POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

CAN Bus Security Analysis: a Fuzzing Approach

Supervisor
Alessandro SAVINO
Co-Supervisors
Nicolò MAUNERO
Giacomo D'AMICO

Candidate
Mattia DE ROSA

Academic Year 2023-2024

Abstract

Modern vehicles are equipped with numerous Electronic Control Units (ECUs), each featuring intricate functionalities and being tightly interconnected via internal networks. Among these, the Controller Area Network (CAN) is the most common. Past research has revealed the CAN network to be vulnerable to a multitude of cybersecurity attacks, enabling an attacker to take control of safety-critical ECUs such as the ones managing the engine, steering, or brakes. Securing ECUs connected via the CAN network against cyber threats is of paramount importance since an infected ECU could be used to propagate the attack to the other units on the network.

Fuzz testing is a widely adopted, automated software testing technique that helps identify vulnerabilities and defects in programs. It involves sending a large amount of generated data to the system under test to identify messages that cause crashes, errors, or other incorrect behavior.

Existing fuzzers in the automotive security landscape often fall into two categories: proof-of-concept open-source tools lacking advanced functionalities or closed-source solutions requiring proprietary hardware, hindering interoperability with other tools.

This thesis aims to bridge this gap by developing a modular fuzzer tailored for robustness and future extensibility. The primary focus is on enhancing the ease of integration while providing a versatile tool for cybersecurity testing.

Organizing the fuzzer into multiple modules facilitates the concurrent development of different features. Additionally, relying on a higher abstraction of the CAN protocol ensures interoperability among the developed components. A direct outcome of this abstraction is the elimination of dependence on proprietary hardware.

The development of dedicated modules for various network interfaces transforms them into plug-and-play components for the fuzzer. This thesis introduces two interfaces: one for utilizing a virtual CAN bus and the other for interfacing with Intrepid CS devices.

To validate the fuzzer's effectiveness, testing has been conducted on both a simulated virtual ECU and a physical test bench. Finally, testing has been performed to compare various developed fuzz generator modules, highlighting their efficacy under different assumptions.

It is important to note that while the project primarily focuses on the CAN network, the architecture has been designed to seamlessly extend to multiple protocols.

Contents

List of Tables

List of Figures

Acronyms

ABC Abstract Base Class.

API Application Programming Interface.

ASIL Automotive Safety Integrity Level.

BEV Battery Electric Vehicle.

BLF Binary Logging Format.

CAN Controller Area Network.

CAN FD Controller Area Network Flexible Data.

CRC Cyclic Redundancy Check.

CVSS Common Vulnerability Scoring System.

DBC Database CAN.

DLC Data Length Code.

ECU Electronic Control Unit.

EoF End of Frame.

HSM Hardware Security Module.

IDE IDentifier Extension bit.

IDS Intrusion Detection System.

IPS Intrusion Prevention System.

ISO International Organization for Standardization.

MAC Message Authentication Code.

OBD-II On-Board Diagnostics II.

OSI model Open Systems Interconnection model.

SoF Start of Frame.

SUT System Under Test.

V&V Verification And Validation.

Chapter 1

Introduction

The most common method of transportation for people and goods are, by far, road automobiles. According to a study performed by Eurostat [15], in 2011 transport by car accounted for 73.1% of passenger-kilometers[1] with small variations until 2019. After the pandemic we saw a decrease in the usage of public transport which in turn caused the percentage of passenger-kilometers for cars to reach 79.7% in 2021 [Figure 1.1].

Cars have massively evolved over the years, the introduction of advanced technologies in auto vehicles began in the early '70 thanks to the development of integrated circuits and microprocessors which allowed the production of ECUs (Electronic Control Unit) on a large scale. The number of ECUs has steadily increased in road vehicles reaching as many as 150 on luxury models [5], all these ECUs exchange thousand of messages every second and communicate which each other through cabled or wireless communication networks, of which the most important is the CAN bus (Controller Area Network).

In recent years, the automotive industry has undergone a remarkable transformation, propelled by technological advancements that have introduced a new era of intelligent, connected, and autonomous vehicles. Vehicles have become increasingly more sophisticated, incorporating cutting-edge technologies such as advanced driver-assistance systems (ADAS), telematics, vehicle-to-vehicle (V2V) and vehicle-to-everything (V2X) communication. As a result of these transformations, the necessity of addressing cybersecurity concerns in the automotive domain has grown significantly.

[1] The passenger-kilometer is a unit of measurement representing the transport of one passenger by a defined mode of transport (road, rail, air, sea, inland waterways etc.) over one kilometer.

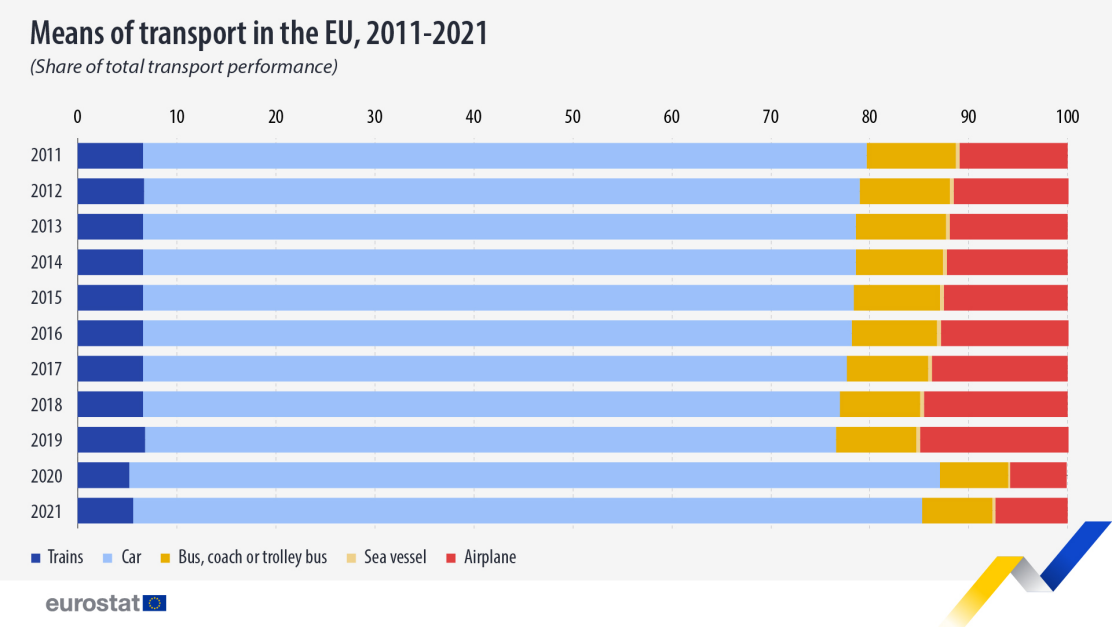


Figure 1.1: Means of transport in the EU, 2011-2021. Image from Eurostat [15].

1.1 Cybersecurity Concerns

Due to the complexity of the underlying systems in an auto vehicles many attacks are possible and there is a need to protect against both practical and theoretical attacks. To understand how to best protect a car against attackers it is important to explain and highlight the different goals an attacker could have and the attack surfaces that could be employed to perform the attack.

1.1.1 Attack Goals

The goals of an attacker are many and varied, but they can be summarized with the mnemonic STRIDE [33], which divides security threats in six categories:

- **S**poofing: Pretending to be something or someone other than yourself, for example can be used to unlock and start a car without the key.
- **T**ampering: Modifying the behavior of a component, for example can be used to take remote control of a car.
- **R**epudiation: Denying the performance of an action without other parties having any way to prove otherwise, for example it could be used to falsify logs.

- **I**nformation disclosure: Exposing information to someone not authorized to access it, for example can be used to obtain GPS data or capturing microphone signals.
- **D**enial of service: Denying a service to the user, for example can be used to prevent a car from unlocking the doors or starting.
- **E**levation of privilege: Allowing someone to do something they are not authorized to do, for example could be used to access locked features of a car or removing safety features.

1.1.2 Attack Surfaces

Examining the diverse attack surfaces of a vehicle, we can distinguish between long-range, short-range, and local attack surfaces.

The long-range attack surfaces encompass technologies such as LTE (Long-Term Evolution) and DSRC (Dedicated Short-Range Communication)[2], extending the potential reach of external threats.

Short-range attack surfaces, though wireless in nature, operate within closer proximity and include technologies like Bluetooth and Wi-Fi.

Finally, local attack surfaces comprise interfaces that necessitate physical access to the vehicle, such as USB ports and the OBD-II connector (On-Board Diagnostics).

It is crucial to safeguard against all these attack surfaces since a breach in any of them could potentially lead to vulnerabilities in the entire system, due to the possibility of privilege escalation exploits.

1.2 Industry Standards

Recognizing the urgency of the situation, both the automotive industry and regulatory bodies have begun to respond. Standards and guidelines for automotive cybersecurity are emerging, shaping the industry's approach to securing vehicles against cyber threats. The most important standards and documents for the automotive industries are AUTOSAR [2], ISO 26262 [28] and ISO/SAE 21434 [30] of which the key points are detailed below. These standards collectively contribute to creating safer and secure automotive systems, keeping up with the industry's evolving needs and challenges.

[2]DSRC is a technology for direct wireless exchange of V2X data between vehicles, other road users (pedestrians, cyclists, etc.) and roadside infrastructure (traffic signals, electronic message signs, etc.)

1.2.1 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a global partnership of leading companies in the automotive and software industry. It is focused on creating and establishing an open and standardized software architecture for automotive ECUs.

AUTOSAR provides specifications basic software modules, focusing on creating common API (Application Programming Interface) and development methodology based on a standardized exchange format. AUTOSAR software architecture can be used by vehicles of different manufacturers and components of different suppliers, increasing interoperability and reducing research and development costs.

AUTOSAR Classic Platform [3] [Figure 1.2] is the standard for embedded ECUs, it is composed by a three layer architecture:

- **Application Software:** designed to be hardware independant, it interacts with the RTE layer.
- **Runtime Environment (RTE):** is a middleware that represents the full interface for applications and abstracts the BSW layer.
- **Basic Software (BSW):** contains the drivers necessary to interface with the hardware.

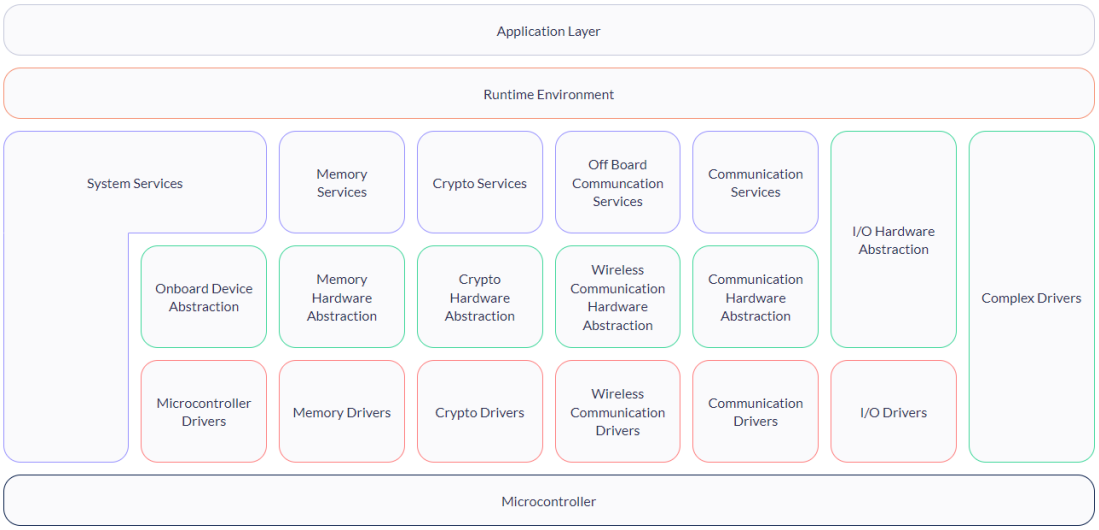


Figure 1.2: AUTOSAR Classic Platform schema. Image from autosar.org [3].

1.2.2 ISO 26262

This document is an international standard that focuses on ensuring the functional safety[3] of electrical and electronic systems in road vehicles. The aim is to minimize the risk of accidents and ensure the proper function of individual components in road vehicles, according to their intended purpose. Additionally it provides an automotive-specific risk classification scheme known as ASIL (Automotive Safety Integrity Level).

ASIL divides risk in four levels, from A to D, which is determined by a combination of Severity, Exposure and Controllability.

ISO 26262 provides a process for managing and reducing risk, based on the concept of "safety life cycle", which includes the following phases:

- **Planning:** the safety requirements are defined and a safety plan is developed.
- **Analysis:** the system is analyzed to identify hazards and potential failures.
- **Development:** the system is designed and implemented to meet the safety requirements defined in the planning phase and to eliminate or mitigate hazards defined in the analysis phase.
- **Verification and Validation:** the system is tested to ensure it complies with the design and behaves as expected in case of failures.
- **Operation, service and decommissioning:** the safety requirements are maintained and the system is decommissioned.

This standard helps ensure that the safety of car components is considered throughout the lifecycle of the vehicle. It provides guidelines for concurrent development and testing of both hardware and software to ensure optimal safety.

1.2.3 ISO/SAE 21434

This document addresses the cybersecurity perspective in engineering of electrical and electronic systems within road vehicles. Its objective is to ensure the appropriate consideration of cybersecurity and to enable the engineering of E/E systems to keep up with state-of-the-art technology and evolving attack methods, without prescribing specific technologies or solutions related to cybersecurity.

ISO/SAE 21434 plays a crucial role in enhancing vehicle security for several reasons:

[3] In the context of the automotive industry, functional safety assures that electronic and electrical systems operate safely, without posing unreasonable risk, even when faced with faults or failures.

- **Unified Framework:** this document provides a unified and standardized framework for addressing cybersecurity concerns in the automotive industry. This standardization helps create a common language and set of practices that can be adopted by manufacturers, suppliers, and other stakeholders.
- **Lifecycle Approach:** the standard takes a lifecycle approach to cybersecurity, encompassing all phases from concept and design through production, operation, maintenance, and decommissioning.
- **Risk Assessment and Management:** the document emphasizes a risk-based approach to cybersecurity, it encourages organization to conduct risk assessments to identify potential threats or vulnerabilities.
- **Adaptability to Technological Advances:** the standard is designed to be adaptable to evolving technologies, expecting vehicles to become more connected and autonomous, while maintaining it secure against cyber attacks.
- **Post-Production Activities:** along the multiple control practices defined for development of products, the standard also requires support post-production by monitoring cybersecurity breaches and the need to release fixes and patches to the developed products.
- **Integration with Existing Standards:** ISO 21434 is designed to integrate with other relevant standards such as ISO 26262 for functional safety.
- **Customer Trust and Confidence:** as cybersecurity concerns become more prominent in the public consciousness, adherence to recognized standards signals a commitment to ensuring the safety and security of vehicle systems and data.

In summary, ISO/SAE 21434 is important for vehicle security because it establishes a standardized and comprehensive approach to addressing cybersecurity throughout the entire lifecycle of automotive development. It provides guidance, best practices, and a common language that fosters collaboration, risk management, and adaptability to the evolving automotive landscape. Adhering to this standard contributes to building resilient and secure vehicles in an increasingly connected and technologically advanced automotive industry.

Chapter 2

State of the Art for Vehicle Cybersecurity

The current state of the art for automotive cybersecurity testing involves a combiation of tools, methodologies and best practices aimed at identifying and mitigating vulnerabilities in the complex landscape of modern vehicles.

To best illustrate the protection measures against cyber-threats in vehicles, we can higlight the distinct procedures and tools implemented through its lifecycle. Threat modeling plays a pivotal role during the planning and design phase, diverse testing methodologies are applied post production and finally a suite of tools is used to consistently verify and maintain the proper operation of a vehicle.

2.1 Threat Modeling

Threat modeling is a process which aims to identify, evaluate, and mitigate potential security threats and vulnerabilities within a system, application, or network. Its purpose is to provide developers with a full analysis of the system highlighting the most likely attack vectors and most desired assets an attacker might want to access or compromise.

A threat model typically includes:

- description of the subject
- assumptions to be checked or challenged
- potential threats to the system
- actions to be taken to mitigate each threat
- a way to validate the models and threats

- a way to verify the success of action taken

It is ideal to establish a threat model early in the planning phase and subsequently evolve it alongside with the system it represents. This implies continuous refinement throughout the entire lifecycle of the product: as more details are added to the system, new attack vectors are created and exposed so updating the model is needed to account for these changes.

