

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



A semi-automated approach to UDS implementation testing

Rodrigo Araújo Castro

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Rui Pedro Ferreira Pinto

July 31, 2023

Abstract

There was a time when vehicles were considered mainly mechanical machines. Now they are closer to being a piece of technology. Every modern vehicle in the market boasts numerous Electronic Control Units (ECUs), that are responsible for the most critical of functions (braking, direction and even safety features). This dependency in software raises the need for diagnostic functions that allow leveraging that software coverage. Unified Diagnostic Services (UDS) was created in order to standardize diagnostic communications among different brands in the automotive industry. UDS plays a critical role in vehicles, by allowing communication between diagnostic testers and ECUs. As with any communication protocol, UDS needs to be implemented and, consequently, have that implementation tested. Normal methods for testing involve manually exchanging messages with an ECU and verifying the responses with a tool such as Vector CANoe. This proves to be expensive in terms of time, reliability and money. In order to make testing faster and more reliable, the work developed aimed to build a tool that takes a semi-automatic approach to testing an UDS stack implementation. The objective was to create an option that poses a faster, more reliable and less error-prone alternative when compared to manual testing. This dissertation was developed in Continental Engineering Services (CES), the company that made the work possible by providing the necessary resources and help for technical development.

The proposed tool was developed using Rust, a programming language that gives the programmer a control similar to C/C++ while offering memory safety by using a borrow-checker system (among other features). The hardware consisted simply of a Raspberry Pi 4 equipped with a Controller Area Network (CAN) interface, making the setup independent from third-party tools. In order to describe the tests, the user creates a JavaScript Object Notation (JSON) file and uses a syntax designed to that effect. The syntax's design was thought to make it as flexible as possible since, although UDS is standardized, its implementation can vary a lot according to client preferences and requirements. After describing the tests, the file is fed to the proposed tool which will execute the tests by exchanging messages with an ECU in the CAN bus and output the results.

The proposed tool was tested with different ECU configurations in order to validate its behavior, which proved to be as expected. A timed comparison was also conducted, with the help of three volunteers from CES' cybersecurity team. The comparison involved executing the same test routine using the proposed tool/syntax vs manually using Vector CANoe. Results showed an average reduction of 15.3% in the time needed to execute the tests and a reduction of 80% in errors during testing.

The work developed represented a solid alternative to manual testing and fulfilled the objectives set for this work, by allowing faster, clearer and more reliable testing on UDS implementation, although work can be done to make the syntax even clearer and easier to use.

Resumo

Outrora, os veículos eram máquinas maioritariamente mecânicas. Hoje em dia, estão mais perto de serem peças de tecnologia. Todos os veículos modernos no mercado possuem numerosas *Electronic Control Units* (ECUs), que são responsáveis pelas mais críticas das funções (travagem, direção e até segurança). Esta dependência em *software* acaba por motivar uma necessidade por funções de diagnóstico que permitam aproveitar essa cobertura. O *Unified Diagnostics Services* (UDS), foi um protocolo criado com o objetivo de padronizar a comunicação de diagnóstico entre as diferentes marcas da indústria automóvel. UDS tem um papel crítico nos veículos, possibilitando a comunicação entre testadores e ECUs. Tal como qualquer outro protocolo de comunicação, o UDS tem de ser implementado e, conseqüentemente, ter essa implementação testada. Uma forma comum de o fazer seria manualmente trocar mensagens com o ECU e verificar as respostas recebidas, numa ferramenta como o Vector CANoe. Isto, no entanto, provou ser caro em termos de tempo, confiabilidade e dinheiro. De forma a tornar a testagem mais rápida e mais confiável, o trabalho desenvolvido nesta dissertação pretende criar uma ferramenta que utiliza uma abordagem semi-automática para testagem da implementação do protocolo UDS. Esta dissertação foi desenvolvida na Continental Engineering Services (CES), a empresa que possibilitou o trabalho desenvolvido, tendo facultado os recursos e ajuda necessários para o desenvolvimento técnico.

A ferramenta foi desenvolvida com Rust, uma linguagem de programação conhecida por dar ao programador um controlo similar ao de C/C++, enquanto oferece garantias de segurança de memória ao utilizar um sistema *borrow-checker* (entre outras vantagens). O *hardware* consistiu num simples Raspberry Pi equipado com uma interface *Controller Area Network* (CAN), que torna o sistema independente de *hardware* externo. Para descrever os testes, o utilizador pode fazê-lo num ficheiro *JavaScript Object Notation* (JSON), utilizando uma sintaxe desenhada para esse efeito. A sintaxe foi desenhada de forma a maximizar a flexibilidade da mesma porque, embora o protocolo UDS seja padronizado, a sua implementação pode sofrer uma grande variabilidade consoante as preferências do cliente. Após a definição dos testes, o utilizador usa a ferramenta proposta para interpretar o ficheiro e executar os testes descritos. A ferramenta irá depois trocar mensagens CAN com o ECU, de forma a verificar se os testes foram bem sucedidos e apresentando o resultado.

A ferramenta proposta foi testada num ECU com diferentes configurações de forma a validar o seu comportamento, que foi o esperado. Uma comparação cronometrada também foi conduzida, com a ajuda de três voluntários da equipa de cibersegurança da CES. Esta comparação envolveu executar uma rotina de testes utilizando a ferramenta proposta vs manualmente no Vector CANoe. Os resultados mostraram uma redução, em média, de 15.3% no tempo necessário para executar os testes, bem como uma redução de 80% nos erros cometidos durante a testagem.

O trabalho desenvolvido representa uma alternativa sólida relativamente à testagem manual, tendo cumprido os objetivos definidos por permitir uma testagem mais rápida, mais clara e mais confiável a uma implementação dos UDS. Trabalho pode ser feito para tornar a sintaxe ainda mais clara e fácil de utilizar.

Acknowledgements

First and foremost, I would like to extend my gratitude to my parents, who have always been by my side. For providing me with as good a life as anyone could ask for and for shaping me into the person I am today. Thank you to my brother who, together with my parents, had to sit through my rants during the development of this dissertation. And a thank you to my uncles, cousin and grandpa, for accompanying me in my life.

Next, a heartfelt thank you to "Toca do Menino", the great friends I met during these crazy 5 years of my life and that made the university an adventure I'll never forget. "Pessoal, isto é o nosso..." and they know the rest. Thanks goes out to my girlfriend, who endured negligence during the last days of this project, and always seems to have spare patience for me. And to the "Faz Esse" group, who since the first grade has been a constant in my life, and from nights out to going for a coffee, has made my life more colorful.

Thank you to Professor Rui Pinto and Gonçalo Ribeiro, for supervising this project and always demonstrating availability to help me as best they could. A big thank you to Continental Engineering Services, mainly the Interiors department, for welcoming me in such a warm way since day 1, and providing me with the biggest learning experience I've probably ever had.

In honor of my maternal grandmother, who passed away during the development of this dissertation

Rodrigo Castro

“If a man knows not to which port he sails to, no wind is favorable”

Seneca

Contents

1	Introduction	1
1.1	Context	1
1.2	Continental Engineering Services	2
1.3	Motivation	2
1.4	Problem	3
1.5	Objectives	3
1.6	Document structure	4
2	Literature Review	7
2.1	Electrical Control Units and Communication	7
2.1.1	Electronic Control Units	7
2.1.2	Network Architecture	7
2.1.3	Controller Area Network	8
2.2	Standards	11
2.2.1	ISO-TP Standard (ISO 15765-2)	11
2.2.2	Unified Diagnostic Service (ISO 14229-1)	11
2.3	Development tools	15
2.3.1	Rust programming language	15
2.3.2	Tokio	17
2.3.3	Serde	17
2.3.4	socketcan_isotp	18
2.3.5	JavaScript Object Notation	18
2.4	Related Work	18
2.4.1	Communicating Sequential Processes	18
2.4.2	TestGen-IF tool	19
2.4.3	Vector CANoe	19
2.5	Summary	19
3	Implementation	21
3.1	Hardware Setup	21
3.1.1	Debug Phase	21
3.1.2	Deployment Phase	22
3.2	Cross-Compilation: from x64 to ARM-32	23
3.3	Software Backbone	24
3.3.1	Architecture	24
3.3.2	Foundations	26
3.4	Test definition syntax	29
3.5	Test Generator	30

3.6	Summary	37
4	Validation	39
4.1	Methodology	39
4.2	Results	40
4.2.1	Template Execution	40
4.2.2	Timed comparison	44
4.3	Discussion	44
4.4	Summary	45
5	Conclusions	47
5.1	Summary of the results	47
5.2	Applications and positive implications	48
5.3	Future work	48
A	File testgen_util.rs	51
B	JSON test definition template for service 0x29	61
C	Guide used for the validation	63
	References	69

List of Figures

1.1	CES' worldwide locations.	2
2.1	Evolution of in-vehicle network architectures.	8
2.2	Example of the Producer-Consumer model.	9
2.3	ISO-TP network exchange for payloads above 7 bytes.	12
2.4	Vector CANoe's GUI.	20
3.1	Raspberry Pi and the CAN interface that was mounted in its GPIO pins.	22
3.2	Vector's VN5610A Ethernet/CAN interface.	23
3.3	Execution phase hardware setup diagram.	24
3.4	Deployment phase hardware setup diagram.	25
3.5	Example of the Command Line Interface.	29
3.6	Test generator flow.	31
4.1	Form to be filled by the volunteers before the validation.	40
4.2	Form filled by the volunteers after the validation.	41
4.3	Executions on target with correctly implemented UDS stack.	42
4.4	Failed template execution with the correct implementation of UDS.	43

List of Tables

2.1	Fields in a CAN Data Frame.	10
2.2	The 4 types of frames defined in the ISO-TP standard.	11
2.3	UDS' functional units and the services included	12
2.4	Request subfunctions in service 0x29 (Request Service ID 0x29).	15
2.5	Reponse subfunctions in service 0x29 (Request Service ID 0x69).	15
3.1	Operators available with the defined syntax	31
4.1	Time comparison between the execution of the same test routine with the proposed tool and syntax vs with Vector CANoe.	44

Listings

2.1	JSON deserialization example using serde.	17
3.1	Linker specification for the <code>armv7-unknown-linux-gnueabi</code> target (<code>.cargo/config</code>).	23
3.2	Building the project for Raspberry Pi execution.	24
3.3	<code>main.rs</code> code snippet.	26
3.4	A look at the the Monitor actor's functioning.	26
3.5	Code snippet of the Monitor's task.	27
3.6	Code snippet of the parser's defining structures (Some parts of the code are omitted for simplification).	28
3.7	Designed syntax applied in a small test for exemplification purposes (the backslash characters only for presentation purposes).	29
3.8	Code snippet of using <code>serde</code> to parse the JSON file.	30
3.9	Code snippet that implements receiving the response message from the ECU.	34
3.10	Code snippet of the splitting of options applied when the <code>'l'</code> operator is used.	34
3.11	Code snippet demonstrating the processing of <code>LEN()</code> operator.	34
3.12	Code snippet demonstrating the processing of the <code>FILE()</code> operator.	35
3.13	Code snippet demonstrating the OpenSSL command for signing a challenge.	35
3.14	Code snippet demonstrating appending the signed challenge.	36
3.15	Code snippet that demonstrates manual wrapping done for ISO-TP compliance.	36
A.1	Test generator file	51
B.1	JSON test definition template for service <code>0x29</code> (character <code>'</code> indicates new line used here for presentation reasons)	61

Abbreviations and Symbols

ACR	Authentication with Challenge-Response
APCE	Authentication with PKI Certificate Exchange
AUTOSAR	Automotive Open System Architecture
CAN	Controller Area Network
CAN-FD	CAN with Flexible Data rate
CES	Continental Engineering Services
CRC	Cyclic Redundancy Check
CSMA/BA	Carrier-Sense Multiple Access with Bitwise Arbitration
DOS	Denial Of Service
ECU	Electronic Control Unit
GUI	Graphical User Interface
IF	Intermediate Format
JSON	JavaScript Object Notation
NRC	Negative Response Code
PKI	Public Key Interface
RQ	Research Question
SSH	Secure Shell
TTCAN	Time-Triggered CAN protocol
UDS	Unified Diagnostic Service
USB	Universal Serial Bus

Chapter 1

Introduction

Modern vehicles rely heavily on software and electronics for the most critical and essential functions. Nowadays, they boast numerous Electronic Control Units (ECUs) that, connected between them and while running hundreds of millions of lines of code, exchange information that makes the car function. The need for reliable testing arises in order to ensure protocol robustness. This chapter attempts to give some context to the problem and includes a brief overview of Continental Engineering Services (CES), where the dissertation was developed. After discussing what motivated the work, the problem at hand is introduced, and the objectives of the work are defined.

1.1 Context

Ever since the birth of the first car at the end of the 19th century, the evolution has been constant and remarkably fast. What started as a purely mechanical machine evolved into what a modern car is today.

Yet, some things never change, and the core requirements of a vehicle are still valid to this day, such as ensuring safety, robustness, and reliability across a wide range of environments and conditions. Quickly, as the first programmable electronic components started to be used inside vehicles, it became necessary to have multiple ECUs working together to achieve the desired functionalities and avoid resource constraints. Consequently, internal network protocols, such as Controller Area Network (CAN) [1], started to be developed and used to enable communication between ECUs.

Autonomous driving capabilities, self-parking, complex infotainment systems, internet connection, or smartphone control are just some of the features in state-of-the-art cars. Although all these features significantly boosted the drivers' and passengers' comfort and the quality of driving, they also created a dependency on the electronics and software inside these vehicles.

This dependency raises the need for diagnostics services that can help manipulate the electronic systems. Unified Diagnostics Services (UDS) is a protocol that is standardized in ISO-14229 [2], and provides a common solution to diagnostics communication between brands.

1.2 Continental Engineering Services

Continental Engineering Services, founded in 2006, mainly provides engineering services in the automotive area, across areas such as automotive electronics, drive and chassis technology, or electrical mobility. The company's support for other industries such as agricultural and construction machinery, aerospace, agriculture, marine, railways, and medical technology is also increasing. With over 2200 employees over 24 different locations worldwide (shown in Figure 1.1 [3]). CES Porto, in particular, benefits from a strategic position when it comes to proximity to engineering universities and research institutes. Also, by having access to Continental's entire technology pool, CES presents itself in an unique position that combines the flexibility and speed that comes associated with small engineering teams, with the resources and strength of an internationally leading company.

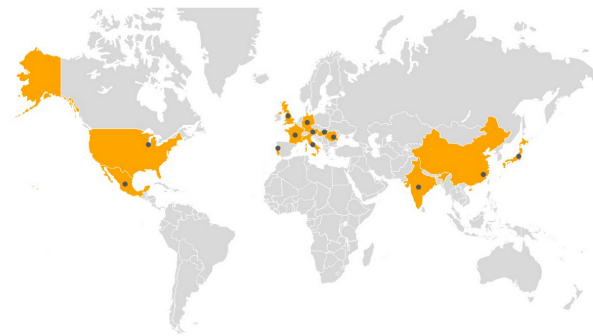


Figure 1.1: CES' worldwide locations.

1.3 Motivation

In order to implement network protocols, researchers and engineers need tools at their disposal that allow them to understand what's happening within the car's networks and manipulate them for testing purposes.

Teams rely heavily on products from external companies in order to conduct their testing and make sure the products developed are as bulletproof as possible.

Although the use of those products saves the teams from workload related to the development of a similar tool, it creates additional financial costs for the company. Also, available solutions greatly limit the flexibility in the creation of testing environments due to licensing systems, and offer no freedom to teams when it comes to the introduction of new features.

Furthermore, when looking at the specific case of UDS, a lack of automation in the testing process was evident after the literature review and after feedback from professionals in the area. Given the importance and the reach of the clearance one can achieve with this protocol [4], the testing to its implementation is paramount and should be done as reliably as possible to avoid the introduction of unexpected behavior or security weaknesses.

It's also worth mentioning two behemoths of the automotive industry and the embedded electronics world in general: C and C++. These two programming languages are the backbone of a big part of automotive systems mainly because of their speed and degree of control they give to the programmer. However, they suffer from a common weakness, which is the lack of memory safety. [5]

1.4 Problem

Whenever a new service is added to an ECU's UDS stack, it must be implemented and, afterward, tested for correct functioning. Typically, testing is done manually by exchanging packets in the network bus and analyzing the ECU's behavior, confirming if it is as expected. This process is painful and somewhat repetitive, making it prone to error.

