function;  $n_{rids}$  is the number of CAN *ids* that receive the given message;  $\lambda$  is the length of the chain (recommended 100);  $id_t$  is the set of all CAN *ids* on which an ECU transmits;

<sup>a</sup> if using CAN extended identifiers.

<sup>b</sup> c2xCAN does not provide details of the input for the MAC computation.

Table 5.7: Comparison of communication overheads, for transmitting 8 bytes of authenticated data, and additional overheads for setup/resync functionality.





---

## PART III

# *Towards Large-Scale Fuzzing of Automotive Electronic Components*

---



## CHAPTER 6

# EXTRACTING THE CONTROL FLOW GRAPH FROM ECU FIRMWARE

In this chapter, we describe a method of representing the control flow of firmware through an annotated Control Flow Graph (CFG). As we will see, one of the issues of analysing ECU firmware is the diversity of the hardware that they are implemented on. Extracting the CFG allows us to abstract away from the underlying architecture and to later build tools on said structure that do not have to deal with the particulars of any architecture.

### 6.1 Related Work

In this section we visit concepts relating to firmware analysis, and review relevant related work, with a focus on embedded device firmware.

Program analysis is widely used for understanding software or firmware functionality, in the absence of the source code. Disassembling is the most common technique

for this, involving a disassembler to translate machine code (from a compiled binary) to a low-level language, assembly. While program analysis of popular architectures (x86, x64, ARM, AArch64, Microprocessor without Interlocked Pipelined Stages (MIPS), PowerPC (PPC)) is a thriving field, with many resources and tools available, ECUs, unfortunately, tend to use obscure architectures which generally lack support. Furthermore, ECU firmwares are bare-metal systems (they do not have OSes) and the chips used are automotive-specific. Even state-of-the-art tools have only limited support for these. Therefore, analysing ECU firmware can be a laborious and tedious manual task.

Firmware is a specific class of computer software which is developed to take advantage of low-level access to the device's hardware. The term firmware is widely used when referring to the program(s) running on MCUs. They are widely used in Internet of Things devices, Industrial Control Systems, routers, smart TVs, etc.

Obtaining the firmware itself can prove to be a difficult task, as noted by [39]. Various methods for this exist, from the firmware being available on the vendor's website, to intercepting a firmware update, or extracting it directly from the hardware, with standard flash programmers [46] or by using glitches (e.g. by using the *ReadMemory-ByAddress* service of the UDS and glitching the *SecurityAccess* check, Milburn *et al.* were able to extract an ECU's firmware in three days [112]).

## Firmware Structure

Firmware may present itself in two versions:

**Operating and file systems:** the firmware is composed of an OS kernel, and associated file system; the file system will contain all software and configuration files that allow the device to operate and fulfil its functionality.

**Single binary:** also known as *bare-metal* firmware, it contains all the functionality of the device; interaction with the hardware is done in a direct manner; this type of firmware is highly tailored to the underlying hardware it runs on.

Most ECUs have firmware which falls in the second category. When referring to ECU firmware, it should be implied it is a reference to the bare-metal type, throughout the rest of the thesis, unless otherwise specified. Assessing a recovered firmware image, in the absence of the source code, is done through *binary program analysis*.

In order to analyse such firmware, the program's assembly instructions must be recovered, though the process of *disassembly*, performed with the help of a *disassembler*, which translates binary data into human-readable assembly code. The process is not perfect, with problems still existing even for the most popular architectures, such as x86, x64 and ARM. Therefore, the quality of the analysis performed is dependent on the accuracy of the disassembler. As ECUs run bare-metal firmware, understanding the hardware is an essential part of the analysis. Datasheets are the primary resource for identifying the memory layout of the MCU (addresses for Read Only Memory (ROM), RAM, Input/ Output (IO), Boot sections) and at what memory address should the firmware be loaded. The datasheets also contain information on where the reset vector is located in memory, as well as mappings of various registers (CAN, interrupts, SPI, Universal Asynchronous Receiver Transmitter (UART)), which could serve as entry points for the program.

Firmware is, generally, written in a high-level language (e.g. C), structured through functions; execution of said firmware means transfer of control between these functions. Control can be transferred through function calls, or through interrupts which signal to the MCU it needs to pause the execution of the current code and allocate time for another particular function (e.g. a CAN controller will use an interrupt when a new CAN message has been received, and the firmware needs to parse its contents and take action according to its contents). Knowing the relationship between the different blocks of code is valuable when performing firmware analysis, as it enables visualisation of the disassembled code in a clearer, more user-friendly manner.

Programs can be represented by a combination of the following:

**Basic Block (*bb*):** the maximal sequence of instructions that does not modify the local control flow of the program; all instructions of a basic block are executed in a sequential manner; basic blocks, generally, end with an instruction which transfers control to another basic block (transfer may be dependent on a condition).

**Control Flow Graph (CFG):** a directed graph representing the transfer of control between the basic blocks; basic blocks are represented as vertices, program branches are represented as edges.

Functions, in the high-level concept, can be viewed as a set of basic blocks, with one of the basic blocks being the entry point of the function. A program can be represented simply, as a set of its functions.

Within the context of CFGs, basic blocks have in-degrees and out-degrees, representing the number of edges directing towards, respectively from the vertex. Given two basic blocks,  $bb_i$  and  $bb_j$ , and a direct edge  $bb_i \rightarrow bb_j$ ,  $bb_j$  is a *successor* of  $bb_i$ , and  $bb_i$  is a *predecessor* of  $bb_j$ .

## Analysis Methods

There are two main methods through which firmware can be analysed:

**Static analysis:** analysing a program without executing it, therefore without knowing the state of the program at each execution step; it involves comprehending the firmware's functionality from listed instructions, retrieved via the disassembly process; as the control flow of a program may be changed, conditional on variables which have values assigned only during execution (e.g. from RAM), assumptions on these values must be made, possibly leading to an overestimation of what code/instructions will actually be executed.

**Dynamic analysis:** analysing a program during its execution, being able to observe the state of the program at each execution step; this also allows for discovery of *exact* interaction between the firmware and the hardware, how output is gener-

ated, or how the control flow may be affected by input, without the need of assumptions. However, while performing dynamic analysis, the control flow coverage is driven by the structure of the program and the inputs provided, therefore areas of code may never be reached, if the correct values are not supplied.

In the context of ECUs and their firmware, dynamic analysis is more difficult, if not impossible in most cases. A key requirement of dynamically analysing firmware, is the ability to run it, and this can be achieved by:

- running the firmware on the original hardware, and interacting with the program through on-board debug interfaces (e.g. Joint Test Action Group (JTAG) [87], Serial Wire Debug (SWD) [102]) if these are available;
- using an *emulator*, which can imitate the behaviour of the hardware, to *some* degree, enough such that the program functionality is not massively affected (i.e. the control flow is not changed).

Emulation tools exist, with the most well known being QEMU [25]. However, as the MCUs in the ECUs are automotive-specific, these are not supported by the emulators. When adding the fact that ECUs use a wide variety of architectures and MCUs, and implement complex combinations of peripherals, each needing its own emulator, it becomes clear that the cost of performing dynamic analysis of ECU firmware at a large scale is prohibitive.

### 6.1.1 Analysing (Embedded) Firmware

Program analysis is concerned with analysing the *behaviour* of a program, generally in an automated way, and deriving *properties* of said program (with the purpose of, for example, confirming that the software in question does not have malicious or unintended behaviour) [122]. The vast amount of research has looked at analysing various programs, and it has been concerned mostly with binaries that run on top of an OS. For the purpose of our research, we are mainly interested in works which have dealt



with recovering program structure, instruction recovery and function detection.

Costin *et al.* present a large scale static analysis of embedded firmware in [38], and they discuss challenges faced while carrying out the research. They mention that due to the diversity of vendors, architectures and OSes in embedded devices, and the difficulties in identifying firmware versions, having a representative sample of firmware was challenging. Moreover, they discuss the issues around unpacking the firmware and dealing with custom file formats. They point out that single binary firmware is especially challenging to deal with. Their analysis was performed on 32,000 firmware images and they discovered 38 new vulnerabilities in 693 of these images. Throughout the analysis they point out inadequacies in existing tools and highlight that, even with their automation, parts of the process remain highly manual tasks, concerns that we also faced while carrying out parts of our own research.

Zaddach *et al.* introduce Avatar [187], a framework for dynamic analysis of embedded systems firmware. They touch on the two main concerns we have faced while conducting research into ECU firmware: the lack of documentation and the extensive variety of hardware used in the components. Their solution combines an emulator with the real hardware, in order to make complex dynamic analysis of embedded devices scalable.

Andriesse *et al.* look into the accuracy of modern disassemblers when applied to x86/x64 binaries in [6]. They study 981 binaries, and 9 of the most popular disassemblers, with a focus on the accuracy of program structure recovery and the challenges they faced. They also compare their findings with the results of similar published research. Their findings show that the literature tends to focus on corner cases or very complex constructs, such as in-line data, overlapping instructions or shared basic blocks, which are not so wide spread in real-world programs, and greatly depend on the compiler used. As such, the effectiveness of binary-based research is often times underestimated.

Andriesse, Slowinska and Bos present Nucleus in [7], a tool aimed at bridging the gap between disassemblers, which have high accuracy in recovering instructions, but are not as effective when it comes to detecting functions, in the absence of symbols (in which case the symbols table contains information about function name, start address and size). For any form of program analysis, obtaining an accurate representation of the program, both in terms of instructions and control flow is highly crucial. Instead of using function signatures, they propose using the Interprocedural CFG, which is determined by the control flow between basic blocks. They show that by using a linear disassembly approach [147] and connected component analysis [65] they are able to detect function entry points and function boundaries better than state-of-the-art tools, such as Interactive DisAssembler (IDA).

Other methods for function detection exist, such as machine learning based approaches. Bao *et al.* [20] propose using supervised learning to identify specific instruction patterns or byte sequences (opcodes) used in either function prologue or epilogue, or in function calls. Neural networks are also applied to function detection, by Shin *et al.* in [151]. Their algorithm is applied directly to the program bytes, therefore eliminating the uncertainty of instruction decoding. They report much better performance results, while achieving similar accuracy to the work of Bao.

### 6.1.2 ECU Firmware Analysis

In the context of automotive security, remote keyless entry systems have received a lot of attention ([175, 58, 67]). Assessing their security usually involves reverse engineering a (generally proprietary) cipher from the firmware of the immobiliser or ECU and finding weaknesses. However, the reverse engineering process is never fully described, as the cryptographic algorithms are the main scope of the research. Similarly, ciphers used in authenticating to Unified Diagnostics Services running on ECUs are reverse engineered in [46], but there is no description of the process.

The only instance of detailed ECU firmware analysis comes from Miller and Valasek [115]. They reverse engineered the firmware from a V850 chip which was responsible for dealing with the CAN communication on the ECU they were investigating. They explain that the procedure took them ‘*several weeks*’, highlighting the extensive effort required. They needed to normalise the IDA database, ensure that functions were correctly created, and amend sections that the disassembler could not figure out on its own. They automated parts of the process by looking for specific opcodes, representing combinations of instructions found in functions’ prologues for the architecture they were working on. They used datasheets to determine the address space and the various register and segment areas. They then proceed to understand how the CAN modules are used within the firmware, with the ultimate goal of modifying the firmware to accept commands via SPI and send CAN messages based on these. Interestingly, the authors mention they had disassembled parts of the firmware incorrectly, treating areas which were supposed to be data as code, and this led to further delays and confusion. From their research we learn important information, such as the fragility of the disassembly process and the value of understanding the hardware specification and architecture.

The diversity of chips and architectures on which ECUs are built makes creating an automated, ‘universal’ firmware analysis solution difficult, and therefore it is hard to scale up the processes. Binary Independent Languages (BILs) are designed to fill this gap (e.g. Binary Analysis Platform (BAP)<sup>1</sup> [29], VEX<sup>2</sup>). They provide an abstraction layer from the underlying instruction set, and use an Intermediate Representation (IR) to describe the operations performed by the MCU. However, existing BILs support a limited number of mainstream architectures, such as ARM, x86, x86-64, PPC, and MIPS for BAP, whereas VEX requires an OS, not providing support for bare-metal

---

<sup>1</sup>The Carnegie Mellon University Binary Analysis Platform (CMU BAP) (<https://github.com/BinaryAnalysisPlatform/bap>)

<sup>2</sup>angr VEX intermediate representation (<https://docs.angr.io/advanced-topics/ir>)

firmware [47]. This means that, in order to use an IR, lifters for all the missing architectures would need to be implemented. This is an extensive task, in the sense that it would require a lot of time investment, and assumes a high level of knowledge about the architectures being lifted, and therefore would require specialist skills. The most suitable solution is to operate directly on the instruction set of the ECU architecture. This requires understanding of a subset of instructions, as opposed to all, for lifting to a BIL, thus requiring less time commitment.

### 6.1.3 Tools and Frameworks

Program analysis requires the help of tools, in order to be able to carry out the task. In this section we review such tools and frameworks, considering the architectures they support and the features they implement.

#### Commercial tools

The tools presented below are commercial solutions. Free versions of the software are often offered, but they have very limited support and lack many essential features.

IDA Pro [66] is a state-of-the-art disassembler, with support for a wide range of architectures, including x86, ARM, MIPS, PPC and Infineon TriCore. IDA has the highest number of supported architectures and processors out of all the tools available. IDA's functionality can be extended and automated through scripts, either in IDC (C-like language developed specifically to be used with IDA) or Python. It can also be run headless, for batch processing. IDA can lift instructions to C-like pseudocode.

Hopper [41] is a disassembler initially developed only for macOS, but can now be run on Linux as well. It supports x86, x64, ARM and PPC architectures. It can be extended to other Central Processing Units (CPUs) and file formats through its SDK, and scripts can be written in Python.

Binary Ninja [171] is a newer cross-platform disassembler, which supports x86, ARM, PPC, MIPS and AVR. It can lift instructions to either a low level or a medium level intermediate representation, the latter being akin to the C programming language. Binary Ninja can also be extended through Python plugins.

### Open source tools

The tools presented below are open source solutions. They benefit from large communities which provide support and extend their functionality.

Ghidra [126] is a new state-of-the-art cross-platform disassembler, open-sourced by the National Security Agency. It provides support for a variety of architectures, including x86, x64, ARM, MIPS, PPC and m68k (Motorola 68000). The software allows extra functionality or automation to be achieved through Java or Python scripts. Ghidra can lift the assembly instructions to P-code, its own intermediate representation. Batch processing of firmwares can be achieved by running the tool in a *headless* configuration, and providing a custom script. It also provides multi-user collaboration support and version control.

Capstone [139] is a disassembler with support for a good range of architectures, ARM, m68k, MIPS, PPC, x86 and x64 including. It provides bindings for scripting in an extensive number of programming languages, Python, Perl, PHP, Lua, Rust and Ruby to name a few. This gives users the flexibility of developing their own tool, tailored to their specific needs and requirements.

Radare2 [130] is an open-source framework which supports disassembling a wide range of architectures, including x86, x64, ARM, AVR, m68k, MIPS, JVM, PPC and SuperH. The framework has a command line text based interface, and therefore a steep learning curve. A graphical user interface was later developed, named Cutter<sup>3</sup>.

---

<sup>3</sup>**Cutter**: free, open-source reverse engineering framework, powered by radare2 (<https://github.com/radareorg/cutter>)

Radare2 can lift instructions to its own intermediate representation, ESIL (Evaluable Strings Intermediate Language). It can be extended with scripts in a variety of languages, such as Python, Go, Rust, Ruby or Java. Radare2 uses Capstone internally as its default disassembler for a number of popular architectures, but it can also provide an assembler by integrating the Keystone project<sup>4</sup> and emulation with the help of the Unicorn project<sup>5</sup> [160].

angr [154] is a binary analysis platform, with an emphasis on allowing users to write their own tool using the angr Application Program Interface (API). The framework mainly aims to be a tool to aid discovery of vulnerabilities and flaws within programs. It has been used widely in the academic community for building various tools, such as Firmalice [153], a framework for detecting authentication bypass backdoors or Driller [157], a vulnerability discovery tool which uses fuzzing and concolic execution. angr supports lifting to the VEX intermediate representation.

**Choosing the appropriate tools for our research.** Based on our overview of existing tools, we decided to use IDA in our research, as it had the best support for the architectures we analysed. Its support for custom scripts helped in automating as much of the firmware analysis process as possible. Choosing any other tool would have required the additional effort of writing a disassembler for any architecture the tool would not support. However, it should be noted that IDA does not have support for automotive profiles of MCUs, and additional annotation with the extra functionality is required.

## 6.2 Motivation and Challenges

Consider a vehicle's electronic components, in their entirety. As a car will have over 70 ECUs [93], the manufacturer does not have the resources needed to design the hard-

---

<sup>4</sup>Keystone – The Ultimate Assembler (<https://www.keystone-engine.org/>)

<sup>5</sup>Unicorn – The Ultimate CPU Emulator (<https://www.unicorn-engine.org/>)

ware and develop firmware for each ECU individually. Instead, they will outsource the tasks to first tier suppliers. The degree of outsourcing will vary depending on the requirements, and a manufacturer may work with any number of suppliers they wish. Manufacturers may request suppliers to design hardware, based on some requirements, and produce the firmware in-house, or the other way around, or may subcontract the whole design of an ECU to one or more suppliers. Similarly, a supplier may work with a number of manufacturers. First tier suppliers may, in turn, outsource some tasks to other companies. The many stakeholders and complexities of the supply chain make tracking of responsibility a difficult task.