The OWASP Foundation defines a "Four Question Framework" [42] to help organize a threat model:

- **What are we working on?** The scope of the project, can be as small as a sprint or as large as the whole system.
- **What can go wrong?** Can be as simple as brainstorming or structured using STRIDE, DREAD or many other strategies.
- **What are we going to do about it?** Resolve, Mitigate, Transfer (to a third party) or Accept the risk.
- **Did we do a good job?** Assess the work done on the project.

Most threat modeling processes start by creating a visual representation of the system being analyzed. It is decomposed into basic blocks to aid the analysis. The representation is then used to enumerate potential attack vectors or other threats to the analyzed system [Figure 2.1].

If done right threat modeling is a powerful tool that provides a clear picture on how to develop a secure system. The threat model allows the developers to take security decision rationally, with all the relevant information on the table.

2.2 Testing Methodologies and Tools

One of the key activities performed by developers during the development cycle is Software Testing. To address all possible vulnerabilities in a system, software testing employs a diverse array of techiniques. Of particular relevance to the automotive sector are penetration testing, fuzz testing and regression testing.

All these different testing techniques complement eachother to form a robust and comprehensive testing framework.

2.2.1 Penetration Testing

Penetration testing (also known as "Ethical Hacking") is a security exercise where a cyber-security expert impersonates the role of an attacker and attempts to find

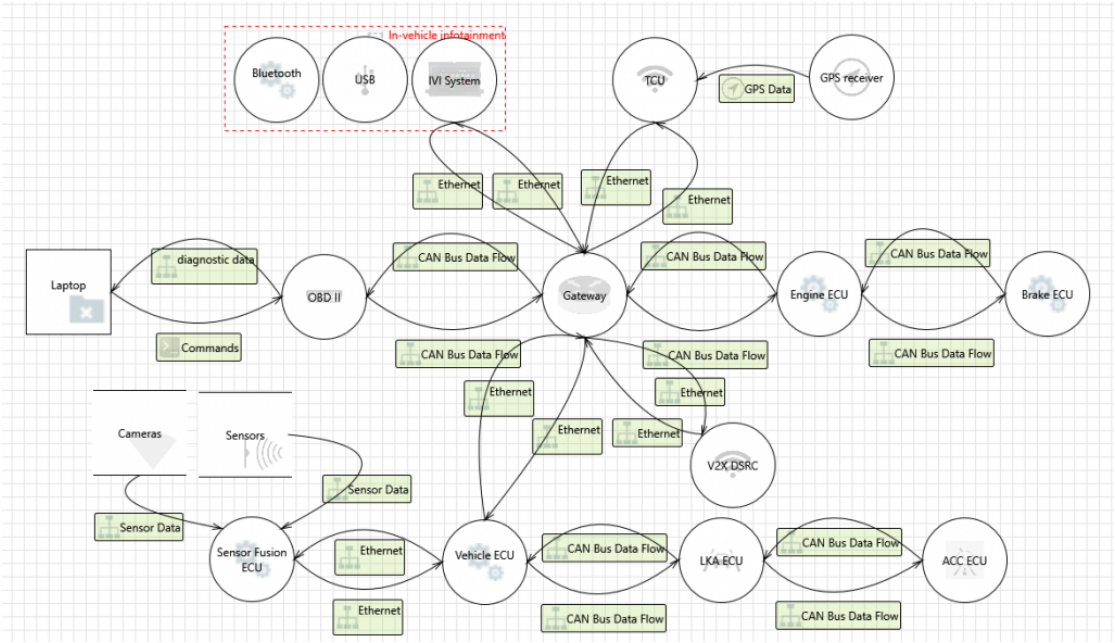


Figure 2.1: Data flow diagram example for a vehicle used for Threat Modeling. Image from Copper Horse [12].

and exploit vulnerabilities in a computer system. Its purpose is to identify weak spots which could be taken advantage of [6].

Penetration testing begin with a phase of reconnaissance, called "scoping" with the objective to gather data to be used in the simulated attack.

After the attack is carried out, the pentester wraps up the test by covering their tracks and producing a report which details the discovered vulnerabilities with associated levels of risk and possible mitigations.

2.2.2 Fuzz Testing

Fuzz testing (or Fuzzing) is an automated software testing technique, which consists in finding implementation bugs using malformed data injection [17].

The biggest advantages of fuzzing are that the test design is extremely simple, often being system-independent, and its complete automation. Additionally the random approach allows this method to find bugs that would have been missed by human eyes.

2.2.3 Regression Testing

Regression testing consists in re-running functional and non-functional tests to ensure that the behaviour of previously developed and tested software still performs as expected. In short regression testing is done to ensure that a achange in the code to fix a defect has not introduced any new defect [4].

2.2.4 Modularity of Tools

The automotive industry utilizes an extensive array of communication protocols, each serving specific functions within the complex ecosystem of vehicles. Given the diversity of protocols present, a key consideration in the selection of testing tools and methodologies lies in its modularity.

Modular tools offer numerous advantages compared to their non-modular counterparts, with the most crucial ones being:

- **Support for multiple protocols:** Modern vehicles employ a multitude of communication protocols. Some examples are CAN, ethernet and LIN for communication between the various ECUs, or Bluetooth, Wi-Fi, USB and cellular data for communication between the vehicle and other devices. An effective tool needs to support as many of these protocols as possible, a modular approach is ideal to easily and effectively add support to each protocol.

- **Adaptability to new technologies:** The automotive industry is dynamic, with rapid advancements in technologies and introduction of new features. A modular tool allows it to rapidly adapt to these emerging technologies, ensuring it remains relevant and effective over time.

- **Scalability:** As the complexity of an automotive system grows, modular tools provide a scalable solution. New modules can be added to work in tandem with existing ones without requiring a complete overhaul of the testing infrastructure.

- **Interoperability:** A modular tool promotes interoperability between different testing tools and system. It allows for easier integration of third-party tools or modules, forstering collaboration within the testing ecosystem.

- **Easier maintenance and updates:** When an update is needed, modularity allows to make the process more efficient. Updates can be applied to specific modules without affecting the entire system, this also means that if an update breaks a specific module, it does not affect the rest of the system negatively.

Due to the numerous outgoing interfaces in a vehicle, an unbalance exists between developers and potential attackers. Developer are required to propertly protect all interfaces, meanwhile an attacker can succeed by exploiting a single vulnerability among the multiple entry points to successfully gain access to the system. Because of this an effective testing tool leverages its own modularity to comprehensively test and protect all possible interfaces.

2.3 Continuous Vehicle Security

After the development of a vehicle and its ECUs is completed, various security measures are implemented to protect it from potential threats. Some key aspects of how a vehicle is protected post-development are:

- **Software Security Updates:** Regular software updates are crucial to address vulnerabilities discovered after the vehicle's release. Manufacturers release updates to fix bugs, enhance performance and address any security weakness that may be identied over time.
- **Over-the-Air (OTA) Updates:** Many modern vehicles support OTA updates, which allows manufacturers to remotely deliver software updates to vehicles. This enables timely deployment of security patches without requiring constant physical visits to service centers.
- **Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS):** These systems' objective is to monitor the vehicle's internal network for unusual or suspicious activities. IDS detects possible security breaches, while IPS intervenes to prevent or mitigate the threat. Both IDS and IPS are crucial to identify threats in real time.
- **Cybersecurity Audits and Assessments:** Regular cybersecurity audits are scheduled to evaluate the overall security of the vehicle. The assessments involve examining the effectiveness of security controls and ensuring compliance with evolving industry standards.
- **Incident Response Plans:** Having a well-defined incident response plan is crucial in the event of a cycbersecurity incident to deploy a patch in a timely manner.

2.4 Fuzz Testing

In this section we delve into the details of fuzz testing to better understand why we chose to develop a fuzzer, we will then explain the inner workings of one, which will be helpful to undertand the design choices taken during development.

As explained previously, fuzz testing is an automated software testing technique with the aim of discovering bug, vulnerabilites and incorrect behaviours in the system under test. Automating these kind of tests allows for discovering bugs in a timely manner, which is of fundamental importance in developing secure software.

Any software accepting user input is potentially vulnerable to multiple security issues. As previously stated, due to the abundance of external interfaces present in a vehicle, cars are prime targets for attacks that rely on untrusted user inputs. Some key examples are protocols relating to the infotainment systems, like bluetooth or wi-fi. An attacker might embed malicious code in a contact on his smartphone so that, when the contacts are synchronized with the vehicle, the injected code is launched and the attack is performed. This type of attack is aptly named code injection and is one of the many kind of attacks that relies on vulnerabilities detected by a fuzzer.

The effectiveness of fuzzing as a testing methodology is apparent when examining the substantial number of bugs found by widely adopted open source fuzzers. For instance, Google's OSS-fuzz, as of August 2023, was able to help identify over 10,000 vulnerabilities across 1,000 projects [23]. Similarly Mozilla's domino identified more than 2,200 defects, including 20 with HIGH or CRITICAL CVSS[1] scores [35].

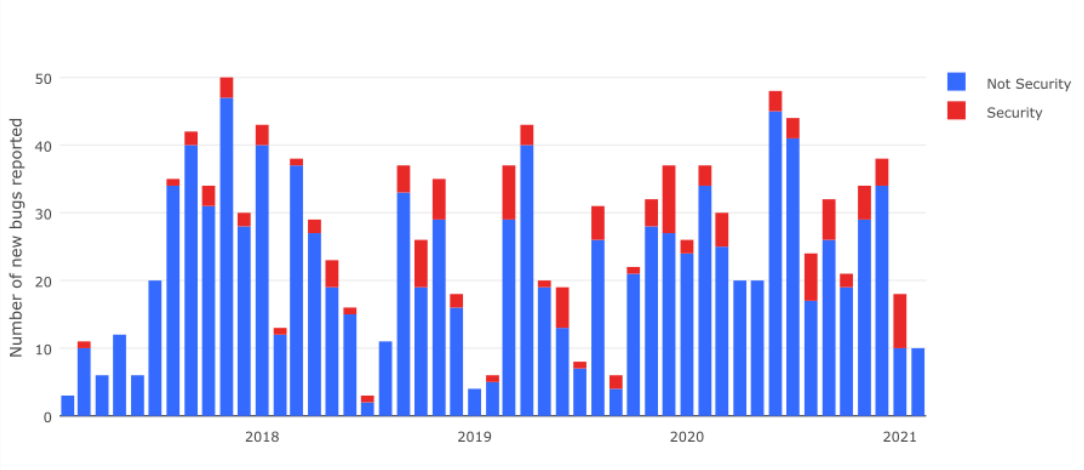


Figure 2.2: Number of bugs reported by Domino over time [34].

[1]The Common Vulnerability Scoring System is a standard for assessing the severity of computer system security vulnerabilities. Scores range from 0 (NONE) to 10 (CRITICAL).

2.4.1 Classification of Fuzzers

When developing a fuzzer from scratch there are many approaches and techniques to consider that can be classified based on the internal knowledge of the target, input type, presence or absence of a feedback and different ways to generate the test cases [31].

Based on the amount of knowledge a fuzzer has available, we can classify fuzzers in three categories:

- **Black-box** fuzzers do not require any additional input, they may be aware of the structure of the input data, but knowledge about inner workings of the target is not needed.

  These kind of fuzzers are the most versatile since they can be applied without developing an ad-hoc solution, but they provide fewer insights on the produced results. To counteract this, there are attempts at developing black-box fuzzers that try to incrementally increase their knowledge about a target's internal structure, like LearnLib [39].

  An additional strong point of black-box fuzzing is that it simulates the point of view of an attacker more accurately.

- **White-box** fuzzers need internal knowledge about the target, usually source code or data flow diagrams. This kind of fuzzer employs program analysis with the main objective of increasing code coverage[2] or reaching critical locations in a program. Other possible techniques for white-box fuzzing include static, taint or concolic analysis, each based on varying amount of knowledge of the target.

  These types of analysis are very effective on targeting specific platform, but one of its main disadvanteges is the time needed to perform them. If a fuzzer takes too long to generate a test case, a black-box fuzzer might be more efficient, or produce results earlier.

- **Gray-box** fuzzers are an attempt to combine the efficiency of black-box with the effectiveness of white-box fuzzers. They usually require minimal knowledge about the target and employ instrumentation[3] to discern information about the target.

[2]Code coverage is a metric that measures the percentage of source code executed when a particular test suite is run. An effective fuzzer tries to reach as close as full coverage as possible.

[3]Instrumentation is a technique used to track application behaviour by detecting errors and obtaining trace information. It employs debug tools to perform profiling of the application and log major events such as crashes.

A major distinction in how fuzzing is approached depends on wheter the fuzzer is aware of the input structure of the target.

- A **smart** fuzzer leverages its knowledge of the input to generate a greater amout of valid input. For example if we have the objective of fuzzing an image processing software we might want to only use images as input since other file types will likely get discarded.

  More advanced smart fuzzers might employ a model or grammar to restrict the input space to only known valid combinations. For instance in the automotive sector we might decide to develop a fuzzer that is aware of the structure of a CAN frame and sends only valid ones to greatly speed up the fuzzing process. Alternatively there is value in sending malformed frames if we believe a vulnerability might be present in the protocol itself.

- A **dumb** fuzzer does not require any kind of input model so it can be employed to fuzz a wider variety of programs. An example of a dumb fuzzer is AFL [22] which modifies a seed file by flipping random bits, moving or deleting portions of it.

  Dumb fuzzers usually generate a lower proportion of valid files, so they might stress the parser more than the core components of the program.

The final important distinction is related to how the input data is generated:

- A **generation-based** fuzzer generates inputs from scratch. The main upside is that this kind of fuzzers do not depend on the quality of the input seed.

- A **mutation-based** fuzzer instead relies on an initial seed which is then "mutated" to produce different results.

  A white or gray-box mutational fuzzer might additionaly rely on the feedback of the program to mutate inputs more effectively.

To conclude, many different choices can be made during the development process of a fuzzer, which impact many different factors, like effectiveness, speed and code coverage. A case study was performed on a small Web application with four planted defects [31]. In table 2.1 we can see the impact of the various employed fuzzing strategies.

2.5 Thesis Objectives

The project detailed in this thesis' work was developed in collaboration with Teoresi S.p.A, a company specialising in engineering consulting in cutting edge global technologies.

Technique	**Effort**	**Coverage**	**Defects Found**
Black-box + Dumb	10 min	50%	25%
White-box + Dumb	30 min	80%	50%
Black-box + Smart	2 hours	80%	50%
White-box + Smart	2.5 hours	99%	100%

Table 2.1: Comparison between different fuzzing techniques.

The main objective of this thesis is the development of a modular black-box fuzzer for automating testing in automobiles. The focus of the project has been the CAN network, but support for CAN FD is also present. Additionally due to the modular structure of the fuzzer, it can be easily adapted to support different protocols both wired and wireless.

More specifically the objectives of this work are the following:

- Researching the state of the art for vehicle security including leading industry standards.
- Analyzing the inner workings and limitations of the CAN network.
- Developing a proof of concept virtual ECU to become acquainted with the libraries interfacing with the CAN network.
- Developing the black-box fuzzer focusing on the modular structure, ease of use and deployment of the tool.
- Testing the fuzzer against a test board including the key components of a Fiat 500, simulating a real car in a lab environment.

Additionally, a tool was created to assist in reverse engineering CAN frames within a vehicle.

Chapter 3

Controller Area Network (CAN) Bus

The Controller Area Network (CAN) bus is an asynchronous serial bus used for realtime communication between multiple microcontrollers with eachother. It is a message based protocol designed for communication in a multi-master architecture.

It was developed by Robert Bosch GmbH, released to the public in 1986 and finally standardized in 1993 by the International Organization for Standardization as ISO 11898 [27]. The CAN standard is still adapting and evolving to rising needs in the industry, in 2015 CAN FD (Flexible Data) was standadrdized and in 2018 development for CAN XL has started.