Automation of this process seems obvious at first sight, but some restrictions have to be considered. Total automation of the testing process of a UDS stack is hard to achieve, mainly due to the high variability of implementations. Each client may specify different behaviors for the ECU, according to their preferences. Since an automated tool would have to "expect" a certain response from the ECU in order to deem the test successful, the tool is then forced to integrate some kind of customization that can account for the different specifications of the protocol.

1.5 Objectives

This dissertation's work focuses on the development of a tool, using the programming language Rust, that enables more reliable and straightforward testing of UDS protocol's implementation on an ECU. Given a semi-automation approach, this tool aims to greatly facilitate testing while maintaining a high level of control over expected responses from an ECU.

At the root of the tool rests the ability to monitor and manipulate a vehicle's internal CAN network. That means the user is able to capture packets being exchanged between ECUs as well as inject them. From that capability, a test generation feature is then developed. In order to execute a set of tests, the user inputs a JavaScript Object Notation (JSON) file describing the tests using a predefined syntax defined later in this document. The tool interprets the file, executes the tests, and returns if they were successful or not. The process is similar to how unit tests work.

The syntax developed must allow some flexibility such as the creation of variables, in order to accommodate the possibility of values that aren't known *a priori*. This is the case of challenge signing, where the server sends a challenge to the client, who then signs it, proving ownership. This process will be explained in more detail later, but since the challenge is determined by the server, the user cannot predict what the value will be nor its length and, as such, cannot expect a certain chain of values.

The choice of Rust comes from the devastating effects of memory-related vulnerabilities, and by considering the sheer complexity of C or C++. C's specification alone is around 800 pages, which is a lot considering it isn't nearly as complex as C++. It's not expected that a developer can

be fully knowledgeable of it, which ends up causing development errors. Rust gives the developer a very similar control to that of C/C++ while enforcing memory safety and quality code.

The software is embedded in a Raspberry Pi that enables a cabled connection to the desired vehicle through a physical CAN interface, eliminating the need for expensive external hardware.

The present work focuses on the service commonly known as "service 29", given its hexadecimal request ID, 0x29. This service was included in the ISO-14229-1 [2] that standardizes UDS's application layer in 2020, and poses as a modern approach to authentication in the UDS stack. This, regardless, does not mean the tool can't be used for many other services in the UDS stack. Service 29 was chosen because of how recent it is and how non-trivial its testing is.

The default template, which can be analyzed in appendix B, was designed according to ISO 14229-1. It tests what should be normal error detection by a correctly implemented UDS stack as well as the happy path (a successful authentication), relative to service 0x29. For example, sending a request without the minimum length should return an error code that is associated with receiving a message that does not meet the minimum length. If the tested ECU returns anything else, that means the implementation isn't according to the standard.

With all this in mind, two research questions (RQs) were formulated, which should be able to determine the success of the work developed:

- **RQ1:** Can the proposed tool achieve the same results as manual testing in a similar time-frame?
- **RQ2:** Can the use of the proposed tool prevent errors while testing?

1.6 Document structure

The document is structured as follows:

Chapter 2 - Literature Review

Provides a comprehensive overview both of the technological background supporting the work developed as well as of related work and tools that might help assess the state-of-the-art in UDS testing methodologies. The chapter mainly serves as a foundation for understanding the challenges and concepts involved in the subsequent chapters.

Chapter 3 - Implementation

Starts off by describing the hardware setup used during the development and deployment phases, followed by a description of the cross-compilation process used. After an overview of the software backbone of the tool, the syntax designed for defining the tests is described and a detailed overview of the test generator module is presented.

Chapter 4 - Validation

This chapter aims to find quantifiable data with which to answer the research questions defined previously. The methodology used to achieve the validation of the proposed solution

is presented. The results are laid out and discussed, mentioning limitations associated with the chosen methodology.

Chapter 5 - Conclusion

In this chapter, a brief recap of the document is constructed, which helps wrap up the information. The results collected in the previous chapter are summarized, and the applications and positive implications of the proposed solution are analyzed. The future work that could be developed is discussed.

Chapter 2

Literature Review

This chapter is intended to build the foundation that will support the work developed in the next chapters. First, a look into ECUs, as well as network architectures and a dive into CAN bus. Next, two essential standards for the work developed are discussed, one standardizing UDS application layer, and one standardizing ISO-TP. In the third section, an overview of some of the tools used during development; this includes an analysis of Rust, the programming language used in this project, some libraries and a look into the JSON format. Finally, some related work found in the literature review is mentioned.

2.1 Electrical Control Units and Communication

2.1.1 Electronic Control Units

Modern vehicles are composed of around 70 ECUs, which are embedded systems that control mechanical or electrical subsystems. [6][7] ECUs are responsible for braking, steering, passenger comfort, safety, overall monitoring of the car, and many more important tasks. They do this by receiving input from sensors installed in the vehicle, processing the data, and controlling actuators in the system. Communication between ECUs is achieved via the use of various technologies, CAN being the most common.

2.1.2 Network Architecture

The architecture of a vehicle's internal network consists of the core organization of its electronic components across the whole system in order to execute the desired function while accounting for security and performance. [8]

In current vehicles, the network is divided into different subsystems. Each subsystem has a number of ECUs destined for a given application, all connected to each other by a network protocol such as CAN. Then, all the subsystems are connected to a centralized gateway that receives information from all ECUs and executes protocol conversions. This creates a choking point in the gateway ECU requiring it to have high-bandwidth connections and high data processing ability.

In order to solve this issue, the industry is moving in-vehicle networks toward a domain-based architecture, in which ECUs are divided by functional domains (Figure 2.1, reproduced from [8]). Each domain gets a domain controller ECU associated that maintains most of the data transfer within a given domain. [7] Domain controllers are then inter-connected using high-bandwidth Ethernet. This ultimately divides the work of the gateway into different devices thus reducing the load and complexity of a central gateway.

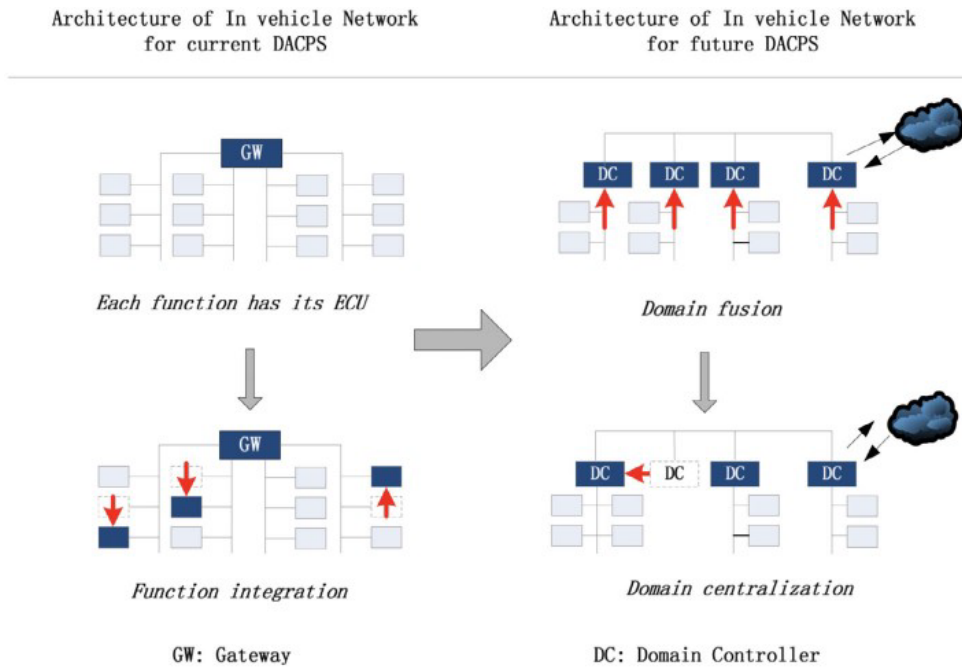


Figure 2.1: Evolution of in-vehicle network architectures.

2.1.3 Controller Area Network

Back in the decade of the 1950s, automotive electronics cost only 1% of the total car cost. In the 2010s it was around 35% and is expected to rise to 50% in the 2030s. [9] CAN is the most widely used protocol with respect to in-vehicle communications. It offers many advantages, mainly in respect of cost-effective wiring, and overall robustness in communications. [1] In contrast, it is weak in terms of security and has low bandwidth.

2.1.3.1 Overview

CAN was created by Bosch in the 80s, specifically for the automotive industry. The original CAN could transmit up to a rate of 1Mbps with a data payload between 8 bytes. In 2012 CAN with Flexible Data rate (CAN-FD) was released, which can go up to 5Mbps with a data payload of up to 64 bytes while being backward compatible with original CAN nodes. [1]

The physical layer supports the dominance property. This means bit collisions are resolved based on a notion of dominant/recessive bits (0 is dominant and 1 is recessive).

The data link layer controls medium access through Carrier Sense Multiple Access with Bit-wise Arbitration (CSMA/BA), using an identifier field in order to determine priority (a lower ID sports benefits of higher priority). If a node is transmitting and detects that the bit in the bus is different from the one it sent, the node switches to receiving mode. CAN is a message based with unique message identifiers protocol. There are five different message (or frame) types, listed below [10]:

- **Data Frame:** it is normally the most exchanged frame in CAN communication. It is used to transmit data between ECUs on the bus. The Data Frame consists of the seven fields denoted in Table 2.1.
- **Remote Frame:** it is sent by a node in order to request data from another ECU on the bus. The structure of this frame is very similar to the data frame's differing only in two aspects: the absence of the data field, and the value of the Remote Transmission Request which is recessive in the remote frame and dominant in the data frame.
- **Overload Frame:** this frame is sent by an ECU to signal that it is overloaded and cannot process more messages. As a consequence, the transmitting node will increase the delay between consecutive data or remote frames. Its structure consists of six dominant bits (overload flag field) followed by the overload delimiter field which consists of eight recessive bits.
- **Interframe Space:** consists of a gap between two consecutive data or remote frames. Composed of at least three recessive bits this time interval allows for the separation of frames as well as synchronization.
- **Error Frame:** This frame is sent by a node as soon as it detects an error while receiving or sending. After this, the bus enters an error state. The frame is composed of six dominant bits, ending with the error delimiter which is a recessive bit.

At the application layer, it uses the Producer-Consumer model, in which producer nodes disseminate information and the consumer nodes access it (Figure 2.2). In this model, transactions are triggered by the producers, they send a message to the bus and the consumer nodes determine whether the message is relevant based on its identifier.

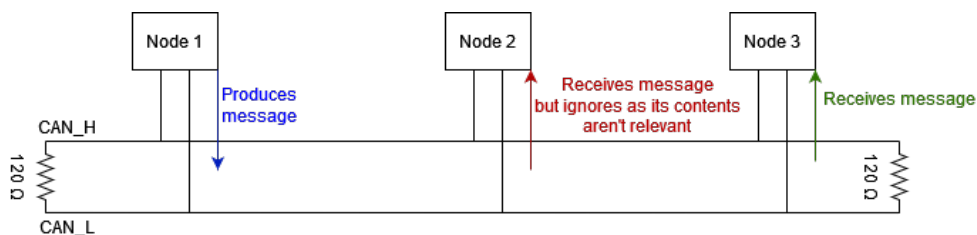


Figure 2.2: Example of the Producer-Consumer model.

Table 2.1: Fields in a CAN Data Frame.

Field name	Length (in bits)	Description
Start of Frame	1	Consists in a solo dominant bit that signals the beginning of a new frame
Identifier	11 or 29 in the case of the extended frame format	Unique message identifier that represents the message's priority
Remote Transmission Request	1	Used to distinguish between a request for data or a response. When set to recessive (1), the frame represents a request for data. When dominant (0) it represents a response.
Data Length Code	4	Indicates the number of bytes of data transmitted by the message (0-8 bytes).
Data	0-8	Data transmitted by the message
Cyclic Redundancy Check	15	Used to detect inconsistencies in the messages that may be caused, for example, by noise in the transmission channels,
End of Frame	7	Consists of seven recessive bits that signal the end of the current frame.

2.1.3.2 Security

CAN works in broadcast, meaning every ECU is connected to the bus and can send/receive messages from it. The protocol boasts, by default, no message authentication, and no encryption. Bozdal et al. [1] set three main characteristics as defining the level of security of a communication protocol:

1. Confidentiality - Communicating data to authorized individuals only. Broken since CAN has no encryption whatsoever. An attacker can access any data in the bus without restrictions;
2. Integrity - related to the validity of the data. Although CAN uses a Cyclic Redundancy Check (CRC) field to check for inconsistencies, it cannot prevent malicious injection of data;
3. Availability- an authorized user's ability to always use the network. Due to how priority is integrated into the protocol, an intruder can deny access by overloading the bus with maximum priority messages. This is known as a Denial Of Service (DOS) attack; [11]

In short, none of these criteria are matched and, as a result, networks running on CAN cannot be considered safe from a security standpoint.

2.2 Standards

2.2.1 ISO-TP Standard (ISO 15765-2)

The ISO-TP standard is defined in ISO 15765-2 [12] and solves the problem related to CAN having a maximum payload of 8 bytes. ISO-TP enables the exchange of messages up to 4095 bytes by fragmenting them into 8-byte CAN frames and adding some metadata. [13] The standard defines 4 types of frames that allow the transmission of longer messages, which are presented in Table 2.2.

Table 2.2: The 4 types of frames defined in the ISO-TP standard.

Type of frame	Byte 0	Byte 1	Byte 2	Bytes 3-7
Single Frame (SF)	First 4 bits set to 0; Second 4 are the size of the data (0-7 bytes)	Data/Padding	Data/Padding	Data/Padding
First Frame (FF)	First 4 bits set to 1; Second 4 bits represent the size of the data with 8 more bits from Byte 1 totaling 12 bits (8-4095 bytes)	Size of data (8-4095 bytes)	Data/Padding	Data/Padding
Consecutive Frame (CF)	First 4 bits set to 2; Second 4 bits, index of the message (1-15)	Data/Padding	Data/Padding	Data/Padding
Flow Control Frame (FC)	First 4 bits set to 3; Second 4 bits flag (0-2)	Block Size	Separation time	Padding

When communicating using ISO-TP, every payload that doesn't exceed 7 bytes can be transmitted with a Single Frame, a frame that is followed by no other. Every payload that has a size between 8 and 4095 bytes will use an exchange of the other three with the flow shown in Figure 2.3. A First Frame is sent, which contains information on the total length of the data that is going to be sent, as well as the initial chunk of the data. After this, the other side responds with a Flow Control Frame, that controls the speed with which the sender must send the Consecutive Frames that contain the rest of the data.

2.2.2 Unified Diagnostic Service (ISO 14229-1)

UDS is a protocol with its application layer specified in ISO 14229-1 [2] and aims to facilitate diagnostics in routine check-ups, covering the application and session layers of the OSI model. [6] The protocol was designed to standardize diagnostics communication between different manufacturers' ECUs.

UDS contains various diagnostic services that each perform a given diagnostic function. These services are divided into six functional units and some are described in Table 2.3, in order to convey

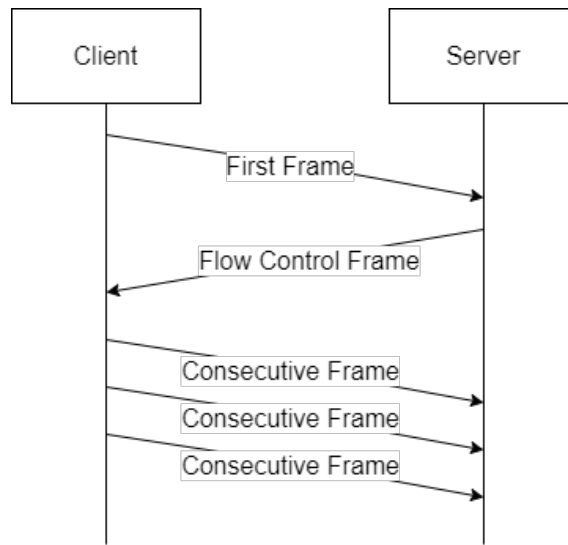


Figure 2.3: ISO-TP network exchange for payloads above 7 bytes.

the general capabilities of the protocol. Each service is commonly known by its request service ID (in parenthesis after each service’s name).