In the cases where firmware is outsourced, the manufacturers receive only the firmware image, which they then program to the dedicated hardware. The source code is not shared with the manufacturer, which means no code auditing can take place, and the testing of the component is limited to what the requirements specify or white-box testing.

The research presented in this part arises from the lack of transparency between suppliers and manufacturers and, ultimately, with the vehicle owner. If a supplier may provide services to multiple manufacturers, and multiple manufacturers request firmware for the same component, the core functionality will be similar, with a few differences. Code reuse is a common practice in software development, but it has been shown it can also enable vulnerabilities to seep into software [62, 135, 182]. Finding bugs is not the only task where static analysis is useful. Representing an ECU's firmware through its CFG can also help in identifying the CAN ids an ECU sends its messages on, or which ids it listens to, which is useful when interacting with an ECU in a grey-box manner, without further information (as we will see in Section 7.6.3). Furthermore, differential static analysis can be used to determine the similarity of two firmwares, e.g. what differences are there in the firmware of a Ford Fiesta 2010 BCM, compared to a 2011 one. It can also help determine what parts of the firmware an

upgrade has modified, especially if the update could not be analysed by itself, for example if it was encrypted.

ECUs are very diverse when considering the underlying hardware and architectures. Furthermore, most ECUs have bare-metal firmware, which means they directly interface with low-level hardware and peripherals. A user is required to have strong knowledge of the device's architecture, as tasks such as distinguishing code from data sections or being able to discern between genuine code and wrongly interpreted code is left to a human. Even simply loading the firmware correctly into a disassembler requires creating the memory layout, identifying the entry point(s), mapping of registers for automotive-specific functionality (e.g. CAN communication), and all of this needs to be done manually.

Analysing bare-metal firmware requires more knowledge about the underlying hardware, and documentation is an important resource. Obtaining the user manuals for the automotive-specific chips proved to be a hard task, as they were often not available.

When loading bare-metal firmware into a disassembler, further information needs to be supplied, such as reset vectors. These are considered the *entry points* of the firmware. Nonetheless, IDA will only be able to disassemble by itself minor parts (if any) of the firmware. To achieve maximal code coverage, we propose a number of solutions in Section 6.6. Extracting an accurate CFG from a binary and ensuring correct function detection is an ongoing open problem ([121, 90, 31, 7, 151, 20]), especially with respect to resolving indirect branch instructions. The solutions are architecture specific, some taking advantage of instructions patterns which appear in the function's prologue or epilogue<sup>6</sup>. Some manual guidance and clean-up is still required, but the time needed to bring the disassembled code to a satisfying level is consider-

---

<sup>6</sup>A function prologue refers to a set of instructions that is found at the beginning of the function; they are usually related to setting up the stack for that function. A function epilogue is the equivalent, but is found at the end of a function, and restores the stack to its previous state



ably reduced. By manual guidance and clean-up, we refer to the user going through the disassembly and making *informed decisions* whether the instructions recovered and functions created are correct.

## 6.3 Contribution

Given the constraints discussed in Section 6.2, we wanted to automate as much as possible of the firmware analysis process. We propose a method to abstract from the underlying architecture by extracting the Control Flow Graph (CFG) from the firmware. The CFG is annotated with information that will later be useful, such as start and end addresses of basic blocks and last instruction of basic blocks. The CFG is then used in building a fuzzer, as described in Chapter 7. We describe the techniques we used in order to reduce the time spent manually guiding the disassembler in function and instruction recovery, and extracting the CFG. Where the solutions are specific to a certain architecture, from the ones studied (ARM, PPC and Infineon Tricore), we state so within the sections of this chapter.

## 6.4 Targets

**Firmware acquisition.** We analysed the firmware of three ECUs. We obtained the firmware from the ECUs' flash memory by using automotive programmers [3].

The ECUs tested are show in Figure 6.1:

- Volkswagen Passat IPC — ARM architecture (ARM CDC 3297G-C MCU);
- Ford Fiesta BCM — PPC architecture (SPC560B);
- Ford Kuga ECM — Infineon TriCore architecture (SAK-TC1793F MCU).

It should be noted that ECUs are not limited to these families of MCUs, with many others being available, on other architectures (e.g. Renesas SH-2A, V850E, M16C; NXP

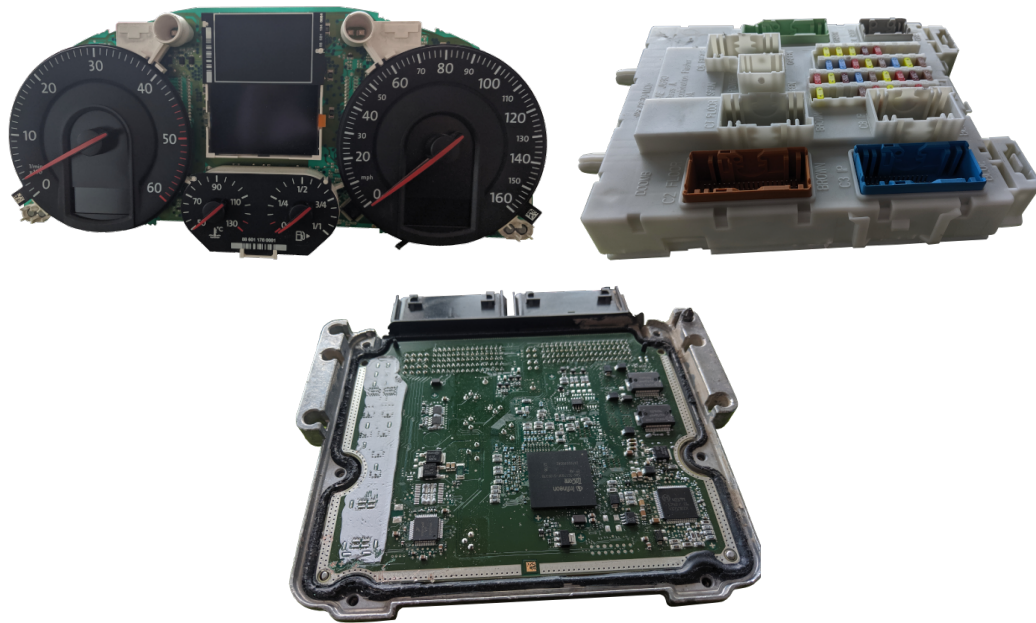


Figure 6.1: Clockwise, the ECUs we have worked with are: VW IPC, Ford BCM, Ford ECM.

HCS12x).

## 6.5 Register Documentation

One of the first issues tackled is IDA's lack of automotive-specific knowledge about the MCUs. For example, for the ARM ECU (ARM CDC 3297G-C MCU), one can choose the ARM processor, and specify the start address and size for the RAM and ROM sections, as well as the offset at which the binary should be loaded. IDA presents the user with a warning that it cannot analyse the binary automatically, and displays the byte values of the firmware. The MCU datasheet provides information of the memory layout, where the RAM, ROM, Input/Output (IO) segments are, etc. This information needs to be manually transferred into IDA, in order to create a correct MCU profile.

In order to help document the registers, we created an IDAPython script. Registers are described in JavaScript Object Notation (JSON) format in a separate file, and

the script will annotate the names, create the appropriate sizes, and comment them. Figure 6.2 shows an excerpt from such a file and Figure 6.3 shows the applied mapping to the IDA database. If the keyword `"payload"` exists, the object is considered a 'group' of registers. The `"base_addr"` keyword contains the address from which the group starts, and each register will contain an `"offset"`, which is added to the base address to obtain the register's address. If the `"payload"` keyword does not exist, the structure is considered one single register. The size of the registers within the group can be specified with the `"reg_size"` property, but it can also be declared for one specific register (e.g. registers in the `"general_ROM"` group have 4 bytes, but the `"ROM_ID"` register has only 2 bytes). The `"code"` keyword means that instead of converting the address to a byte/word/dword, the script will try to create an instruction. Setting the `"include_label"` property to `True` (1) will prepend the `"label"` to the register name. In some cases, objects are mapped to registers, such as in the case of the CAN Communication Objects, for the ARM IPC ECU. Multiple such objects are defined in the CAN-RAM area of the IO segment, and contain CAN messages either received or scheduled to be broadcast. The size of the object can be defined by using the `"size"` keyword, and the number of objects to be mapped can be specified through the `"iterations"` property.

Mapping interesting registers, such as CAN, CAN message buffers, Interrupt Source Nodes (ISNs) or ports, helps improve the readability of the code when manually inspecting it. By documenting them in JSON format, together with the mapping script, the aim is to have a structured and extensible way of dealing with the lack of support and of easily adding the missing information to IDA. The method can be improved by combining the script with a Portable Document Format (PDF) scraper, which could automatically generate the JSON file, as many datasheets have tables which specify the base address, offset and name of each register. We have not tackled creating a PDF scraper, as it is a programming exercise and it was not needed for the work carried out

```

{
    "general_ROM": {
        "label" : "ROM",
        "base_addr" : "0x00",
        "reg_size" : "4",
        "payload": {
            "Reset" : {
                "offset": "0x00",
                "code": "1"
            },
            "SWI": {
                "offset": "0x08",
                "code": "1",
                "comment": "Software Interrupt"
            },
            "ROM_ID": {
                "offset": "0x28",
                "reg_size": "2"
            }
        },
    },
    "can_comm_objects": {
        "label" : "CO",
        "base_addr" : "0xF80000",
        "include_label": "1",
        "iterations" : "5",
        "reg_size" : "1",
        "size": "0x10",
        "payload": {
            "CTRL": "0x00",
            "ID28.21": "0x01",
            "ID20.13": "0x02",
            "ID12.05": "0x03",
            "ID04.00_CTRL": "0x04",
            "DLC_CTRL": "0x05",
            "Data": {
                "offset": "0x06",
                "iterations": "8"
            },
            "timestamp_low": "0x0d",
            "timestamp_high": "0x0e"
        }
    }
}

```

Figure 6.2: Example describing registers as JSON, for the ARM CDC 3297G MCU.

```

IO:F80000 CO0_CTRL           % 1
IO:F80001 CO0_ID28.21        % 1
IO:F80002 CO0_ID20.13        % 1
IO:F80003 CO0_ID12.05        % 1
IO:F80004 CO0_ID04.00_CTRL   % 1
IO:F80005 CO0_DLC_CTRL       % 1
IO:F80006 CO0_Data0          % 1
IO:F80007 CO0_Data1          % 1
IO:F80008 CO0_Data2          % 1
IO:F80009 CO0_Data3          % 1
IO:F8000A CO0_Data4          % 1
IO:F8000B CO0_Data5          % 1
IO:F8000C CO0_Data6          % 1
IO:F8000D CO0_Data7          % 1
IO:F8000E CO0_timestamp_low  % 1
IO:F8000F CO0_timestamp_high % 1

```

Figure 6.3: IDA excerpt, after applying the registers mapping for the first CAN Communication Object. The %1 value for each register denotes the size (1 byte).

in this chapter.

IDA allows additional annotation of registers, interrupts and ports through its own `.cfg` files. However, in the `.cfg` files, each register needs to be defined individually. In the case of the example above, for CAN Communication Objects, each register would have to be defined 5 times (as the number of iterations is 5). Our method is much more flexible, and more compact. It can, in fact, be used to generate IDA `.cfg` files, if so desired, in a much more efficient way.

## 6.6 Disassembling Electronic Control Unit Firmware

As previously mentioned, we used IDA for disassembling the firmware, automated by developing our own IDAPython scripts. We used the `igraph` Python module for storing and working with the CFG, as it allows labelling of vertices and edges, and has good support for efficiently saving the graph to storage.

**A word on the IDA Pro interface.** IDA comes with a multitude of useful functions and options, as one would expect from the state-of-the-art disassembler and program analysis tool. While leveraging the IDAPython API has helped us automate many parts of the process, we found one graphical user interface feature particularly useful. This is the *program navigation bar* (Figure 6.4). The navigation bar displays the entire address space of the program being analysed and colour-codes areas. Therefore, whenever we had to do manual code inspection, or wanted to look for code areas which did not belong to functions, as we will see in the following section, it was very useful to use the navigation bar and look for chunks coloured as *Instructions*. The navigation bar can also be zoomed, and this allows for revealing small areas that require inspection.

As IDA does not recognise the entry point of the firmware binary, guidance needs to be provided. A number of heuristics have been tested, some architecture dependent,



Figure 6.4: IDA navigation bar.

and some which can be applied across architectures.

Firstly, the reset vector is given as entry point. Based on the datasheets, the reset vector is, generally, at the first address in ROM. This will reveal the main loop of the firmware. Other registers can be considered entry points, such as the ISNs, Software Interrupt vector or CAN Interrupt Index Register. This applies to all MCUs in the test set and can be used across architectures. However, using these entry points does not guarantee that the whole program will be covered. Indirect jumps or jump tables may not be recognised by the disassembler, and therefore would need manual intervention. Also, there may be areas of unreachable code, possible leftovers from testing the firmware. The remainder of this section explains the methods tested on each architecture.

### 6.6.1 Disassembling ARM

First of all, it needs to be mentioned that most ARM MCUs support both ARM and Thumb instruction sets. The difference between them is that while ARM instructions are 32 bits, Thumb instructions are 16 bits. All Thumb instructions have corresponding ARM instructions. Thumb code is about 65% of the size of the ARM code, and provides better performance, especially if targeting MCUs with low memory (up to 160% improvement) [101].

With respect to our ARM ECU, once we have loaded the firmware, other architecture options can be set, such as the version of the ARM instruction set, whether the board has a Variable Floating Point co-processor, or it uses the advanced single instruction multiple data extension (mainly used in signal processing for media applications).

All these pieces of information require a strong understanding of the hardware we are working with. The datasheet of the CDC 3297G chip specifies it uses ARM7TDMI MCU, which leads us to believe it uses the ARMv4T instruction set (Thumb enabled)<sup>7</sup>.

One of the heuristics tested was *forcefully creating functions* at each address in the ROM segment of the firmware. As IDA cannot distinguish between `.code` and `.data` areas in the ROM, there will be areas where data is stored, but is wrongly disassembled as code. In order to overcome this, the functions that were successfully created are logged into a file, and a clean database of the firmware is loaded. Code is created only at the addresses which were previously logged.

Next, *instructions patterns* which are used in the *prologue of functions* are identified, and used to recover more code. Thumb instructions make up the majority of the firmware, therefore these instructions are targeted for analysis. Specifically, combinations of the following instructions are sought within the firmware:

- `PUSH` [...] `SUB`
- `PUSH` [...] `BL`
- `PUSH` [...] `LDR`
- `PUSH` [...] `MOVS`

The instructions are represented by 2 bytes (except `BL`) and are identified based on the opcodes they use. The `PUSH` instruction has the opcode `0xb5` (if the Link Register (LR) is also pushed) or `0xb4` (without LR), followed by a byte representing which of the registers `R0–R7` are used. For the other instructions, masks and filters are defined, in order to determine if the instruction following a `PUSH` is indeed one of the previously mentioned ones. Table 6.1 shows the values defined. The filter determines which bits are of interest, and the mask determines what the value of those bits should be. The

---

<sup>7</sup>ARM7TDMI first appears in MCUs from the 1990s, which used the ARMv3 instruction set. However, it also appears in MCUs from the 2000s and later, which use the ARMv4 instruction set. No further information is given in the datasheets, but the brief document for the CDC 32xxG MCU family, detailing their suitability as car dashboard controllers, is dated 2000 (<http://pdf.datasheetcatalog.com/datasheet/MicronasIntermetall/mXvsrxz.pdf>), and therefore it is more likely our component uses ARMv4.

Instruction	(bits)								Filter	Mask
	7	6	5	4	3	2	1	0		
<b>SUB</b>	1	1	1	1	1	0	1	0	✓	
	0	0	0	1	1	0	1	0		✓
<b>LDR</b>	1	1	1	1	1	0	0	0	✓	
	0	1	0	0	1	0	0	0		✓
<b>MOVS</b>	1	1	1	1	1	0	0	0	✓	
	0	0	0	1	1	0	0	0		✓
<b>BL</b>	1	1	1	1	1	0	0	0	✓	
(offset low)	1	1	1	1	1	0	0	0		✓
<b>BL</b>	1	1	1	1	1	0	0	0	✓	
(offset high)	1	1	1	1	0	0	0	0		✓

Table 6.1: Filter and mask for determining Thumb instructions. Together with the **PUSH** instruction, they are used for recovering more code from the target firmware.

branch with link instruction (**BL**) is represented by 4 bytes, first containing the upper 11 bits of the target address, then the 11 bits of the lower half of the address.

The result is a fairly clean version of the firmware with code made at relevant addresses. However, due to the mixed usage of both ARM and Thumb instructions, some parts of the code were created as ARM, when they were in fact Thumb, and thus are not recognised as valid functions. This behaviour is observed due to the misinterpretation between ARM and Thumb, and it still appears, even if the ROM segment is set to Thumb mode, or the even if the segment register Thumb is set before each function creation. Switching between ARM state and Thumb state (and inverse) is done by the MCU when it encounters a **BX** *rA* (or **BLX**) instruction, where *rA* is a register. If the least significant bit of the register *rA* is 0, the MCU changes to (or remains in) ARM state. If the bit is 1, the MCU changes to (or remains in) Thumb state.