In the modern automotive industry CAN is still the de-facto leading standard for ECU communication in vehicles. The development of newer protocols, like

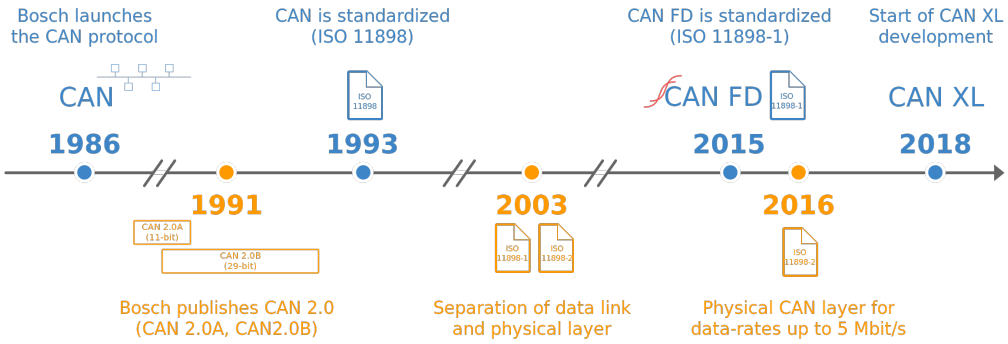


Figure 3.1: CAN adoption timeline. Image from CSS Electronics [14].

SAE J1939[1] and LIN[2], designed to operate either in conjunction with or on top of the CAN bus, further solidifies the integral role of the CAN bus in automotive ECU communication. These protocols exemplify the enduring significance and adaptability of the CAN bus in the evolving landscape of automotive connectivity.

This chapter will provide the motivation for the focus of this thesis on the CAN protcol and an in depth summary of the CAN protocol. Starting with an overlook on the fundamental properties behind its design, followed by a description of its phyiscal and data link layers, and concluding by highlighting the cybersecurity concerns and measures taken to mitigate them.

3.1 Why focus on the CAN bus?

As detailed in section 2.5, this thesis is centered around creating a versatile and modular fuzzer for testing various protocols, with a predominant emphasis on the CAN bus. The rationale for this focus can be summarized by the following motivations.

- **The CAN protocol is universally adopted in cars**.

  CAN bus is one of five different protocols used in the OBD-II (On-Board Diagnostics) standard. OBD-II has been mandatory for all cars and light trucks in the United states since 1996 and for all petrol and diesel vehicles in the EU since 2001 and 2004 respectively [44].

  The CAN standard is by far the preferred protocol among the five for cars, additionally since 2008 it has been mandatory in the US for vehicles below 8500 pounds (about 3500 Kg) [41].

- **CAN is the "Central Nervous System" of a car**.

  The CAN bus, akin to a vehicle's central nervous system, orchestrates the communication between the various ECUs. Every ECU relies on it to communicate with the other components of the vehicle. Therefore, an attack that gains control or distrupts the CAN bus has its effect propagated on the entire network.

  Conversely, safeguarding the CAN bus may reduce the efficacy of attacks on individual ECUs, as the impact could be confined to the affected ECU or a more limited segment of the network.

[1]Society of Automotive Engineers J1939 is the standard in-vehicle network for heavy-duty vehicles like trucks and buses.

[2]The LIN bus (Local Interconnect Network) has been introduced to complement CAN for non-critical subsystems like air conditioning and infotainment.

- **CAN is inherently less secure than higher level protocols**.

  Despite its pivotal role, the CAN bus is susceptible to multiple cybersecurity threats owing to its design choices.

  The original proposal for CAN dates back to the 80s, with widespread adoption in the 90s and early 2000s. During this period, cybersecurity concerns were relatively less prominent, as the primary focus was on establishing a reliable communication protocol.

  The CAN protocol lacks several vital security features found modern protocols, making it susceptible to a range of attacks that would be impractical against more contemporary communication standards.

  Therefore, there is a motivation for attackers to avoid targeting higher-level protocols equipped with more modern security standards. Instead, the focus tends to shift towards directly exploiting the vulnerabilities present in the CAN protocol, which makes protecting it all the more important.

3.2 CAN Design Overview

According to the original speification by Bosch, CAN has the following main properties:

- **Noise immunity**: the signal needs to be resistant against electric disturbances and electromagnetic interference.
- **Multicast**: CAN messages can receive multiple ECUs.
- **Multimaster**: there is no central bus master, every ECU can act like one.
- **Priority Handling**: CAN messages are prioritized so that critical ECUs get immediate bus access.
- **Time Synchronization**: necessary to correctly prioritize messages.
- **Error Detection and Automatic Retransmission**.
- **Fault Confinement**: achieved by detecting and switching off defect nodes.
- **Low Cost and Lightweight**: allows to save on both cost and weight of copper wiring in a vehicle.
- **Ease of access**: one point of entry to communicate with all ECUs.

The ISO 11898 standard covers both the physical and data link layers and is composed of four parts [27].

Part 1 describes the data link layer composed of the logic link control (LLC), media access control (MAC) and physical coding sub layers.

Part 2 describes the high-speed physical layer, it is by far the most popular standard for the physical layer.

Part 3 describes the low-speed physical layer, it allows communication to continue even if there is a fault in one of the two wires. It is also referred to as "fault tolerant CAN".

Part 4 describes time-triggered communication.

Part 5 and **6**, pertaining to power modes and selective wake-up functionality, previously existed independently but have since been withdrawn and merged into part 2.

The role of CAN is often presented in the 7 layer ISO OSI model [Figure 3.2].

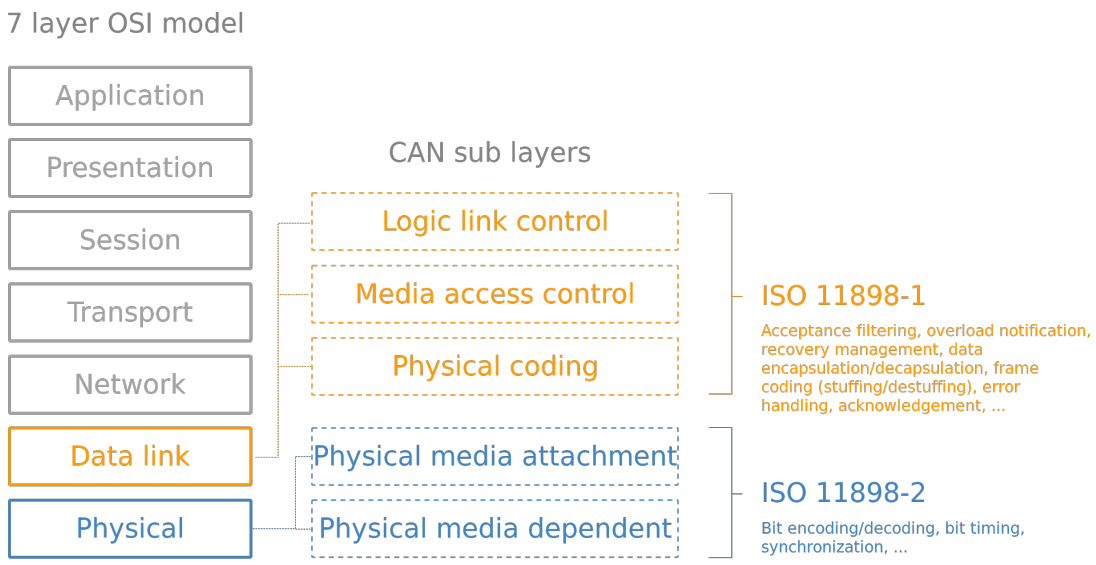


Figure 3.2: CAN physical and data link layers in relation to the ISO OSI model. Image from CSS Electronics [14].

3.3 Physical Layer

The original CAN specification (ISO 11898-1) did not provide specific requirements for the physical layer, instead it relied on abstract criteria. This decision

aimed to promote broader adoption thanks to the additional freedom given to the manufacturers.

However, this approach resulted in potential interoperability challenges among CAN bus implementations. To address this issue, two standards were subsequently introduced: ISO 11898-2 (established in 2003) and ISO 11898-3 (established in 2006).

3.3.1 Logic Signaling

All nodes are connected to eachother though a two-wire bus. The wires are a twisted pair and the specification requires a nominal impedence of 120 Ω. The CAN bus must be terminated using resistors, which are needed to suppress signal reflection and to return the bus to its idle recessive state.

Two signals, CAN high (CANH) and CAN low (CANL) can be either driven to represent a dominant state, or not driven to represent a recessive state. By comparing the voltages of CANH and CANL we can determine if the bus is in the dominant ($CANH > CANL$) or recessive ($CANH \leq CANL$) state.

A dominant signal is always able to overwrite a recessive one. In practical terms, if a dominant bit is transmitted at the same time of a recessive one, all nodes on the bus, including the senders, will read the dominant bit.

Finally a bit is assigned to denote the logical state of the bus using a wired-AND configuration, wherein 0 represents a dominant state, and 1 represents a recessive one [Figure 3.3].

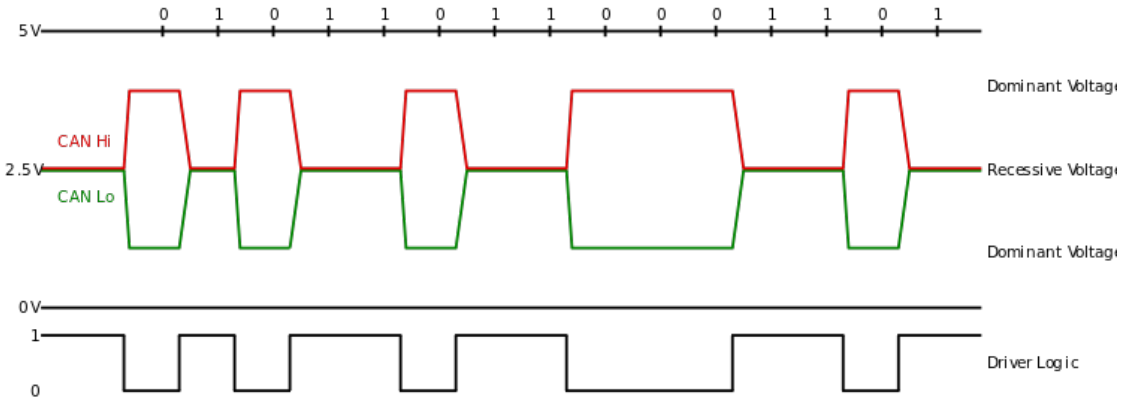


Figure 3.3: High-speed CAN signaling. ISO 11898-2. Image from Wikipedia, the free encyclopedia [43].

3.3.2 Electrical specifications

ISO 11898-2, also called **high-speed CAN**, uses a linear bus terminated at each end by 180 Ω resistors. It supports bitrates from 10 kbit/s to 1 Mbit/s.

Nodes can drive the CANH wire to 3.5 V and the CANL wire to 1.5 V, signaling a dominant state. If no device is transmitting a dominant bit the resistors passively return the voltage of CANH and CANL to a nominal 0 V, usually considering a difference less than 0.5 V to be recessive. The dominant differential voltage is 2 V.

ISO 11898-3, also called **low-speed** or **fault-tolerant CAN**, can use different bus combinations. Approved topologies include linear bus, star bus or multiple star buses connected by a linear one.

In contrast to high-speed CAN, each node in a low-speed CAN bus requires its own individual termination resistance, contributing to an overall resistance that should be close, but not inferior to 100 Ω. Low-speed bus support bitrates from 10 kbit/s to 125 kbit/s.

Low-speed signaling operates similarly to high-speed CAN, but with larger voltage swings. The dominant state is transmitted by driving CANH towards the power supply voltage (vdd) and CANL to 0 V. In a recessive state, CANH voltage is pulled towards 0 V and CANL is pulled to vdd.

This allows for a simplification of the CAN transciever, which needs to only consider the sign of $CANH - CANL$.

Another key difference from the high-speed bus is that in the event of a fault in one of the two wires in the twisted pair, the low-speed bus can still communicate.

Today, the adoption of ISO 11898-3 remains low and usage has been limited to a few niche segments in the automotive industrys.

3.3.3 Node Architecture

In both high-speed and low-speed can, nodes within a CAN network necessitate three components to properly interface with the bus [Figure 3.4].

- **Microcontroller**: it is the Central Processing Unit of the node.

  Its role involves parsing and processing the received complete CAN frames, determining appropriate actions to take. Given the broadcast nature of communication on the bus, the microcontroller must decide whether the received message is pertinent to its function.

  Moreover, the microcontroller is tasked with relaying responses on the bus or transmitting sensor data if that corresponds to the node's designated role.

- **CAN controller**: is in charge of correctly implement the CAN specification. This includes automatically adding stuffing bits, perform error handling and

acting as a buffer for the serial communication.

More specifically, when receiving, the controller stores the received serial bits until a full CAN frame is available in his internal buffer. Then it triggers an interrupt to the microcontroller, prompting it to retrieve the message from the buffer.

When the microcontroller sends a message to the CAN controller, the latter waits until the bus is free and then transmits the bits serially.

- **CAN transciever**: serves as as an interface between the CAN controller and the bus. Its role is to translate the logical bits into an electrical voltage and vice versa.

Often the CAN controller is integrated into the microcontroller for cost-effectiveness and space efficiency reasons. Additionally, microcontrollers with the entire node stack embedded (including the CAN controller and CAN transciever) are available.

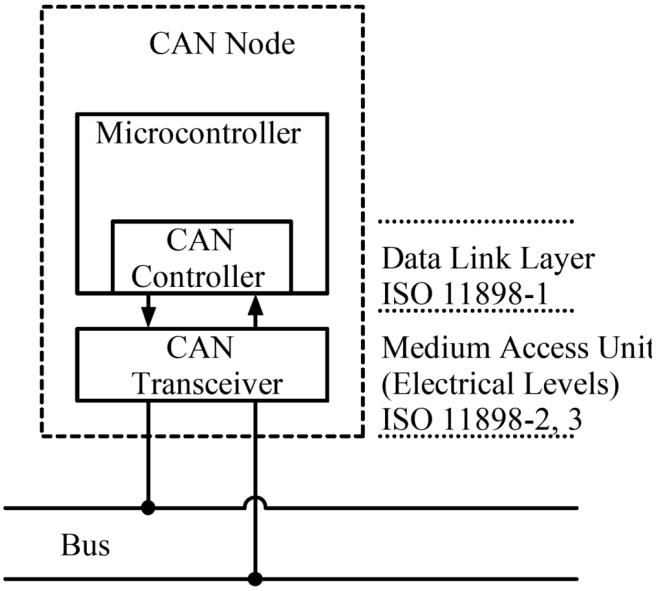


Figure 3.4: CAN bus Node. Image from Wikipedia, the free encyclopedia [43].

3.4 Data Link Layer

The focus of the original Bosch specification is the data link layer, its objective is to act as an interface between the physical layer and the upper ones. The current standard for this layer is ISO 11898-1, which is laregely unchanged from the Bosch specification, except for the clarification of a few ambiguities.

3.4.1 Arbitration

As stated previously, the CAN protocol is a multi-master protocol, so to avoid conflicts between the different nodes the data transmission is preceeded by an arbitration phase. The objective of this phase is to establish which node is the master for the current transmission, so that nodes with lower priority will yield control of the bus to the higher priority master.

The arbitration process makes use of the dominant/recessive property of the bits transmitted on the bus. After the start of frame identifier, the nodes start transmitting their arbitration id on the bus.

If a node transmit a dominant bit, while another transmit a recessive one, there is a collision and the dominat bit wins. This method of automatic collision resolution allows for no additional delays during transmission.

After writing a bit on the bus each node reads the current value of the bus, if a node reads the same bit it wrote it will continue by writing the next bit of the arbitration sequence. Conversely if a node writes a recessive bit, but reads a dominat one it realizes that there has been a contention with an higher priority node, so it will yield the control of the bus and stop transmitting [Figure 3.5].

A node that loses arbitration will re-queue its message and attempt retransmission six-bit cycles after the end of the dominant message, restarting the arbitration process.

Because the dominant bit is the one with logical value 0 and the bits of the arbitration id are transmitted from the most significant to the least, messages with lower arbitration id will always win arbitration against higher ids by transmitting more zeros at the start.

This means that the arbitration id is used not only to identify the message sent, but also to set its priority relative to the other messages.

An important consequence of this arbitration process is that arbitration ids must be unique on a single CAN bus. If that were not the case, transmission would continue for multiple nodes after the arbitration phase, causing contention and subsequently an error.

A single node may send frames with different arbitration ids, but a single arbitration id cannot be shared by multiple nodes.

To allow the arbitration method to work properly, all nodes on the bus are required to be synchronized. Nonetheless CAN is not considered a synchronous protocol because data is transmitted in an asynchronous format, meaning without a clock signal. This implies the necessity of a synchronization mechanism.