Table 2.3: UDS’ functional units and the services included .

Functional Unit	Description	Service	Description
Diagnostics and communication management	Allows controlling of the diagnostic and communication-related operations in the ECU	Diagnostic Session Control (0x10)	Allows changing which session is active. UDS utilizes different operating sessions, each one with different services available. When an ECU starts, the default session is loaded; if, for example, the user wants to re-program the ECU, the services needed will only be available in the programming session and the user would need to change the session

Functional Unit	Description	Service	Description
		ECU Reset (0x11)	<p>Allows for resetting the ECU. There are three forms of resetting:</p> <ul style="list-style-type: none"> • Hard reset - simulates the shutdown of the power supply; • Key off-on reset - simulates turning the ignition off and on; • Soft reset - simulates the equivalent of a watchdog reset in a microcontroller;
Data Transmission	Contains services designed to read and write data to the ECUs	Read data by Identifier (0x22)	Using this service, data can be gathered from the ECU. Each value has a data identifier associated which is represented by a 2-byte unsigned integer (0 to 65535)
		Read memory by address (0x23)	Reads the data contained in the specified memory address
		Dynamically defined data identifier (0x2C)	Used to read the response of a pre-defined list of needed values. When this new identifier is polled, the ECU responds with the values of all the pre-defined identifiers
		Write data by identifier (0x2E)	Writes the value associated with the specified data identifier
		Write memory by address (0x3D)	Writes in the specified memory address

Functional Unit	Description	Service	Description
Remote Activation of Routine	Allows starting a specific routine in an ECU	Routine Control(0x31)	<p>This service can be used to perform a variety of actions, namely:</p> <ul style="list-style-type: none"> • Managing devices such as actuators and sensors; • Running specific test sequences; • Mutating values in the ECU's systems.

As anyone could access the CAN bus and exchange messages complying with the UDS standard, it also introduces security measures in order to guarantee that only authorized individuals are able to alter an ECU's software data. The SecurityAccess service is used to achieve this authentication. The client first requests the ECU for a seed that will then be used to compute a key, using a cryptographic function chosen by the vehicle's manufacturer. The key is sent back to the ECU, and if it is as expected, then access is unlocked.

Due to the weak processing capability of ECU computers, the complexity of these cryptography processes is not great, which opens up the way to brute force attacks. To prevent this, sometimes a delay is applied when a client requests for a seed and has already failed authentication multiple times. In some systems, going beyond a certain number of failed authentications can trigger automatic denial of authorization. [6]

2.2.2.1 Service 0x29

Since we will focus on service 0x29 for the development of the proposed tool, it is worth taking a more detailed look into it. Introduced in the 2020 update to the ISO-14229, this service serves as a replacement for service 0x27 and enables a more modern approach to authentication, bringing bi-directional authentication with certificate exchange based on a public key infrastructure.

It provides a way for the client to prove its identity and thus protect restricted access functions and data. Service 0x29 supports two concepts:

- Based on Public Key Interface (PKI) with certificate exchange procedures using asymmetric cryptography: Authentication with PKI Certificate Exchange (APCE);
- Based on challenge-response procedure without PKI certificates by using either symmetric cryptography or asymmetric cryptography with software authentication tokens: Authentication with Challenge-Response (ACR);

The work is focused on the APCE concept that supports both unidirectional authentication (client against the server) as well as bidirectional authentication (client against the server and server against the client). Tables 2.4 and 2.5 contain some of the relevant request and response subfunctions, respectively.

Table 2.4: Request subfunctions in service 0x29 (Request Service ID 0x29).

Name	ID	Description
deAuthenticate	0x00	Simple request to deauthenticate
verifyCertificateUnidirectional	0x01	This request initiates an exchange for unidirectional authentication. It carries a certificate that will be checked by the ECU in order to verify the tester's legitimacy.
proofOfOwnership	0x03	This request transmits the solution of the challenge sent by the ECU in response subfunction 0x01. It aims to confirm the tester's ownership of the certificate by having the tester sign a challenge sent by the ECU

Table 2.5: Reponse subfunctions in service 0x29 (Request Service ID 0x69).

Name	ID	Description
verifyCertificateUnidirectional	0x01	This is the response to the 0x29 0x01 subfunction request. It informs the tester if the certificate that the certificate is valid and sends a challenge that can be signed by the tester in order to prove ownership of the certificate.
proofOfOwnership	0x03	This request concludes an authentication process, signaling if it was successful

2.3 Development tools

2.3.1 Rust programming language

Rust [14] was created by software developer Graydon Hoare in a personal project while working at Mozilla Research back in 2006. The project was sponsored by Mozilla in 2009, and its first stable release was launched in May 2015. Since then, Rust has been growing fast, starring in Mozilla's proprietary Firefox browser and even becoming, in December 2022, the second high-level language to be included in Linux's kernel code, after C.

2.3.1.1 Borrow Checker

Rust is a language that focuses on high-performance, safe and concurrent code. One of its most notable features is the borrow checker, Rust's way of managing memory. The borrow checker is

the part of the Rust compiler that enforces ownership and makes Rust memory-safe. Ownership, as defined in Rust's official book [15], is the "set of rules that govern how a Rust program manages memory". While languages like Java, C#, or Python have a garbage collector that frees unused memory while the program is running, languages like C, C++ or Rust do not boast this mechanism.

However, Rust's ownership rules are enforced by the compiler, and if a program violates a single rule across its whole extension, then compilation will be unsuccessful. This means Rust has no runtime overhead associated with memory management, as all processing done to ensure memory safety goes on at compile time. The ownership rules are defined as the following [15]:

- Each value has an owner;
- A variable can only have one owner at a time;
- As soon as the owner goes out of scope, the value is dropped;

With these rules, Rust is able to avoid common errors such as buffer overflows, null pointer dereferencing, or even data races. In fact, in Rust, null pointers don't exist, and as a substitution the standard library offers an `Option` enum that either is variant `Some(value)` or variant `None`. This is important as even Tony Hoare, who invented the null reference while working in the ALGOL W language, calls it his "billion-dollar mistake" stating it lead to "innumerable errors, vulnerabilities, and system crashes" since its first appearance.

Rust then supports borrowing, by giving references to a value. You can either have one mutable reference to a value or any number of immutable references to a value. This way values can be accessed without changing their ownership.

2.3.1.2 Cargo

Cargo is another one of Rust's key features. It is the package manager and build tool for the Rust language. It provides an easy and convenient way to manage dependencies, build projects, even enabling cross-compilation. Cargo supports versioning of dependencies which prevents compatibility issues while making it easy to upgrade dependencies to their new versions. The dependencies are sourced from `crates.io`, a central repository that hosts a large number of Rust packages. Cargo also makes it easy for developers to publish their packages in it and make them available for the community to use them.

Cargo also boasts a built-in testing framework that makes it straightforward to write unit and integration tests. As of the writing of this document, a benchmark feature is also available in nightly Rust (the experimental version of the language), and enables the execution of benchmarks.

2.3.1.3 Why Rust?

Why should one choose Rust over other performant languages such as C++ or C? These last two offer no memory safety of any kind, which makes it possible and common for developers to "shoot themselves in the foot". In fact, Google's team over at the Chromium project found

that 70% of their serious security bugs were memory safety problems. [16] This dangerous lack of memory safety, coupled with the fact that Rust programs boast a comparable performance to C/C++ programs [17][18], makes it a very viable substitute.

Rust's type system, its intelligent compiler that carefully and extensively details compilation errors, and its harmonious combination of speed and safety is the reason why the language is regarded, as of StackOverflow's 2022 Developer Survey [19], as the most loved programming language by developers.

2.3.2 Tokio

Tokio [20] provides a runtime to enable asynchronous programming with Rust, as well as asynchronous versions of the blocking APIs in the standard library. It is one of the most known libraries in the Rust world, and a must for writing asynchronous programs. tokio, following Rust's theme, sports zero-cost abstractions that make it bare-metal fast. Asynchronous TCP and UDP sockets are also available with this library by leveraging operating systems' event queues (such as epoll and kqueue).

Asynchronous programming is strong in solving the blocking I/O problem. [21] Using the example of a TCP socket, the operation of waiting to receive a message can become non-blocking by reverting access to the handler. When a message becomes available to receive, the operating system's event queue will notify the runtime and the task scheduler may carry on with receiving the message.

2.3.3 Serde

Serde [22] is a popular and widely used Rust library for deserialization and serialization. It serves as a way to convert data between different formats such as JSON, TOML, YAML or DynamoDB items. serde's role in this work will consist of deserializing the JSON file inputted by the user. That way, the program can interpret the test described in the file and execute it. Listing 2.1 represents a code snippet demonstrating the straight-forward deserialization of a JSON file [22]:

```
1 use serde::{Serialize, Deserialize};
2
3 #[derive(Serialize, Deserialize, Debug)]
4 struct Point {
5     x: i32,
6     y: i32,
7 }
8
9 fn main() {
10     let point = Point { x: 1, y: 2 };
11
12     // Convert the Point to a JSON string.
13     let serialized = serde_json::to_string(&point).unwrap();
```

```
14
15 // Prints serialized = {"x":1,"y":2}
16 println!("serialized = {}", serialized);
17
18 // Convert the JSON string back to a Point.
19 let deserialized: Point = serde_json::from_str(&serialized).unwrap();
20
21 // Prints deserialized = Point { x: 1, y: 2 }
22 println!("deserialized = {:?}", deserialized);
23 }
```

Listing 2.1: JSON deserialization example using serde.

2.3.4 socketcan_isotp

socketcan_isotp [23] is Rust a library that provides CAN sockets and methods that include an abstraction from the ISO-TP layer, by wrapping data according to the standard, which is essential as all messages need to be according to the ISO-TP standard in order to be received by the ECU, even if they are 8 bytes or less. Under the hood, the library utilizes `can-isotp`, a Linux kernel module developed in C. The library also takes care of the reception of Flow Control Frames, and adjusts the parameters accordingly.

2.3.5 JavaScript Object Notation

JavaScript Object Notation (JSON) [24] is a text-based, lightweight data format. Due to its simplicity and compatibility with a large number of programming languages, in the last few years, JSON became one of the most common ways to exchange data over the web. It is widely used in Application Programming Interfaces. In the context of this dissertation, JSON will be used by the user of the program to describe the tests they want to execute. This file will then be interpreted by serde (Subsection 2.3.3).

2.4 Related Work

2.4.1 Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a formal model that can be used to describe and model concurrent systems. Thus, it can also be applied to modern vehicles due to the systems of systems approach to building them. As explored in [25], a vehicle can be modeled as a composition of processes, with processes exchanging messages between them as in automotive networks.

In the aforementioned work, Ran Q. et al. model the networks and bus systems found in a modern car and also describe attack vector models that all in all enable the automatic generation of tests that can evaluate the response of a system to attacks.

Timed CSP is applied in [26] in order to model and verify the Time-Triggered CAN protocol (TTCAN). In a nutshell, TTCAN is an extension of CAN in which a reference message is transmitted periodically, enabling nodes to calculate their timeslots for the transmission and reception of messages. Timed-CSP is an extension of CSP that adds time operators.

2.4.2 TestGen-IF tool

TestGen-IF [27] was created by a team at Telecom SudParis and, similarly to the work developed in this dissertation, aims to facilitate active testing, enabling capabilities like checking a protocol's or a service's implementation. In this case, the system under test is described using the Intermediate Format (IF) language and composed of parallelized processes running asynchronously and communicating through shared variables and message exchanging through either channels or direct addressing.

The tool was shown working with the Dynamic Route Planner, a service used to solve transportation problems, mainly providing the driver with the best route to their destination. In order for the driver (client) to activate the service, a certificate exchange takes place, with the server verifying if the client has the authorization to activate the service. This is the process tested with the TestGen-IF tool.

2.4.3 Vector CANoe

CANoe [28] is a software tool developed and maintained by Vector. It's one of the most used tools in the automotive industry and allows the development, testing and analysis of individual ECUs and even whole ECU networks. One of its main features is the ability to support multiple bus systems, such as CAN, LIN, FlexRay and Ethernet, enabling developers to test various kinds of systems. The program also supports Automotive Open System Architecture (AUTOSAR), J1939 standard and OBD-II. CANoe's main advantages lie in the user-friendly Graphical User Interface (GUI) (represented in Figure 2.4) and being a single tool that fulfills most development and testing tasks needed. However, besides the heavy price tag of the software, one must also acquire a CAN Box, in which the license is contained, to use the tool.

2.5 Summary

In this chapter, an overview of ECUs was presented, as well as of CAN communication and vehicle network architecture. An analysis of both the ISO 15765-2 and ISO 14229-1 was executed, two standards on which the project is heavily based. Regarding development tools, Rust was discussed, the programming language used during development and that allows similar control and performance to C++ while offering memory safety through a borrow-checker system. Furthermore the key Rust libraries in the project, such as tokio for asynchronous programming, serde for parsing JSON files and socketcan-isotp for providing wrapping according to the ISO-TP standard, were analyzed. The JSON file format was chosen to describe the tests, given its popularity. All this

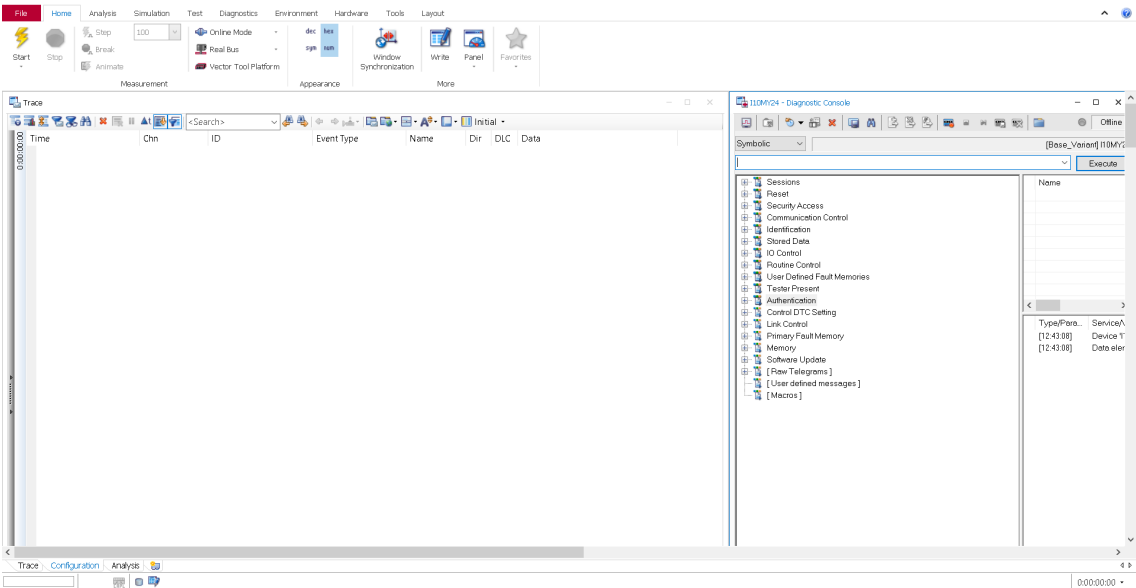


Figure 2.4: Vector CANoe’s GUI.

aims to build a foundation for the following chapters. Closing the chapter, some related work that was found during literature review was analyzed. The lack of related work lead to the conclusion that a gap in the literature does, in fact, exist.

Chapter 3

Implementation

In this chapter, the process for the implementation of the tool is presented. Firstly, the hardware setup is described, followed by a look into the cross-compilation pipeline used and an analysis of the software architecture. This is important in order to understand how the tool functions as a whole, and the way the different modules are interconnected. Furthermore, the core components in the code are analyzed, as to discuss the key logic that runs the software. After discussing the syntax created for the description of the tests, a detailed look into the code that drives the generation of tests is presented.