In order to deal with this, we wrote a small IDAPython script, which determines the changes in the Thumb segment register, undefines the sections which are ARM code, and re-sets the Thumb register. It then iterates through the whole ROM segment and logs the addresses where the changes from Thumb to ARM still happen, and these can then be used to manually inspect the code around that address. Some parts of the



recovered code will belong to genuine functions, but the end of the functions is set at an earlier address. This is solved by adding the chunks of instructions to the original functions. The code navigation bar helped in identifying end areas for the code being inspected. We found that the script greatly aided the process by targeting specific areas. Using these methods, 60.21% of the firmware was converted into explored addresses<sup>8</sup>. 17.52% of the firmware is a block of `0xFFs`, therefore only 22.27% of the firmware code could not be automatically disassembled and requires further manual inspection.

## 6.6.2 Disassembling PowerPC

For the PPC firmware, we had to make sure the IDA settings are correct. We set the processor as big-endian PPC, select the Signal Processing Engine (SPE) instruction set and enable Variable Length Encoding (VLE). IDA has support for a few PPC devices, and we can select `mpc5xx`, as this is the generic version of the board our ECU uses.

For this ECU we were able to obtain the flash and the RAM contents. Therefore, we have to create the RAM segment, at the correct address, according to the datasheet. We also create the IO segment. From the datasheet we know that the reset vector is located at address `0x04`, and this is the point from which we start our analysis. The vector points to ROM address `0x160`. Once we create code at that address, we can analyse the functions that spawn from there (in our case, 94 new functions were created). We can notice that for the created functions, 66% of them started with the `se_mflr r0` instruction, which moves the contents of the Link Register into register 0. We wrote a script which looks for the opcode of the instruction and operand, `0x00 0x80`. The last nibble of the opcode indicates the register number to be moved in LR (in our case, 0). 5063 new functions are created using this method. Using `MakeFunction(ea)`, from the IDAPython API, we can tell IDA to create a function at a specific address, `ea`. If the function returns false, it means it has failed, and we use `MakeUnkn(ea)`

---

<sup>8</sup>IDA marks an address as explored if it is correctly recognised as code or data.

```
e_lwz      rA, DMR
se_mtctr   rA
se_bctrl
```

where `rA` represents a general register, and `DMR` represents a Direct Memory Reference.

Figure 6.5: Load address to control register pattern and branch to address in control register.

to undefine it. As with the ARM code, some of the functions have the ending set before the function actually ends, leaving some instructions on the outside. Therefore we need to manually adjust these, and the code navigation window comes in handy. Identifying whether the instructions which do not belong to any function should, in fact, be part of the function that exists before them is fairly straightforward, as in this case functions tend to end with `se_blr`, which is the branch to the address contained in the LR instruction (or, in a few other cases, some other form of branch instruction). Otherwise, we mark the set of instructions as their own function, and inspect the code around it. Next, we observe that indirect jumps are used (a control register is used in controlling the flow of the firmware). A set of three instructions is used (Figure 6.5); first, a general register is loaded with an address from the ROM. Then, the control register is loaded with the address the register points to. The flow of the program then branches to the address contained in the control register. Therefore, by extracting the addresses referenced by the Direct Memory Reference (DMR), we can discover more functions of the firmware.

Lastly, the processor options require us to supply the address for the Table of Contents, which is the address of the second general register, and the address of the Small Data Area, which is also the address of the 13th general register. Searching within the firmware for a load instruction into `R2` or `R13` reveals the addresses.

Using this method 52.77% of the ROM was recognised as explored addresses. The firmware contains two large blocks of bytes with the value `0xFF`, and they are located towards the end of the ROM address space. The blocks represent 23.21% of the size

of the firmware. Therefore, 24.02% of the firmware needs to be further manually inspected.

### 6.6.3 Disassembling Infineon TriCore

The Infineon TriCore ECU was the most challenging out of the components we worked with. The documentation for the MCU was sparse, and the Infineon website did not contain the user manual for the TC1793 MCU, and instead linked to the user manual of the TC1798 MCU. The two MCUs are from the same family, though comparing the datasheets of the two revealed a few minor differences, such as the TC1793 having fewer Analog-to-Digital Converters (ADCs) and fewer General Purpose Input/Outputs (GPIOs) lines. Conflicting information was found in datasheets and user manuals with respect to the location of the reset vector, with some specifying the reset vector is at offset  $0 \times 0$  in the flash memory, and others placing the Interrupt Table Vector at that address. Regardless, our firmware had a bank of zeroes as the first 16393 bytes, so neither sources seemed to be correct.

Furthermore, while IDA does have support for TriCore, none of the device profiles match the TC1793/TC1798. Therefore, we start off with a device IDA does know, TC1797, remove all segments which do not match the specification of the TC1793, and create the correct ones. The flash memory of the TC1793 is divided between two address ranges, each of 2 Kbytes, and so we need to split and load the firmware at the appropriate offsets. Once the memory map was correctly recreated and the firmware loaded, we started considering retrieving functions.

Unlike ARM and PPC, TriCore has eliminated the need for function prologues and epilogues, through code optimisation [162]. Therefore, searching for opcodes would not help for this particular architecture. We used the forceful code creation method, combined with manual inspection of code. While more time consuming, this method successfully led to 54% of the total address space being recognised as explored. The

firmware contained three large blocks of byte values 0xC3, amounting to 11.81% of its size. 34.19% of the firmware required further inspection.

## 6.7 Control Flow Graph Extraction

This section describes how we extract the CFG, in order to later use it with the fuzzer.

In order to create an accurate CFG, we use an intermediary object, named a Control Flow List (CFL). The CFL is obtained by calling the `Flowchart()` function provided by the IDAPython API. The function returns a list of `BasicBlock` objects, which contain information about the start and end address of the basic block, as well as two lists, one being the predecessors of the basic block, and one containing the successors.

Once satisfied the CFL is accurate, it can be used to create the graph, using the Python `igraph` module. The start and end address of the basic block are added as vertex attributes.

The vertices will later be labelled with a static data value, if the branch of the basic block is dependent on a comparison with it (more details in Section 7.5). The edges will be labelled with the condition that needs to be met in order to go down that path. Edges that point from a vertex to itself are ignored.

### 6.7.1 CFG Extraction from ARM Firmware

As previously mentioned in Section 6.6.1, calls to Thumb addresses are not correctly handled by IDA. It does not correctly identify a call to a Thumb function as the end of a basic block. Therefore, the CFG would not be an accurate representation of the firmware running on the ECU without amending these. Thumb functions are called using a pattern of three instructions, as represented in Figure 6.6. The value of the DMR determines the address of the Thumb function. These are logged to console, and can be manually inspected, to make sure they are correctly formed.

```
LDR rA, DMR
MOV LR, PC
BX rA
```

where `rA` represents a general register, and `DMR` represents a Direct Memory Reference.

Figure 6.6: Pattern for branch to THUMB function, from ARM code.

Given a CFL created strictly with the results returned by the `Flowchart()` function, along with a list of the previously identified Thumb function calls, we amend the CFL by splitting the basic blocks containing calls to Thumb functions, adding the Thumb function to the successors of the first new basic block, and the second new basic block as a successor to the first one.

### 6.7.2 CFG Extraction from PowerPC Firmware

As discussed in Section 6.6.2, some functions are accessed by being referenced through a DMR. The `Flowchart()` function of the IDAPython API does not recognise these as part of the control flow of the firmware.

Similarly to the procedure used for the ARM firmware, we create the CFL by calling the `Flowchart()` function, then we look for the pattern of instructions presented in Figure 6.5. We extract the address referenced by the DMR, and amend the basic blocks to include the function correctly.

### 6.7.3 CFG Extraction from Infineon TriCore Firmware

The CFG extraction is straightforward for the TriCore firmware, as there are no functions loaded through DMR accesses. The `Flowchart()` function returns the correct CFG, and we only need to create the `Graph` object, based on its information. Figure 6.7 shows the extracted CFG of *one* function, from the ECM ECU (Infineon TriCore). It shows the labelling of the vertices with basic block information, such as start and end addresses and the last instruction of the basic block, as well as static data used in deciding the control flow of the firmware.

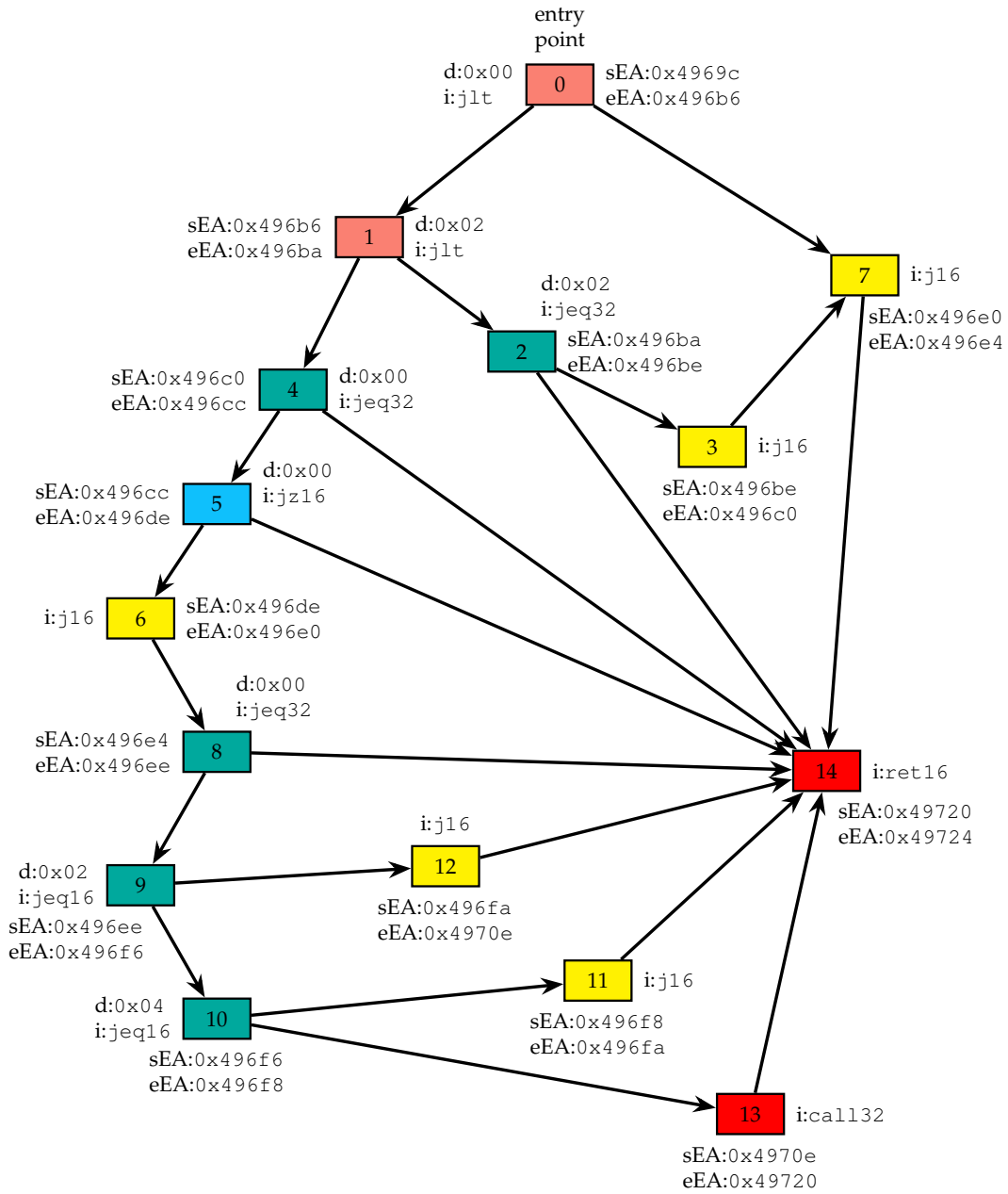


Figure 6.7: Extracted CFG, representing the basic blocks of *one* function. Each basic block was tagged with information about the start address (sEA), end address (eEA), static data used in comparisons (d) and the last instruction of the basic block (i). The function is from the ECM ECU, Infineon TriCore architecture.

## 6.8 Summary

In this chapter we have proposed a method for analysing ECU firmware by extracting the CFG, thus abstracting away from the underlying architecture. The CFG can be annotated with information useful for creating other tools (such as a fuzzer). Due to the large number of architectures ECUs are built on, the sparsity of documentation and the insufficiently documented attempts at ECU firmware analysis, this has been a challenging task, and we believe it presents a high entry barrier for possible researchers having an interest in the field. As there is only one other description of analysing ECU firmware (Miller and Valasek, see Chapter 3), we strongly believe the methodologies presented in this chapter add significant value to the field of automotive research.

## CHAPTER 7

# FUZZING ELECTRONIC CONTROL UNITS

The role of the work described in Chapter 6, is to enable the design of a CAN fuzzer which uses *static data* extracted from the ECU firmware to create CAN messages. This chapter describes how the data is extracted and the design of the fuzzer, together with the evaluation of running the fuzzer on the three devices we mentioned in the previous chapter.

### 7.1 Related Work

Fuzzing is the technique of automating the testing of programs, with the aim of discovering bugs and vulnerabilities. The principle of fuzzing is providing the program under analysis with inputs which may not be valid, or which the program does not expect, and observing the behaviour of the program.

American Fuzzy Lop (AFL) [188] is a state-of-the-art fuzzer, employing several deterministic mutation strategies, such as bit flip, byte flip, arithmetics, and randomised strategies. AFL uses a modified version of QEMU [25], which informs it whether an



input has discovered a new path in the program being tested. AFL has been used as the basis of many other fuzzers in the literature, such as Ifuzzer [172], FairFuzz [100] or Neuzz [150], which shows the high impact AFL has had on advancing the field of fuzz-testing. As AFL requires the fuzzed program to be emulated, or to be fed information about what execution path the given input has triggered within the program, it is not particularly suitable for use with ECUs, as there are no viable emulators for automotive MCUs.

Driller [157] uses a hybrid method of fuzzing and selective concolic execution (port-manteau for symbolic + concrete execution) in order to efficiently discover bugs or vulnerabilities at a deeper level within the evaluated program. Their approach aims to solve the issues each method has on their own: for fuzzing, generating a good set of inputs to drive the program execution, and for concolic execution, the path explosion issue. Driller uses fuzzing to explore initial paths, then relies on concolic execution to overcome *complex checks*. It then goes back to fuzzing the newly discovered paths, until it encounters again a check it cannot resolve, applying concolic execution once more. This cycle is used repeatedly until a vulnerability is discovered or all paths have been explored. Driller's evaluation shows its hybrid approach allows it to perform better than fuzzing or concolic execution used independently.

VUzzer [142] is an *application aware* fuzzer, which aims to create inputs that are more interesting, or relevant, to the program being tested. They use lightweight static and dynamic analysis to extract the program's control flow and data flow features. VUzzer uses control flow information to determine priorities for paths, and data flow information to extract bytes from branch constraints. Therefore, VUzzer can target paths with higher importance, and can determine which input offset enables the program to go via a specific path. The authors argue their approach is more scalable, as compared to Driller, as it does not use symbolic execution. Their evaluation shows that VUzzer performs significantly better, as compared to AFLPIN [163], an AFL-based fuzzer.

Yun *et al.* introduce QSYM [186], a lightweight symbolic execution engine, which aims to solve the performance issues hybrid fuzzers suffer from, increasing their scalability. Their approach removes the usage of intermediate representations used as abstraction layers to the underlying architecture, by other symbolic execution engines. Instead, they focus on a specific, small subset of instructions to be symbolically executed, as opposed to traditional methods which use a basic block approach. The evaluation of the engine shows it outperforms Driller in approximately 82% of the tested binaries.

Muench *et al.* [119] point out in their research that fuzz-testing embedded devices is different from fuzzing desktop systems, due to the limited IO and computing power. They argue that *silent memory corruptions* are much more frequent on embedded devices and they pose a challenge when trying to detect *what* was the exact state of the device when it crashed. They also define six categories for the unexpected behaviour of a device, with various implications for the difficulty of dealing with them: observable crash or reboot (immediate observable effect), hang or late crash (they imply a device could take some time to crash, and therefore the input that triggered it is hard to identify), malfunction (there are no crashes per se, but memory corruption causes the device to not compute some requests correctly) and no effect (no observable effect in the time window immediately following the memory corruption, but unforeseen behaviour might occur at some point in the future). Their arguments are entirely transferable to ECU fuzzing.

### 7.1.1 ECU Fuzzing

A few instances of applying fuzzing to test the security of automotive components and the CAN bus exist, and are presented below.

When fuzzing an ECU, performing an exhaustive search over the whole space would not be feasible, as  $2^{75}$  values (for the CAN 2.0A 11-bit identifier + 64 bits of

payload), would take too long computation-wise. Also, as an ECU typically listens to only a handful of ids, it would mean a lot of wasted resources. Randomly choosing the payload does yield better results, but it cannot reason about the meaning of clusters of bytes, and their potential relations to one other.