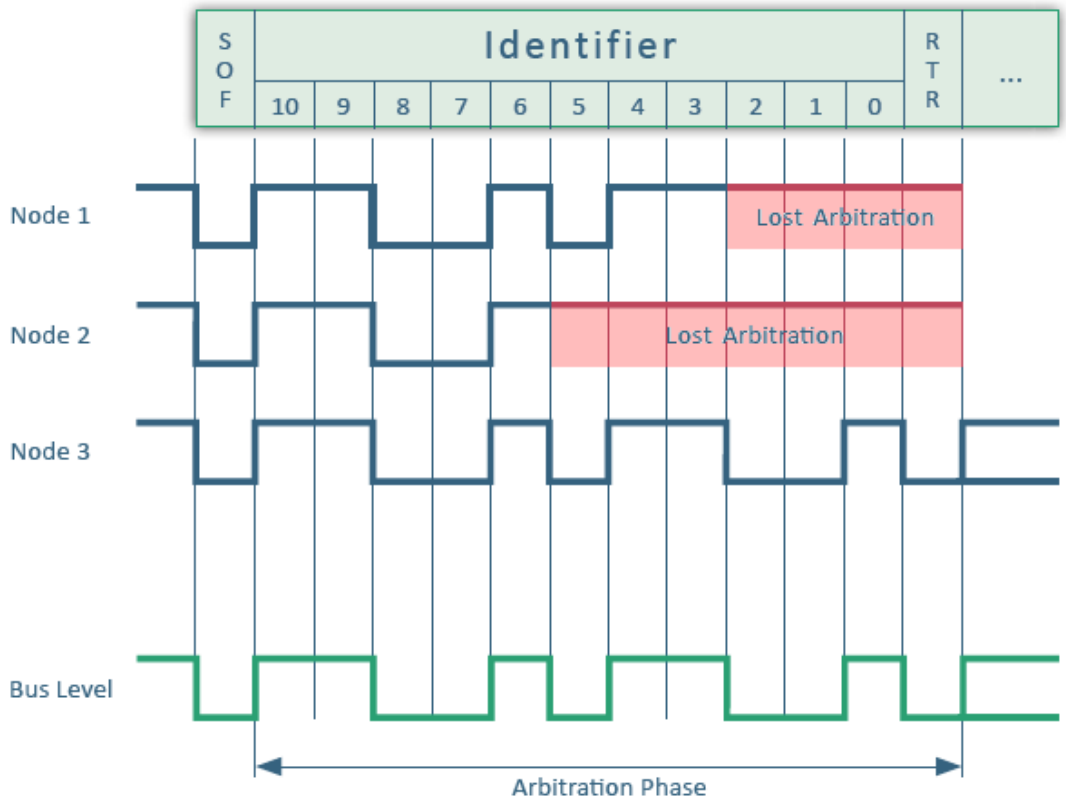


Figure 3.5: Arbitration example between three nodes in an 11-bit ID CAN network. Node 1 transmits a message with id 1631, node 2 uses id 1663 and node 3 uses id 1625. Node 2 is the first to lose arbitration since it has the highest id, while node 3 wins arbitration and procees with transmitting the rest of the frame.

3.4.2 Bit Timing and Synchronization

All nodes in the CAN network must operate at the same nominal bit rate, but the presence of noise or tolerance between the components, means that the actual bit rate may vary from the nominal bit rate.

An absence of synchronization can lead nodes to read or write incorrect data on the bus, rendering the arbitration process ineffective and resulting in errors.

Due to the lack of a clock, synchronization is performed using the signal data on every recessive to dominant transition.

Synchronization is achieved in two ways:

- **Hard Synchronization**: occurs on the first recessive to dominant transmission, which always happens at the start of the message since the bus goes

from an idle state (recessive) to the start of frame bit (dominant).

- **Resynchronization**: occurs on every recessive to dominant transition during the frame. The CAN controller checks if the transition happens at a multiple of the nominal bit rate and, if not, it adjusts the nominal bit rate accordingly.

To calculate the necessary adjustment, the controller divides the nominal bit time into slices called quanta, the amount of which varies according to the controller.

The quanta are then allocated among four different segments: synchronization, propagation, phase 1 and phase 2. The sample point, crucial for calculating the bit rate, is positioned between the phase 1 and phase 2 segments [Figure 3.6].

The synchronization segment is alway 1 quantum long and the propagation segment is used to account for physical delay among nodes. The lengths of the two phase segments can be extended or shortened by an amount calculated by the CAN controller to compensate for desynchronizations.

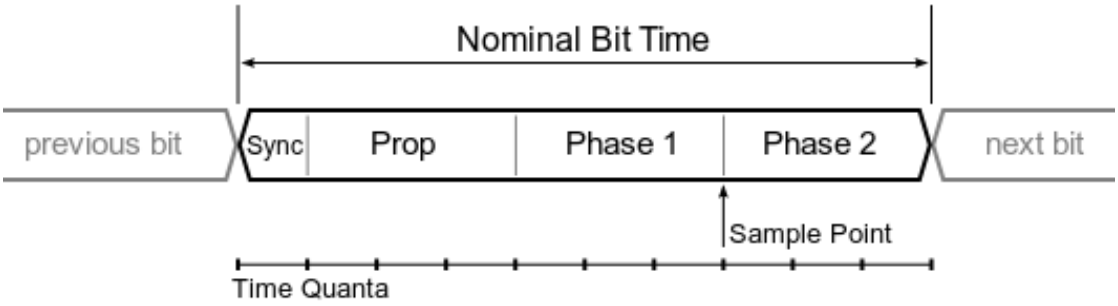


Figure 3.6: CAN bit timing example with 10 time quanta per bit. Image from Wikipedia, the free encyclopedia [43].

3.4.3 Bit Stuffing

CAN uses a non-return-to-zero encoding, indicating the absence of a neutral or rest condition for the bus. The bus idle states corresponds to a recessive bit, holding logical value of 1.

Consequently, messages may feature several consecutive dominant or recessive bits, with the signal level remaining unchanged for an extended duration. As synchronization relies on a recessive to dominant transition, a constant signal may lead to node desynchronization.

To alleviate this issue, CAN employs a technique called bit stuffing: after five consecutive bits of the same polarity, a bit of the opposite polarity is inserted.

Every field in a CAN frame except CRC delimiter, ACK and end of frame field are subject to bit stuffing. In fields where stuffing is used, six consecutive bits of the same polarity are considered an error.

Receiver nodes are tasked with destuffing the signal, by discarding the next bit each time five consecutive bits of the same polarity are received.

This stuffing and destuffing process guarantees that the original bit sequence remains unaltered, while still providing consistent re-syncrhonization.

3.4.4 CAN Frames

The CAN standard describes two frames formats that can be supported by a CAN network, the standard or base format and the extended frame format.

The only difference between the two formats is the length of the arbitration id field: the base frame supports an 11 bit identifier (base identifier) and the extended frame supports a 29 bit identifier. The extended frame identifier is made up of the 11 bit base identifier and an 18 bit identifier extension.

To distinguish between a base and an extended frame, the CAN protocol makes use of the IDE (IDentifier Extension) bit, which is dominant for base frames and recessive for extended ones.

Finally the CAN standard describes four different frame types: data frames, remote frames, error frames and overload frames.

Data Frame

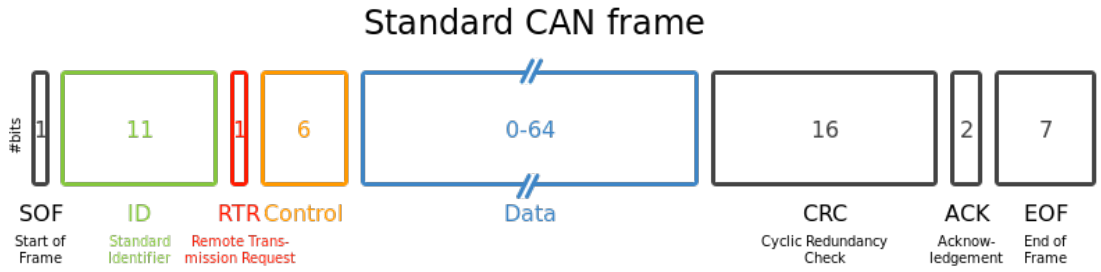


Figure 3.7: Base CAN data frame, excluding stuff bits.

A data frame, represented in figure 3.7, is composed of:

- **Start of Frame** (**SOF**): 1 bit, must be dominant. It denotes the start of the frame and is used to synchronize sender and receivers.
- **Arbitration ID**: An 11 bit long unique identifer. It also serves as priority for the message.

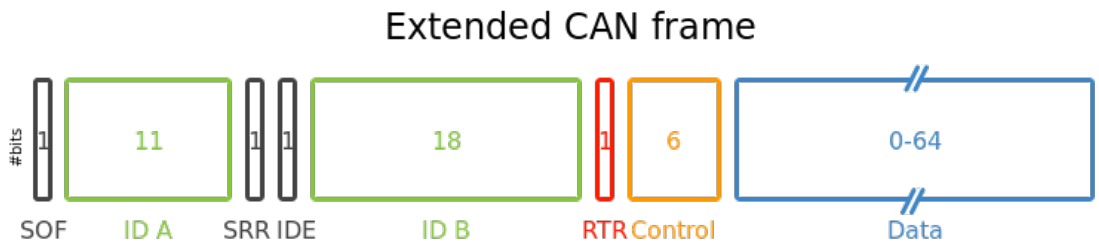


Figure 3.8: ID, Control and Data portions of an extended CAN bus frame, to highlight the differences with the base frame.

- **Remote Transmission Request** (**RTR**): 1 bit flag. Must be dominant for data frames or recessive for remote frames (desribed below).
- **Control**: 6 bits with multiple purposes.
  - **Identifier extension** (**IDE**): 1 bit flag. Must be dominant for 11 bit standard CAN frames or recessive for 29 bit extended CAN frames (described below).
  - 1 **reserved bit**. Must be dominant, but accepted as either.
  - **Data Length Code** (**DLC**): 4 bits. Describes the length of the data field in bytes, from 0 to 8[3].
- **Data**: 0-64 bits. Data to be transmitted, the length of this field is dictated in bytes by the DLC field.
- **Cyclic Redundancy Check** (**CRC**): 15 bits for the CRC sequence, followed by 1 recessive bit for the CRC delimiter. Used for checking the integrity of the received message. Uses the CRC-15-CAN polynomial.
- **ACK**: 1 bit for the ACK and 1 recessive bit for the ACK delimiter. The sender writes a recessive bit, receivers can write a dominant bit to acknowledge the correct reception of the message. The transmitter can read the ACK bit to verify if *any* node received the sent frame, and possibly re-send it.
- **End of Frame** (**EOF**): 7 consecutive recessive bits. Used to signal the end of the frame.

[3]It is possible to put in 4 bits values higher than 8, although the maximum length of the data field still remain 8 bytes. The behaviour for invalid lenghts depends on the implementation of the controller.

Extended data frames share many of the fields with base data frame, with the following distinctions:

- **Arbitration ID A**: 11 bits. Contains the first part of the arbitration id.
- **Substiture Remote Request** (**SRR**): 1 bit. Replaces the RTR bit, must be recessive.
- **Identifier extension** (**IDE**): 1 bit flag. Must be recessive for extended frames.
- **Arbitration ID B**: 18 bits. Contains the second part of the arbitration id. The full 29 bit id must be unique and represents the priority of the message.
- **Control**: 2 reserved bits, followed by 4 bits for the DLC.

Remote Frame

Typically, nodes autonomously transmit data; for instance, a sensor might continuously send data frames to be processed by another ECU. However, the CAN standard, offers a method to request data from the source by sending a remote frame.

A remote frame is similar to a data frame, with a few key differences:

- the RTR bit is recessive instead of dominant;
- the data field is never present;
- the DLC indicates the length of the requested data, not the transmitted one.

If a data frame and a remote frame with the same id begin arbitration at the same time, the data frame will win arbitration since the dominant bit indicates a data frame.

Interframe Spacing

Data frames and Remote frames are separated from preceding frames by a field called interframe space. It consists of at least three recessive bits.

The next dominant bit received after the interframe spacing is considered a start of frame.

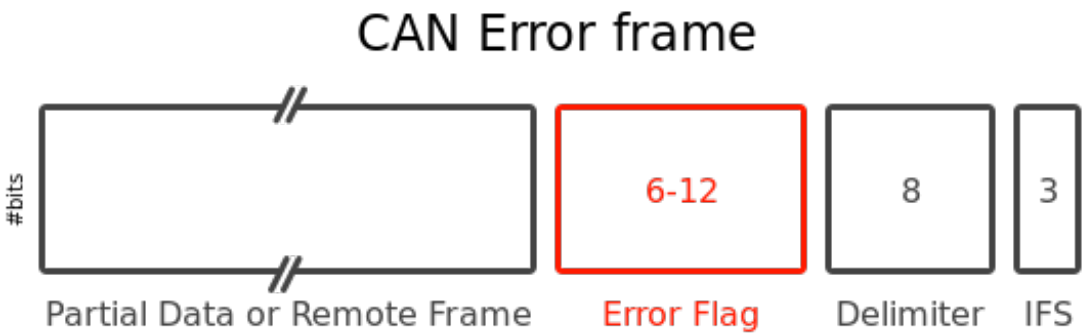


Figure 3.9: CAN error frame.

Error Frame

Error frames are different from data and remote frames: instead of being transmitted after the interframe spacing, they are allowed to be transmitted at the same time of an ongoing data frame or remote frame, interrupting it.

Error frames rely on purposefully violating the bit stuffing rule and transmitting six consecutive dominant or recessive bits to signal the error.

Error frames are composed of two fields:

- Superimposition of **error flags**: 6-12 dominant or recessive bits, which are contributed by different nodes, each transmitting 6 bits possibly at different times.
- **Error delimiter**: 8 recessive bits, signaling the end of the error frame.

The error flag bits can be all dominant or all recessive depending on the state of the node when the error flag is sent.

To determine the current state of the node, each one has two different counters: the **transmitted error count** (**TEC**) and **received error count** (**REC**). These two counters are increased respectively when a node transmits or receives an error flag by varying amounts based on the type of error received. The counters can be decreased if an error is not detected for some time.

The states the node can be in are the following:

- **Active error state**: the starting state of the node. In this state the node transmits active error flags (six dominant bits) whenever it detects an error.
- **Passive error state**: the node enters this state when $TEC \geq 128$ or $REC \geq 128$. In this state the node transmits passive error flags (six recessive bits) whenever it detects an error.

- **Bus off state**: the node enters this state when $TEC \geq 256$. In this state the node is not allowed to transmit anything on the bus and can only read from it.

In conclusion there are five possible error types:

- **Bit error**: detected when a node transmits a bit on the bus, and the value of the read bit differs from the one that was sent. An exception is made for when a recessive bit is sent, but a dominant one is read in the arbitration field or ACK flag of the frame.
- **Stuff error**: detected when the sixth consecutive bit of the same value is read on the bus, violating the stuffing rule.
- **CRC error**: detected when receiving a frame and the CRC calculated from the message does not match the received one.
- **Form error**: detected when a fixed form-bit is the incorrect one. For example if the CRC delimiter, which needs to always recessive, is read as dominant.
- **ACK error**: detected by the sender of a message when no other node transmits a dominant bit for the ACK flag.

Overload Frame

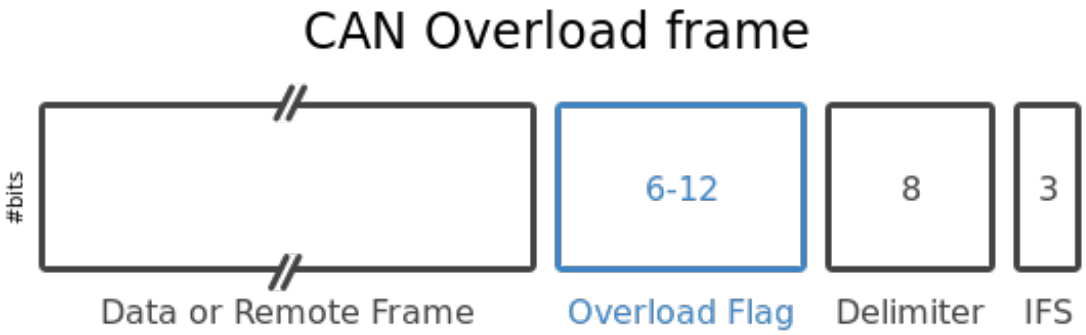


Figure 3.10: CAN overload frame.