3.1 Hardware Setup

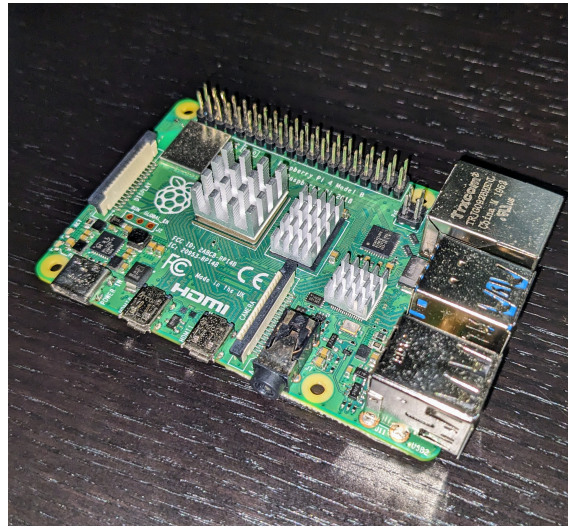
When referring to the hardware dimension of the project, we can divide it in two phases according to the timeline. Moving forward we will refer to the first one as "Debug Phase" and the second one as "Deployment Phase".

3.1.1 Debug Phase

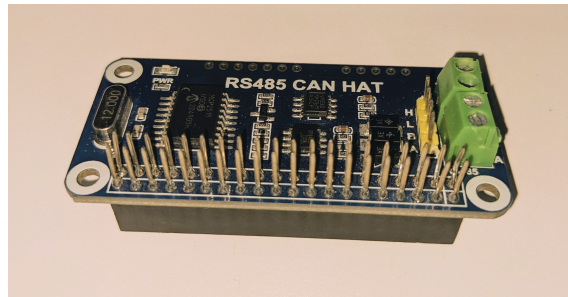
The debug phase was the first phase of the project and was directed toward developing functioning foundations for the planned work. CAN communication such as being able to send and receive simple messages, as well as the JSON parsing that enabled interpreting the tests described by the user were done during this phase.

The core component of the setup is a Raspberry Pi 4 Model B [29], depicted in Subfigure 3.1a. A small computer with an ARM architecture processor, that allowed connection via a CAN interface. The computer was loaded with the 32-bit Raspberry Pi OS, the device's official operating system, version Bullseye from 30th of October 2021 for toolchain compatibility reasons that will be discussed later in the 3.2 section. The use of a Linux-based operating system allowed taking advantage of its user space toolset of CAN utilities.

As the Raspberry does not boast a CAN interface from the get-go, a CAN HAT manufactured by Waveshare (Subfigure 3.1b) was mounted in its GPIO pins, thus achieving the ability to communicate in the CAN bus. As the CAN HAT already included the necessary 120 Ω terminal



(a) The Raspberry Pi 4 Model B used.



(b) CAN HAT interface.

Figure 3.1: Raspberry Pi and the CAN interface that was mounted in its GPIO pins.

resistor, the only setup needed was installing `wiringPi`, a GPIO interface library for the Raspberry Pi. The Raspberry was then accessed headlessly through Secure Shell (SSH), meaning there wasn't a display connected to the device.

In order to debug CAN communication while developing, the Raspberry was connected via the CAN HAT to a VN5610A network interface through a CAN connection. The VN5610A (Figure 3.2), manufactured by Vector, was connected to a laptop via Universal Serial Bus (USB) and running Vector CANoe. Through CANoe, it was then possible to analyze messages sent from the Raspberry, as well as to transmit them. A diagram of the full setup is represented in Figure 3.3.

3.1.2 Deployment Phase

During the deployment phase, the setup was similar, although with two new components. A remote computer and the ECU board. Now, the only connection in the laptop is to the remote computer (via Remote Desktop Protocol), to which everything else is connected. The Raspberry was connected via Ethernet to the remote and assigned a static IP so that an user can take advantage of the tool just by connecting to the remote. The ECU board was also connected via serial port to the remote for monitoring. The Vector CAN interface was connected via USB to the remote as



Figure 3.2: Vector's VN5610A Ethernet/CAN interface.

well as via a CAN split cable to both the CAN HAT and the ECU Board. A diagram of the full setup is represented in Figure 3.4.

The role of the CAN interface in this phase is simply to be able to examine the messages exchanged between the CAN HAT and the ECU board. In order to connect both together (since they were in different channels), a script was written so the messages sent from one channel were redirected to the other and vice-versa.

3.2 Cross-Compilation: from x64 to ARM-32

In order for the application to run on the Raspberry Pi, a cross-compilation setup had to be arranged. While development was being done in a x64 computer with Windows, the Raspberry Pi's processor uses ARM architecture while loaded with a Linux distro-based OS. Also, the code includes libraries related to CAN communication that require Linux kernel applications, which renders the code uncompileable in the Windows computer. For that a Windows wrapper was built so that the project could be compiled in windows (although not having any real functionality).

To enable cross-compilation, a prebuilt Windows toolchain for the Raspberry Pi was installed and its binaries listed in the PATH variable. After adding the target to rustup with `rustup target add armv7-unknown-linux-gnueabi` in the command line, the linker that should be used for the `armv7-unknown-linux-gnueabi` was specified in the `.cargo/config` file with the entry described in Listing 3.1.

```
1 [target.armv7-unknown-linux-gnueabi]
2 linker = "arm-linux-gnueabi-gcc-10.exe"
```

Listing 3.1: Linker specification for the `armv7-unknown-linux-gnueabi` target (`.cargo/config`).

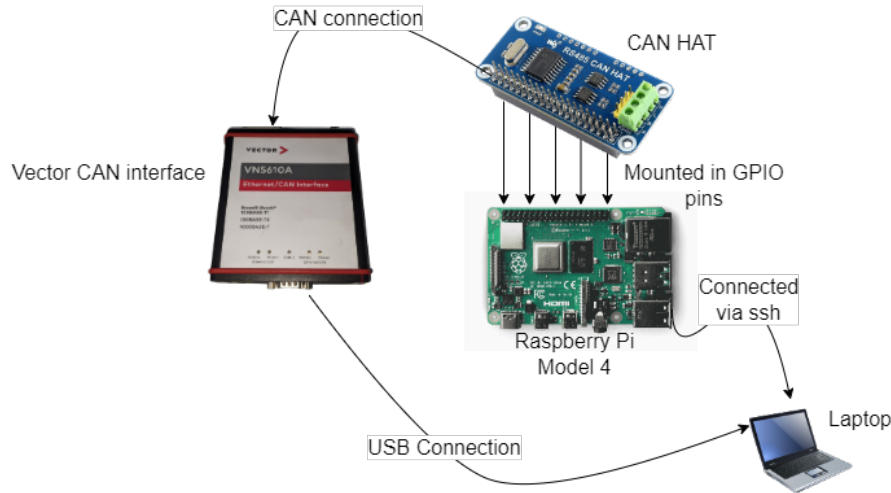


Figure 3.3: Execution phase hardware setup diagram.

After this configuration, the project can be built for the Raspberry Pi running Listing 3.2's command in the command line while inside the project's root directory.

```
1 cargo build --target armv7-unknown-linux-gnueabi
```

Listing 3.2: Building the project for Raspberry Pi execution.

The Raspberry Pi OS image used was the one recommended by the toolchain developers, as it guaranteed compatibility with the toolchain.

3.3 Software Backbone

The project has, since the beginning, pointed in the direction of facilitating horizontal scalability in order to leverage the distribution of computation nodes and improve flexibility in the creation of testing environments. Before taking a look at the modules that compose the system, we shall look at the actor model on which this software's architecture rests.

3.3.1 Architecture

The actor model [30] is a mathematical model applied in concurrent systems that has the `Actor` as its most basic primitive and messages as the root of communication. When an actor receives a message, it can proceed to do the following:

- Create new Actors;
- Decide how to handle the next message that gets received;
- Send messages to other Actors, as long as it possesses their addresses;

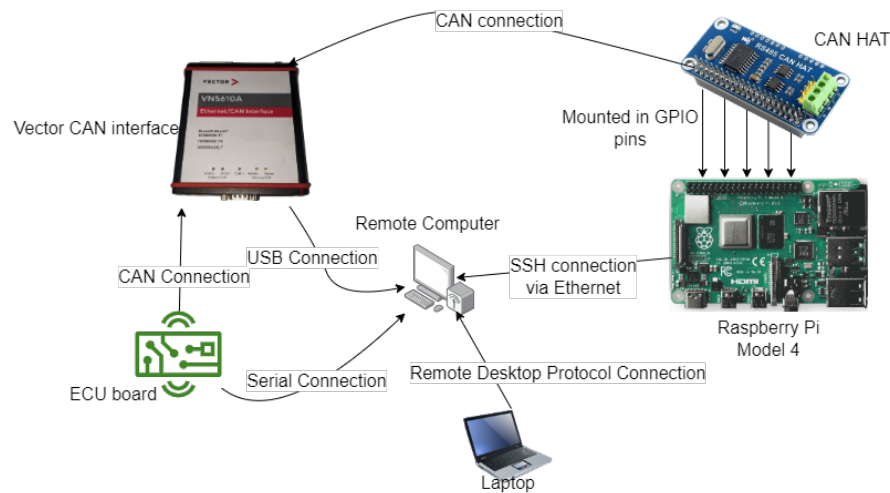


Figure 3.4: Deployment phase hardware setup diagram.

- Modify its own state;

The most important takeaways of the model are its ability to isolate actors in the sense that each one executes their tasks concurrently, enabling efficient resource usage and ensuring an actor's state remains unaffected by processes running in other actors. This is particularly important as it allows for much greater control over what happens in the system. Applying the actor model to the software's architecture resulted in 6 different actors:

- Monitor: the monitor actor is the root actor of the architecture. It's the only actor that isn't created by another actor. The monitor actor communicates directly with the "Ctrl-C"/console actors only, and its sole purpose is monitoring the system's cycle.
- "Ctrl-C": this actor has the sole function of monitoring when the user performs a Ctrl+C on the keyboard, a keybind combination that is supposed to terminate the program immediately. It communicates directly with the monitor actor only;
- Console: The console actor takes care of monitoring the user's input on the console. It parses the input and then decides which path to follow. Depending on the command the user invokes, either one of the following three actors receives a message.
- Sender: the sender actor serves the purpose of sending a message specified by the user to the CAN bus.
- Receiver: the receiver actor serves the purpose of receiving a message from the CAN bus.
- Test generator: this actor is the spotlight of the developed work. By receiving a file path from the console actor that points to the test description, it exchanges messages in the CAN bus in order to execute and verify if the tests pass or fail, displaying the results back to the user;

Every actor is represented by two objects: an handler and an object that produces the expected results. In the case of the test generator, it is represented by the `TestGenHandle`, which receives messages from the console actor (the test file) and forwards it to the `TestGen` object that then interprets it.

3.3.2 Foundations

A Rust application starts at the `main.rs` file, and this one is no exception. Given the nature of the actor model and the way the architecture was structured, the only functionality built into `main.rs` is starting the `Monitor` actor and calling for the "Ctrl-C"/Console actors' initialization.

```
1  #[tokio::main]
2  async fn main() {
3      std::env::set_var("RUST_LOG", "debug");
4
5      env_logger::init();
6
7      info!("Starting runtime");
8
9      let mut monitor = MonitorHandle::new();
10
11     monitor
12         .spawn_ctrlc_watcher()
13         .await
14         .expect("could not spawn ctrl c watcher");
15
16     monitor
17         .spawn_console()
18         .await
19         .expect("could not spawn console actor");
20
21     monitor.wait_to_die_like_in_life().await;
22 }
```

Listing 3.3: `main.rs` code snippet.

The code in Listing 3.3 starts with the attribute macro `#[tokio::main]`, and it's provided by the Tokio library in order to facilitate calling asynchronous functions by initiating a runtime automatically. After tuning some logging functionalities, the `Monitor` actor is created by initializing a new instance of the `MonitorHandle` struct. After spawning both the `Ctrl-C` and the `Console` actors, the handler then waits for a shutdown signal that marks program termination.

```
1  enum MonitorMessages {
2      SpawnCtrlC,
3      SpawnConsole,
4      UglyExit,
```

```
5     CleanExit,  
6 }  
7 pub struct MonitorHandle {  
8     inbox: mpsc::Sender<MonitorMessages>,  
9     shutdown: watch::Receiver<bool>,  
10 }  
11 impl MonitorHandle {  
12     pub fn new() -> Self {  
13         let (sender, receiver) = mpsc::channel(3);  
14         let (tx, shutdown) = watch::channel(false);  
15  
16         let handle = Self {  
17             inbox: sender,  
18             shutdown,  
19         };  
20  
21         let actor = Monitor::new(receiver, tx, handle.clone());  
22  
23         tokio::spawn(run(actor));  
24  
25         handle  
26     }  
27     pub async fn spawn_console(&self) -> Result<()> {  
28         let msg = MonitorMessages::SpawnConsole;  
29  
30         self.inbox.send(msg).await.expect("failed to send message");  
31  
32         Ok(())  
33     }  
34 }  
35 }
```

Listing 3.4: A look at the the Monitor actor's functioning.

Listing 3.4 shows basic functioning behind an actor with the `Monitor` as the example. On initialization, the handler starts off by creating a channel that will allow communication with the `Monitor` instance via messages defined in the `MonitorMessages` enum. The `watch` channel is opened to monitor a `bool` variable that corresponds to the shutdown signal. The handler gets the sending end and the receiving end of the first and second channels, respectively, while the `Monitor` struct is initialized with the receiving/sending ends, respectively. Line 23 spawns a new asynchronous task that makes the actor actually start to function. The `spawn_console` method shows that the handler does not have any inherent functionality, as it simply transmits intended behaviors through the communication channel. In this case, a `SpawnConsole` message is passed, that will be interpreted by the `Monitor` instance.

```

1 pub async fn run(mut actor: Monitor) {
2     while let Some(msg) = actor.inbox.recv().await {
3         if !actor.handle_message(msg) {
4             break;
5         }
6     }
7     // tell CtrlC actor to shutdown
8     if let Some(ctrlc) = actor.ctrlc_actor {
9         ctrlc.clean_shutdown().await;
10    }
11    // tell Console actor to shutdown
12    if let Some(console) = actor.console_actor {
13        console.shutdown().await;
14    }
15    actor.shutdown.send(true).unwrap();
16 }

```

Listing 3.5: Code snippet of the Monitor’s task.

Listing 3.5 contains the task running asynchronously for the Monitor actor. The while loop will iterate every time the handler sends a new message through the channel. The `handle_message` method will then check which kind of `MonitorMessages` was passed. In the case of an `UglyExit` or a `CleanExit`, the method will return false, which will trigger a shutdown. In the case of a `SpawnCtrlC` or a `SpawnConsole`, the respective actors will be spawned, the return will be true, and the while loop will continue.

Now that the backbone behind the architecture is understood, it is important to delve into how the user’s input is parsed by the Console actor. Console parsing was achieved by using `clap` [31], a Rust library designed specifically for that effect. By defining a few structs and enums, one can have a solid and polished command line interface set up that includes automatic help generation (`-help`), and fix suggestions for the user.

```

1 #[derive(Parser)]
2 #[clap(version = "0.2.0", setting=AppSettings::NoBinaryName)]
3 pub struct Opts {
4     #[clap(short, long)]
5     config: Option<String>,
6     #[clap(subcommand)]
7     subcmd: SubCommand,
8 }
9
10 #[derive(Parser)]
11 enum SubCommand {
12     GenTest(Test),
13 }
14

```



```

15 #[derive(Parser)]
16 struct Test {
17     file_path: String,
18 }

```

Listing 3.6: Code snippet of the parser's defining structures (Some parts of the code are omitted for simplification).

Analyzing Listing 3.6, the `Opts` enum is required for the library, and besides giving the option for inputting a config file, it points to a second enum, `SubCommand`. This last is where the integration of the intended user commands is materialized. In this case, the process for entering the test generation case is exemplified. The `GenTest` option on the enum creates the `gen-test` subcommand, which accepts an argument for the path to the JSON file describing the intended tests. The result of this configuration in Windows' Powershell can be visualized in Figure 3.5.

```

gen-test
error: The following required arguments were not provided:
  <FILE_PATH>

USAGE:
  gen-test <FILE_PATH>

For more information try --help

```

Figure 3.5: Example of the Command Line Interface.