Lee *et al.* [98] demonstrated they could sniff the packets from a vehicle, through an OBD-II Bluetooth dongle, and then leverage the knowledge acquired about which CAN ids are in use. They then fuzzed each byte of the 8 bytes of the possible payload, setting the rest to zero. They used the random byte strategy. They observed changes in the IPC signals or in physical components in the vehicle (e.g. lights).

Bayer and Ptok [22] present a UDS fuzzer, which is able to create messages with sequence numbers, as expected by the protocol. They claim the fuzzer is block-based, in that it understands and is aware of what specific fields within the message mean. It can therefore automatically produce correct values for these (e.g. for a checksum field). No other information is given about the strategies the tool uses. The fuzzer was evaluated on an ECU simulator and performed approximately 40,000 tests. It uncovered 6 results which the authors categorise as ‘exploitable’, though no exact definition for such a result is given. A further 695 results were categorised as ‘probably exploitable’ (with 29% of them being reproducible), and they involved no timely negative response from the ECU, which is in contradiction with the expected behaviour, as defined by the standards.

Fowler *et al.* [56] argue that the existing design process should be extended with automated fuzz-testing, informed by the in-vehicle network design (what ECUs are inter-connected, their interfaces and the directions of the data-flow). They then describe in [57] such a fuzzer. Their tool fuzzes CAN packet payloads and ids, and uses the bit flip strategy. The fuzzer can be configured to flip from one single bit in one message to every bit in every message. Therefore, the CAN payloads are randomised. They evaluated the tool against a simulator, an actual ECU (which was an IPC) and

against a vehicle, though with limited scope. The simulation and IPC tests revealed that the fuzzer was able to trigger odd behaviour, such as negative RPM, activating warning lights and sounds, and displaying a *crash* message. The latter behaviour persisted through power cycles of the IPC. In light of this, the authors decided to test the fuzzer only with a small subset of CAN ids on the target vehicle, as to not render it completely useless.

Patki, Gothindikar and Mane [131] present another UDS fuzzer, which uses valid UDS messages and ‘mutates’ the payload in order to create invalid messages, which are then sent to the ECU being tested. They create invalid messages by using invalid values in the DLC field, invalid values for the service sub-functions or invalid inputs in which all bytes are 0x00 or 0xFF. Similarly to Bayer and Ptok’s UDS fuzzer [22], the authors report that some services do not respond according to the specification of the standard.

### 7.1.2 Tools and Frameworks

Caring Caribou [1] is a security exploration tool for the vehicular CAN bus, initially developed as part of the HEAVENS project [2]. The tool has a fuzzer module, which supports random payload generation, brute-forcing, and mutation (randomising nibbles in CAN ids and/or payloads). The fuzzer also allows for replay of previously sent messages and features an *identify* method, which provides supports in determining which message may have triggered a specific behaviour in the ECU being tested. The strategies used by the tool completely rely on randomising CAN messages, even if a user can define *which* part of the message should be subject to said randomisation.

Peach Fuzzer [161] is a commercial fuzzer by Peach Tech<sup>1</sup>, which claims to be able to interface with the CAN bus and can be extended to support any custom protocol. They also claim ‘over 50 algorithms’ for mutation strategies. However, no further in-

---

<sup>1</sup>Peach Tech Website (<https://peach.tech>)

formation on what the strategies are is given for the non-paying audience. Therefore, it is hard to draw any conclusions on whether the commercial version of the tool is suitable for our intended research. A Community Edition is also available, with stripped-down functionality. Sequential and random mutation strategies are included within this version.

## 7.2 Motivation and Challenges

The goal of this research is to create a tool that can automatically test the ECU for bugs, vulnerabilities or hidden functionality, in the absence of the firmware source code. This could expose implementation vulnerabilities, incorrect configurations or backdoors into systems, either inserted maliciously or being the results of forgotten debug functionality.

For the fuzzer, one of the main challenges pertained to defining what it means for an ECU to misbehave or crash. In embedded systems, unexpected or unintended behaviour can take many forms. One of the most common ways of determining crashes is the output the program-under-test gives. But when working with ECUs via CAN communication, the output from the devices is very limited. The CAN bus is a broadcast network, where ECUs have pre-established message ids they use to send their data on, and there is no bi-directional communication (except for UDS). Therefore, there is no real method of establishing the internal state of the ECU, or if any error occurred. Also, obtaining execution traces from a component is a difficult, if not impossible task, as the ECUs have limited debugging support.

Furthermore, when fuzzing CAN communication, the ids the component listens on must be taken into account. In a scenario where nothing is known about the ECU, the ids become part of the search space. Sending a payload on all possible ids significantly increases the time needed for fuzzing. The communication matrix of a vehicle, or of

an ECU, describes which signals are sent and received by which ECU, and on which ids. Such knowledge would be very useful for the fuzzing process, as it would allow us to target only ids the ECU listens to. However, a communication matrix is a well kept industry secret, and therefore not commonly available. One possible solution to reduce the id search space is to sniff CAN communication from a car that uses the same ECU that will be tested. The ids that appear in the trace can then be targeted for use with the fuzzer. Experimentally, this helped reduce the number of ids to test from  $2^{11}$  values to only 50, for a Ford ECU.

### 7.3 Contribution

Continuing on the work from Chapter 6, we present a fuzzer for ECUs that uses the CFG extracted in the previous chapter. The CFG is annotated with more information. We annotate it with the static data that is used in comparisons which influence the control flow of the firmware, and which branch instruction was used. We also compute probability and weight metrics for the edges and vertices of the CFG. The probability of an execution path (edge in the CFG) represents the likelihood that the edge will be taken, given the number of successors its originating basic block has. The weight of a basic block (vertex in the CFG) is used in order to cover execution paths which are hard to reach, and is calculated as the inverse of the weight of a basic block. This information is used in order to compute a set of input seeds for the fuzzer. It is also used in adapting the input messages during the fuzzing process. Using the CFG in creating messages enables us to retain message structure and the relationships between the bytes in a message. We evaluate the fuzzer on three ECUs, with three different architectures, and from different manufacturers. To the best of our knowledge, this work presents the first fuzzer which uses static analysis in guiding the fuzzing of ECUs and does not solely rely on randomness in forming CAN messages.

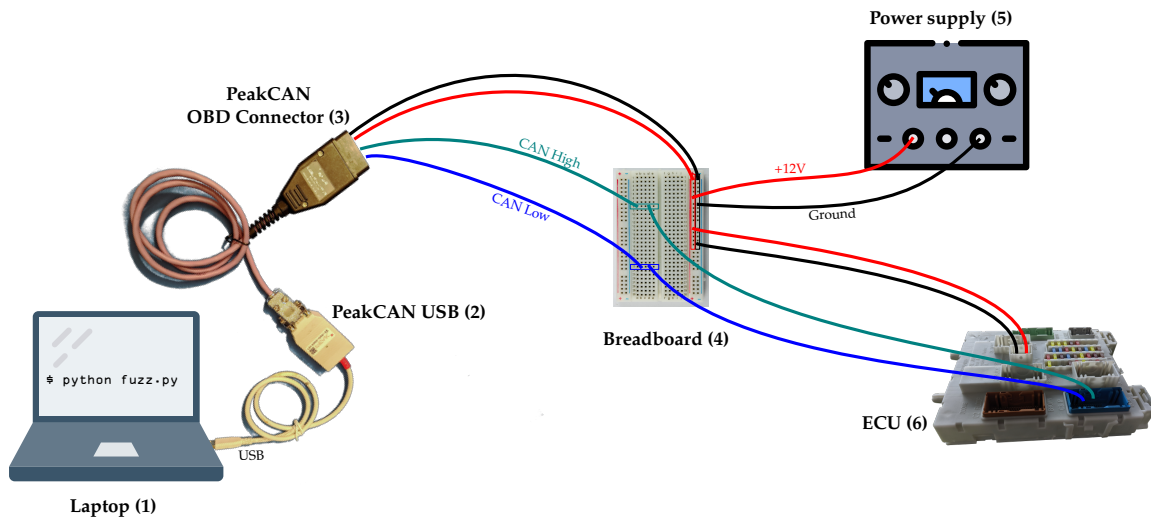


Figure 7.1: Fuzzer hardware setup.

## 7.4 Tools and Setup

The diagram in Figure 7.1 shows the hardware setup required by the fuzzer. In order to communicate with an ECU we used the PeakCAN USB interface (2) together with the OBD-II connector (3), which are connected to the laptop (1) running the fuzzer. We used a breadboard (4) for wire management. The PeakCAN is connected to the target ECU through the CAN pins on the OBD-II connector (3) and on the device under test (6). An external power supply (5) provides the required 12V. The PCAN Python API provides the module `PCANBasic`, which handles the initialisation and configuration of communication channels, as well as transmitting and receiving CAN messages. We continue using the `igraph` module for manipulating and working with the program CFG and we use the `pickle`<sup>2</sup> module for storing the graph to disk.

<sup>2</sup>Python module for serialising and de-serialising object structures ([Python pickle documentation: https://docs.python.org/3/library/pickle.html](https://docs.python.org/3/library/pickle.html)).

ARM Architecture		
<b>CMP</b> rA, IMMED	<b>MOVS</b> rA, IMMED	
<b>BEQ</b> ADDR	<b>CMP</b> rB, rA	
	<b>BGT</b> ADDR	
PPC architecture		
<b>se_cmpli</b> rA, IMMED	<b>e_cmpi</b> cr0, rA, IMMED	<b>e_lis</b> rB, IMMED1
<b>se_bge</b> ADDR	<b>se_bne</b> ADDR	<b>e_addl6i</b> rB, rB, IMMED2
		<b>se_cmpl</b> rA, rB
		<b>se_bne</b> ADDR
Infineon TriCore architecture		
<b>jne32</b> rA, IMMED, ADDR	<b>jz16</b> rA, ADDR	<b>mov16</b> rA, IMMED
		<b>jge.u</b> rA, rB, ADDR

Figure 7.2: Example of instruction patterns for identifying comparisons with static data for the three architectures studied. **IMMED** refers to an immediate value, **rA** and **rB** are general registers and **ADDR** is a ROM address within the firmware.

## 7.5 Data Extraction

The first step towards creating the fuzzer is the data extraction step. Labelling the CFG with the data still requires the firmware disassembly, but after this step, any further work can be done without the need of IDA.

### 7.5.1 Control Flow Graph Tagging

The CFG is labelled with the static data values used in comparisons by looking for instruction patterns. These patterns are architecture-dependent, as shown in Figure 7.2. The data is then set as an attribute for the vertex corresponding to the basic block. The edges of the graph are labelled with the instruction the program control would take in order to go down that path. For this purpose we define an **Antonyms** dictionary (detailed in Appendix A), through which we map the opposite instruction for each branch or jump instruction we encounter (e.g. **BEQ** – **BNE**). The instructions will later be used by the fuzzer in determining what values could a byte take, while still respecting the condition of the branch or jump instruction.



Lastly, the vertices are also labelled with *probability* and *weight* metrics, and the edges with the *probability* metric. Inspired by VUzzer [142] (see Section 7.1), we calculate the probability that a basic block will be reached. For this, we take as starting point a vertex which has an in-degree of 0 and we identify all vertices that can be reached from it, by performing a breath first search. Then we iterate over the search result and calculate the probabilities accordingly. For vertices, the weight is the inverse of the probability. The probability of the edges is dependent on the out-degree of the originating node:

$$p_{e(i,j)} = \frac{1}{out\_deg(i)} \quad (7.1)$$

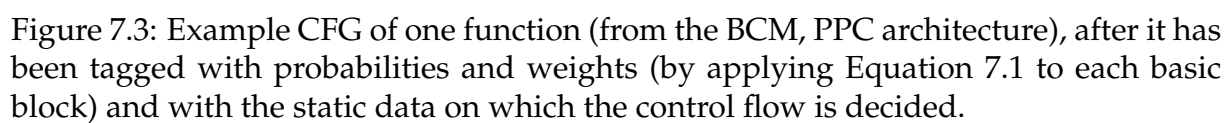
where  $p_{e(i,j)}$  is the probability of the edge between vertices  $i$  and  $j$ , and  $out\_deg(i)$  is the out-degree of vertex  $i$ .

The module `sympy` is used to solve the probabilities of vertices which are part of a cycle. The metrics will be used by the fuzzer to determine which byte of the payload to fuzz. Bytes corresponding to basic blocks with high weights will be given priority. From this step onwards the firmware disassembly is not required, as all the information needed is contained within the CFG. The CFG file can now be saved to storage, in *pickle* format, and will be later given as input to the fuzzer.

Figure 7.3 shows the CFG for *one* function, from the BCM firmware, having its vertices tagged with basic block start address, static data that influences the branch condition, probability and weight, and its edges tagged with the probability and the instruction satisfying the specific paths.

## 7.5.2 Forming Input Seeds for the Fuzzer

In creating the data chains we take advantage of the small payload of the CAN frames, by creating payloads with at most 8 bytes. Algorithm 7.1 describes the process of



extracting the data chains. We iterate through the vertices of the CFG and look for those which have been tagged with data (function `startEA_with_data`). These will be the starting points for the exploration algorithm. For each starting point, we then perform a depth-first search of the CFG, through the function `find_paths` (line 6). We begin at a given vertex `start_vertex`, up to a maximum depth of `max_depth`, and search for as long as comparisons with static data occur in consecutive vertices. The function stores a list of vertex sequences that have been explored in `paths_found`. This leaves us with a list of vertex sequences stored in `vids`. In lines 9–13, we extract the static data associated with those vertices and build the list `data`. Each path in `vids` has a corresponding path in `data`.

```

def explore_cfg(cfg):
    Input: cfg                                * CFG of firmware analysed *
    Output: vids, data

    * vids contains the lists of vertex indexes, *
    * data contains the lists of data payloads *

1   data_startEAs = startEA_with_data(cfg)      * retrieve a list of vertices *
    * where comparisons with static data occurs *
2   vids = list()                             * declare new list *
3   for start_vertex in data_startEAs:
4       explored_vids = list()                 * declare new list *
5       paths_found = list()                  * declare new list *
6       find_paths(cfg, start_vertex, explored_vids, paths_found, max_depth)
7       vids.extend(paths_found)              * add all found paths to list *
8   data = list()                             * declare new list *
    * for each path of vertices found, create a corresponding list of the static data values *
9   for path in vids:
10      path_data = list()                    * declare new list *
11      for vertex in path:
12          path_data.append(cfg.vs[vertex]["data"]) * extract data from CFG *
13          data.append(path_data)            * add the list with the data sequence to the list *
14  return (vids, data)

```

**Algorithm 7.1:** Exploring the CFG, extracting data and vertices chains.

After the chains have been extracted, a list of `Payload` objects is created. Each instance has as attributes the data chain list, the vertex IDs list corresponding to the

data, a `fuzzed` list which will keep track of which bytes have been modified, as well as a probabilities list and a weights list for the basic blocks corresponding to the data. The list can now be saved to storage, in *pickle* format, and given as input to the fuzzer.

## 7.6 Fuzzer Design

The following section is concerned with the design and implementation of the fuzzer. It presents the heuristics for the seed transformation, as well as the options the program has implemented.

### 7.6.1 Fuzzer Prerequisites

The program takes as input the following arguments:

```
> python2 fuzz.py <arch> <suffix> <cfg-file> <queue/payload>  
↪ <USB-device> [resume]
```

where

**<arch>** is the architecture of the target ECU firmware (`arm/ppc/tricore`) – mandatory argument;

**<suffix>** is the suffix of the vertices and data file; it helps distinguish files in the case multiple ECUs with the same architecture have been analysed – mandatory argument;

**<cfg-file>** is the CFG file of the firmware, saved in pickle format – mandatory argument;

**<queue/payload>** is the method of fuzzing; the program can either fuzz one byte for each payload in the queue and wrap around or can fuzz all bytes in each payload, then move on to the next payload; argument can take values `q/p` – mandatory argument;

**<USB-device>** is the number of the PCAN interface, as seen on the host computer – mandatory argument;

**[resume]** signals to the fuzzer it should continue from the last known state – optional argument.

Based on the arguments specified, the program looks for the following files:

**payloads\_<arch>\_<suffix>.pickle:** the file contains the list of `Payload` objects, containing the data chains, vertex IDs, probabilities and weights;

**payloads\_resume\_<arch>\_<suffix>.pickle:** the file contains the list of fuzzed `Payload` objects (see Section 7.6.3); this file is sought only if the `resume` option is enabled; if it does not exist, but the option has been specified, it notifies the user and proceeds as if the option was disabled;

**ids\_<arch>\_<suffix>.pickle:** the file contains the CAN ids the fuzzer will send messages on; if the file does not exist, it will try all possible ids.

## 7.6.2 Input Transformation

For each branch/jump instruction, a function is defined that will choose a random value such that the condition is still respected. This is used in the input transformation.

In Python, a `list` object can also be treated as a queue, with methods of popping the first element and appending to the end. We use the notion of queue below, as to emphasise the first-in-first-out order of the elements and to allow for a language-independent explanation of the algorithms. Python does provide the module `queue`, which is thread-safe, but for our usage this is not necessary.