An overload frame has the same format of an *active* error frame, but a very different purpose:

- Superimposition of **overload flags**: 6-12 *dominant* bits, contributed by different nodes.

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

An overload frame can only be sent during the interframe spacing and can be sent for the following reasons:

- A node requires a delay of the next data frame or remote frame due to some internal condition. In this case a node has to start sending the overload frame on the first bit of the interframe spacing. Additionally a node is allowed to send a maximum of two consecutive overload frames with the purpose of delaying successive frames.

- A node detects a dominant bit in the second or third bit of the interframe spacing. In this case the node will send an overload frame with a delay of 1 bit.

3.5 Information Security in CAN Networks

CAN was originally conceived and designed with a primary focus on safety rather than security. CAN aimed to facilitate reliable communication among Electronic Control Units, enhancing system functionality and safety.

However the original design lacked robust cybersecurity features, as the prevalent concern at the time was ensuring the dependability and real-time performance of communication in safety-critical automotive applications.

Nowadays, evaluating every network through the lens of information security has become imperative.

Information security is the practice of protecting information by mitigating risks. It is often defined as "Preservation of confidentiality, integrity and availability of information. Note: In addition, other properties, such as authenticity, accountability, non-repudiation and reliability can also be involved." (ISO/IEC 27000:2018 [29]).

The original CAN specification violates most information security principles, therefore making it extremely vulnerable against cybersecurity attacks.

- **Confidentiality** is the ability to provide data only to authorized actors, it is achieved with encryption.

  However, due to the broadcast nature of the CAN protocol and its lack of encryption, there are no means to guarantee confidentiality. This allows any attacker to simply listen in the conversation between ECUs, causing a violation of privacy.

- **Integrity** is the accuracy, completeness and validity of the data, it is usually achieved by using a digest.

  The CAN protocol uses a 15-bit CRC to protect against transmission errors. However, it does not protect against attacks due to the possibility of collisions and the lack of authentication.

- **Availability** is a concept that focuses on ensuring that a system is available when requested.

  Due to the nature of dominant/recessive bits in CAN and the priority based arbitration, instigating a denial of service attack in the CAN network is straightforward for an attacker, thus limiting availability.

- **Authenticity** is the ability to verify who the author of a message is, it is usually achieved with MAC (Message Authentication Code), digital signature or encryption with a shared key.

  The CAN protocol does not provide any of the aforementioned methods for assuring authenticity, so it is trivial for an attacker to impersonate a node on the network and send arbitrary messages.

- **Accountability** and **non-repudiation** are concepts centered around establishing responsibility and preventing denial of involvement in specific actions.

  While less critical in the automotive industry when compared to online communications, these concepts remains relevant for tracking specific operations, such as firmware updates.

- **Reliability** is the ability of the system to deliver its intended functionality accurately and without errors.

  Reliability is provided by the CRC, ACK and the ability of the ECUs to automatically re-queue failed messages.

3.6 Threat Mitigation Technologies

Restructuring the CAN standard to inherently incorporate essential cybersecurity features presents a significant challenge, given its extensive adoption and critical role in automotive communication systems.

For this reason, emerges a necessity to implement measures to mitigate existing threats. A study, performed by the United States Government Accountability Office [40] in 2016, identified possible countermeasures.

- **Cryptography**: encryption and authentication can be used to protect the data field of the frame and verify the legitimacy of message senders and receivers.

  The limited size of the data field poses a challenge for CAN encryption. This issue can be overcome by splitting the encrypted message over multiple CAN frames. But, as the number of ECUs rises, so do the amount of messages, making this solution non-scalable in the long term.

  Another concern arises from the restricted computing power of ECUs, limiting the ability to use more complex encryption algorithms.

- **Hardware Security Module** (**HSM**): an external phyisical security unit that is installed into an ECU. An HSM is equipped with its own CPU and is able to perform various cryptographic functions, like encryption, decryption and digital signatures.

  The HSM effectively manages the computational workload associated with cryptographic functions, mitigating the previously mentioned challenges. However, its adoption implies the installation of an additional component for every ECU seeking to encrypt its traffic, significantly raising costs of production.

- **Network Segmentation**: consists in decoupling the CAN network into distinct sections based on their criticality and interconnecting them through a gateway. Additionally the gateway can be equipped with a firewall to ensure only approved messages are sent across it [Figure 3.11].

  This network segmentation practice enables the isolation of non-critical systems (e.g., infotainment, air conditioning) from critical systems (such as engine control). As a result, even in the event of a compromise in a non-critical system, the potential for transmitting malicious messages to critical systems is minimized.

- **Intrusion Detection System** (**IDS**): security device that monitors network traffic and analyzes it for potential security issues. Additionally an **Intrusion Prevention System** (**IPS**) can be employed to automatically intervene and halt ongoing threats.

  The main advantage of an IDS is that it requires little to no changes in the network structure or ECUs and the bus traffic does not increase.

  IDS rely on a database of known attack signatures to detect a threat, so keeping the database up to date is necessary to make effective use of an IDS.

Among these methods network segmentation and IDS are the ones believed to be the most cost-effective solutions. IDSs are especially useful when safeguarding

older vehicles due to the minimal modifications needed for integration with the CAN bus.

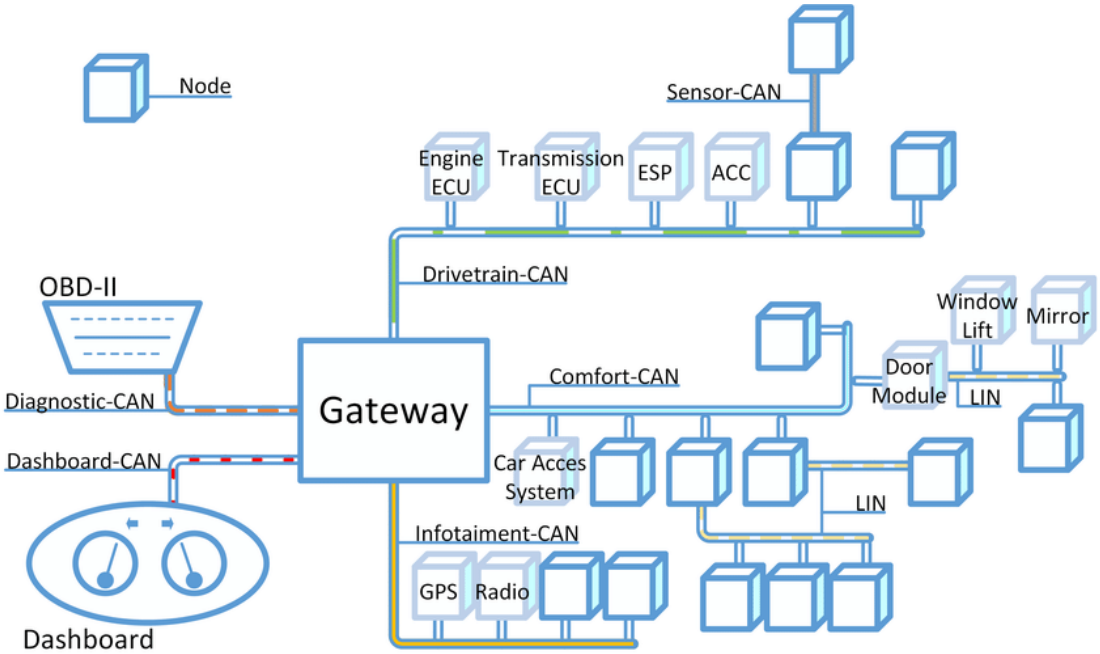


Figure 3.11: A gateway is used to decouple the CAN network in multple sub-networks. Image from Van Barel [24].

Chapter 4

Solution Design

The aim of this project is to develop a modular CAN fuzzer with dual functionality: as a standalone tool for conducting fuzz testing on automotive networks and as a foundational base for future future developments and integrations with other tools.

A survey of existing solutions in this domain reveals a gap in the availability of modular CAN fuzzers. Existing results from web searches yielded limited outcomes, including open-source proof-of-concept projects lacking advanced features [19, 1] or closed-source commercial products with limited modularity and integration capabilities [26].

The objective is to recreate the many features available in commercial products, while eliminating the necessity of using proprietary hardware that typically is necessary to run the associated commercial solution.

4.1 Key design principles

The project has been designed and developed with adhering to the following principles, acting as foundations for the decisions concerning its structure and components.

- **Ease of Use**: the finished product is designed to be intuitive and easy to learn, requiring only basic knowledge about the CAN protocol. To facilitate usability, a straightforward graphical user interface has been created, allowing the user to oversee every aspect of the fuzzer without the need for expertise in command line operations or programming languages.
- **Ease of Integration**: the modularity of the software facilitates the incorporation of additional modules or the utilization of a single module as part of a larger project. This is achieved by adhering to the best coding practices,

with the purpose of avoiding technical debt, and comprehensive documentation which covers all functionalities offered by the modules.

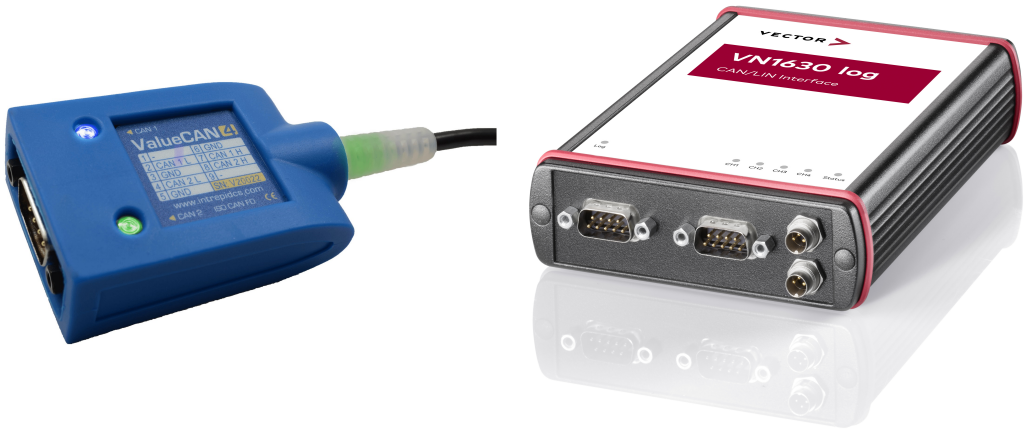
- **Platform Agnostic**: the fuzzer is designed to work independently of operating system, so no platform specific function were used.
- **Protocol Agnostic**: the fuzzer is a black box fuzzer, so it is designed to work independently of car model or protcol. Developed modules reflect this decision by using generic types that can be easily replaced or modified if support for an additional protocol is required.

4.2 Key components

As mentioned previously, the fuzzer has been developed with a emphasis on modularity. Certain modules have been designed to be used with another, while some can be used independently. Additionaly some modules offer different options to choose from to perform the same task [Figure 4.2].

- **CAN Interfaces**: this module is the foundation for the communication with the hardware. It is often required by other modules. Multiple options are available to allow the program to interface with different CAN devices or virtual interfaces [Figure 4.1].
- **Fuzz Generators**: the main objective of this component is to prepare and generate data for the fuzzer. Multiple generation are available depending on the need of the fuzzer.
- **Fuzzer Loop**: the core of the fuzzer. It requires both a CAN interface and a fuzz generator. The purpose of this module is to continously generate data from a fuzz generator and send it via the CAN interface. Multiple instances of the fuzzer loop can be run in parallel if fuzzing to multiple interfaces at the same time is required.
- **Logger**: this module is responsible for logging both the sent and received data on the CAN bus. It requires the user to specify the CAN interface to log and supports multiple file types and log outputs.
- **User Interfaces**: the user interface module is optional and can be omitted on headless devices for a more lightweighted application.

The CAN interface and fuzz generator modules have been developed with the intent of being easily extensible. Adding support to different protocols and fuzzing strategies is straightforward.



(a) ValueCAN 4-2 [37] (b) Vector VN1630 [21]

Figure 4.1: Example of CAN network interfaces.

4.3 Additional components

To complement the fuzzer, additional modules were designed to work alongside it.

- **Virtual ECU**: to facilitate testing of the fuzzer during development, a simple virtual ECU has been developed. This ECU requires the CAN interface module to communicate with the fuzzer.
- **CAN Reverse**: this module allows a user to easily find the CAN frame triggering a specific event. It requires both a CAN interface to communicate with the device under test and a fuzz generator to supply the messages for exploration.

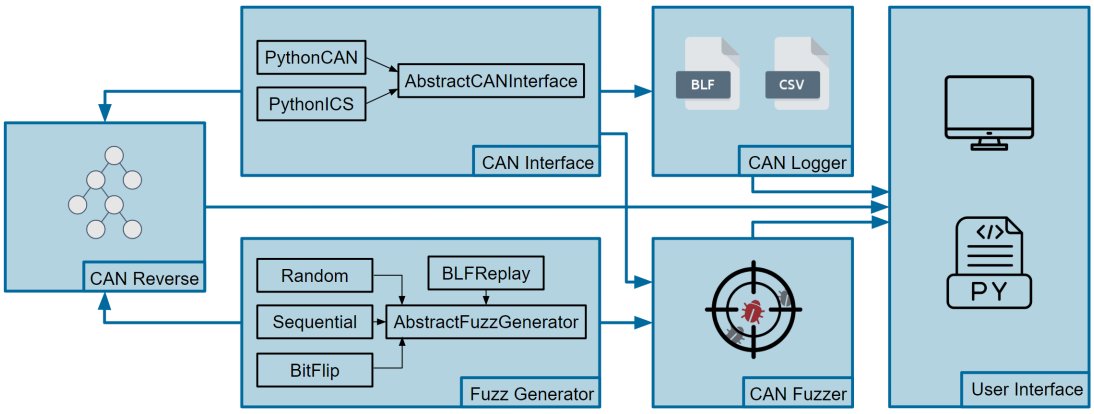


Figure 4.2: Developed software modules and their relationship.

Chapter 5

Implementation

In this chapter we will introduce the technologies used during the development of this project, followed by an in-depth expaination of the implementation of the various modules, including the rationale behind the decisions and the challenges faced during this process.

5.1 Technologies and Libraries

The project has been developed from scratch in python, using the $python-can$ and $python-ics$ libraries for handling the communication with the hardware, and the $PySide6$ library to create an intuitive user interface.

5.1.1 Python

A key requirement for this fuzzer is its capability to run on multiple platforms, for this reason the Python programming language was selected.

Python [18] is a high-level, multi-paradigm, known for its simplicity, readability, and versatility. It provides a dynamic, object-oriented approach to software development and emphasizes code readability, making it easier for developers to express concepts in fewer lines of code.

Python was conceived in the late 1980s by dutch programmer Guido van Rossum and saw its first official release in 1991. Over the years, its substantial evolution owes much to the open-source model and collaborative efforts within the community. Presently, python has a myriad of community-created libraries encompassing diverse domains, including web development, scientific and numeric computing, desktop GUIs, cybersecurity tools and many more [25].

The rich community environment contributes greatly to the "ease of integration" requirement, enhancing this fuzzer's versatility and adaptability across a wide

spectrum of applications.

The fuzzer has been developed using python 3.11, the latest stable release at the time of writing this thesis.

5.1.2 python-can

The python-can library [7] provides Controller Area Network support for Python, including common abstractions to different hardware devices and utilities for sending and receiving messages.

The core of this library relies on the *Bus* and *Message* classes which provide an abstraction for communicating with different virtual or hardware CAN devices.

Other notable features of this library include:

- error handling for identifying CAN error frames;
- compatibility with both base and extended CAN frames;
- synchronous and asynchronous message management;
- logging messages in a variety of text based or binary formats.

Moreover, the python-can library proved to be an excellent educational resource thanks to its easy to learn and straightforward design. The added advantage lies in the possibility to simulate virtual CAN devices, facilitating testing scenarios without the need for physical hardware. This feature enchances the library's utility as a learning tool for those seeking hands-on experience with CAN communication.

5.1.3 python-ics

Python-ics [13] is a python library for interfacing with Intrepid Control System [38] hardware.

Primarily functioning as a wrapper around libicsneo, a C++ API for interfacing with various hardware devices from IntrepidCS, this library exhibits a more intricate interface compared to python-can. Its usage demands a deeper comprehension of the CAN bus, adopting a lower-level approach reminiscent of C programming.

The selection of this library was dictated by the project's network interface requirements, specifically the utilization of the ValueCAN 4-2 interface [Figure 5.1] developed by IntrepidCS.

5.1.4 PySide6

PySide6 [16] is the official python module for the Qt for Python project [11].