3.4 Test definition syntax

As previously discussed, the approach to test generation in the context of this work involves a file written by the user that describes the desired tests. With the goals of ease of understanding and flexibility, a syntax for the JSON file was defined. Listing 3.7 contains a small example of a test description using the designed syntax. It's important to note that the ``\`` symbol introduces a new line that isn't present in a regular file and serves only for presentation purposes.

```

1 {
2     "TestSuitName": "0x29 Service",
3     "ID": "Extended, 0x18DB33F1, 0x18DAF100",
4     "Tests": [
5         {
6             "Sequence": [
7                 {
8                     "TestName": "Happy Path"
9                 },
10                {
11                    "PairArray": ["0x29", "0x01", "0x00", "0x08", "0x00", "\
12                    "FILE(client_certificate.der)", "0x00", "0x00", "\

```

```

13     "Response", "0x69", "0x01|0x02", "RANGE(0x11,0x15)", "LEN(CHA)", "CHA", \
14     "0x00", "0x00"]
15     },
16     {
17     "PairArray": ["0x29", "0x03", "LEN(RES(CHA))", "RES(CHA)", \
18     "0x00", "0x00", "Response", "0x69", "0x03", "0x12", "0x00", "0x00"]
19     }
20     ]
21     }
22     ]
23     }

```

Listing 3.7: Designed syntax applied in a small test for exemplification purposes (the backslash characters only for presentation purposes).

The first key-value pair, `TestSuitName`, is completely optional and offers no functionality; its only goal is identifying what the object of the tests described in the present file is (in this case, service 0x29 is being tested). The `ID` pair sets the CAN IDs for the tests that will be executed in the file, and it is required. The user can use both Standard (11-bit) or Extended (29-bit); the first ID is the source ID and the second one is the destination ID.

The `Tests` array contains objects that, in turn, contain only a `Sequence` array. This last array is where an actual individual test is described and may start with the optional `TestName` key-value pair that identifies the test. Following it, comes however many `PairArray` pairs are needed to execute the desired test sequence. Each `PairArray` contains a request message and a response message, the division between the two being marked by the "Response" string. All values are expressed as strings, and besides the standard 1-byte hexadecimal value (e.g 0x2A), each item in a `PairArray` can use the operators described in Table 3.1.

3.5 Test Generator

In this section, the test generator will be looked at in detail, since it does represent the main part of this work. Following the flow represented in Figure 3.6, the test generation starts off by the actor defined in `test_gen.rs` (with the same logic as explained in the Subsection 3.3.2) calling the `exec_test()`, defined in the `testgen_util.rs` file (contents fully laid out in appendix A). This function takes a `file_path` argument, which is a `String` of a path that points to the JSON file describing the tests. The contents of the file are then parsed using `serde` (Subsection 2.3.3) as shown in Listing 3.8.

```

1 let mut file = File::open(file_path).expect("Couldn't open test file");
2 let mut contents = String::new();
3 file.read_to_string(&mut contents)?;
4 let json: Value = serde_json::from_str(&contents)?;

```

Listing 3.8: Code snippet of using `serde` to parse the JSON file.

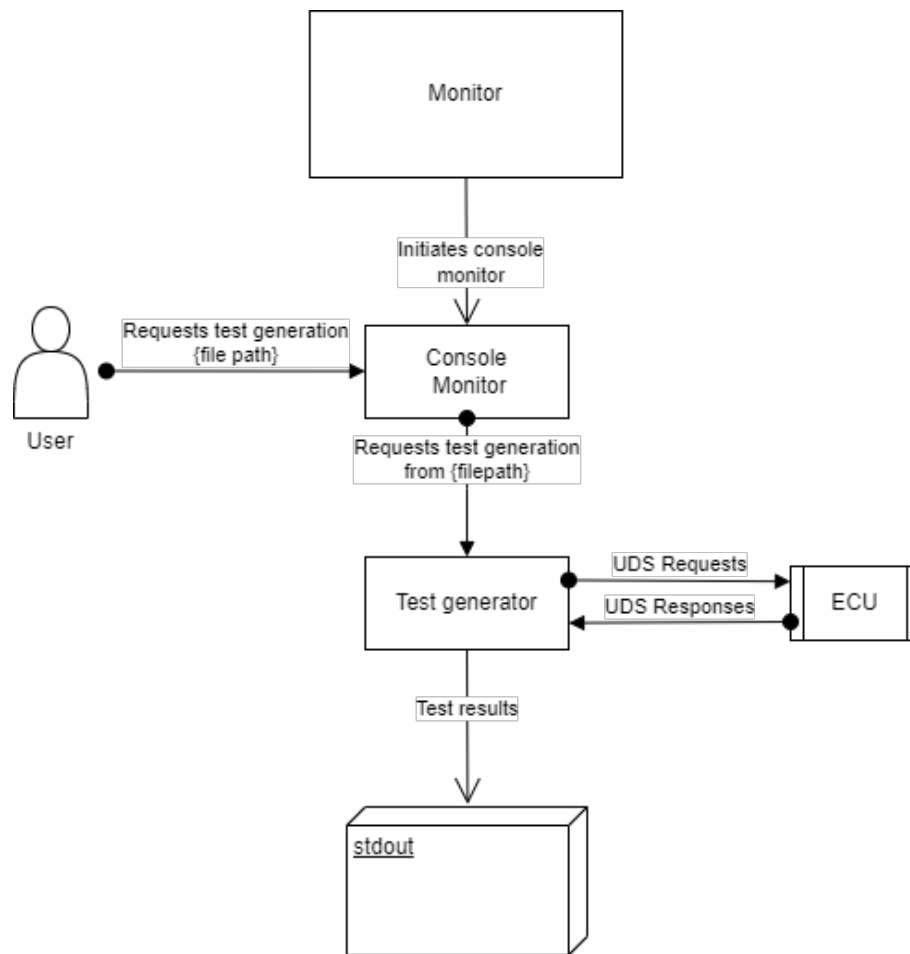


Figure 3.6: Test generator flow.

Table 3.1: Operators available with the defined syntax .

Operator	Description	Example
RANGE(min, max)	This operator can only be used in the Response part of a PairArray, and allows expecting any number that fits between the min and max limits in a given index of an incoming message	"RANGE(0x03,0x0A)"-> this would allow any value between 0x03 and 0x0A

Operator	Description	Example
	This operator can only be used in the Response part of a <code>PairArray</code> , and allows expecting multiple values in a given index of an incoming message. It can also be used in conjunction with the <code>RANGE()</code> operator, as in the example	<code>"0x01 0x02 RANGE(0x03,0x0A)"</code> -> this is the equivalent to accepting values from 0x01 to 0x0A
FILE(file_path)	This operator can only be used in the Request part of a <code>PairArray</code> , and channels the contents of the file that <code>file_path</code> points to. This is particularly useful in order to use, for example, a certificate file while trying to authenticate.	<code>"FILE(/certificate_xpto.der)"</code> -> loads the contents of the file specified in <code>certificate_xpto.der</code> into the to-be transmitted message
LEN(VAR)/VAR	This operator can only be used in the Response part of a <code>PairArray</code> , and allows to save a variable <code>VAR</code> for later use in the sequence. This is particularly helpful when the tester is expecting unknown length data from the ECU and allows for later use as well. The use of this kind of syntax is possible given the way fields work on UDS. If an ECU wants, for example, to send a challenge for the client to solve, then there are two bytes describing the size of the challenge and the challenge is transmitted after	<code>.."LEN(CHA)","CHA"</code> -> this would capture a message of size <code>LEN(CHA)</code> to the variable <code>CHA</code>

Operator	Description	Example
LEN(RES(VAR, priv_key_path)) / RES(VAR)	This pair of operators can only be used in the Request part of a <code>PairArray</code> , must be used together and allows signing a challenge <code>VAR</code> provided by the ECU that will prove ownership of the tester. This can be used in the context of the challenge discussed in the previous row. The ECU provides a challenge, it gets saved in variable <code>VAR</code> and signed using the private key that <code>priv_key_path</code> points to. <code>LEN(RES(VAR))</code> represents the length in bytes of the signature <code>RES(VAR)</code> in 2 bytes.	.." <code>LEN(RES(CHA))</code> "," <code>RES(CHA)</code> ","..-> this would solve and transmit the solution to challenge <code>CHA</code>

This parse results in a `json` variable of type `serde_json::Value`, an enum that contains 6 variants: `Value::Null`, `Value::String`, `Value::Number`, `Value::Array`, `Value::Object`, and `Value::Bool`. Given that the array variant is an array of `serde_json::Values` and the object variant is a map that pairs a string key and a value, recursion was used in order to iterate through the JSON, in function `process_json()`. It's also in this function that the `ID` key-value pair is processed: depending on the kind of IDs the user chooses, they are constructed and assigned to variables that were declared before starting to process the JSON (in the `exec_test()` function, for lifetime reasons).

The `process_sequence()` is called whenever a `Sequence` array is detected in the JSON, and starts by initiating a variable of type `HashMap` that will save variables for the entire duration of the sequence. Both the sending and receiving CAN sockets are also opened, using the `src` and `dest` IDs assigned earlier. Next, each `PairArray` in the sequence is divided in both the request and the response sections by using the "Response" value as the delimiter, feeding each one to the `process_request()` and `process_response()` functions, respectively.

Function `process_response` returns a `bool` that represents the result of the `PairArray` that is being exchanged, and takes as an argument the expected message as well as the variables `HashMap` and the receiving socket. A false return from the function deems the entire test as having failed, even if there are more `PairArrays` to go through. The first action the function takes is to receive the incoming response to the request exchanged prior. As shown in Listing 3.9, there is a

loop that enables filtering unwanted messages, such as the Negative Response Code 78 that is sent by the ECU to inform the tester that it is still processing the request.

```

1  let mut message: Vec<u8>;
2  loop {
3      message=receive_isotp_frame(socket).await?;
4
5      if message[0]==127 && message[2]==120 {continue;} //ignore NRC 78
6          case
7      break;
    }

```

Listing 3.9: Code snippet that implements receiving the response message from the ECU.

After receiving the message from the ECU, the function reads the expected message from the JSON in order to compare both. This is done by iterating through the values of the response section of the `PairArray`. For each value two variables are created:

- `acceptable_values`, a vector of byte-sized integers that will hold the possible values in the respective position that won't deem the test as failed;
- `options`, a vector of `Strings` that will hold the possible options, if the user used the `'|'` operator. For example, a value of `"0x29|Range(0x30,0x31)"` would result in `options[0]="0x29"` and `options[1] = "Range(0x30,0x31)"`.

Splitting of the options is achieved with the code represented in Listing 3.10

```

1  if value.contains('|') {
2      let value = value.replace(" ", "");
3      let values= value.split("|").collect_vec();
4      for (j,val) in values.iter().enumerate() {
5          options.push(val[j*4+3..j*4+2].to_owned());
6      }
7  }
8  else {options.push(value);}

```

Listing 3.10: Code snippet of the splitting of options applied when the `'|'` operator is used.

When related to the `options` vector processing, it's important to highlight the case where the user uses the `LEN()` operator. Demonstrated in Listing 3.11, the code extracts the variable name, and gets the size of the variable from the response message. Having the length of the message, the message itself is also extracted and pushed into the `variables` `HashMap`, where it will stay available for the rest of the sequence.

```

1  else if option.starts_with("LEN(") {
2      let map_key = option[4..option.len()-1].to_owned();
3      let var_len=(message[i] as u16)<<8 | (message[i+1] as u16);
4

```

```

5         let var_vec = message[i+2..i+2+var_len as usize].to_owned();
6         i+=2+var_len as usize;
7
8         variables.insert(map_key, var_vec);
9
10        continue 'mainloop;
11    }

```

Listing 3.11: Code snippet demonstrating the processing of LEN() operator.

Function `process_request()` returns no data, and takes as an argument the String values that the user input in the JSON, as well as the variables `HashMap` and the sending socket. The function starts by allocating a vector of 8-byte unsigned integers, on which the bytes that are going to be transmitted will be pushed. Then a `for` loop is initiated that will convert the String values from the JSON into values that will be pushed in the vector.

Listing 3.12, shows how the `FILE()` operator is handled. The program simply verifies if the provided path is valid and, if it is, its contents are extracted and appended onto the final vector.

```

1     else if value.starts_with("FILE("){
2         let value_len=value.chars().count();
3         if !Path::new(&value[5..value_len-1]).is_file() {
4             panic!("{}", isn't a file", &value[5..value_len-1]);
5         }
6
7         let mut cert_vec =
8             fs::read(&value[5..value_len-1]).expect("failed to read
9             certificate");
10        can_frame_vec.append(&mut cert_vec);
11    }

```

Listing 3.12: Code snippet demonstrating the processing of the FILE() operator.

In order to sign a challenge, prompt by the `LEN(RES(..))/RES(..)` pair of operators, the program creates a temporary file on which the challenge, contained in a variable chosen by the user, is dumped. This file will be used to execute the OpenSSL command in Listing 3.13, which will output a file `signature.bin` containing the signed challenge.

```

1     openssl_cmd.args(&[
2         "dgst",
3         "-sha256",
4         "-sign",
5         priv_key_path,
6         "-out",
7         "./signature.bin",
8         "./challenge"
9     ])

```

```
10 .output().expect("Couldn't execute openssl command");
```

Listing 3.13: Code snippet demonstrating the OpenSSL command for signing a challenge.

After extracting the signature from `signature.bin`, and appending itself and its length to the final vector, both the signature file and the temporary challenge file are deleted, as shown in Listing 3.14

```
1 let mut signature: Vec<u8> =
2     fs::read("signature.bin").expect("Couldn't read signature");
3
4 let len_var = signature.len() as u16;
5
6 can_frame_vec.push(((len_var & 65280)>>8) as u8);
7 can_frame_vec.push((len_var & 255) as u8);
8 can_frame_vec.append(&mut signature);
9
10 fs::remove_file("signature.bin"?;
11 fs::remove_file("challenge"?;
```

Listing 3.14: Code snippet demonstrating appending the signed challenge.

In order to send the requests, the `socketcan-isotp` library was being used, however, after trying to transmit a message that had more than 83 bytes, it was noticed that the message was truncated and not being fully delivered. After much investigation, it was understood that the speed at which the program was sending the CAN frames was overloading the buffers, so no more than 11 consecutive frames would go through. In order to go around this issue, and because the program was abstracted from the ISO-TP layer, manual implementation of the protocol was hard coded, in which mandatory delays were set. Since in development, frame control frames don't have much importance, it was assumed the ECU wouldn't force any specific separation time between frames.

```
1 if can_frame_vec.len()>7 { //multi frame case
2     let byte1 = ((1 as u8)<<4) | (((can_frame_vec.len() as u16) &
3         0xf00)>>8) as u8);
4     let byte2 = ((can_frame_vec.len() as u16) & 0x0ff) as u8;
5     let mut dummy: Vec<u8> = vec![byte1, byte2];
6     dummy.extend_from_slice(&can_frame_vec[0..6]);
7     let frame=CANFrame::new(*id, dummy.as_slice(), false,
8         false).expect("Couldn't construct SF");
9     send_can_frame(socket, frame).await;
10
11     std::thread::sleep(std::time::Duration::from_millis(5));
12
13     let n_cfs = ((can_frame_vec.len()-6) as u16)/7+1; //number of
14         consecutive frames to be sent
15     for i in 0..n_cfs{
16         let frame_index = if i % 16 == 15 { 0 } else { (i % 16) + 1 };
```



```

14
15     let mut dummy:Vec<u8> = vec![0x20+frame_index as u8];
16     if i==n_cfs-1{//last frame
17         dummy.extend_from_slice(&can_frame_vec[6+7*i as usize..]);
18         while dummy.len()<8 {
19             dummy.push(0 as u8);
20         }
21     }
22     else {
23         dummy.extend_from_slice(&can_frame_vec[6+7*i as usize..13+7*i
24             as usize]);
25     }
26
27     let frame = CANFrame::new(*id, dummy.as_slice(), false,
28         false).expect("Couldn't construct CF");
29     send_can_frame(socket, frame).await;
30     std::thread::sleep(std::time::Duration::from_millis(10));
31 }
32
33 else{ //single frame case
34     let mut dummy: Vec<u8> = vec![can_frame_vec.len() as u8];
35     dummy.append(&mut can_frame_vec);
36
37     while dummy.len()<8 {dummy.push(0 as u8)}
38     let frame = CANFrame::new(*id, dummy.as_slice(), false,
39         false).expect("Couldn't mount single frame");
40     send_can_frame(socket, frame).await;
41 }

```

Listing 3.15: Code snippet that demonstrates manual wrapping done for ISO-TP compliance.