The fuzzing process is iterative and works as follows. Given a queue of `Payload` objects (Listing 7.1), the first value is popped, and the payload is subject to transformation. If the program is run with the `queue` option, one of the bytes of the payload is chosen to be modified. A new value is chosen, such that the instruction condition is respected. The new message is sent over the CAN interface, and a new `Payload` object,

```
class Payload(object):
    def __init__(self, payload, vids, probability_score, weight):
        self.payload = list(payload)
        self.vids = list(vids)
        self.fuzzed_bytes = [ False for _ in len(self.payload) ]
        self.probability_score = list(probability_score)
        self.weight = list(weight)
```

Listing 7.1: Definition of the `Payload` class, with its member attributes.

with the modified payload, is added to the end of the queue. If the *payload* option is chosen, all the bytes of the payload are subject to transformation, and this whole new value is sent over CAN to the ECU, and added to the queue.

The transformation function operates on one `Payload` object at a time. It looks if there are any bytes which have not yet been fuzzed, then looks at the probabilities of these bytes. It will choose the byte with the lowest probability, and it will choose a random value that still respects the condition of the instruction that defined it. For example, if the instructions sequence was:

```
CMP rA, 0x40
BGT ADDR
```

and we are on a path that takes the branch, the value has to be in the interval  $(0x40, 0xFF]$ . Once the byte has been fuzzed, it is marked accordingly in the `fuzzed_bytes` list. If a new value cannot be chosen, the initial value of the payload is retrieved and the process is repeated<sup>3</sup>. Once all the bytes have been fuzzed, in the *queue* mode, the `fuzzed_bytes` list is reset. For the *payload* mode, the `fuzzed_bytes` list is not used, as all bytes are fuzzed in each iteration.

---

<sup>3</sup>There may be cases in which a byte cannot get another new value, for example if the immediate value of a `BLT` instruction has been fuzzed to `0x0` in a previous iteration, the values interval would be  $[0, 0)$ . In order to avoid this, we retrieve the original value of the payload and operation can resume normally.

### 7.6.3 Additional Features

As the fuzzing process can be a lengthy one, we implemented functionality for the program to be able to *resume* from a previous state. The fuzzer saves the state of the payloads queue to disk after a number of payloads have been fuzzed (in a file `payloads_resume_<arch>_<suffix>.pickle`). This parameter can be chosen as desired, reflecting the trade-off between how recent a state is needed and the time it takes for the payloads file to be written to disk (approx. 160 ms in our tests<sup>4</sup>).

The fuzzer also implements logging of the fuzzed payloads sent on the CAN network in a human-readable format. The program writes a number of files to a `logs-sent` folder, with the name format `log_<arch>_<suffix>_<n>`, where `<n>` is the index of the payload being fuzzed, within the order of the corresponding Payloads *pickle* file. When resuming, the `vids` list of the first payload object in the `payloads_resume_<arch>_<suffix>.pickle` file is used to determine the index value of the payload in the original `payloads_<arch>_<suffix>.pickle` file, therefore assuring the correctness of the logging process. The logs are also useful when encountering an unexpected behaviour. They are used in order to *replay* the messages previously sent, in order to see if the condition can be reproduced.

As mentioned in Section 7.2, knowing which CAN ids an ECU listens to is not a trivial task. The ideal case scenario is having access to a vehicle which has the ECU under test. Then, a trace from the car can be used. The CAN traffic can be sniffed via the OBD-II port, and all ids that are seen on the network can be used as candidates. This reduces the search space considerably. For the BCM we used a trace from a Ford Fiesta, same year as the vehicle our component came from (2015). While there is no guarantee the CAN ids are all the same (even though the two vehicles have the same make, model and year, additional features of the car may be reflected in additional

---

<sup>4</sup>Run on Kaby Lake Intel Core i7-7500U 2.70GHz with Hyper-V enabled, 4 vCPUs.

ECUs or messages), there will be a significant overlap. The trace reduced the number of ids to try from  $2^{11}$  to 50.

Searching within the disassembled firmware may also yield the CAN ids. We experimentally tested the feasibility of this idea, as we had both a trace and the firmware for the BCM. We looked for comparisons with static data with a value lower than  $0 \times 800$ , the maximum a CAN id can take. We also checked for comparisons with bytes in two consecutive basic blocks which, when put together, would give a value lower than  $0 \times 800$ . We retrieved 473 values matching the requirements from the firmware, and 26 of them were also found in the trace from the vehicle. We were unable to verify the correctness of the ids, as we did not possess the communication matrix for the ECU. However, the overlap between the extracted data and the trace ids signals could be a good method for reducing the CAN ids search space (in our case, to 23% of all possible values).

Finally, we provide a `utils.py` file, with various functions we have found useful throughout the development of the fuzzer. The algorithm for computing the probabilities and weights is included, as well as the CFG exploration algorithm. We also provide a function which removes *identical* `Payload` objects. If two paths have the same data and the same branch/jump instructions, we allow for their removal, in order to reduce the number of elements to fuzz. A function for transforming a PeakCAN View<sup>5</sup> trace to the format expected by the `ids_<arch>_<suffix>.pickle` file is provided; this is helpful in the previously mentioned case of access to a vehicle with an ECU of interest. The quickest way to obtain a trace from the car is by using the PCAN View software, which has its own custom format, and we supply a parser for it. Lastly, we include the function for extracting potential CAN ids from the firmware.

---

<sup>5</sup>PCAN-View: Windows Software for Displaying CAN and CAN FD Messages (<https://www.peak-system.com/pcan-view.242.0.html?&l=1>)



## 7.7 Crash Detection

As discussed in Section 7.2, knowing what a *crash* means with respect to an ECU is difficult, due to the limited output and feedback they provide. Without access to the internal state of the ECU, the whole task becomes even more difficult. Furthermore, as ECU firmware logic is mostly driven by interrupts, reproducibility of a result is problematic. This is due to the difficulty of guaranteeing the ECU is in the same state on two different execution runs, and becomes a problem of perfectly tuning the timing. For a large scale approach, this method significantly increases the time required for testing.

Our fuzzer uses two methods of detecting crashes, depending on the information available about the ECU: message *timeout* or the *UDS Tester Present* service. Both methods detect crashes based on the CAN traffic the ECU under test outputs. Before the fuzzing process starts, the program listens for all the messages the ECU sends data on and keeps a record of them. While sending fuzzed payloads, it also listens to incoming traffic. If a message on a new CAN id is recorded, it will flag this up for inspection.

For the timeout method, if the ECU stops sending messages for a specified time (parameter), it considers the ECU to have crashed. For the Tester Present service, the CAN id for UDS needs to be known. After each fuzzed payload is sent to the ECU, a request is made to the Tester Present service (0x3E). If the service responds, regardless of whether it has a positive or negative reply, it means the ECU is still functioning. If there is no response, within a specified time, it considers the ECU to have crashed. The delay in response is also logged, and this can be analysed and potentially provide clues as to whether certain messages introduce a greater delay. This could mean the ECU was in a possibly unexpected state.

## 7.8 Evaluation and Results

The evaluation process consisted of two steps: first, we tested whether the input seeds we extracted from the firmware were relevant to the appropriate ECUs, without any transformation being applied; we then ran the fuzzer on the ECUs. We use the same ECUs mentioned in Section 6.4.

For the first step, after we extracted the byte sequences, we tested the validity of the data chains. As previously mentioned, working with ECUs is tricky, due to the lack of feedback, as CAN communication is unidirectional by design. The best target for testing was the Volkswagen IPC with an ARM chip, as it has a multitude of visual outputs, in the form of gauges and indicators. As we did not know the CAN ids the ECU was programmed to listen to, we sent the data chains extracted from the firmware to all possible ids, without any further modification. We could then monitor the IPC and see if any of the gauge needles moved, if indicators changed status or if the information of the display was modified. The test was successful, as seen in Figure 7.4. The sent CAN messages did indeed trigger the various lights the panel had and moved the speed and the tachometer needles, in rapid succession. Therefore, our initial hypothesis that *data on which the control flow of the firmware is decided is more likely to be meaningful to the ECU than randomly chosen messages* is validated.

We then tested the fuzzer on the three ECUs, with positive results. We reiterate that due to the limited feedback the ECUs provide, the results cannot always be explained, and we can only hypothesise about them. During the tests we have observed unexpected behaviours from the ECUs, where they stopped communicating via the CAN bus. For most situations, rebooting the ECU enabled it to resume normal function. However, for the BCM, on three separate occasions, the ECU did not work as expected after reboot. It required a cool-off time of about 2-4 hours, but we cannot explain the need for it. The ECU PCB does not have any large capacitors, the one possible expla-

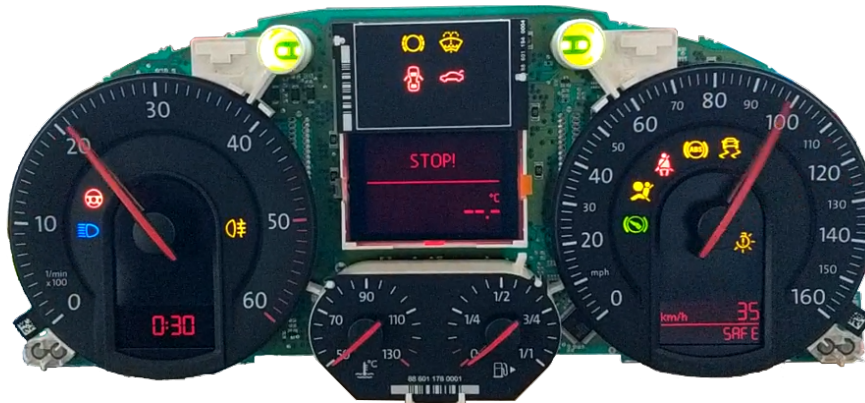


Figure 7.4: Volkswagen IPC in testing.

nation we could consider feasible for the ECU to be able to retain its state for such a long time. Furthermore, the results were not reproducible by replaying the same set of CAN messages to the BCM. This is not highly surprising; as previously discussed, the ECUs are driven by interrupts, therefore making sure it is in the exact same state twice is highly difficult without debug capabilities.

Nonetheless, our results are highly promising. We demonstrate that data extracted from the ECU firmware is more meaningful than randomly chosen data, and that fuzzing with this data as input seeds does indeed lead to crashes and unexpected behaviours. This research lays the ground work for what we hope will be further research into the security of ECUs. As long as there is a lack of transparency between automotive suppliers and manufacturers, and the end users, creating automated frameworks and tools for analysing the firmware of electronic components is crucial.

## 7.9 Summary

We have presented a fuzzer for ECUs, which communicates with a target component via the CAN bus. The fuzzer uses data extracted from the firmware running on the ECU and is guided by the CFG of the firmware in its seed transformation process. We have evaluated the fuzzer on three ECUs, with positive results. To our knowledge, this is the first automotive fuzzer available in the literature that relies on something more complex than randomly choosing the messages to send. Our tests show that indeed this methodology yields better results than the purely random strategy.



---

## PART IV

### *Concluding Remarks*

---



## CHAPTER 8

# DIRECTIONS FOR FUTURE RESEARCH

To conclude this thesis, in this chapter we highlight various problems that are still open questions within the area of vehicle security. Furthermore, we discuss possible improvements to the fuzzer we presented in Chapter 7.

### 8.1 Key Management for Secure CAN Communication

Key management is a critical part of an authentication protocol. Currently, ECUs do not have enough memory to securely store a large set of keys efficiently, making key management in the context of automotive security even more challenging. As described in Chapter 4 and discussed in Chapter 5, most of the proposed CAN authentication protocols in the literature rely on symmetric key cryptography. However, many do not deal with key management in the design of the protocol, simply assuming that the keys were somehow securely loaded onto the ECUs. The few that do approach the subject of key management propose the use of a central node to serve as a *key server* and distribute the necessary keys to the other ECU as needed (e.g. LiBrA-



CAN, MaCAN, CaCAN).

Without an appropriate key management solution, any of the existing protocols are difficult to implement in a vehicle network, and manufacturers have no incentive to adopt an authentication protocol, if it is not a complete solution. Furthermore, the distribution of trust and control among the various stakeholders (vehicle manufacturers, repairs shops and owners) is hard to satisfy. The right to repair legislation in force in the European Union means that owners are not obliged to service or repair their cars at a franchised dealer. This implies that any vehicle repairs shop should be able to replace an ECU, and update the cryptographic material on it, while manufacturers might not feel inclined to share such information. Therefore, research is needed, in partnership with manufacturers, to design a system that would encompass these opposing requirements. A hierarchical key derivation system could be considered, as it would reconcile the requirements for control of the manufacturers. The system would need to allow for quick key revocation, such that if any of the involved parties misbehaved (i.e. keys were leaked), they could be removed from the chain.

## **8.2 The Future of Automotive Networks and Architectures**

Looking at the future trends within the automotive industry, there are two areas which are tightly entwined: in-vehicle network topologies and technologies, and Vehicle-to-Everything (V2X) networks. While we will not discuss advances on the latter, as this thesis is concerned with in-vehicle networks, it is worth mentioning that requirements coming from V2X will influence decisions regarding future in-vehicle technologies.

As vehicles inter-connectivity is increased, and more functionality is delegated to automation software, the communication bandwidth requirement increases. As cars become more autonomous, they require more sensors – radar, lidar, cameras – and

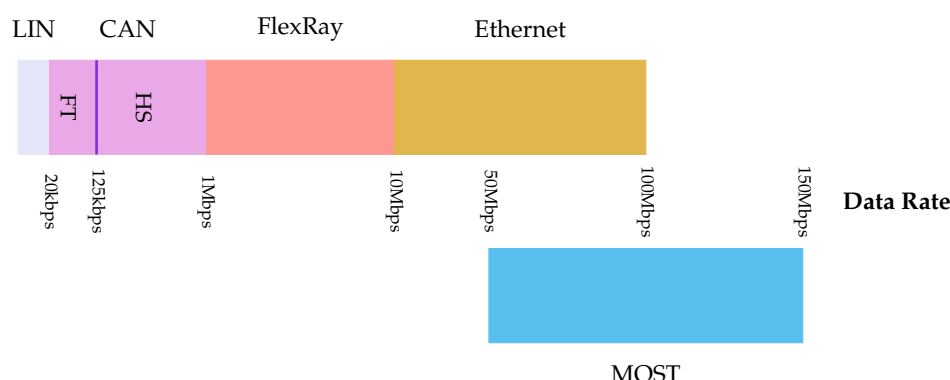


Figure 8.1: In-vehicle network technologies by data rate [129].

faster communication between these components and the ECU processing and taking real-time decisions based on the incoming data. It has long been known that current in-vehicle network technologies are a bottleneck for speed and performance. Figure 8.1 presents an overview of the current network technologies used in vehicles, by their data rate. While MOST is widely used in cars for networking infotainment ECUs, and would provide sufficient speed for the future vehicle architectures, it is a network designed for audio, video and voice data transfer and has high resource demand [109], making it unsuitable for the more resource-constrained ECUs. For these reasons, the industry's interest in *Automotive Ethernet* has increased.

Automotive Ethernet started being used in cars in the early 2000s, when manufacturers began seeking out a way of improving the ECU update process. As networks had more ECUs, and the firmware increased in size, updates became a very time consuming task, which could take up to 16 hours [109]. Standardised in 2011, as ISO 13400 [78], Ethernet is mainly used for Diagnostics over IP, and could soon see wider adoption. The differences between normal Ethernet and automotive Ethernet mainly reside in the physical layer. Automotive Ethernet is able to function as full duplex (simultaneous transmit and receive) on a cable containing a single twisted pair, with improved electromagnetic characteristics, keeping both the manufacturing costs and the overall added weight to the car low. Of interest will be the changes in network

architecture that will come with automotive Ethernet. While CAN is a broadcast network, IP over Ethernet communication will be addressed, therefore improving one of the major flaws of CAN (lack of source identification). Furthermore, while domain (e.g. Body & Gateway, Powertrain & Chassis) separation is currently achieved by physical separation in CAN, Ethernet will allow for virtual configurations, which will be more flexible, and easier to maintain and modify. The trend seems to indicate Ethernet will eventually be used as the “backbone” communication technology in cars, connecting the various domains, with CAN, LIN or FlexRay being used within the domains [61]. But a more imminent use of Ethernet will be for Advanced Driver-Assistance Systems (ADAS), infotainment and camera systems. As Ethernet is an already established technology, solutions for securing it from the IT and IoT areas can be used (virtual networks, firewalls, IDSes).