Figure 5.1: ValueCAN 4-2 [37] developed by IntrepidCS.

Qt is a cross-platform framework for GUI design. he selection of this library was influenced by its capacity to develop a unified GUI for various platforms. Additionally, Qt Designer, a visual design tool, allows programmers to quickly create the user interface through intuitive graphical representations.

5.2 CAN Interfaces

The first and perhaps most important component implemented is the CAN Interface, serving as a crucial abstraction layer for hardware communication while providing an API for higher level modules to use.

In the initial stages of development, the python-can library was deemed sufficient for handling a virtual CAN bus.

As illustrated by this brief example, the library makes use of the *can.Bus* class to easily define an interface, exposing the *send* and *recv* methods that higher level components can leverage for basic communication. Additionally, the *can.Message* class is used as a generic container for different types of CAN frames.

```
import can

bus = can.Bus("default", interface="virtual")

msg = can.Message(arbitration_id=0x01, data=[0, 1, 2, 3])
bus.send(msg)

msg2 = bus.recv()
```

Python-can offers comprehensive support for a multitude of virtual and physical CAN network interfaces [10, 9] including the $NeoViBus$ tailored for the ValueCAN 4-2 interface, which we will be using for testing the fuzzer.

Despite the compatibility with this network adapter, an alternative wrapper for interface communication was developed. This decision was motivated by a desire to improve the extensibility of the module and to gain access to the lower-level functionalities provided by network devices. This choice not only enhances the module's flexibility, but also provides the programmer with additional customization options.

Moreover, python-can includes support for numerous interfaces and features that are not essential for our application. By opting not to rely on it, we can streamline our application, creating a more lightweight solution tailored to our requirements.

To abstract the communication between higher-layer modules and the hardware, two classes were creted: $MessageEnvelope$ and $AbstractCANInterface$. The former serves as a generic container for various types of CAN frames, including support for base and extended IDs, CAN FD, and error frames. The latter, an abstract class, is defined as follows.

```
class AbstractCANInterface(ABC):

    @abstractmethod
    def close(self):
        pass

    @abstractmethod
    def send(self, message: MessageEnvelope):
        pass

    @abstractmethod
    def receive(self, waitForever: bool = False) ->
        tuple[List[MessageEnvelope], int]:
        pass
```

Expanding this module to support a new network interface is a straightforward process. It involves extening the $AbstractCANInterface$ class and implementing the three required methods, alongside any additional function necessary for the specialized interface.

Two child classes were implemented within this module: $PythonICSInterface$, supporting any device produced by Intrepid CS, such as the ValueCAN 4-2, and $PythonCANInterface$, designed to work with any device supported by the python-can library. The relationship between these classes and other modules is illustrated in Figure 5.2.

The $PythonCANInterface$ intruduces an additional layer of abstraction on top of the one provided by python-can. Therefore, its direct usage is not recomended. Instead, this module should be extended by implementing another child class specific to a particular device. Despite this, it has been included anyway for compatibility reasons and its utility in testing, thanks to the provided virtual interfaces.

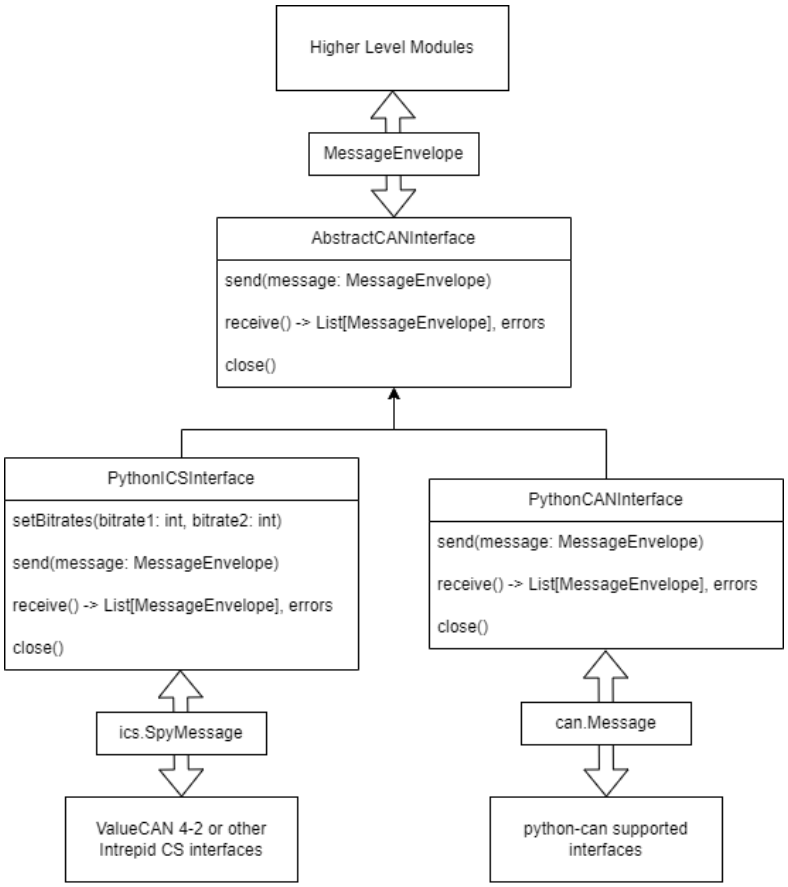


Figure 5.2: CAN Interface overview.

5.3 Virtual ECU

To assess the correct functionality of the implemented CAN Interface, a virtual ECU has been developed, structured around a simple state machine. The ECU is instantiated through a class named *Node*, which necessitates any implementation of *AbstractCANInterface*, a list of states, an initial state, and a *stateTransition* function.

The *Node* class automatically manages incoming messages received via the specified CAN interface. It consults the *stateTransition* function to determine whether the received message should trigger a transition to the next state.

Moreover, the ECU can transition to an idle state, to avoid unnecessary polling, if no communication is detected on the bus within a specified timeout and can be awakened by receiving any message.

Nodes are also capable of sending messages. By connecting multiple virtual ECUs to the same interface, a rudimentary CAN network can be replicated.

Although the ECU's functionality is too basic to provide profound insights into the fuzzer's efficacy, it serves as a useful tool for testing the functionality and reliability of the other modules.

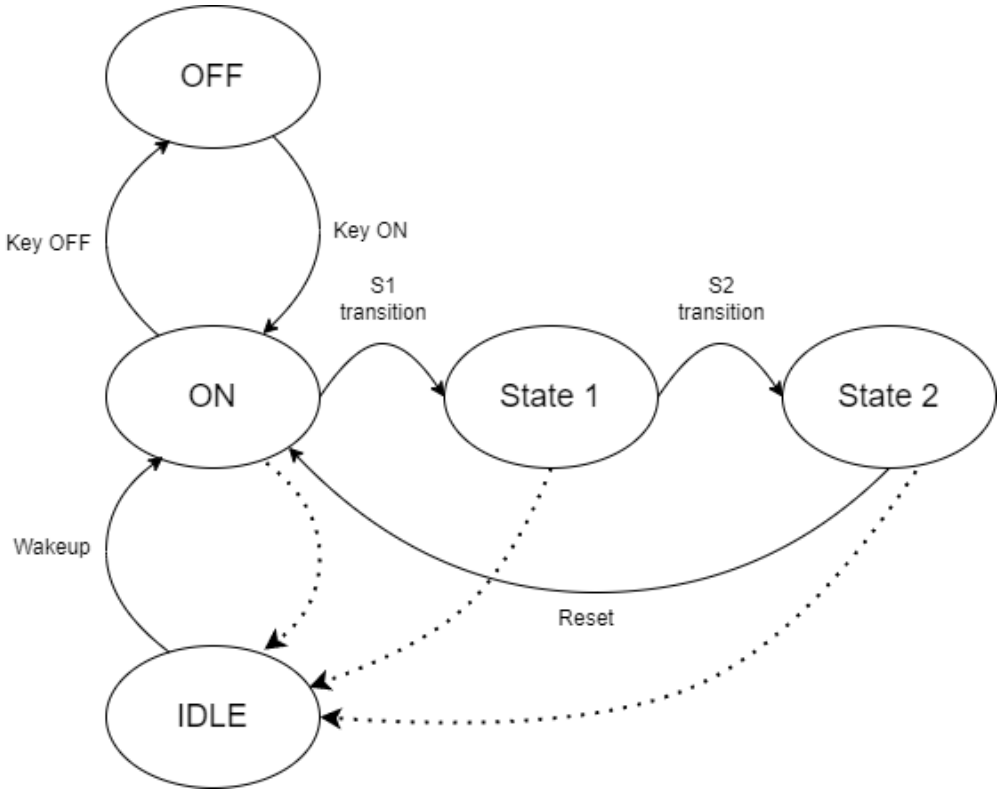


Figure 5.3: Virtual ECU state diagram.

5.4 Fuzz Generators

A fundamental element of the fuzzer is the generator responsible for producing the fuzzed messages. This module is built upon the abstract class *AbstractFuzzGenerator* defined below.

```
1   class AbstractFuzzGenerator(ABC):
2
3       @abstractmethod
4       def __next__(self) -> Optional[MessageEnvelope]:
5           pass
```

Opting for a custom iterator class, as presented in this implementation, is preferred over a function generator (implemented in Python with the *yield* keyword).

This choice aligns with the strict typing employed in this project. By making all fuzz generators child classes of *AbstractFuzzGenerator*, we can annotate attributes in the fuzzer with this abstract class, rather than using the generic *Generator* type. This enhances the code's consistency.

Similarly to the CAN interfaces, developing a new fuzz generator involves extending this abstract class. The current roster of implemented fuzz generators includes:

- **RandomFuzzGenerator**: a straightforward fuzz generator, it generates CAN frames in a completely random manner. It requires a seed, minimum and maximum arbitration ids to randomize, minimum and maximum DLC, whether it should employ extended ids or CAN FD, and optionally a non-RandomData field for fixed data bytes.
- **SequentialFuzzGenerator**: another straightforward generator, it brute forces all possible combinations of CAN frames in specified ranges. It requires the same parameters as the *RandomFuzzGenerator*.
- **BitFlipFuzzGenerator**: this generator is designed to mutate a given starting message by flipping bits in the data field. It requires a *MessageEnvelope* to mutate, the maximum number of bit flips to perform and optionally a mask to specify if some bits should not be flipped.
- **BLFReplayFuzzGenerator**: this generator is designed to replay the content of a BLF file[1], encompassing a CAN conversation that was previously

[1] A BLF file (Binary Logging Format) is a message-based format for storing CAN messages, developed by Vector Informatik GmbH.

recorded by the Logger module or other compatible applications. It requires two parameters: the file name and a flag indicating if only "transmitted" messages should be replayed.

5.5 Fuzzer

The fuzzer component is fairly straightforward, its main purpose is to serve as an orchestrator for lower level modules. These modules, in turn, manage the majority of the logic.

To instantiate a $Fuzzer$, two objects are required: an implementation of any $AbstractCANInterface$ and an implementaiton of any $AbstractFuzzGenerator$. The purpose of the fuzzer is to simply iterate over the fuzz generator, obtain a CAN frame, and send it though the CAN interface.

Additionally, the $Fuzzer$ class provides methods to start and stop the fuzzing loop. It requires a parameter to determine the delay between messages and optionally accepts callback functions for handling events, such as sending a message or reaching the end of the generator.

The fuzzer is a subclass of a python $Thread$ so instantiating multiple fuzzers is as simple as calling the $Fuzzer$ constructor multiple times.

5.6 Logger

The logger component operates similarly to the fuzzer. To instantiate a $Logger$, an implementation of any $AbstractCANInterface$ is required, as it will be utilized to continuously monitor the CAN bus and log all transmitted and received messages on it.

The $Logger$ class provides methods to start and stop it and is a subclass of $Thread$, similar to the $Fuzzer$ class.

Initially, there was consideration to include the logging functionality directly inside the fuzzer module since they share a common CAN interface and structure. However, keeping them in separate modules allows for greater flexibility. For instance, it is possible to run multiple fuzzers on the same interface and use only a single logger instead of having to use multiple of them. Alternatively, one can run the logger without a fuzzer if only monitoring of the CAN bus is required.

The logger currently supports three different logging methods:

- Python's builtin *logging* module: by default the log is output on stdout, but can be configured to write to files instead. A human readable format is used for this output, its main utility is debugging if few messages were exchanged over the network.

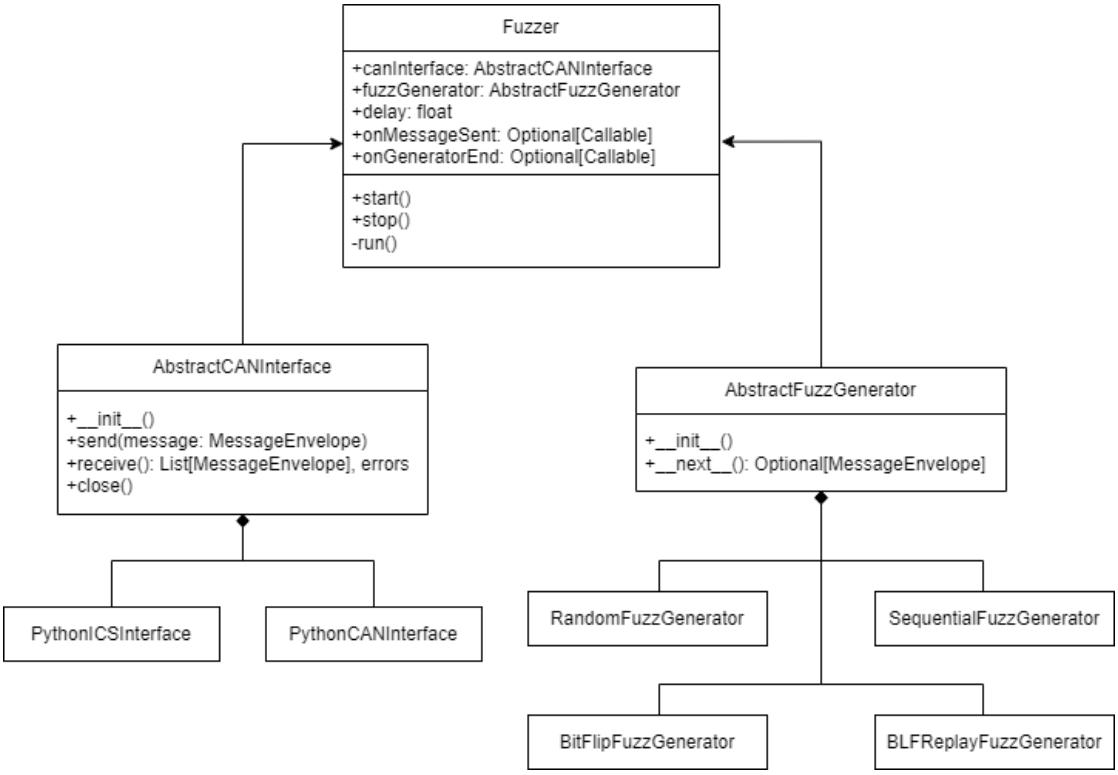


Figure 5.4: Fuzzer UML diagram.

- CSV file: useful because it can be processed by many tools. Since it is an ASCII based format, file sizes for these log files tend to be larger.
- BLF file: proprietary Binary Logging Format developed by Vector Informatik GmbH. Since it is a binary file, the size is a lot smaller compared to CSV, but fewer tools support this format. Mainly used to analize the conversation using CANalyzer or as an input file for the $BLFReplayFuzzGenerator$. The code for the $BLFWriter$ and $BLFReader$ classes was adapted from python-can's blf utility [8]. This was done to allow compatibility with the $MessageEnvelope$ object instead of $can.Message$, removing the dependence on the python-can library.

5.7 CAN Reverse

The CAN reverse module has been developed with the intent of facilitating reverse engineering of CAN frames.

Upon instantiation of the $CANReverse$ class, an interactive session is started, taking manual input to quickly identify a single message causing an observed effect.

This is performed by first loading a specified number of messages into the $CANReverse$ instance by using any fuzz generator (typically by replaying a BLF file, although data can be randomly generated if preferred).

Subsequently, all stored messages are transmitted to the CAN interface. If the desired effect is not observed, we can proceed by loading the next section of messages, until the fuzz generator reaches its end. If the desired effect is observed, the CAN reverse module can replay only the first half of the messages and we can indicate to the CAN reverse module wether the effect is observed. By proceeding in this manner, an efficient binary search is performed and it is possible to correctly identify the CAN frame causing the desired effect in $log_2(section\_size)$ steps [Figure 5.5].