Listing 3.15 demonstrates the process behind manually wrapping and dividing a CAN message for ISO-TP transmission. In case the message is 7 bytes or less, a simple single frame will be sent, just with the number of bytes in the first byte and the rest in the following bytes, padding with zeros if needed. If a message is more than 7 bytes, then a first frame will be sent, followed by however many consecutive frames are needed. A delay of 5 ms was introduced after the first frame, and of 10 ms after every consecutive frame, in order to ensure no overloading of the buffers.

3.6 Summary

This chapter described the process behind the development and had the objective of providing a general understanding of the technical work developed. In terms of hardware, the setup used in the "Debug Phase" allowed for a high degree of control and monitoring, whilst the "Development Phase"s allowed for testing with a real target in real conditions. The implemented architecture,

being based in the actor model, represented a highly flexible and modular option, perfect for this application. These same characteristics were of the utmost importance when defining the syntax for the description of the tests, with the special operators accommodating potential special requirements for the testing. The test generator, by interpreting the JSON file, and wrapping it according to the ISO-TP standard, allows for the execution of the described tests.

Chapter 4

Validation

This chapter focuses on the validation of the proposed tool, with the research questions defined in Section 1.5 in mind. The goal is to verify if the tool achieved the objectives and can indeed be a valuable resource for real-world applications. Firstly, an overview of the methodology used for validating the tool is presented. The results of the application of that methodology are then presented, analyzed, and discussed.

4.1 Methodology

In order to understand if the tool can reliably test the implementation of an UDS stack, the template in Appendix B was fed to the test generator with two different configurations flashed in the target ECU. This should help understand if the tests are being correctly executed.

To understand the time efficiency of the tool when compared to the manual way of testing, three professionals from CES volunteered to execute the same tests in two different scenarios: describing them in a JSON file using the proposed syntax and feeding it to the developed tool vs manually using Vector CANoe. The pool of volunteers consisted in software engineers that were integrated in the cybersecurity team. All responded to a form (represented in Figure 4.1) where they self-evaluated with respect to Vector CANoe and UDS knowledge, in a 0 to 5 Likert scale (0 being no experience and 5 extensive everyday usage).

It's important to note that none of the volunteers had any previous contact with the proposed tool, and it was their first time using it. For this reason, a guide (fully laid out in Appendix C) was elaborated that contained the tests to be executed (common to both scenarios), a tutorial on the JSON test definition syntax and the process for manual testing in Vector CANoe. Volunteers were then timed using a chronometer from the moment they started writing the tests until the moment they were fully executed and analyzed.

Whilst testing with the proposed tool consisted in describing the tests in the JSON and simply feeding them to the tool, manual testing was less static. The volunteers had to manually write and send the packets in Vector CANoe's console and analyze the responses. They also needed to extract challenges, sign them and write them back in CANoe manually.

Your name and employee identifier

A sua resposta _____

What would you say is your degree of expertise in CANoe?

0 1 2 3 4 5

Never used it Use it everyday extensively

How would you evaluate your knowledge in UDS

0 1 2 3 4 5

Never used it Use it everyday extensively

Choose the days you are available (I'll contact you after to discuss the time)

21/06

22/06

23/06

24/06

25/06

26/06

27/06

Figure 4.1: Form to be filled by the volunteers before the validation.

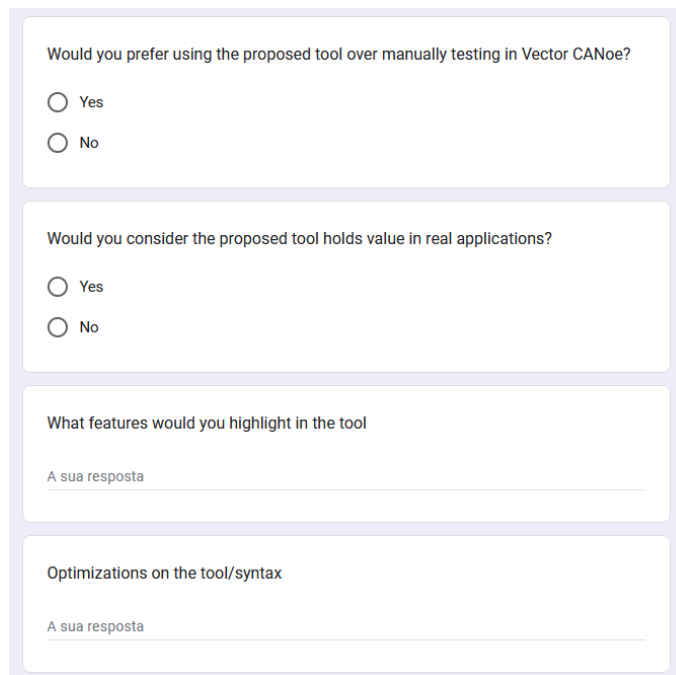
Susceptibility to error was evaluated by analyzing the subjects during both test scenarios. Errors in the context of the proposed tool are related to the syntax of the tool or mistyping, while in the context of Vector CANoe may be related to wrongful conclusions of the outcome of a test or mistyping.

Volunteers also filled out a form after the procedure (represented in Figure 4.2), where they expressed their educated opinion on the tool and syntax, their usability in real applications, optimizations that could be done and highlights.

4.2 Results

4.2.1 Template Execution

The first execution with the template JSON file describing the tests involved an ECU that was flashed with what was thought as a correct implementation of UDS. The output of the tool, shown in Subfigure 4.3a demonstrates an accidentally failed test. Every test until the `HappyPath` passed; however, the `HappyPath`, which proved to work before in standalone testing, failed due to a



Would you prefer using the proposed tool over manually testing in Vector CANoe?

Yes

No

Would you consider the proposed tool holds value in real applications?

Yes

No

What features would you highlight in the tool

A sua resposta

Optimizations on the tool/syntax

A sua resposta

Figure 4.2: Form filled by the volunteers after the validation.

timeout. The ECU wasn't responding back, and was stuck at sending a Negative Response Code (NRC) 78 that signals the ECU is busy. After further investigation, this issue proved to be ECU-sided, and although it isn't solved as of the writing of this document, it is believed the cause was the forcing of errors done before the happy path and sent the ECU into a blocking state, since the second execution described next worked flawlessly.

The second execution used a revised version of the template where the `Happy Path` sequence was being executed first and, as shown in Subfigure 4.3b, all tests passed. The ECU still blocked for further correct requests, but since the `Happy Path` sequence was before the error-inducing sequences, the ECU responded correctly.

The third execution involved an incorrect implementation of UDS where the NRCs returned by the ECU aren't the expected in the `Minimum Length Check` and `Invalid Certificate Check` tests. Figure 4.4 demonstrated the expected behavior: a failed `Minimum Length Check` test and a failed `Invalid Certificate Check` test.

```

[2023-06-23T14:08:51Z INFO caravel] Starting runtime
[2023-06-23T14:08:51Z INFO caravel::actors::monitor] Running
[2023-06-23T14:08:51Z INFO caravel::actors::ctrlc] Running
[2023-06-23T14:08:51Z INFO caravel::actors::receiver_can] Running
[2023-06-23T14:08:51Z INFO caravel::actors::sender_can] Running
[2023-06-23T14:08:51Z INFO caravel::actors::test_gen] Running TestGen
[2023-06-23T14:08:51Z INFO caravel::actors::stdin] Running
gen-test test_example.json
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Initiating tests to 0x29 Service
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Initiating test Minimum Length Error Check
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Passed Minimum Length Error Check
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Initiating test Total Length Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Passed Total Length Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Initiating test Sequence Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Passed Sequence Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Initiating test Invalid Certificate Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Passed Invalid Certificate Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Initiating test Happy Path

```

(a) Accidentally failed test execution (happy path after error-inducing tests).

```

[2023-06-23T14:08:51Z INFO caravel] Starting runtime
[2023-06-23T14:08:51Z INFO caravel::actors::monitor] Running
[2023-06-23T14:08:51Z INFO caravel::actors::ctrlc] Running
[2023-06-23T14:08:51Z INFO caravel::actors::receiver_can] Running
[2023-06-23T14:08:51Z INFO caravel::actors::sender_can] Running
[2023-06-23T14:08:51Z INFO caravel::actors::test_gen] Running TestGen
[2023-06-23T14:08:51Z INFO caravel::actors::stdin] Running
gen-test test_example.json
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Initiating tests to 0x29 Service
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Initiating test Minimum Length Error Check
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Passed Minimum Length Error Check
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:01Z INFO caravel::util::testgen_util] Initiating test Total Length Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Passed Total Length Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Initiating test Sequence Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Passed Sequence Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Initiating test Invalid Certificate Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Passed Invalid Certificate Error Check
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:09:03Z INFO caravel::util::testgen_util] Initiating test Happy Path

```

(b) Successful template execution (happy path before error-inducing tests).

Figure 4.3: Executions on target with correctly implemented UDS stack.

```

[2023-06-23T14:40:06Z INFO caravel] Starting runtime
[2023-06-23T14:40:06Z INFO caravel::actors::monitor] Running
[2023-06-23T14:40:06Z INFO caravel::actors::ctrlc] Running
[2023-06-23T14:40:06Z INFO caravel::actors::receiver_can] Running
[2023-06-23T14:40:06Z INFO caravel::actors::sender_can] Running
[2023-06-23T14:40:06Z INFO caravel::actors::stdin] Running
[2023-06-23T14:40:06Z INFO caravel::actors::test_gen] Running TestGen
gen-test test_example.json
[2023-06-23T14:40:18Z INFO caravel::util::testgen_util] Initiating tests to 0x29 Service
[2023-06-23T14:40:18Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:40:18Z INFO caravel::util::testgen_util] Initiating test Happy Path
[2023-06-23T14:40:20Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:40:20Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:40:20Z INFO caravel::util::testgen_util] Passed Happy Path
[2023-06-23T14:40:20Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:40:20Z INFO caravel::util::testgen_util] Initiating test Minimum Length Error Check
[2023-06-23T14:40:20Z INFO caravel::util::testgen_util] Messages didn't match
[2023-06-23T14:40:20Z INFO caravel::util::testgen_util] Failed Minimum Length Error Check
[2023-06-23T14:40:20Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:40:20Z INFO caravel::util::testgen_util] Initiating test Total Length Error Check
[2023-06-23T14:40:22Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:40:22Z INFO caravel::util::testgen_util] Passed Total Length Error Check
[2023-06-23T14:40:22Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:40:22Z INFO caravel::util::testgen_util] Initiating test Sequence Error Check
[2023-06-23T14:40:22Z INFO caravel::util::testgen_util] Messages matched
[2023-06-23T14:40:22Z INFO caravel::util::testgen_util] Passed Sequence Error Check
[2023-06-23T14:40:22Z INFO caravel::util::testgen_util] Starting to process sequence
[2023-06-23T14:40:22Z INFO caravel::util::testgen_util] Initiating test Invalid Certificate Error Check
[2023-06-23T14:40:23Z INFO caravel::util::testgen_util] Messages didn't match
[2023-06-23T14:40:23Z INFO caravel::util::testgen_util] Failed Invalid Certificate Error Check

```

Figure 4.4: Failed template execution with the correct implementation of UDS.

4.2.2 Timed comparison

Regarding the timed comparison between volunteers executing the same tests describing them in a JSON and inputting them in the proposed tool vs manually in Vector CANoe, the obtained results are presented in Table 4.1. The associated self-evaluated UDS and Vector CANoe expertise grade is also represented.

Table 4.1: Time comparison between the execution of the same test routine with the proposed tool and syntax vs with Vector CANoe.

Volunteer number	Proposed tool time to execution (in minutes:seconds)	Vector CANoe time to execution (in minutes:seconds)	Knowledge of UDS (0-5)	Expertise of Vector CANoe
1	14:43 (1 error)	16:05 (3 errors)	2	1
2	12:21 (0 errors)	14:32 (2 errors)	4	3
3	11:37 (0 errors)	14:41 (0 errors)	5	4

4.3 Discussion

From the first three executions, it's possible to conclude that the tool was successful at fulfilling its main objective of reliably testing the UDS stack and ended up uncovering a bug in the ECU's implementation. It's important to keep in mind this was an ECU being developed for a real project, so this finding ended up being useful for the development team. After a change to the ECU's configuration, the tool also detected failed tests and faulty NRC responses, indicating that the implementation wasn't as expected.

Analyzing Table 4.1, an advantage is evident on the proposed tool's side, in terms of time efficiency and errors. The proposed tool reduced the time to execute the same routine of tests by an average of 15.3%, a significant reduction. Between volunteers 1 and 3, a reduction of 21% in the time needed to execute the tests with the proposed tool was detected, most likely due to the increase in self-evaluated expertise of the UDS protocol. This leads to the belief that knowledge of UDS helps users understand the somewhat complex nature of some of the operators defined in the proposed syntax.

Also evident is a reduction of errors, with the tool obtaining a result of 80% fewer errors, on average. While describing the tests with the proposed tool, the only error encountered was misusing operators such as the `LEN(..)/.` or the `LEN(RES(..))/RES(..)`; mistyping was not found to happen. In CANoe, the errors were 80% mistyping byte values, and 20% originated from not understanding the underlying UDS logic.

All in all, the advantages of the proposed tool are clear, and from the feedback received from the volunteers in the post-validation form, all participants agreed on the value of the proposed tool and would use it instead of Vector CANoe for testing UDS. Features highlighted by the volunteers were the ease of testing, not needing third-party tools, easy reproducibility, and how visually clearer the tests are. Volunteers pointed out that some parts of the syntax could be optimized;

this is discussed in Section 5.3. Volunteers also stated that the use cases are evident and prove themselves much clearer after the first use, which can be another indicator of space to make the syntax better.

The validation methods used in this work determined the fulfillment of the objectives, however, the need to test with real hardware meant only collaborators from CES could validate the work developed, due to confidentiality concerns with the ECUs. This ultimately resulted in a small sample of volunteers, a consequence of the big time investment needed to complete the validation of the proposed tool, and could have limited the accuracy of the results.

4.4 Summary

Validation was split into two main parts: template execution and timed execution with volunteers. The first tested the provided template (Appendix B) on two different ECU configurations, one correctly implemented and one wrongly implemented. After finding an ECU-sided bug in the (supposedly) correct implementation, a revised version of the template allowed the tests to pass anyways. The team responsible for the development of the ECU was informed of the unexpected behavior. The wrongly implemented ECU provoked two failed tests, as expected. This part confirmed the intended behavior of the tool in detecting wrong implementations of the stack.

The timed execution involved three volunteers from CES' cybersecurity team executing the same tests with the proposed tool and syntax vs manually using Vector CANoe. After answering a form where they self-evaluated on UDS and Vector CANoe knowledge, the volunteers read a guide (Appendix C) that contained the information needed to execute the validation (a tutorial on the tool's syntax, on the manual testing in CANoe and the intended tests) and performed the tests in both scenarios. They were timed from the moment they started testing until all tests were concluded and analyzed. The proposed tool reduced by 15.3% the average time to execute the same routine of tests. Volunteer 3 took 21% less time to execute the tests with the proposed tool than volunteer 1, which is thought to be tied to the higher UDS knowledge (5 vs 2 in the 0-5 scale). The proposed tool also resulted in 80% fewer errors.

Post-validation feedback received from the volunteers established the value of the tool in terms of real-world usability and preference over manual testing with Vector CANoe. Volunteers highlighted the tool's ease of testing, independence from third-party tools and visual clarity as strong points. Optimizations to the syntax were also suggested and discussed in Section 5.3.

Limitations in the validation arose from the need to test with collaborators from CES for confidentiality purposes and the big time investment needed to complete the validation. This resulted in a small sample that could have affected the results.

Chapter 5

Conclusions

Throughout this document, the work that culminated in a functional tool for UDS implementation following a semi-automated approach was described. In the introduction, after discussing the context of the dissertation, the motivation that led to the development of this dissertation was presented. After describing the problem to tackle, the objectives for the work were defined. Also, a brief presentation of the company that enabled the development of this dissertation, Continental Engineering Services, was delivered.

The literature review chapter aimed to provide some background on which to cling for the more technical chapters to follow. It began by providing some knowledge of ECUs and communication between them, especially CAN bus. Crucial standards such as ISO 15765-2 and ISO 14229-1 were examined. The main development tools were discussed, such as the programming language used, Rust, and its libraries. A short introduction to JSON was also included. Finally, some related work was mentioned and analyzed.

The implementation chapter provided a comprehensive view of the process of implementing the proposed tool. The hardware setup used in both stages of development (debug and deployment) was described, followed by a look into the cross-compilation setup. In the software backbone section, the architecture employed in the construction of the software (based in the actor model) was detailed, and its implementation analyzed. Finally, after a description of the syntax designed for defining the tests, the code behind the test generator was examined in detail.