Besides changes to in-vehicle networks, functional architectural shifts are expected. Currently, each ECU fulfils one specific function. As more powerful ECUs are introduced, it is anticipated that functionality will be aggregated, with one ECU fulfilling multiple tasks. This will reduce the complexity of the systems, and therefore the cost. However, as the application software would still be delivered by various suppliers questions about liability and security arise. Who would be responsible if an ECU fails? Who performs testing of the final, integrated ECU? Who is responsible for protecting supplier IP and how should it be done? If one application contains vulnerabilities, how to prevent a potential attacker to reach other co-located applications? Virtualisation could provide a solution that would ensure application separation, increased availability in the case of ECU failure, separation of critical driving-relevant applications from less critical ones, as well isolation for security-critical functionality [132]. One of the key advantages of virtualisation is the flexibility to host various types of guest OSes or bare-metal applications. Additionally, Trusted Execution Environments (TEEs) are very good candidates for supporting security services (e.g. the AUTOSAR Crypto

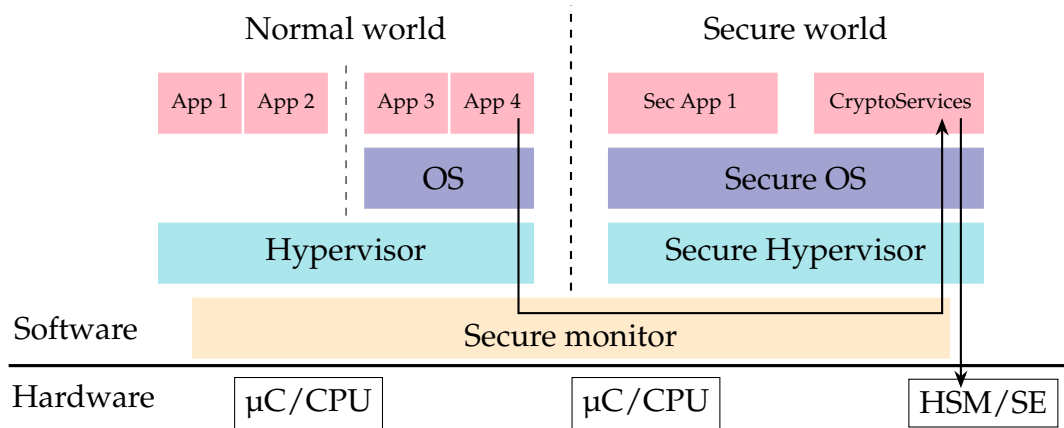


Figure 8.2: Example of virtualised ECU architecture, running a guest OS and bare-metal applications, supported by a TEE.

Stack and SecOC). Briefly, a TEE is an isolated environment (secure world) which runs in parallel to other OSes or applications (normal world) on a processor, and which has confidentiality and integrity guarantees with respect to data or code loaded within it. Moreover, each world has a number of hierarchical execution modes. Figure 8.2 shows how such a system could look, based on the ARM TrustZone TEE [11], which has four execution modes. One important mention is that scheduling becomes of utmost importance for such applications, due to the real-time communication requirements of an automotive system, but also due to the fact that vulnerabilities or bugs in higher privileged layers may result in undesired outcomes (e.g. a call to a Crypto Service residing in the secure-world from a normal world application may not return if there is a bug in the hypervisor, with the application never regaining any execution time on the CPU).

To summarise, the move to Ethernet can be seen as imminent, at least for parts of the in-vehicle communication. It can also be seen as an improvement, both in terms of bandwidth offered and security over the more popular automotive networks. However, Ethernet was not designed with security as one of its core principles, and this should not be disregarded. The technology is sufficiently wide-spread that security solutions from other areas can be ported to Automotive Ethernet. Additionally, the nodes that will be connected through Ethernet are seeing architectural changes as well. While

virtualisation can reduce costs for manufacturers and can increase security by providing isolation mechanisms, the real-time communication requirements of ECUs need to be taken into consideration.




## 8.3 Security in Autonomous Vehicles

As mentioned, the automotive industry is moving towards more advanced automated driving systems. Figure 8.3 depicts the driving automation levels, according to the standard SAE J3016 [146]. The first three levels are defined as *driver support features*, with the human still being the primary driver of the vehicle, their full attention being required. The latter three levels are defined as *automated driving features*, with the automation system performing the driving task, and human intervention being progressively eliminated.

Currently, commercial vehicles offer driver assistance of the Levels 0-2, with level 2 solutions found in Tesla, General Motors, Nissan or Mercedes-Benz vehicles. Audi announced in 2017 plans to bring a Level 3 system, AI Traffic Jam, but have since abandoned their plans due to legislation shortcomings [45]. Vehicles aiming for levels 4 and 5 exist only in prototype form, with the most notable projects being Weymo [178] by Google, Level 5 [107] by Lyft and Advanced Technologies Group [167] by Uber.

Uber's fatal accident in 2018 drew new attention to the safety of autonomous vehicles, even though the incident was deemed to have been a result of the driver's error [97]. With driving decisions being increasingly reliant on data from sensors, such as lidar, radar, GPS and cameras for automated driving systems, securing the input of said sensors becomes of interest. It has already been shown that radar and lidar signals can be jammed, spoofed or relayed [134, 152, 33, 183]. This could result in a vehicle not sensing an obstacle in its vicinity and failing to act in a safe manner. GPS spoofing would affect a vehicle's ability to reach a target destination, effectively "hijacking" the

Levels 0 - 2 : Driver support features – the human performs all or part of the driving task.

		
No automation	Driver assistance	Partial automation
The driver performs the entire driving task. Assistance systems <i>may</i> provide warnings (e.g. lane departure).	Either steering <i>or</i> break support for the driver (e.g. lane centering <i>or</i> cruise control).	ADAS. Both steering <i>and</i> break support for the driver (e.g. lane centering <i>and</i> cruise control).

Levels 3 - 5: Automated driving features – automated driving system performs the driving task when engaged.




		
Conditional automation	High driving automation	Full driving automation
Automated system can perform driving task under limited conditions. Driver required to take over driving when automated system requests (e.g. traffic jam chauffeur).	Automated driving system can perform driving task in most circumstances, without driver intervention (e.g. driverless taxi). Manual override possible if desired.	Automated system can perform driving task in all circumstances and do not require human attention and intervention (i.e. the human is only a <i>passenger</i> ). No controls.

Figure 8.3: Driving automation levels, according to SAE J3016 [146].

car and guiding it to an attacker-desired location [189]. Furthermore, camera sensors are heavily used for object detection and reading road-side signs, and adapting the vehicle driving style to the road restrictions. This is done through computer vision and machine learning algorithms that “see” the sign and then use classifiers to predict its information [54, 155, 137, 26]. However, what happens if the algorithms’ predictions can purposefully be manipulated? Nguyen, Yosinski and Clune [120] were among the first to show it is possible to fool a computer vision system into believing, with high confidence, that e.g. TV static is a motorcycle. Other researchers have also shown that current algorithms can be tricked into wrongly classifying traffic signs by making small modifications to the signs (Figure 8.4) [156, 53].



Speed limit 120 sign which was classified as speed limit 30 in [156], 85% attack success rate.



Stop sign which was classified as speed limit 80 in [53] with 80% success rate.

Figure 8.4: Example of modified traffic signs which are wrongly classified.

To summarise, considerable research needs to investigate how to prevent sensors inputs from being tampered with, and how to build more adversary-resilient machine learning algorithms, before we can *safely* enjoy the comfort of a level 5 autonomous driving vehicle.

## 8.4 Automotive Fuzzing

As we discussed in Chapter 7, the field of automotive fuzzing is still in its infancy. While inspiration can be drawn from fuzzing embedded devices, ECU firmware comes with its own challenges. From a high-level view, the tasks of creating a robust automotive fuzzing system can be broken into three main areas: firmware analysis, firmware execution and feedback loop. We discuss below each of these areas, exploring how they can be improved.

**Firmware analysis** still has a number of open research questions, such as control flow recovery, function identification and data structure recovery, the former two upon which we touch in this thesis and are pertinent to embedded bare-metal firmware. While control flow recovery is a feature of some existing tools (including IDA), the solutions are often architecture-dependent, and custom heuristics need to be hand-

written for non-trivial control flow transfer. Handling indirect function calls and indirect branch instructions are the main challenge for this, especially in cases where control flow is transferred via function pointers or the destination address may be dynamically computed [48]. Function pointers complicate function detection as well, therefore a heuristic cannot only rely on explicit function calls. Tail calls also need to be considered, as the functions called in this manner might be mistakenly included in the parent function [48]. Lastly, data structure recovery in the context of ECU firmware could have as prime use the identification of parsers, such as the one for parsing CAN messages. Improvements in these areas will allow for a more accurate CFG recovery, having a better representation of the firmware analysed, and therefore improving any heuristics inferred from said analysis.

**Firmware execution** Evidently, executing the firmware we are trying to test is a central part of a fuzzer. In our solution, we chose a hardware-in-the-loop system, but this is not a highly scalable option, as it requires additional hardware to parallelise the procedure. Firmware emulation would be able to overcome this problem. However, emulating ECUs, with their architectural diversity and their many peripheral configurations is a challenge. Existing tools for full emulation support only a limited set of architectures and would require significant engineering commitments to bring ECU firmware support to them, an effort which academic views seem to deem not worthwhile. Partial emulation could cover some of the pitfalls of full emulation. For example, the core can be emulated using an existent tool, and a model for the peripherals can be created from actual executions on the device under test, if the firmware was written using hardware abstraction libraries [36]. Frankenstein [143] solves the issue of unknown memory map of embedded devices by emulating the firmware and delegating any memory map functionality back to the device, such that it is retrieved straight from an execution run. While both these solutions would improve our system,



their main limitation remains that they support a limited number of architectures.

**Feedback loop** The feedback loop mechanism is a crucial part of a fuzzer, as this is the way the fuzzer knows whether the input provided to the firmware under testing has triggered new behaviour. For our solution, we used physically observable behaviours or gaps in the communication with the device. However, the loop can be improved by identifying the root cause of the events and, as we mentioned earlier, by establishing what the state of the device is when a crash occurs, which would also improve reproducibility. Using instrumentation in order to track the specific execution path taken when provided with an input would also increase the performance of the fuzzer, as this would allow for better control flow coverage, and enable a more efficient exploration of the firmware functionality. Alas, this method would currently suffer from pollution from the additional paths that time-based interrupts would introduce, and a method of distinguishing them within the trace would need to be developed. Lastly, an interesting idea to be explored is using side-channels (e.g. electromagnetic measurements, power analysis) for the feedback loop. This could be done in a coarse manner, in the sense of distinguishing if two executions follow the same control flow, or finer-grained, profiling specific parts of the code and detecting whether that code was executed.

To summarise, in order to create an efficient, accurate and scalable fuzzer for ECUs, significant engineering efforts need to be invested in the area of reverse engineering and emulating automotive embedded devices.

## CHAPTER 9

# CONCLUSION

In this thesis we have studied the security of the in-vehicle network, focusing on the Controller Area Network, as it allows ECUs to communicate with each other, and the security of the ECU firmware.

We have explored CAN bus authentication, both by presenting a new design, and by evaluating and reviewing other protocols available in the literature. Our protocol, LEIA, provides uni-directional source authentication for CAN messages, protecting against attacks that rely on the ability of an adversary to replay messages or impersonate other ECUs. LEIA was designed to be AUTOSAR compliant, and backwards compatible with the CAN standard. It is a software-only solution, and therefore vehicles can benefit from it by a simple firmware upgrade. It has flexibility at its core, and allows for authenticated and unauthenticated messages to co-exist and not disturb each other's communication. This means the upgrade to a secure CAN bus can be done gradually, softening the costs for a manufacturer. From our evaluation, LEIA emerges as the best authentication protocol option, and should be considered for adoption within in-vehicle networks.

After researching the security of the in-vehicle network communication, we turned our attention to the components connected to it. Due to the complexity of the ECUs supply chain, manufacturers often do not have the ability to review the code of the firmware that runs on their ECUs. Furthermore, ECU firmware has not been extensively researched. The diversity of hardware and architectures used in building the ECUs is a contributing factor to the lack of research. In order to bridge this gap, we presented an automated tool, a fuzzer, which can be used to test ECUs without the need for a large amount of knowledge about the device. The tool abstracts from the underlying ECU architecture by extracting the Control Flow Graph of the firmware and augmenting it with information which is then used to adapt the input messages.

The tool currently relies only on static analysis and there are a number of possible directions for further research. Dynamic analysis would greatly improve the accuracy of the fuzzer. Information about the execution path a specific input has triggered could be incorporated into the input transformation process of the tool. This could be achieved by running the firmware in an emulator; however, there are no mature projects for automotive emulators. Existing emulators could be extended, by adding automotive MCU profiles, but this would be a tremendous task, due to the large number of MCUs used in ECUs, as well as the the difficulty of managing the many peripherals ECUs have. Writing an emulator involves both knowledge about the architecture of the device, as well as expert understanding of the hardware of the device.

From the attack-related literature we discussed in this thesis, to the many different solutions for securing the in-vehicle network, it is clear the academic world has been invested in improving the security of cars. However, it is now up to the manufactures to implement these solutions and usher in the era of *cyber secure vehicles*. This is important, as consumers deserve to be protected against cyber attackers, especially due to the grave implications successful breaches have been shown to have.

---

## Appendices

---



APPENDIX A

ANTONYMS

Antonyms for ARM architecture	Antonyms for PowerPC architecture
<pre>ANTONYMS = { \     "BLE" : "BGT", \     "BGT" : "BLE", \     "BEQ" : "BNE", \     "BNE" : "BEQ", \     "BGE" : "BLT", \     "BLT" : "BGE", \     "BPL" : "BMI", \     "BMI" : "BPL", \     "BCC" : "BCS", \     "BCS" : "BCC", \     "BHI" : "BLS", \     "BLS" : "BHI", \     "BL" : "", \     "B" : "", \ }</pre>	<pre>ANTONYMS = {     "e_b" : "", \     "e_beq" : "e_bne", \     "e_bge" : "e_blt", \     "e_bgt" : "e_ble", \     "e_ble" : "e_bgt", \     "e_blt" : "e_bge", \     "e_bne" : "e_beq", \     "se_b" : "", \     "se_beq" : "se_bne", \     "se_bge" : "se_blt", \     "se_bgt" : "se_ble", \     "se_ble" : "se_bgt", \     "se_blt" : "se_bge", \     "se_bne" : "se_beq" }</pre>

## Antonyms for Infineon TriCore architecture

```

ANTONYMS = {
    "j16"      : "", \
    "j32"      : "", \
    "ja"       : "", \
    "jeq.a"    : "jne.a", \
    "jeq16"    : "jne16", \
    "jeq32"    : "jne32", \
    "jge"      : "jlt", \
    "jge.u"    : "jlt.u", \
    "jgez16"   : "jltz26", \
    "jgtz16"   : "jlez16", \
    "ji16"     : "", \
    "jlez16"   : "jgtz16", \
    "jlt"      : "jge", \
    "jlt.u"    : "jge.u", \
    "jltz16"   : "jgez16", \
    "jne.a"    : "jeq.a", \
    "jne16"    : "jeq16", \
    "jne32"    : "jeq32", \
    "jned"     : "", \
    "jnz16"    : "jz16", \
    "jnz16.a"  : "jz16.a", \
    "jnz16.t"  : "jz16.t", \
    "jnz32.a"  : "jz32.a", \
    "jnz32.t"  : "jz32.t", \
    "jz16"     : "jnz16", \
    "jz16.a"   : "jnz16.a", \
    "jz16.t"   : "jnz16.t", \
    "jz32.a"   : "jnz32.a", \
    "jz32.t"   : "jnz32.t"
}

```

---

## List of References

---





- [1] (NO DATE) Caring caribou. URL <https://github.com/CaringCaribou/caringcaribou> [Accessed 25/09/2019].
- [2] (NO DATE) HEAVENS: HEALing Vulnerabilities to ENhance Software Security and Safety. URL [http://www.sp.se/en/index/research/dependable\\_systems/heavens/Sidor/default.aspx](http://www.sp.se/en/index/research/dependable_systems/heavens/Sidor/default.aspx) [Accessed 25/09/2019].
- [3] (NO DATE) Tools for automotive repairing. URL <http://www.usprog.ru/index.php/en/news/usp.html> [Accessed 27/07/2020].
- [4] (2005) Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication. URL <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38B.pdf> [Accessed 20/07/2019].
- [5] Al-Jarrah, O.Y., Maple, C., Dianati, M. et al. (2019) Intrusion detection systems for intra-vehicle networks: A review. **IEEE Access**, 7: 21266–21289
- [6] Andriesse, D., Chen, X., Van Der Veen, V. et al. (2016) “An in-depth analysis of disassembly on full-scale x86/x64 binaries.” In **25th USENIX Security Symposium (USENIX Security 16)**. pp. 583–600
- [7] Andriesse, D., Slowinska, A. and Bos, H. (2017) “Compiler-agnostic function detection in binaries.” In **2017 IEEE European Symposium on Security and Privacy (EuroS&P)**. IEEE. pp. 177–189
- [8] Apvrille, L., Muhammad, W., Ameer-Boulifa, R. et al. (2006) “A uml-based environment for system design space exploration.” In **2006 13th IEEE International Conference on Electronics, Circuits and Systems**. IEEE. pp. 1272–1275
- [9] ARINC (2006) ARINC specification 812: Definition of standard data interfaces for galley insert (GAIN) equipment, CAN communications. Standard, Aeronautical Radio, Incorporated (ARINC).
- [10] ARINC (2007) ARINC specification 825: General standardization of CAN (controller area network) bus protocol for airborne use. Standard, Aeronautical Radio, Incorporated (ARINC).
- [11] ARM (NO DATE) Arm truszone. URL <https://developer.arm.com/ip-products/security-ip/trustzone> [Accessed 27/07/2020].
- [12] AUTOSAR (2015) AUTOSAR classic platform specification 4.2. URL <http://www.autosar.org/specifications/release-42/> [Accessed 24/04/2017].
- [13] AUTOSAR (2019a) AUTOSAR classic platform specification 4.4. URL <https://www.autosar.org/standards/classic-platform/> [Accessed 30/07/2019].