For instance, a practical scenario could involve connecting the logger to the CAN network, then use the car's key to unlock the doors and finally use the CAN reverse module to correctly identify the message used to unlock the car. This method allows the identification of any frame needed, proving especially valuable for cybersecurity testing by simulating the process an attacker might employ to target a vehicle without prior knowledge.

Moreover, automated detection of the desired effect enables the complete automation of the CAN reverse procedure. This is possible in cases where the effect is transmitted over the CAN network (for example error frames or specific messages) or if the effect can be observed with specialized sensors and its output transmitted to the CAN reverse module.

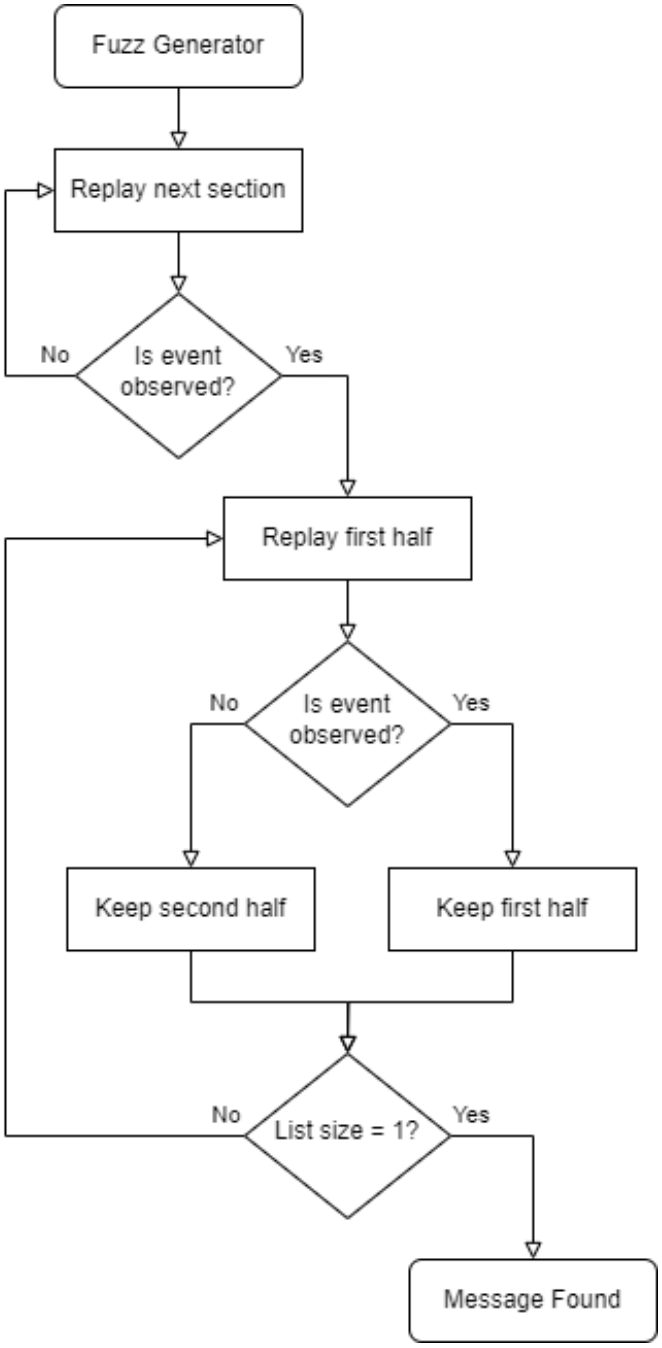


Figure 5.5: Identifing a CAN frame using CAN Reverse.

5.8 User Interface

This project has been mainly developed to be run automatically in an headless environment, but a proof of concept graphical user interface has been created in PySide6 to showcase the main features of the fuzzer and CAN reverse modules.

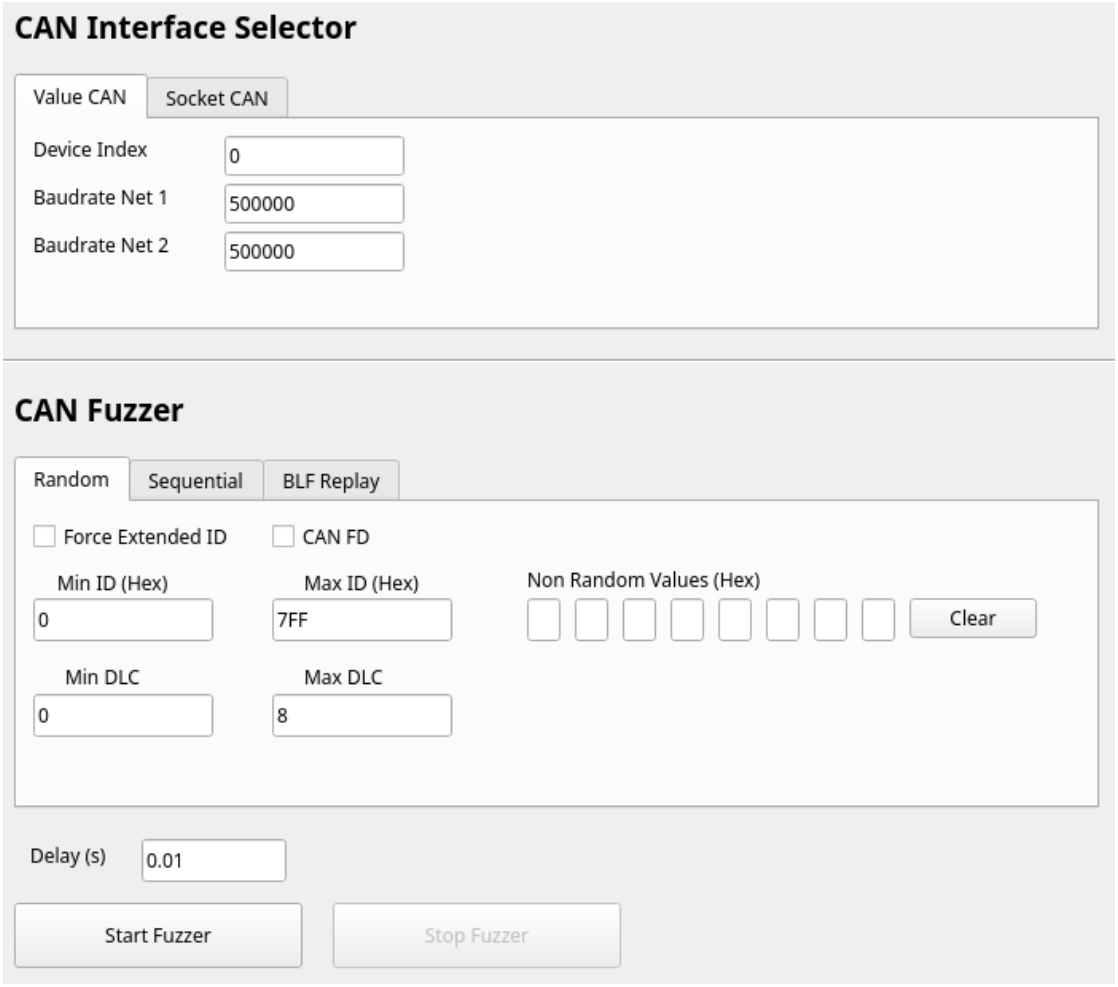


Figure 5.6: Fuzzer graphical user interface. The interface selector allows the user to specify which CAN interface to use and its configuration options. The CAN fuzzer section, allows the user to specify the Fuzz Generator and its parameters.

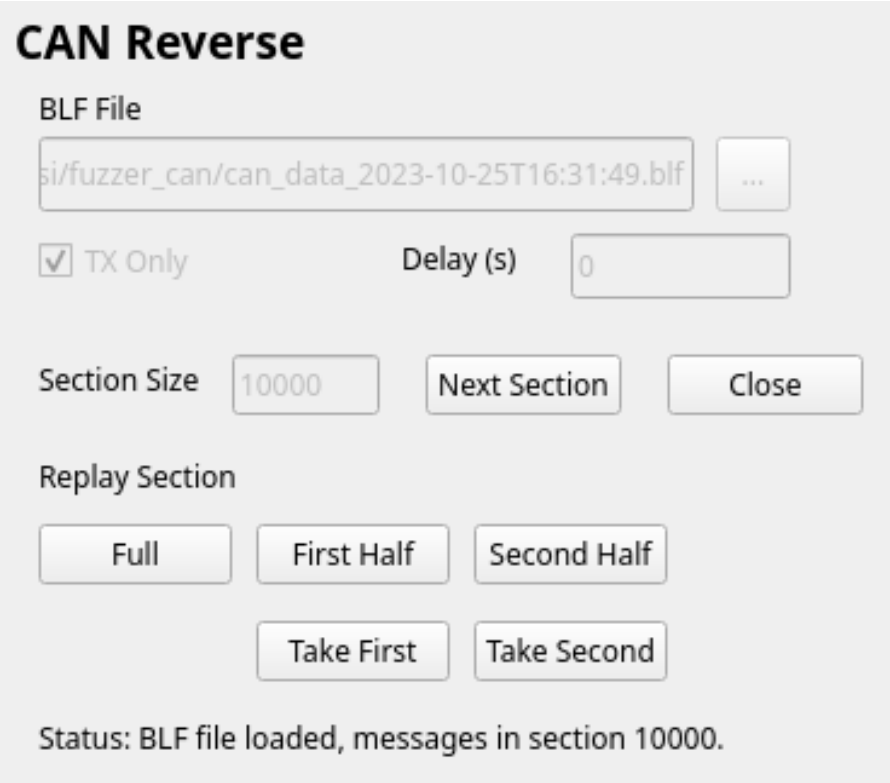


Figure 5.7: CAN Reverse graphical user interface. This module allows for loading sections from the BLF file to replay, buttons are available to replay the two halves of the section and then choose which of the two to keep. Refer to figure 5.5 for the usage.

Chapter 6

Testing and Results

To validate the fuzzer's effectiveness, testing has been conducted on a Dell Latitude E6540, a laptop with an Intel Core i5-4200M and 8 GB of RAM, running Arch Linux 6.7.4.

During testing, different setupts were employed, some leveraging the developed virtual ECU, while other using a testing bench with physical ECUs.

The physical CAN interface used during testing is a ValueCAN 4-2 developed by Intrepid Control Systems, supporting multiple CAN based protocols and OBD2 diagnostics. The ValueCAN 4-2 supports simultaneously two different CAN network, but during testing only the first of the two networks has been employed.

The objective of the experimental evaluation is to verify the correct functioning and performance of the fuzzer. The main objective of the virtual ECU is to allow the fuzzer to detect known crashes in a white-box environment, while the bench has been used to emulate a real world black-box scenario and benchmark the performance.

6.1 Full virtual setup

This preeliminary testing setup makes use of SocketCAN [36] and was employed for early tests of the developed CAN interfaces and fuzzer, without the use of the logger.

SocketCAN is an open source CAN driver and networking stack contributed by Volkswagen Research to the Linux kernel. This kernel module, along other command line utilities, is available under the package $can - utils$. Upon installation, a new network device virtualizing a CAN bus can be instantiated and fuzzing can begin.

```
#!/bin/sh

# load the vcan kernel module
modprobe vcan

# add and enable new vcan device named vcan0
ip link add dev vcan0 type vcan
ip link set up vcan0
```

Listing 6.1: Loads the vcan kernel module and creates a new virtual interface named *vcan*0.

```
#!/bin/python

vcan0 = PythonCANInterface(can.Bus(channel="vcan0",
    type="socketcan"))

randomGenerator = RandomFuzzGenerator()

fuzzer = Fuzzer(vcan0, randomGenerator, delay=0.25)

fuzzer.start()
```

Listing 6.2: Instantiates a fuzzer with a random generator for the virtual interface *vcan*0.

The command line utility *candump* can be used to visualize the output of the fuzzer.

```
$ candump vcan0
vcan0  30B   [6]  36 24 27 BE E3 AA
vcan0  7D7   [1]  DF
vcan0  0A0   [4]  2A 63 E0 47
vcan0  523   [6]  04 FC E2 AD FA 28
vcan0  7FE   [2]  6B C1
vcan0  1BA   [5]  F4 D6 FA 6E 82
vcan0  5E4   [8]  CF 1C 78 61 50 45 25 F0
```

Listing 6.3: Example of the fuzzer output obtained with the utility *candump*. The output includes the arbitration ID, DLC and data of the CAN frame.

6.2 CANalyzer

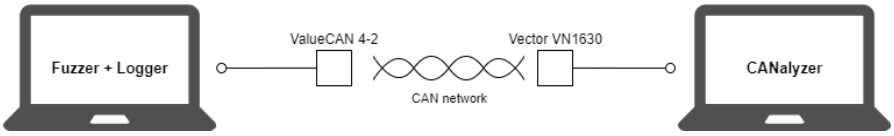


Figure 6.1: CANalyzer testing setup.

To ensure correct behaviour of the developed logger and accurate implementation of the .blf file format, the next test was performed using CANalyzer [20], a powerful analysis and diagnostic tool developed by Vector.

A Vector VN1630 network interface was used on the CANalyzer workstation to receive data from the fuzzer. CANalyzer can log the received CAN frames in a .blf file and, by using a simple script to compare the log files produced by CANalyzer and the Logger module, it was possible to determine that no message was lost, indicating a correct implementation of the BLF format.

An additional test was conducted by allowing the fuzzer to send CAN frames without any delay between messages, with the objective of stressing the CAN network and the hardware interfaces. The fuzzer was left running for 5 minutes and different baudrates for the network interfaces were compared (Table 6.1). Even by stressing the network we were able to determine that no messages were lost and no error frames generated, assuring us of the reliability of the software and hardware devices.

Baudrate	**Messages**	**Message/s**	**Estimated message/s**	**% of estimate**
virtual	3.707M	12,359	N/A	N/A
125.000	423k	1,411	1,563	90.3%
250.000	829k	2,763	3,125	88.4%
500.000	1.630M	5,435	6,250	86.9%

Table 6.1: Fuzzer performance compared to different baudrates.

During this test, the runtime has been set at 5 minutes, the messages have been generated with a *RandomFuzzGenerator* with default parameters and the estimated messages per second is based on an average frame size of 80 bits (standard arbitration ID and 4 data bytes). The virtual interface does not support a baudrate because it does not abstract the transmission delay of a real network, for this reason the thoughput is much higher. Lower baudrates achieve results slightly closer to the estimated maximum due to the overhead of the communication having less impact at lower baudrates.

6.3 Testing Bench

The next test was executed on a test bench equipped with components sourced from a Fiat 500 BEV. The purpose of this test was to emulate a real-world black-box scenario.

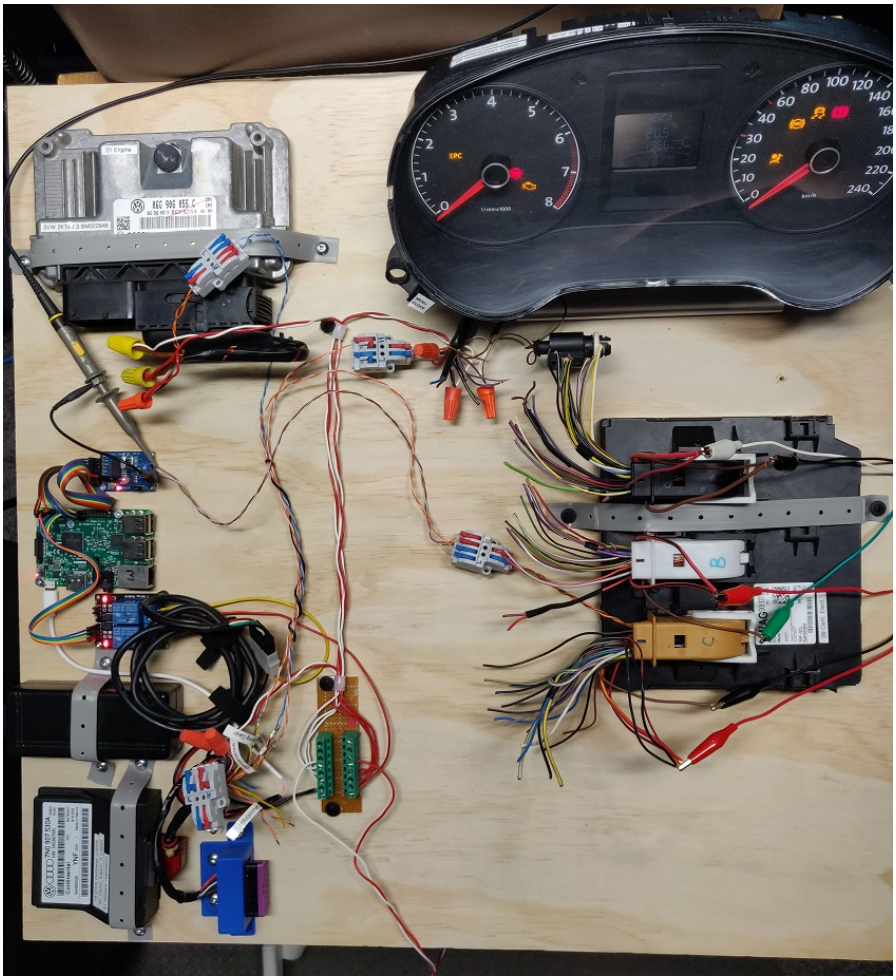


Figure 6.2: Example of a test bench using Volkswagen components. Image from Ross-Tech Forum [32].