In the validation chapter, the methodology used to validate the work developed was described. It consisted in testing the proposed tool with different ECU configurations as well as timing volunteers at executing the same routine of tests using the tool vs manually using Vector CANoe. The results obtained are discussed in Section 5.1.

5.1 Summary of the results

The results obtained in the validation process suggest that the work was indeed successful. While testing the tool, a malfunction was found in the target ECU's implementation of UDS, an ECU

that was being used in a real project. Also, different configurations on the target ECU resulted in correct evaluations from the tool.

Regarding RQ1, the time comparison between executing the same tests using the proposed tool vs manually using Vector CANoe showed the former achieving an average reduction of 15.3% in the time needed to execute the tests. It was concluded that in one case, greater knowledge of UDS resulted in a reduction of 21% in the time needed to execute the tests with the tool. This goes in line with what can be considered a down-fall of semi-automation: there is still human intervention.

Regarding RQ2, using the proposed tool resulted in 80% fewer errors. This was attributed by volunteers (in the post-validation form) to the static definition of the tests, which provided a clearer visualization environment of the tests described.

5.2 Applications and positive implications

The tool has the ability to facilitate testing for teams developing software for ECUs. Some of the advantages are the following:

1. Cost saving: Alternatives to the developed tool are very expensive. As mentioned previously, there isn't only the cost of the software itself, but also of the hardware on which the software is embedded. The software developed needs only a Raspberry Pi with a CAN interface installed.
2. Reliability: when compared to the previous testing solution, the use of the developed tool, by requiring less effort and being more easily reproduced, poses as a more reliable alternative.
3. Speed: the tool developed offers an advantage in terms of time efficiency, by providing an interface (the JSON describing the tests) that's easier to interact with. Reproducibility is also an important factor.
4. Flexibility feature-wise: having an in-house tool makes sure it fits the team's needs. In case a feature is lacking in the tool, it can be added. The downside of time consumption in developing those new features is obvious, however, the tool was built in order to facilitate the introduction of new features.

5.3 Future work

Future work would involve optimizing the proposed syntax to make it clearer. Although the designed syntax produced results that indicated less susceptibility to errors and increased speed in executing the tests, there are optimizations possible. Volunteers indicated the `PairArray` key-value pair as being a confusing and hard-to-read design option, and would rather have two independent arrays, one for the request and one for the expected response.

Another suggestion was the inclusion of a graphical user interface to describe the tests. This would result in an even clearer interface, that could include details of the operators, possibly

producing even better results. However, given the development nature of this tool and its relatively limited use, a better command line would probably suffice without requiring too much effort.

Appendix A

File testgen_util.rs

```
1 use std::collections::HashMap;
2 use std::fs::{File, self};
3 use std::process::{Command, Stdio};
4 use hex;
5 use itertools::Itertools;
6 use std::io::{Read, Write};
7 use std::path::{Path};
8 use anyhow::Result;
9 use log::{info, error, debug};
10 use serde_json::{self, Value};
11 use async_recursion::async_recursion;
12
13
14 use super::canutil::{send_isotp_frame, IsoTpSocket, ExtendedId, StandardId, Id,
15     receive_isotp_frame, FlowControlOptions, send_can_frame, CANSocket, CANFrame};
16
17 pub async fn exec_test(file_path: String) -> Result<()>{
18     let mut file = File::open(file_path).expect("Couldn't open test file");
19     let mut contents = String::new();
20     file.read_to_string(&mut contents)?;
21
22     let json: Value = serde_json::from_str(&contents)?;
23
24     let mut dest:u32=0;
25     let mut src_isotp: Id;
26     let mut dest_isotp: Id;
27     if let Some(struc) = StandardId::new(0){
28         src_isotp=Id::Standard(struc);
29         dest_isotp=Id::Standard(struc.clone());
30     } else {panic!("Couldn't create ids");}
```

```

31 process_json(json, None, &mut src_isotp, &mut dest_isotp, &mut dest).await?;
32
33
34 Ok(())
35 }
36
37 #[async_recursion]
38 async fn process_json(json: Value, key_op: Option<String>, src_isotp: &mut Id,
39 dest_isotp: &mut Id, dest:&mut u32) -> Result<()>{
40 match json {
41     Value::String(value) => {
42         if let Some(key) = key_op {
43             match key.as_str() {
44                 "TestSuiteName" => {
45                     info!("Initiating tests to {value}");
46                 }
47                 "ID" => {
48                     let ids= value.split(',').collect_vec();
49                     *dest=u32::from_str_radix(&ids[2][2..],16).unwrap();
50                     match ids[0] {
51                         "Extended" => {
52                             let src_struct_opt =
53                                 ExtendedId::new(u32::from_str_radix(&ids[1][2..],16)\
54                                     .unwrap());
55                             if let Some(src_struct) = src_struct_opt {
56                                 *src_isotp= Id::Extended(src_struct);
57                             } else {panic!("Panicked creating id from {}", ids[1])}
58                         }
59                         let dest_struct_opt =
60                             ExtendedId::new(u32::from_str_radix(&ids[2][2..],16)\
61                                     .unwrap());
62                         if let Some(dest_struct) = dest_struct_opt {
63                             *dest_isotp = Id::Extended(dest_struct);
64                         } else {panic!("Panicked creating id from {}", ids[2])}
65                     }
66                 "Standard" => {
67                     let src_struct_opt =
68                         StandardId::new(u16::from_str_radix(&ids[1][2..],16)\
69                             .unwrap());
70                     if let Some(src_struct) = src_struct_opt {
71                         *src_isotp= Id::Standard(src_struct);
72                     } else {panic!("Panicked creating id from {}", ids[1])}

```



```
71         let dest_struct_opt =
72             StandardId::new(u16::from_str_radix(&ids[2][2..],16)\
73             .unwrap());
74         if let Some(dest_struct) = dest_struct_opt {
75             *dest_isotp = Id::Standard(dest_struct);
76         } else {panic!("Panicked creating id from {}", ids[2])}
77     }
78     _ => {
79         panic!("Unknown ID type {}", ids[0]);
80     }
81 }
82 _ => {info!("Unknown key {}", key);}
83 }
84 }
85 }
86 Value::Array(values) => {
87     if let Some(key) = key_op{
88         match key.as_str(){
89             "Tests" => {
90                 for value in values{
91                     process_json(value, None, src_isotp, dest_isotp, dest).await?;
92                 }
93             }
94             "Sequence" => {
95                 process_sequence(values, src_isotp, dest_isotp,
96                     dest).await.expect("Failed at processing sequence");
97                 std::thread::sleep(std::time::Duration::from_millis(50));
98             }
99             _ => {}
100         }
101     }
102 }
103 Value::Object(obj) => {
104     for (key, value) in obj {
105         process_json(value, Some(key), src_isotp, dest_isotp, dest).await?;
106     }
107 }
108 _ => {}
109 }
110
111 Ok(())
112 }
113
```

```

114 async fn process_sequence(objects: Vec<Value>, src_isotp: &mut Id, dest_isotp: &mut
    Id, dest: &mut u32) -> Result<()> {
115     info!("Starting to process sequence");
116
117     let mut vars: HashMap<String, Vec<u8>> = HashMap::new();
118     let mut testname: String = "Placeholder Test Name".to_owned();
119
120     let req_socket = CANSocket::open("can0").expect("Failed to open request socket");
121     let mut res_socket = IsoTpSocket::open(
122         "can0",
123         src_isotp.to_owned(),
124         dest_isotp.to_owned(),
125     ).expect("Failed to open ISO-TP socket");
126
127     for object in objects {
128         if let Value::Object(obj) = object {
129             for (_key, value) in obj {
130                 if let Value::String(string) = value {
131                     info!("Initiating test {string}");
132                     testname=string;
133                 }
134                 else if let Value::Array(elems) = value {
135                     let mut divide: bool = false;
136                     let mut request_vec: Vec<String>=Vec::new();
137                     let mut response_vec: Vec<String>=Vec::new();
138
139                     for elem in elems {
140                         if let Value::String(string) = elem {
141                             if string == "Response" {
142                                 divide=true;
143                                 continue;
144                             }
145
146                             if divide {
147                                 response_vec.push(string);
148                             }
149                             else{
150                                 request_vec.push(string);
151                             }
152                         }
153                     }
154
155                     process_request(request_vec, &mut vars, &req_socket, dest).await?;
156                     let res = process_response(response_vec, &mut vars, &mut
                        res_socket).await?;

```

```
157         if !res {
158             info!("Messages didn't match");
159             info!("Failed {}", testname);
160             return Ok(());
161
162         }
163         else {
164             info!("Messages matched");
165         }
166
167     }
168 }
169 }
170 }
171 }
172 info!("Passed {}", testname);
173
174 Ok(())
175 }
176
177 async fn process_request(request_vec: Vec<String>, variables:&mut HashMap<String,
178     Vec<u8>>, socket: &CANSocket, id:&u32) -> Result<()>{
179     let mut can_frame_vec: Vec<u8>= Vec::new();
180     let mut len_next_flag = false;
181
182     for value in request_vec {
183         let value = value.replace(" ", "");
184
185         if value.starts_with("0x"){
186             can_frame_vec.push(u8::from_str_radix(&value[2..], 16).unwrap());
187         }
188         else if value.starts_with("FILE("){
189             let value_len=value.chars().count();
190             if !Path::new(&value[5..value_len-1]).is_file() {
191                 panic!("{}", isn't a file", &value[5..value_len-1]);
192             }
193
194             let mut cert_vec = fs::read(&value[5..value_len-1]).expect("failed to read
195                 certificate");
196             can_frame_vec.append(&mut cert_vec);
197         }
198         else if value.starts_with("LEN(RES(") {
199             let len_key = value.chars().count();
200             let pair = value[8..len_key-2].to_owned();
```

```

200     let values= pair.split(",").collect_vec();
201
202     let challenge = variables.get(values[0]).unwrap();
203     let priv_key_path = values[1];
204     if !Path::new(&priv_key_path).is_file() {
205         panic!("{}", isn't a file", &priv_key_path);
206     }
207
208     let mut file = File::create("challenge").unwrap();
209     file.write_all(challenge.as_slice())?;
210
211     let mut openssl_cmd = Command::new("openssl");
212     openssl_cmd.args(&[
213         "dgst",
214         "-sha256",
215         "-sign",
216         priv_key_path,
217         "-out",
218         "./signature.bin",
219         "./challenge"
220     ])
221     .output().expect("Couldn't execute openssl command");
222
223     let mut signature: Vec<u8> = fs::read("signature.bin").expect("Couldn't
224         read signature");
225
226     // dbg!("signature is {}", signature.len());
227
228     let len_var = signature.len() as u16;
229
230     if len_next_flag {
231         // dbg!("Introducing length of next field");
232         let len_nextfield = (2+len_var).to_be_bytes();
233         can_frame_vec.extend_from_slice(&len_nextfield);
234         len_next_flag=false;
235     }
236     can_frame_vec.push(((len_var & 65280)>>8) as u8);
237     can_frame_vec.push((len_var & 255) as u8);
238
239     can_frame_vec.append(&mut signature);
240
241
242     fs::remove_file("signature.bin")?;
243     fs::remove_file("challenge")?;

```

```

244     }
245     else if value.starts_with("LEN_NEXT") {
246         len_next_flag=true;
247     }
248 }
249
250
251 //println!("Request frame: {:?}", can_frame_vec);
252 // send_isotp_frame(socket, can_frame_vec.as_slice()).await;
253 if can_frame_vec.len()>7 { //multi frame case
254     let byte1 = ((1 as u8)<<4) | (((can_frame_vec.len() as u16) & 0xf00)>>8) as
        u8);
255     let byte2 = ((can_frame_vec.len() as u16) & 0x0ff) as u8;
256     let mut dummy: Vec<u8> = vec![byte1, byte2];
257     dummy.extend_from_slice(&can_frame_vec[0..6]);
258     let frame=CANFrame::new(*id, dummy.as_slice(), false, false).expect("Couldn't
        construct SF");
259     send_can_frame(socket, frame).await;
260
261     std::thread::sleep(std::time::Duration::from_millis(5));
262
263     let n_cfs = ((can_frame_vec.len()-6) as u16)/7+1; //number of consecutive
        frames to be sent
264     for i in 0..n_cfs{
265         let frame_index = if i % 16 == 15 { 0 } else { (i % 16) + 1 };
266
267         let mut dummy:Vec<u8> = vec![0x20+frame_index as u8];
268         if i==n_cfs-1{//last frame
269             dummy.extend_from_slice(&can_frame_vec[6+7*i as usize..]);
270             while dummy.len()<8 {
271                 dummy.push(0 as u8);
272             }
273         }
274         else {
275             dummy.extend_from_slice(&can_frame_vec[6+7*i as usize..13+7*i as
                usize]);
276         }
277
278         let frame = CANFrame::new(*id, dummy.as_slice(), false,
            false).expect("Couldn't construct CF");
279         send_can_frame(socket, frame).await;
280         std::thread::sleep(std::time::Duration::from_millis(10));
281     }
282 }
283 else{ //single frame case

```

```

284     let mut dummy: Vec<u8> = vec![can_frame_vec.len() as u8];
285     dummy.append(&mut can_frame_vec);
286
287     while dummy.len()<8 {dummy.push(0 as u8)}
288     let frame = CANFrame::new(*id, dummy.as_slice(), false,
289         false).expect("Couldn't mount single frame");
290     send_can_frame(socket, frame).await;
291 }
292 Ok(())
293 }
294
295 async fn process_response(response_vec: Vec<String>, variables:&mut HashMap<String,
296     Vec<u8>>, socket: &mut IsoTpSocket) -> Result<bool>{
297     // debug!("Reached response");
298     let mut message: Vec<u8>;
299     loop {
300         message=receive_isotp_frame(socket).await?;
301
302         if message[0]==127 && message[2]==120 {continue;} //ignore NRC 78 case
303         break;
304     }
305
306     //debug!("Received message: {:?}", message);
307     ////////////////Testing////////////////////////////////////
308     // let message: Vec<u8> = vec![105, 3, 17, 0, 2, 1, 1, 0, 0];
309     // // let message = message_vec.as_slice();
310     let mut skip_iter:bool=false;
311     let mut i=0;
312     'mainloop: for value in response_vec {
313         let mut acceptable_values: Vec<u8> = Vec::new();
314         let mut options: Vec<String> = Vec::new();
315
316         if skip_iter {skip_iter=false; continue;}
317
318         if value.contains('|') {
319             let value = value.replace(" ", "");
320
321             let values= value.split("|").collect_vec();
322             for (j,val) in values.iter().enumerate() {
323                 options.push(val[j*4+3..j*4+2].to_owned());
324             }
325         }
326         else {options.push(value);}

```

```
327
328     for option in options {
329         let option=option.replace(" ", "");
330
331         if option.starts_with("0x"){
332             acceptable_values.push(u8::from_str_radix(&option[2..],
333                 16).expect("Unknown value {option}"));
334         }
335         else if option.starts_with("LEN(") {
336             let map_key = option[4..option.len()-1].to_owned();
337             let var_len=(message[i] as u16)<<8 | (message[i+1] as u16);
338
339             let var_vec = message[i+2..i+2+var_len as usize].to_owned();
340             i+=2+var_len as usize;
341
342             variables.insert(map_key, var_vec);
343             skip_iter=true;
344             continue 'mainloop;
345         }
346         else if option.starts_with("RANGE(") {
347             let lower = u8::from_str_radix(&option[8..10],16).unwrap();
348             let higher = u8::from_str_radix(&option[13..15], 16).unwrap();
349
350             for val in lower..=higher{
351                 acceptable_values.push(val);
352             }
353         }
354         else {
355             error!("Command {option} unknown");
356             continue 'mainloop;
357         }
358     }
359
360     for acc_value in acceptable_values {
361         if acc_value == message[i] {
362             i+=1;
363             continue 'mainloop;
364         }
365     }
366     return Ok(false);
367 }
368 Ok(true)
369 }
```