- [14] AUTOSAR (2019b) Requirements on crypto stack. URL [https://www.autosar.org/fileadmin/Releases\\_TEMP/Classic\\_Platform\\_4.4.0/Crypto.zip](https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/Crypto.zip) [Accessed 30/07/2019].
- [15] AUTOSAR (2019c) Specification of crypto interface. URL [https://www.autosar.org/fileadmin/Releases\\_TEMP/Classic\\_Platform\\_4.4.0/Crypto.zip](https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/Crypto.zip) [Accessed 30/07/2019].
- [16] AUTOSAR (2019d) Specification of crypto service manager. URL [https://www.autosar.org/fileadmin/Releases\\_TEMP/Classic\\_Platform\\_4.4.0/Crypto.zip](https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/Crypto.zip) [Accessed 30/07/2019].
- [17] AUTOSAR (2019e) Specification of key manager. URL [https://www.autosar.org/fileadmin/Releases\\_TEMP/Classic\\_Platform\\_4.4.0/Crypto.zip](https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/Crypto.zip) [Accessed 30/07/2019].
- [18] AUTOSAR (2019f) Specification of secure onboard communication. URL [https://www.autosar.org/fileadmin/Releases\\_TEMP/Classic\\_Platform\\_4.4.0/Communication.zip](https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/Communication.zip) [Accessed 30/07/2019].
- [19] Avatefipour, O. and Malik, H. (2018) State-of-the-art survey on in-vehicle network communication (can-bus) security and vulnerabilities. **arXiv preprint arXiv:1802.01725**.
- [20] Bao, T., Burket, J., Woo, M. et al. (2014) "BYTEWEIGHT: Learning to recognize functions in binary code." In **23rd USENIX Security Symposium (USENIX Security 14)**. pp. 845–860
- [21] Basin, D., Cremers, C., Dreier, J. et al. (2013) Tamarin prover. URL <https://tamarin-prover.github.io/> [Accessed 27/07/2020].
- [22] Bayer, S. and Ptok, A. (NO DATE) Don't fuss about fuzzing: Fuzzing controllers in vehicular networks.
- [23] BBC (2015) Fiat chrysler recalls 1.4 million cars after jeep hack. URL <https://www.bbc.co.uk/news/technology-33650491> [Accessed 12/08/2019].
- [24] Bella, G., Biondi, P., Costantino, G. et al. (2019) "Toucan: A protocol to secure controller area network." In **Proceedings of the ACM Workshop on Automotive Cybersecurity**. ACM. pp. 3–8
- [25] Bellard, F. (2009) QEMU the FAST! processor emulator. URL <https://www.qemu.org/> [Accessed 05/08/2019].
- [26] Berkaya, S.K., Gunduz, H., Ozsen, O. et al. (2016) On circular traffic sign detection and recognition. **Expert Systems with Applications**, 48: 67–75

- [27] Bogdanov, A. (2007) "Linear slide attacks on the Keeloq block cipher." In **Information Security and Cryptology, Third SKLOIS Conference, Inscrypt 2007, Xining, China, August 31 - September 5, 2007, Revised Selected Papers**. pp. 66–80
- [28] Bozdal, M., Samie, M. and Jennions, I. (2018) "A survey on can bus protocol: Attacks, challenges, and potential solutions." In **2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)**. IEEE. pp. 201–205
- [29] Brumley, D., Jager, I., Avgerinos, T. et al. (2011) "Bap: A binary analysis platform." In **International Conference on Computer Aided Verification**. Springer. pp. 463–469
- [30] Bruni, A., Sojka, M., Nielson, F. et al. (2014) "Formal security analysis of the MaCAN protocol." In **Integrated Formal Methods**. Springer. pp. 241–255
- [31] Bruschi, D., Martignoni, L. and Monga, M. (2006) "Detecting self-mutating malware using control-flow graph matching." In **International conference on detection of intrusions and malware, and vulnerability assessment**. Springer. pp. 129–143
- [32] Cervin, A., Henriksson, D., Lincoln, B. et al. (2003) Analysis and simulation of controller timing. **IEEE Control Systems Magazine**, 23 (3): 16–30
- [33] Chauhan, R. (2014) A platform for false data injection in frequency modulated continuous wave radar.
- [34] Checkoway, S., McCoy, D., Kantor, B. et al. (2011) "Comprehensive experimental analyses of automotive attack surfaces." In **20th USENIX Security Symposium**. San Francisco.
- [35] CiA (2011) CANopen application layer and communication profile. Standard, CAN in Automation.
- [36] Clements, A.A., Gustafson, E., Scharnowski, T. et al. (2020) "Halucinator: Firmware re-hosting through abstraction layer emulation." In **29th USENIX Security Symposium (USENIX Sec)**. pp. 1–18
- [37] Committee, S.V.E.S.S. et al. (2016) Sae j3061-cybersecurity guidebook for cyber-physical automotive systems. **SAE-Society of Automotive Engineers**.
- [38] Costin, A., Zaddach, J., Francillon, A. et al. (2014) "A large-scale analysis of the security of embedded firmwares." In **23rd USENIX Security Symposium (USENIX Security 14)**. pp. 95–110
- [39] Costin, A., Zarras, A. and Francillon, A. (2016) "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces." In **Proceedings of**

- the 11th ACM on Asia Conference on Computer and Communications Security.** ACM. pp. 437–448
- [40] Courtois, N., Bard, G.V. and Wagner, D. (2008) “Algebraic and slide attacks on Keeloq.” *In Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers.* pp. 97–115
  - [41] CrypticApps (NO DATE) Hopper v4, the macos and linux disassembler. URL <https://www.hopperapp.com/> [Accessed 19/09/2019].
  - [42] Daily, J. (NO DATEa) Dodge can messages. URL <http://tucrrc.utulsa.edu/DodgeCAN.html> [Accessed 01/10/2019].
  - [43] Daily, J. (NO DATEb) Interpreting the can data for a 2010 toyota camry. URL <http://tucrrc.utulsa.edu/ToyotaCAN.html> [Accessed 01/10/2019].
  - [44] Dariz, L., Costantino, G., Ruggeri, M. et al. (2018) A joint safety and security analysis of message protection for can bus protocol. **Advances in Science, Technology and Engineering Systems Journal**, 3 (1).
  - [45] Davies, C. (2020) Audi abandons self-driving plans for current flagship. URL <https://www.slashgear.com/audi-a8-traffic-jam-pilot-level-3-cancelled-tech-self-driving-legislation/> [Accessed 27/07/2020].
  - [46] den Herrewegen, J.V. and Garcia, F.D. (2018) “Beneath the bonnet: A breakdown of diagnostic security.” *In 23rd European Symposium on Research in Computer Security (ESORICS 2018), Proceedings, Part I.* Springer. Lecture Notes in Computer Science, vol. 11098, pp. 305–324. URL <https://doi.org/10.1007/978-3-319-99073-6>.
  - [47] Developers, V. (2017) Valgrind supported architectures. URL <http://www.valgrind.org/info/platforms.html> [Accessed 23/07/2019].
  - [48] Di Federico, A., Payer, M. and Agosta, G. (2017) “rev. ng: a unified binary analysis framework to recover cfgs and function boundaries.” *In Proceedings of the 26th International Conference on Compiler Construction.* pp. 131–141
  - [49] Di Natale, M., Zeng, H., Giusto, P. et al. (2012) **Understanding and using the controller area network communication protocol: theory and practice.** Springer Science & Business Media.
  - [50] Dolev, D. and Yao, A.C. (1983) On the security of public key protocols. **Information Theory, IEEE Transactions on**, 29 (2): 198–208
  - [51] Elektrobit (NO DATE) Elektrobit and autosar. URL <https://www.elektrobit.com/products/ecu/technologies/autosar/> [Accessed 20/09/2019].

- [52] ESOL (NO DATE) Aubist classic platform. URL [https://www.esol.com/embedded/aubist\\_cp.html](https://www.esol.com/embedded/aubist_cp.html) [Accessed 20/09/2019].
- [53] Eykholt, K., Evtimov, I., Fernandes, E. et al. (2018) "Robust physical-world attacks on deep learning visual classification." **In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. pp. 1625–1634
- [54] Farag, W. and Saleh, Z. (2018) Traffic signs identification by deep learning for autonomous driving.
- [55] Farag, W.A. (2017) "Cantrack: Enhancing automotive can bus security using intuitive encryption algorithms." **In 2017 7th International Conference on Modeling, Simulation, and Applied Optimization (ICMSAO)**. IEEE. pp. 1–5
- [56] Fowler, D.S., Bryans, J. and Shaikh, S. (2017) Automating fuzz test generation to improve the security of the controller area network.
- [57] Fowler, D.S., Bryans, J., Shaikh, S.A. et al. (2018) "Fuzz testing for automotive cyber-security." **In 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)**. IEEE. pp. 239–246
- [58] Garcia, F.D., Oswald, D., Kasper, T. et al. (2016) "Lock it and still lose it - on the (in)security of automotive remote keyless entry systems." **In 25nd USENIX Security Symposium (USENIX Security 2016), to appear**. USENIX Association.
- [59] Greenberg, A. (2015) Hackers remotely kill a jeep on the highway – with me in it. URL <http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/>.
- [60] Groza, B., Murvay, S., Van Herrewege, A. et al. (2012) "LiBrA-CAN: A lightweight broadcast authentication protocol for Controller Area Networks." **In Cryptology and Network Security**. Springer. pp. 185–200
- [61] Hank, P., Müller, S., Vermesan, O. et al. (2013) "Automotive ethernet: In-vehicle networking and smart mobility." **In 2013 Design, Automation Test in Europe Conference Exhibition (DATE)**. pp. 1735–1739
- [62] Hanna, S., Huang, L., Wu, E. et al. (2012) "Juxtapp: A scalable system for detecting code reuse among android applications." **In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment**. Springer. pp. 62–81
- [63] Hartkopp, O., Reuber, C. and Schilling, R. (2012) "MaCAN - Message authenticated CAN." **In 10th Int. Conf. on Embedded Security in Cars (ESCAR 2012)**.
- [64] Hazem, A. and Fahmy, H.A. (2012) "LCAP - A lightweight CAN authentication protocol for securing in-vehicle networks." **In 10th escar Embedded Security in Cars Conference, Berlin, Germany**. vol. 6.

- [65] He, L., Ren, X., Gao, Q. et al. (2017) The connected-component labeling problem: A review of state-of-the-art algorithms. **Pattern Recognition**, 70: 25–43
- [66] Hex-Rays (NO DATE) About ida. URL <https://www.hex-rays.com/products/ida/> [Accessed 19/09/2019].
- [67] Hicks, C., Garcia, F.D. and Oswald, D. (2018) Dismantling the aut64 automotive cipher. **IACR Transactions on Cryptographic Hardware and Embedded Systems**, pp. 46–69
- [68] High Speed, C. (1999) for vehicle applications at 500kbps. **SAE J2284**.
- [69] Hoppe, T., Kiltz, S. and Dittmann, J. (2008) “Security threats to automotive can networks—practical examples and selected short-term countermeasures.” In **International Conference on Computer Safety, Reliability, and Security**. Springer. pp. 235–248
- [70] Hoppe, T., Kiltz, S. and Dittmann, J. (2011) Security threats to automotive can networks—practical examples and selected short-term countermeasures. **Reliability Engineering & System Safety**, 96 (1): 11–25
- [71] IEC (2008) 61162-3:2008 Maritime navigation and radiocommunication equipment and systems – Digital interfaces – Part 3: Serial data instrument network. Standard, International Electrotechnical Commission.
- [72] IEC (2014) 62026-3:2014 low-voltage switchgear and controlgear – controller-device interfaces (CDIs) – part 3: DeviceNet. Standard, International Electrotechnical Commission.
- [73] IEC/ISO (1994) ISO/IEC 7498-1: 1994 information technology – open systems interconnection—basic reference model: The basic model. Standard, Recommendation, ITUTX.
- [74] Indesteege, S., Keller, N., Dunkelman, O. et al. (2008) “A practical attack on Keeloq.” In **Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings**. pp. 1–18
- [75] ISO (2003) 11898-1: 2003 - Road vehicles - Controller Area Network. **International Organization for Standardization, Geneva, Switzerland**.
- [76] ISO (2008) 26021: 2008 – Road Vehicles – End-of-life activation of on-board pyrotechnic devices. Standard, International Organization for Standardization.
- [77] ISO (2010) 15031: 2010 – Road Vehicles – Communication between vehicle and external equipment for emissions-related diagnostics. Standard, International Organization for Standardization.



- [78] ISO (2011) Road vehicles. diagnostic communication over internet protocol (DoIP). Standard, International Organization for Standardization.
- [79] ISO (2012) 27145: 2012 – Road Vehicles – Implementation of World-Wide Harmonized On-Board Diagnostics (WWH-OBD) communication requirements. Standard, International Organization for Standardization.
- [80] ISO (2013a) 14229-1: 2013 – Road Vehicles – Unified diagnostic services (UDS) – Part 1: Specifications and requirements. Standard, International Organization for Standardization.
- [81] ISO (2013b) 14229-2: 2013 – Road Vehicles – Unified diagnostic services (UDS) – Part 2: Session layer services. Standard, International Organization for Standardization.
- [82] ISO (2013c) 14229: 2013 – Road Vehicles – Unified diagnostic services (UDS). Standard, International Organization for Standardization.
- [83] ISO (2013d) 14229: 2013 – Road Vehicles – Unified diagnostic services (UDS) – Part 3: Unified diagnostic services on CAN implementation (UDSonCAN). Standard, International Organization for Standardization.
- [84] ISO (2015) 11898-1: 2015 – Road Vehicles – Controller Area Network – Part 1: Data link layer and physical signalling. Standard, International Organization for Standardization.
- [85] ISO (2016) 11898-1: 2016 – Road Vehicles – Diagnostic communication over Controller Area Network (DoCAN). Standard, International Organization for Standardization.
- [86] ISO (2019) ISO/SAE CD 21434: Road Vehicles – Cybersecurity engineering. Standard, International Organization for Standardization.
- [87] Johnson, R. and Christie, S. (2009) JTAG 101 IEEE 1149.x and software debug. URL <http://intel.com/content/dam/www/public/us/en/documents/white-papers/jtag-101-ieee-1149x-paper.pdf> [Accessed 05/08/2019].
- [88] Jukl, M. and Čupera, J. (2016) Using of tiny encryption algorithm in can-bus communication. **Research in Agricultural Engineering**, 62 (2): 50–55
- [89] Kasper, M., Kasper, T., Moradi, A. et al. (2009) “Breaking Keeloq in a flash: On extracting keys at lightning speed.” **In Progress in Cryptology – AFRICACRYPT 2009: Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 403–420



- [90] Kinder, J. and Veith, H. (2008) "Jakstab: A static analysis platform for binaries." *In International Conference on Computer Aided Verification*. Springer. pp. 423–427
- [91] Kleberger, P., Olovsson, T. and Jonsson, E. (2011) "Security aspects of the in-vehicle network in the connected car." *In 2011 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. pp. 528–533
- [92] Knorreck, D., Apvrille, L. and Pacalet, R. (2009) "Fast simulation techniques for design space exploration." *In International Conference on Objects, Components, Models and Patterns*. Springer. pp. 308–327
- [93] Koscher, K., Czeskis, A., Roesner, F. et al. (2010) "Experimental security analysis of a modern automobile." *In Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. pp. 447–462
- [94] Kurachi, R., Matsubara, Y., Takada, H. et al. (2014) "CaCAN – Centralised authentication system in CAN (Controller Area Network)." *In 12th Int. Conf. on Embedded Security in Cars (ESCAR 2014)*.
- [95] Larson, U.E. and Nilsson, D.K. (2008) "Securing vehicles against cyber attacks." *In Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*. ACM. p. 30
- [96] Lawrenz, W. (1997) CAN system engineering. *From theory to practical applications*, New York.
- [97] Lee, D. (NO DATE) Uber self-driving crash 'mostly caused by human error'. URL <https://www.bbc.co.uk/news/technology-50484172> [Accessed 27/07/2020].
- [98] Lee, H., Choi, K., Chung, K. et al. (2015) "Fuzzing can packets into automobiles." *In 2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. IEEE. pp. 817–821
- [99] Lee, H., Jeong, S.H. and Kim, H.K. (2017) "Otidis: A novel intrusion detection system for in-vehicle network by using remote frame." *In 2017 15th Annual Conference on Privacy, Security and Trust (PST)*. vol. 00, pp. 57–5709. URL [doi.ieeecomputersociety.org/10.1109/PST.2017.00017](https://doi.ieeecomputersociety.org/10.1109/PST.2017.00017).
- [100] Lemieux, C. and Sen, K. (2018) "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage." *In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM. pp. 475–485
- [101] Limited, A. (2004) Arm7tdmi technical reference manual.