Listing A.1: Test generator file

Appendix B

JSON test definition template for service 0x29

```
1 {
2   "TestSuitName": "0x29 Service",
3   "ID": "Extended, 0x18DAF017, 0x18DA17F0",
4   "Tests": [
5     {
6       "Sequence": [
7         {
8           "TestName": "Minimum Length Error Check"
9         },
10        {
11          "PairArray": ["0x29", "0x01", "Response", "0x7F", "0x29", "0x13"]
12        }
13      ]
14    },
15    {
16      "Sequence": [
17        {
18          "TestName": "Total Length Error Check"
19        },
20        {
21          "PairArray": ["0x29", "0x01", "0x00", "0x08", "0x00", \
22            "FILE(client_certificate.der)", "0x00", "0x00", \
23            "Response", "0x7F", "0x29", "0x13"]
24        }
25      ]
26    },
27    {
28      "Sequence": [
29        {
```

```

30     "TestName": "Sequence Error Check"
31   },
32   {
33     "PairArray": ["0x29", "0x03", "0x00", "0x02", "0x12", "0x34", \
34     "0x00", "0x00", "Response", "0x7F", "0x29", "0x24"]
35   }
36 ]
37 },
38 {
39   "Sequence": [
40     {
41       "TestName": "Invalid Certificate Error Check"
42     },
43     {
44       "PairArray": ["0x29", "0x01", "0x00", "0x00", "0x02", "0x00", \
45       "0x00", "0x00", "0x00", "Response", "0x7F", "0x29", "0x54"]
46     }
47   ]
48 },
49 {
50   "Sequence": [
51     {
52       "TestName": "Happy Path"
53     },
54     {
55       "PairArray": ["0x29", "0x01", "0x00", "0x02", "0xCD", \
56       "FILE(client_certificate.der)", "0x00", "0x00", "Response", \
57       "0x69", "0x01", "0x11", "LEN(CHA)", "CHA", "0x00", "0x00"]
58     },
59     {
60       "PairArray": ["0x29", "0x03", \
61       "LEN(RES(CHA, ./client_private_key.pem))", \
62       "RES(CHA)", "0x00", "0x00", "Response", "0x69", "0x03", "0x12"]
63     }
64   ]
65 }
66 ]
67 }

```

Listing B.1: JSON test definition template for service 0x29 (character ' indicates new line used here for presentation reasons)

Appendix C

Guide used for the validation

General module (execute in order)

Tests to service 0x29

ID src: 0x18daf017 (Extended)

ID dest: 0x18da17f0 (Extended)

Happy Path

Send: 0x29 0x01 0x00 0x02 0xCD client_certificate.der 0x00 0x00

Expect: 0x69 0x01 0x11 (2bytes with the length of the challenge) challenge 0x00 0x00

Send: 0x29 0x03 (2 bytes with the length of the signed challenge) signed_challenge 0x00 0x00

Expect: 0x69 0x03 0x12

Minimum Length Error Check

Send: 0x29 0x01

Expect: 0x7F 0x29 0x13

Tutorial Caravel

```
{
  "TestSuitName": "Conjunto de testes exemplo",
  "ID": "Extended/Standard, ID source, ID dest",
  "Tests": [
    {
      "Sequence": [
        {
          "TestName": "Teste exemplo"
        },
        {
          "PairArray": ["0x01", "0x02", "Response", "0x03", "0x04"]
        },
        {
          "PairArray": ["0x01", "0x02", "Response", "0x03", "0x04"]
        }
      ]
    },
    {
      "Sequence": [
        .....
      ]
    },
    {
      "Sequence": [
        .....
      ]
    }
  ]
}
```

JSON structure that describes the tests. Fields TestSuitName and TestName aren't mandatory but help organizing the output.

ID: First field to be specified is the ID type. You can either choose Extended or Standard IDs (both have to be the same type. Second field is the source ID, and third is the destiny ID.

It's in the Tests array that each sequence is included in (each sequence is equivalent to one test). In this case, the sequence that executes "Teste Exemplo" contains two PairArrays, which means that ,for each PairArray, the message before "Response" is going to be sent, and the message after "Response" is going to be the expected response.

In case the test fails, a message will be transmitted to the user informing of the pass/failure of the test.

Special Operators

- **FILE(file_path)** only available at send/request, and imports the contents from the file pointed to by the specified file_path. In the case of having a file called ola.der with a content 0x1234, and a PairArray of the type:

"0x30", "0x40", "FILE(ola.der)"

This would be equivalent to:

"0x30", "0x40", "0x12", "0x34"

-
-
- **LEN(VAR)/VAR**, only available for expect/response and allows saving a variable for later use in the sequence. If we are waiting to receive a value with an unknown length, we can use this pair of operators to process it in the following way:

`"0x30","0x40","LEN(CHA)","CHA","0x05"`

What happens in this case is:

The first and second bytes are verified in order to know if they match with 0x30 and 0x40;

The length of a field in UDS is always described with 2 bytes, so LEN uses the two bytes in the received message to extrapolate the size of variable CHA.

With CHA's length known, the program extracts CHA and saves it in a variable list for later use in the following PairArrays.

The last byte is checked for match against 0x05.

- **LEN(RES(VAR, path_priv_key))/RES(VAR) –**, only available for send/request, it is used in conjunction with the previous pair of operators. This pair of operators allows using a saved variable (that must be a challenge for this pair to be useful) and signing it using the private key pointed to by path_priv_key. As in the previous example LEN(RES(..)) represents the 2 bytes that contain the length of RES(..) (the signed challenge). This pair of operators could be used in the following way:

`"0x30","0x40","LEN(RES(CHA, private_key.pem))","RES(CHA)"`

What happens in this case is:

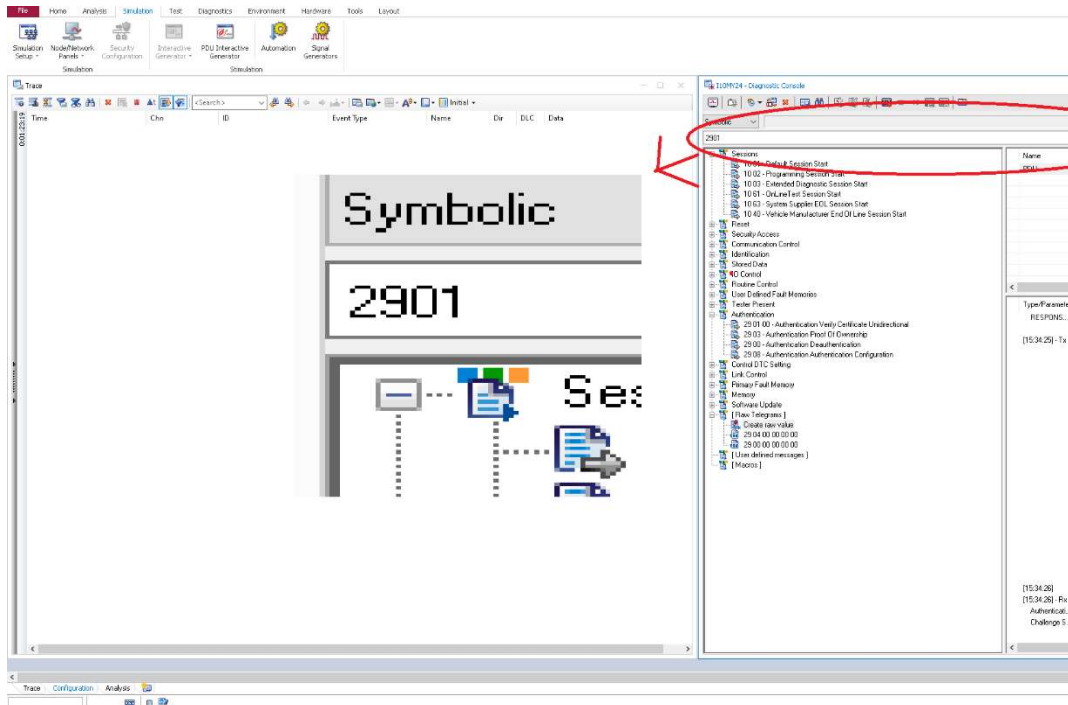
The first and second bytes are set to 0x30 and 0x40

The challenge contained in variable CHA is signed. The length of the signed challenge is written in 2 bytes and the signed challenge is inserted after.

For example, for RES(CHA)=0x12 0x34, the sent message would be:

`"0x30","0x40","0x00","0x02","0x12","0x34"`

Tutorial CANoe



It's in the red circled box that the hexadecimal byte can be written in, in order to be sent to the CAN bus. In the figure a 29 01 request is represented.

In case you wish to extract a message from CANoe, you must pause the simulation, right-click in the intended message and copy it to the clipboard.

To sign the challenge you must write the challenge into a file and process it using the following command in the Raspberry Pi.

```
openssl dgst -sha256 -sign path_private_key -out path_to_signed_challenge path_to_challenge
```

You can then check the signed challenge using the command

```
xxd -p path_to_signed_challenge
```

Which dumps its hexadecimal contents in the terminal

You can also check the file's size using the command:

```
du -h path_to_file
```


References

- [1] Mehmet Bozdal, Mohammad Samie, Sohaib Aslam, and Ian Jennions. Evaluation of can bus security challenges. *Sensors 2020, Vol. 20, Page 2364*, 20:2364, 4 2020. [Online; accessed October 2022]. URL: <https://www.mdpi.com/1424-8220/20/8/2364/html><https://www.mdpi.com/1424-8220/20/8/2364>, doi:10.3390/S20082364.
- [2] Iso - iso 14229-1:2020 road vehicles — unified diagnostic services (uds) — part 1: Application layer. [Online; accessed March 2023]. URL: <https://www.iso.org/standard/72439.html>.
- [3] Continental Group. World Wide Locations - Continental Engineering Services. https://conti-engineering.com/ces_locations/, 2023. [Online; accessed June 2023].
- [4] Markus Zoppelt and Ramin Tavakoli Kolagari. What today’s serious cyber attacks on cars tell us: Consequences for automotive security and dependability. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11842 LNCS:270–285, 2019. [Online; accessed May 2023]. URL: https://link.springer.com/chapter/10.1007/978-3-030-32872-6_18, doi:10.1007/978-3-030-32872-6_18/TABLES/2.
- [5] Mahmood Jasim Khalsan and Michael Opoku Agyeman. An overview of prevention/mitigation against memory corruption attack. *ACM International Conference Proceeding Series*, 9 2018. [Online; accessed June 2023]. URL: <https://dl.acm.org/doi/10.1145/3284557.3284564>, doi:10.1145/3284557.3284564.
- [6] Liis Jaks. Security evaluation of the electronic control unit software update process. 2014. [Online; accessed November 2022]. URL: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-188171>.
- [7] Md Swawibe Ul Alam, Shahrear Iqbal, Mohammad Zulkernine, and Clifford Liem. Securing vehicle ecu communications and stored data. volume 2019-May, pages 1–6. IEEE, 5 2019. [Online; accessed February 2023]. URL: <https://ieeexplore.ieee.org/document/8762043/>, doi:10.1109/ICC.2019.8762043.
- [8] Yong Xie, Yu Zhou, Jing Xu, Jian Zhou, Xiaobai Chen, and Fu Xiao. Cybersecurity protection on in-vehicle networks for distributed automotive cyber-physical systems: State-of-the-art and future challenges. *Software: Practice and Experience*, 51:2108–2127, 11 2021. Página 4- domain controller centralized architecture. URL: <https://onlinelibrary.wiley.com/doi/10.1002/spe.2965>, doi:10.1002/spe.2965.
- [9] Martin Placek. Car costs - automotive electronics costs world-wide 2030 | statista, 2019. [Online; accessed November 2022].

- URL: <https://www.statista.com/statistics/277931/automotive-electronics-cost-as-a-share-of-total-car-cost-worldwide/>.
- [10] J A Cook and J S Freudenberg. Controller area network (can) eecs 461, fall 2008 *. [Online; accessed April 2023]. URL: https://www.eecs.umich.edu/courses/eecs461/doc/CAN_notes.pdf.
- [11] András Gazdag, Csongor Ferenczi, and Levente Buttyán. Development of a man-in-the-middle attack device for the can bus, 2020. [Online; accessed October 2023]. URL: <http://illmatics.com/remote>.
- [12] Iso 15765-2:2016 - road vehicles — diagnostic communication over controller area network (docan) — part 2: Transport protocol and network layer services. [Online; accessed June 2023]. URL: <https://www.iso.org/standard/66574.html>.
- [13] Razvan G. Lazar and Constantin F. Caruntu. Simulator for the automotive diagnosis system on can using vector canoe environment. *2020 24th International Conference on System Theory, Control and Computing, ICSTCC 2020 - Proceedings*, pages 705–710, 10 2020. [Online; accessed June 2023]. doi:10.1109/ICSTCC50638.2020.9259683.
- [14] William Bugden, Ayman Alahmar, and Corresponding Author. Rust: The programming language for safety and performance. *Igscong'22*, 6 2022. URL: <https://arxiv.org/abs/2206.05503v1>.
- [15] What is ownership? - the rust programming language. [Online; accessed April 2023]. URL: <https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html>.
- [16] Memory safety. [Online; accessed April 2023]. URL: <https://www.chromium.org/Home/chromium-security/memory-safety/>.
- [17] Nikolay Ivanov. Is rust c++-fast? benchmarking system languages on everyday routines. 9 2022. [Online; accessed April 2023]. URL: <https://arxiv.org/abs/2209.09127v1>.
- [18] Ignas Plauska, Agnius Liutkevičius, and Audronė Janavičiūtė. Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller. *Electronics 2023, Vol. 12, Page 143*, 12:143, 12 2022. [Online; accessed April 2023]. URL: <https://www.mdpi.com/2079-9292/12/1/143/htmhttps://www.mdpi.com/2079-9292/12/1/143>, doi:10.3390/ELECTRONICS12010143.
- [19] Stack overflow developer survey 2022. [Online; accessed April 2023]. URL: <https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-language-love-dread>.
- [20] Lerche Carl. Tokio - an asynchronous rust runtime. [Online; accessed June 2023]. URL: <https://tokio.rs/>.
- [21] Lorcan Leonard. Event-driven servers using asynchronous, non-blocking network i/o: Performance evaluation of kqueue and epoll. *Dissertations*, 1 2021. [Online; accessed June 2023]. URL: <https://arrow.tudublin.ie/scschcomdis/238>, doi:<https://doi.org/10.21427/3K70-ZZ90>.

- [22] Overview · serde. [Online; accessed April 2023]. URL: <https://serde.rs/>.
- [23] socketcan-isotp - crates.io: Rust package registry. URL: <https://crates.io/crates/socketcan-isotp>.
- [24] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. *25th International World Wide Web Conference, WWW 2016*, pages 263–273, 2016. [Online; accessed April 2023]. URL: <https://dl.acm.org/doi/10.1145/2872427.2883029>, doi:10.1145/2872427.2883029.
- [25] Eduardo dos Santos, Andrew Simpson, and Dominik Schoop. A formal model to facilitate security testing in modern automotive systems. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, 271:95–104, 5 2018. [Online; accessed April 2023]. URL: <http://arxiv.org/abs/1805.05520><http://dx.doi.org/10.4204/EPTCS.271.7>, doi:10.4204/EPTCS.271.7.
- [26] Qinwen Ran, Xi Wu, Xin Li, Jianqi Shi, Jian Guo, and Huibiao Zhu. Modeling and verifying the ttcan protocol using timed csp. *Proceedings - 2014 International Symposium on Theoretical Aspects of Software Engineering, TASE 2014*, pages 90–97, 12 2014. [Online; accessed March 2023]. doi:10.1109/TASE.2014.8.
- [27] Mohamed H.E. Aouadi, Khalifa Toumi, and Ana Cavalli. An active testing tool for security testing of distributed systems. pages 735–740. IEEE, 8 2015. [Online; accessed March 2023]. URL: <http://ieeexplore.ieee.org/document/7299986/>, doi:10.1109/ARES.2015.97.
- [28] Canoe | ecu and network testing | vector. [Online; accessed April 2023]. URL: <https://www.vector.com/int/en/products/products-a-z/software/canoe/#>.
- [29] Raspberry pi 4 model b – raspberry pi. [Online; accessed June 2023]. URL: <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>.
- [30] Carl Hewitt. Actor model of computation: Scalable robust information systems. 8 2010. [Online; accessed June 2023]. URL: <https://arxiv.org/abs/1008.1459v38>.
- [31] Kevin Knapp. clap - crates.io: Rust package registry. [Online; accessed June 2023]. URL: <https://crates.io/crates/clap>.