- [102] Limited, A. (2018) ARM debug interface architecture specification ADIv5.0 to ADIv5.2. URL [https://silver.arm.com/download/ARM\\_and\\_AMBA\\_Architecture/AR551-DA-70001-r0p0-01rell/debug\\_interface\\_v5\\_2\\_architecture\\_specification\\_IHI0031E.pdf](https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR551-DA-70001-r0p0-01rell/debug_interface_v5_2_architecture_specification_IHI0031E.pdf) [Accessed 05/08/2019].
- [103] Lin, C.W. and Sangiovanni-Vincentelli, A. (2012) "Cyber-security for the controller area network (can) communication protocol." In **Cyber Security (Cyber-Security), 2012 International Conference on**. pp. 1–7
- [104] Lin, C.W., Zhu, Q., Phung, C. et al. (2013) "Security-aware mapping for can-based real-time distributed automotive systems." In **2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. pp. 115–121
- [105] Lin, C.W., Zhu, Q. and Sangiovanni-Vincentelli, A. (2015) Security-aware modeling and efficient mapping for can-based real-time distributed automotive systems. **IEEE Embedded Systems Letters**, 7 (1): 11–14
- [106] Ltd, K.T. (NO DATE) K-sar – autosar suite. URL [https://www.mathworks.com/products/connections/product\\_detail/k-sar-autosar-suite.html](https://www.mathworks.com/products/connections/product_detail/k-sar-autosar-suite.html) [Accessed 20/09/2019].
- [107] Lyft (NO DATE) Level 5: Imagining the future and then building it. URL <https://self-driving.lyft.com/level5/> [Accessed 27/07/2020].
- [108] Mansor, H., Markantonakis, K. and Mayes, K. (2014) "Can bus risk analysis revisit." In **IFIP International Workshop on Information Security Theory and Practice**. Springer. pp. 170–179
- [109] Matheus, K. and Königseder, T. (2017) **Automotive ethernet**. Cambridge University Press.
- [110] Microchip (NO DATE) Microchip and autosar. URL <https://www.microchip.com/design-centers/automotive-solutions/software/autosar> [Accessed 20/09/2019].
- [111] Microelectronics, S. (NO DATE) Stsw-spc56as003. URL <https://www.st.com/en/embedded-software/stsw-spc56as003.html> [Accessed 20/09/2019].
- [112] Milburn, A., Timmers, N., Wiersma, N. et al. (2018) There will be glitches: Extracting and analyzing automotive firmware efficiently. URL [https://www.riscure.com/uploads/2018/11/Riscure\\_Whitepaper\\_Analyzing\\_Automotive\\_Firmware.pdf](https://www.riscure.com/uploads/2018/11/Riscure_Whitepaper_Analyzing_Automotive_Firmware.pdf) [Accessed 12/02/2019].
- [113] Miller, C. and Valasek, C. (2013) Adventures in automotive networks and control units. **DEF CON**.
- [114] Miller, C. and Valasek, C. (2014) A survey of remote automotive attack surfaces. **Black Hat USA**, 2014: 94

- [115] Miller, C. and Valasek, C. (2015a) Remote exploitation of an unaltered passenger vehicle. URL <http://illmatix.com/Remote%20Car%20Hacking.pdf>.
- [116] Miller, C. and Valasek, C. (2015b) A survey of remote automotive attack surfaces.
- [117] Monfared, M.S., Noori, H. and Abazari, M.A. (2019) "Design space exploration of the aes encryption algorithm implementation for securing can protocol." **In 2019 9th International Conference on Computer and Knowledge Engineering (ICCCKE)**. IEEE. pp. 380–385
- [118] Mouha, N., Mennink, B., Van Herrewege, A. et al. (2014) "Chaskey: an efficient mac algorithm for 32-bit microcontrollers." **In International Conference on Selected Areas in Cryptography**. Springer. pp. 306–323
- [119] Muench, M., Stijohann, J., Kargl, F. et al. (2018) "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." **In NDSS**.
- [120] Nguyen, A., Yosinski, J. and Clune, J. (2015) "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images." **In Proceedings of the IEEE conference on computer vision and pattern recognition**. pp. 427–436
- [121] Nguyen, M.H., Nguyen, T.B., Quan, T.T. et al. (2013) "A hybrid approach for control flow graph construction from binary code." **In 2013 20th Asia-Pacific Software Engineering Conference (APSEC)**. IEEE. vol. 2, pp. 159–164
- [122] Nielson, F., Nielson, H.R. and Hankin, C. (2015) **Principles of program analysis**. Springer.
- [123] NIST, S. (2019) Transitioning the use of cryptographic algorithms and key lengths.
- [124] Noorman, J., Bulck, J.V., Mühlberg, J.T. et al. (2017) Sancus 2.0: A low-cost security architecture for iot devices. **ACM Transactions on Privacy and Security (TOPS)**, 20 (3): 7
- [125] Noureldeen, P., Azer, M.A., Refaat, A. et al. (2017) "Replay attack on lightweight can authentication protocol." **In 2017 12th International Conference on Computer Engineering and Systems (ICCES)**. pp. 600–606
- [126] NSA (2019) Ghidra, sre suite of tools. URL <https://ghidra-sre.org/> [Accessed 19/09/2019].
- [127] Nürnberger, S. and Rossow, C. (2016) "vatiCAN–vetted, authenticated can bus." **In International Conference on Cryptographic Hardware and Embedded Systems**. Springer. pp. 106–124

- [128] NXP (NO DATE) Autosar-4-3: Autosar 4.3.x (classic platform) software. URL <https://www.nxp.com/design/automotive-software-and-tools/autosar-/autosar-4.3.x-classic-platform-software:AUTOSAR-4-3> [Accessed 20/09/2019].
- [129] NXP (2011) Introduction in-vehicle-networking – NXP elearning content. URL <https://www.youtube.com/watch?v=DeQb8Q6hEkA> [Accessed 27/07/2020].
- [130] pancake (NO DATE) The unix-friendly framework for reverse engineering. URL <https://rada.re/> [Accessed 19/09/2019].
- [131] Patki, P., Gotkhindikar, A. and Mane, S. (2018) “Intelligent fuzz testing framework for finding hidden vulnerabilities in automotive environment.” In **2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)**. IEEE. pp. 1–4
- [132] Pelzl, J., Wolf, M. and Wollinger, T. (2008) Virtualization technologies for cars. **Tech. Rep.**
- [133] Perrig, A., Canetti, R., Tygar, J.D. et al. (2000) “Efficient authentication and signing of multicast streams over lossy channels.” In **Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000**. IEEE. pp. 56–73
- [134] Petit, J., Stottelaar, B., Feiri, M. et al. (2015) Remote attacks on automated vehicles sensors: Experiments on camera and lidar. **Black Hat Europe**, 11: 2015
- [135] Pham, N.H., Nguyen, T.T., Nguyen, H.A. et al. (2010) “Detection of recurring software vulnerabilities.” In **Proceedings of the IEEE/ACM international conference on Automated software engineering**. ACM. pp. 447–456
- [136] Popov, A. (2015) Rfc 7465 – prohibiting rc4 cipher suites. URL <https://tools.ietf.org/html/rfc7465>.
- [137] Prabhakar, G., Kailath, B., Natarajan, S. et al. (2017) “Obstacle detection and classification using deep learning for tracking in high-speed autonomous driving.” In **2017 IEEE Region 10 Symposium (TENSYP)**. IEEE. pp. 1–6
- [138] Preneel, B. (1997) “Cryptanalysis of message authentication codes.” In **International Workshop on Information Security**. Springer. pp. 55–65
- [139] Quynh, N.A. (NO DATE) Capstone, the ultimate disassembler. URL <https://www.capstone-engine.org/> [Accessed 19/09/2019].
- [140] Radu, A.I. and Garcia, F.D. (2016) “LeiA: A lightweight authentication protocol for CAN.” In **21st European Symposium on Research in Computer Security (ESORICS 2016)**. Springer-Verlag. Lecture Notes in Computer Science, vol. 9879, pp. 283–300

- [141] Radu, A.I. and Garcia, F.D. (2020) "Grey-box analysis and fuzzing of automotive electronic components via control-flow graph extraction." **In The 4th ACM Computer Science in Cars Symposium (CSCS 2020)**. ACM.
- [142] Rawat, S., Jain, V., Kumar, A. et al. (2017) "VUzzer: Application-aware evolutionary fuzzing." **In NDSS**. vol. 17, pp. 1–14
- [143] Ruge, J., Classen, J., Gringoli, F. et al. (2020) "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets." **In 29th USENIX Security Symposium (USENIX Security 20)**. pp. 19–36
- [144] Ryan, M., Arapinis, M. and Ritter, E. (2013) Statverif: Verification of stateful processes. URL <https://markryan.eu/research/statverif/> [Accessed 27/07/2020].
- [145] SAE (1994) Class c application requirements. **SAE Handbook**, 2: 23.366 – 23.272
- [146] SAE, J. (NO DATE) 3016: 2014 taxonomy and definitions for terms related to on-road motor vehicle automated driving systems. **Society of Automotive Engineers**.
- [147] Schwarz, B., Debray, S. and Andrews, G. (2002) "Disassembly of executable code revisited." **In Ninth Working Conference on Reverse Engineering, 2002. Proceedings**. IEEE. pp. 45–54
- [148] Schweppe, H., Idrees, S., Roudier, Y. et al. (2008) Evita: Secure on-board protocols specification.
- [149] Schweppe, H., Roudier, Y., Weyl, B. et al. (2011) "Car2x communication: securing the last meter-a cost-effective approach for ensuring trust in car2x applications using in-vehicle symmetric cryptography." **In 2011 IEEE Vehicular Technology Conference (VTC Fall)**. IEEE. pp. 1–5
- [150] She, D., Pei, K., Epstein, D. et al. (2018) Neuzz: Efficient fuzzing with neural program learning. **arXiv preprint arXiv:1807.05620**.
- [151] Shin, E.C.R., Song, D. and Moazzezi, R. (2015) "Recognizing functions in binaries with neural networks." **In 24th USENIX Security Symposium (USENIX Security 15)**. pp. 611–626
- [152] Shin, H., Kim, D., Kwon, Y. et al. (2017) "Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications." **In International Conference on Cryptographic Hardware and Embedded Systems**. Springer. pp. 445–467
- [153] Shoshitaishvili, Y., Wang, R., Hauser, C. et al. (2015) "Firmallice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware." **In Proceedings of the 2015 Network and Distributed System Security Symposium**.

- [154] Shoshitaishvili, Y., Wang, R., Salls, C. et al. (2016) "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis." *In IEEE Symposium on Security and Privacy*.
- [155] Shustanov, A. and Yakimov, P. (2017) Cnn design for real-time traffic sign recognition. *Procedia engineering*, 201: 718–725
- [156] Sitawarin, C., Bhagoji, A.N., Mosenia, A. et al. (2018) Darts: Deceiving autonomous cars with toxic signs. *arXiv preprint arXiv:1802.06430*.
- [157] Stephens, N., Grosen, J., Salls, C. et al. (2016) "Driller: Augmenting fuzzing through selective symbolic execution." *In NDSS*. vol. 16, pp. 1–16
- [158] Studnia, I., Nicomette, V., Alata, E. et al. (2013) "Survey on security threats and protection mechanisms in embedded automotive networks." *In Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*. IEEE. pp. 1–12
- [159] Team, A.W. (2018) AUTOSAR introduction. URL [https://www.autosar.org/fileadmin/ABOUT/AUTOSAR\\_Introduction.pdf](https://www.autosar.org/fileadmin/ABOUT/AUTOSAR_Introduction.pdf) [Accessed 28/07/2019].
- [160] Team, T.R. (NO DATE) Radare2 and capstone. URL <http://radare.today/posts/radare2-capstone/> [Accessed 20/09/2019].
- [161] Tech, P. (NO DATE) Peach fuzzer. URL <https://www.peach.tech/products/peach-fuzzer/> [Accessed 25/09/2019].
- [162] Technologies, I. (2003) Tricore compiler writer's guide. URL [https://www.infineon.com/dgdl/inf0010\\_v1\\_4Dec2003\\_1.pdf?fileId=db3a304412b407950112b40f8aad1423](https://www.infineon.com/dgdl/inf0010_v1_4Dec2003_1.pdf?fileId=db3a304412b407950112b40f8aad1423) [Accessed 25/07/2019].
- [163] Thompson, P. (NO DATE) Aflpin. URL <https://github.com/moثرan/aflpin> [Accessed 21/09/2019].
- [164] Tindell, K. and Burns, A. (1994) "Guaranteeing message latencies on control area network (can)." *In Proceedings of the 1st International CAN Conference*. Citeseer.
- [165] Tindell, K., Burns, A. and Wellings, A.J. (1995) Calculating controller area network (can) message response times. *Control Engineering Practice*, 3 (8): 1163–1169
- [166] Tindell, K., Hanssmon, H. and Wellings, A.J. (1994) "Analysing real-time communications: Controller area network (can)." *In RTSS*. Citeseer. pp. 259–263
- [167] Uber (NO DATE) Self-driving car technology. URL <https://www.uber.com/us/en/atg/technology/> [Accessed 27/07/2020].



- [168] Van Bulck, J., Mühlberg, T. and Piessens, F. (2017) "Vulcan: Efficient component authentication and software isolation for automotive control networks." In Proceedings of the 33th Annual Computer Security Applications Conference (ACSAC 2017). ACM.
- [169] Van Herrewege, A., Singelee, D. and Verbauwhede, I. (2011) "CANAuth - A simple, backward compatible broadcast authentication protocol for CAN bus." In ECRYPT Workshop on Lightweight Cryptography 2011.
- [170] Vaudenay, S. (2007) "On privacy models for RFID." In Advances in Cryptology-ASIACRYPT 2007. Springer. pp. 68–87
- [171] Vector35 (NO DATE) Binary ninja: a new king of reversing platform. URL <https://binary.ninja/> [Accessed 19/09/2019].
- [172] Veggalam, S., Rawat, S., Haller, I. et al. (2016) "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming." In European Symposium on Research in Computer Security. Springer. pp. 581–601
- [173] Verdult, R. and Garcia, F.D. (2015) "Cryptanalysis of the Megamos Crypto automotive immobilizer." In USENIX ;login:. USENIX Association. vol. 40/6, pp. 17–22
- [174] Verdult, R., Garcia, F.D. and Balasch, J. (2012) "Gone in 360 seconds: Hijacking with Hitag2." In 21st USENIX Security Symposium (USENIX Security 2012). pp. 237–252
- [175] Verdult, R., Garcia, F.D. and Ege, B. (2015) "Dismantling Megamos Crypto: Wirelessly lockpicking a vehicle immobilizer." In 22nd USENIX Security Symposium (USENIX Security 2013). USENIX Association. pp. 703–718
- [176] Viswanathan, T. and Bhatnagar, M. (2015) **Telecommunication switching systems and networks**. PHI Learning Pvt. Ltd.
- [177] Wang, Q. and Sawhney, S. (2014) "Vecure: A practical security framework to protect the can bus of vehicles." In Internet of Things (IOT), 2014 International Conference on the. pp. 13–18
- [178] Waymo (NO DATE) Waymo: the world's most experienced driver. URL <https://waymo.com/> [Accessed 27/07/2020].
- [179] Wolf, M., Weimerskirch, A. and Paar, C. (2004) "Security in automotive bus systems." In Workshop on Embedded Security in Cars.
- [180] Wolf, M., Weimerskirch, A. and Wollinger, T. (2007) State of the art: Embedding security in vehicles. **EURASIP Journal on Embedded Systems**, 2007 (1): 074706

- [181] Woo, S., Jo, H.J. and Lee, D.H. (2014) A practical wireless attack on the connected car and security protocol for in-vehicle can. **IEEE Transactions on intelligent transportation systems**, 16 (2): 993–1006
- [182] Xia, P., Matsushita, M., Yoshida, N. et al. (2014) Studying reuse of out-dated third-party code in open source projects. **Information and Media Technologies**, 9 (2): 155–161
- [183] Yeh, E., Choi, J., Prelcic, N. et al. (2016) Security in automotive radar and vehicular networks. **submitted to Microwave Journal**.
- [184] Yomsi, P.M., Bertrand, D., Navet, N. et al. (2012) “Controller area network (can): Response time analysis with offsets.” **In 2012 9th IEEE International Workshop on Factory Communication Systems**. IEEE. pp. 43–52
- [185] Young, R. (2015) Hackers cut a corvette’s brakes via a common car gadget. URL <https://www.wired.com/2015/08/hackers-cut-corvettes-brakes-via-common-car-gadget/>.
- [186] Yun, I., Lee, S., Xu, M. et al. (2018) “QSYM: A practical concolic execution engine tailored for hybrid fuzzing.” **In 27th USENIX Security Symposium (USENIX Security 18)**. pp. 745–761
- [187] Zaddach, J., Bruno, L., Francillon, A. et al. (2014) “Avatar: A framework to support dynamic security analysis of embedded systems’ firmwares.” **In NDSS**. pp. 1–16
- [188] Zalewski, M. (NO DATE) American fuzzy lop. URL <http://lcamtuf.coredump.cx/afl> [Accessed 21/09/2019].
- [189] Zeng, K.C., Liu, S., Shu, Y. et al. (2018) “All your GPS are belong to us: Towards stealthy manipulation of road navigation systems.” **In 27th USENIX Security Symposium (USENIX Security 18)**. pp. 1527–1544
- [190] Ziermann, T., Wildermann, S. and Teich, J. (2009) “CAN+: A new backward-compatible Controller Area Network (CAN) protocol with up to 16x higher data rates.” **In Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE’09**. IEEE. pp. 1088–1093