The evaluation utilized a *RandomFuzzGenerator*, mirroring the setup of the previous CANalyzer test, with baudrates ranging from 125,000 to 1,000,000. Despite producing no visible response on the test bench, the ValueCAN's LED indicated transmission of error frames.

By utilizing a Y CAN connector and introducing the PC with CANalyzer on the network verified the presence of bit errors, indicating the absence of a termination

Instrument Cluster	Displays vital vehicle information to the driver, including tachometer, fuel gauge and turn signals indicators.
Body Control Module (BCM)	Manages various body-related functions like lighting, door locks, and windows.
Infotainment Control Module (ICM)	Manages audio, video, navigation, and communication features.
Airbag Control Module (ACM)	Controls airbag deployment and emergency lights based on crash sensor inputs.

Table 6.2: Components of the Fiat 500 BEV test bench and their respective functions.

resistance for the ValueCAN device. The issue was addressed by setting the flag *termination_enables* to 1 in the library's configuration, activating a software-controlled termination.

Following this adjustment, the correct transmission of CAN frames was verified by the ValueCAN LED, when transmitting frames with a baudrate of 500,000. All further testing was performed with this rate.

Subsequent fuzzer runs elicited diverse reactions from the test bench, including random fluctuation in the tachometer, activation of the rear view camera, various alarms related to factors like seatbelt status, imminent collision, and others linked to Bluetooth connectivity.

6.3.1 Results

A fuzzer efficacy is measured by the amount of errors or program crashes found in a given amount of time. In the CAN protocol, program crashes translate to an ECU resetting or violating the CAN protocol standard, thus causing error frames to be sent. Program errors are instead communicated to other ECUs by using error flags in the data block of a message.

During the multiple fuzzing runs executed, no error frames were reported, indicating a correct implementation of the CAN standard by all ECUs considered. Additionally, due to the black-box nature of this analysis, databases to interpret the byte sequences received by the ECUs were non used, leaving the logged data impossible to check for the presence of error flags.

Nonetheless, basic traffic analysis was performed on the frequency of received arbitration IDs, to determine the influence of the fuzzer on the CAN network.

Figure 6.3 shows the increase or decrease in the number of receieved CAN frames measured before and after the fuzzer session has started. The number of the received CAN frames has been collected and averaged over multiple 5 minutes

runs, and the ratio between the number of frames during and before the fuzzing session has been plotted. As seen in the figure, only the CAN frames with ID 0x64D[1] showed any noticeable increase in frequency, going from 1201 to 1985 received frames in a 5 minutes interval.

After empirical testing, frame 0x64D is confirmed to be the frame controlling the ticking of the bench's emergency lights. Additionally pressing the respective button on the board does not turn off the emergency lights, so resulting to a board reset is necessary.

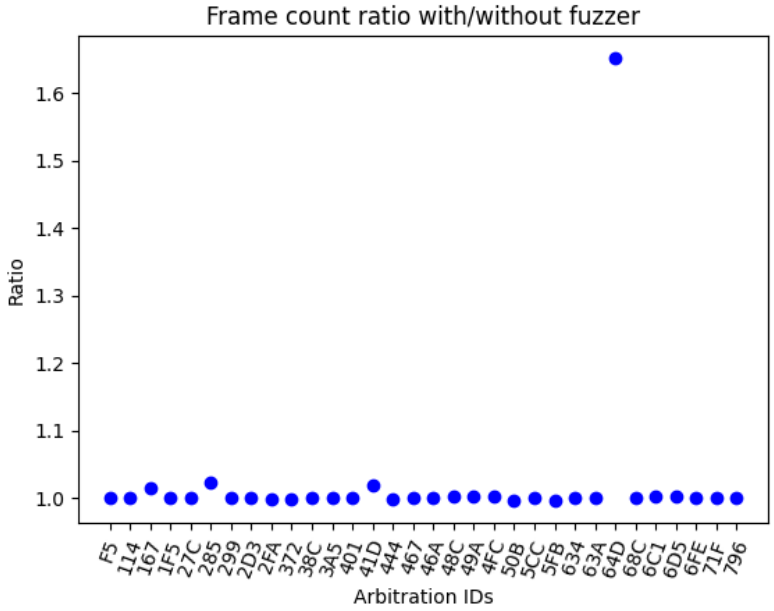


Figure 6.3: Ratio between received CAN frames during and before a fuzzing session. Fuzzing does not seem to affect the frequency of received CAN frames, with the exception of frames with arbitration ID 0x64D, belonging to emergency lights.

6.3.2 Reverse Engineering of CAN Frames

The CAN reverse module, explained in section 5.7, has been purposefully developed to gain some additional insights about the emergency lights CAN frame.

By using the CAN reverse module to replay the .blf log file obtained while fuzzing the test bench, it was possible to quickly identify the specific CAN frame

[1] Arbitration IDs have been replaced and frame data bits have been swapped in these results, to avoid revealing the reverse engineered data.

responsible for turning on the emergency lights.

Through multiple cycles of fuzzing and reversing, a collection of CAN frames with the same arbitration ID, all responsible for activating the emergency lights, was obtained. All the reversed frames shared a uniform length of 4 bytes, but no discernible pattern emerged among the data bytes.

To invesigate this frame further, random bits in one of the found frames were flipped. However none of the resulting frames succeeded in triggering the emergency lights, suggesting the presence of a CRC in the data portion of the message.

Consulting the CAN database for the test bench confirmed the hypothesis: the identified frame's purpose is to activate the emergency lights in the event of an accident. The first three bytes of the data field contain multiple flags, specifying the type of incident and which sensors were triggered, while the last byte serves as a CRC-8 for the preceding three bytes.

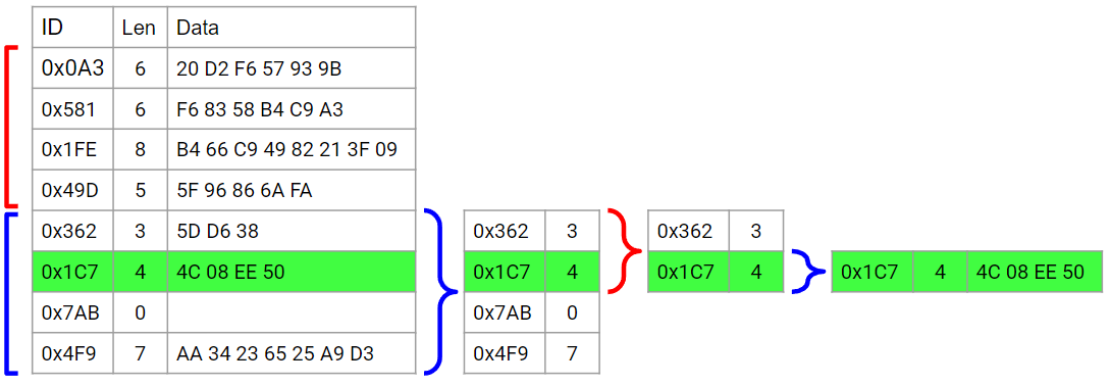


Figure 6.4: Example Usage of the CAN reverse module. The highlighted frame is the one responsible for triggering the emergency lights.

6.4 Virtual ECU on a Physical CAN Bus

To overcome the black-box limitation of bench testing, the next objective was to conduct tests in a white-box environment, achieved by specifically crafting a virtual ECU with known behaviour.

Unlike the initial test that relied on the virtual CAN bus, two ValueCAN devices connected to each other were utilized. Finally, the two devices were connected to different USB ports on the same PC, and a $PythonCANInterface$ was assigned to each based on the connected ValueCAN's index.

Similar to the previous test, one of the interfaces was connected to the fuzzer, while the other was used to instantiate an $EqualityTestECU$, a subclass of the $Node$ class outlined in section 5.3.

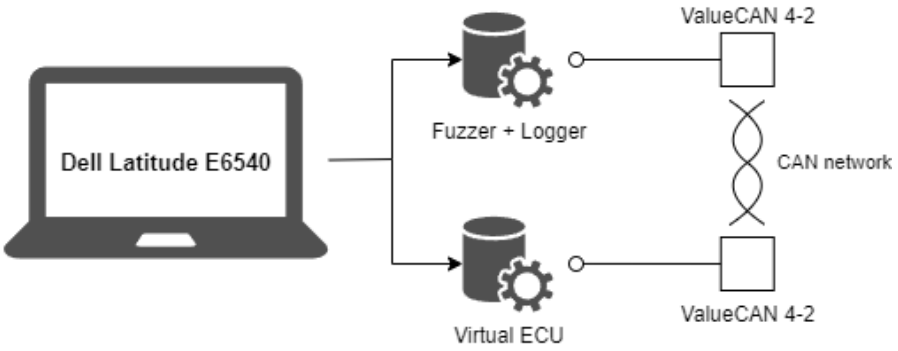


Figure 6.5: Virtual ECU testing setup.

Several of assumption were made when designing the *EqualityTestECU* to mimic the behaviour of the observed ECUs in the test bench:

- The ECU works in cycles of 100 milliseconds.
- The ECU can receive any number of messages per cycle.
- The ECU can be in two states "OK" and "ERROR".
- The ECU exclusively accepts messages with arbitration ID 0x100 and will transition to the "ERROR" state if, during the cycle time, at least one message matches the *messageDataRule*.
- The *messageDataRule* is defined as a dictionary where keys represent bit indices and values represent bit values. A received message matches the rule if all indicated bits match the specified value; all other bit values and DLC are ignored.
- The ECU will transmit a single message with arbitration ID 0x101 every cycle, indicating the current state of the ECU in the data field of the message ("OK" = 0x00, "ERROR" = 0xFF).
- The ECU is reset to the "OK" state at the start of evert cycle.

The setup script used for testing is reported below.

```
#!/bin/python

int0 = PythonICSInterface(0)
int1 = PythonICSInterface(1)

messageDataRule = {
    5: 1,
    6: 0,
    7: 1,
    8: 0,
}

ecu1 = EqualityTestECU(int1, messageDataRule,
    acceptedArbitrationIDs=[0x100])

fuzzGenerator = RandomFuzzGenerator(minId=0x100, maxId=0x100)
logger = Logger(int0)
fuzzer = Fuzzer(int0, fuzzGenerator)
```

6.4.1 Results

When fuzzing an ECU with a fixed response rate, there are two possible approaches: sending messages at a faster or slower rate than the ECU's message rate. Fuzzing at a slower rate requires sending a maximum of one message from the fuzzer for every ECU cycle. This ensures that the logger can accurately pair a "request" and a "response", allowing a straightforward approach to identify which message caused an error. On the other hand, if the fuzzer exceeds the ECU's rate, sent messages must be grouped into "batches" for each cycle. The specific message in the batch causing the error is subsequently identified using CAN reverse.

Figure 6.6 shows the difference in mean time needed to find the error between the two strategies. As expected, the batch fuzzing technique yields better results, even when factoring in the time required to execute CAN reverse. This is due to the overall higher volume of messages sent.

Figure 6.7 compares the mean time needed to find the error among batch fuzzers with different sizes. The results indicate that larger batch sizes result in improved efficiency when dealing with larger error sizes. However, it is important to note that excessively large batches can lead to a decrease in the number of messages sent by the ECU, as the fuzzer may take precedence over the ECU response due to the arbitration feature of CAN.

The earlier findings are applicable only to an ECU that is able to maintain the "ERROR" state. If an ECU that is not able to, and instead responds solely to the last received message in each cycle, sending messages in batches becomes impossible. This forces the fuzzer to operate using the slower-rate strategy.

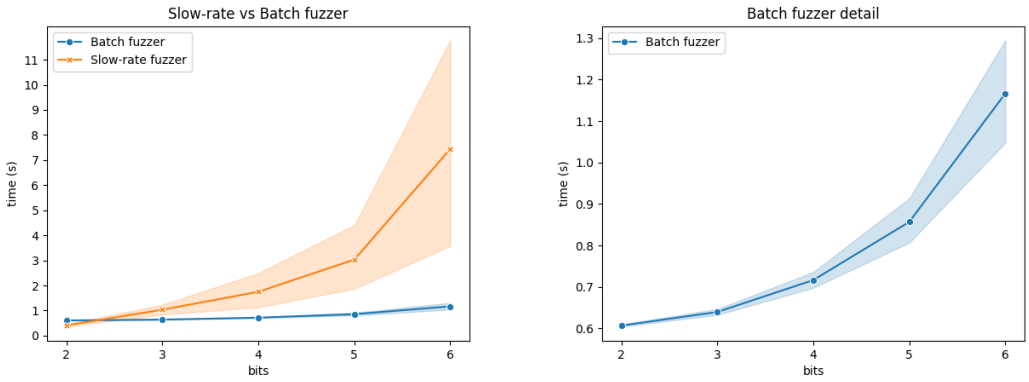


Figure 6.6: Slow-rate vs batch fuzzing. The mean time needed to find an error is plotted. The virtual ECU uses a 100 ms cycle time, the batch fuzzer uses a batch of size 10 for every ECU cycle. In the batch fuzzer case, the time needed to run CAN reverse is included.

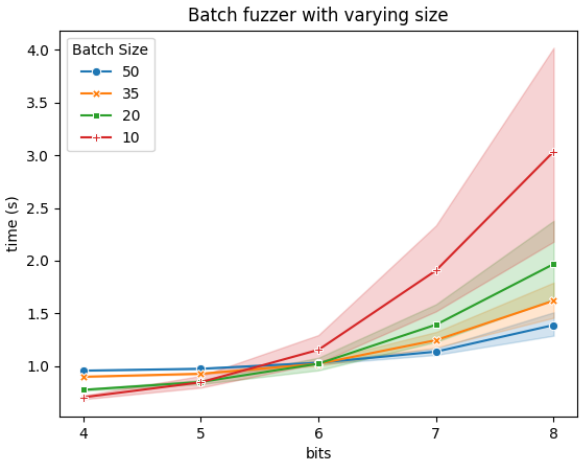


Figure 6.7: Batch fuzzer with varying batch size. The mean time needed to find an error is plotted. The virtual ECU uses a 100 ms cycle time. Note that a slow-rate fuzzer would be equivalent to a batch fuzzer with size = 1. The time needed to run CAN reverse is included.

6.4.2 Error Detection

Allowing the fuzzer to run for an extended period allows the collection of multiple errors, providing a dataset for analysis. To gain additional insights into the behavior causing a specific error, two strategies can be employed.

First, by fixing the arbitration ID and DLC, and by continuously sending random messages, we can collect a larger dataset relating to that specific error. Alternatively, it is possible to use a *BitFlipFuzzGenerator* to mutate the original message in a more controlled way.

The bitflip strategy requires sending fewer messages compared to the randomized strategy. While the first proves to be very effective in pinpointing the specific bits responsible for the error in simple cases, its effectiveness diminishes in more complicated scenarios. In contrast the random strategy may be more useful at identifying errors in more complex situations, such as those involving a CRC. However, due to its random nature, a larger volume of messages needs to be exchanged to accurately identify the bits causing errors.

To identify patterns in the message structure, one effective approach is to generate heatmaps of the data portion of the messages. Each heatmap illustrates the cumulative value of individual bits across all error-inducing messages. The minimum value (always 0) is depicted on the heatmap only if all error-causing messages had that bit set to 0. Conversely, the maximum value (equal to the reported number of errors) emerges when all messages exhibit that bit set to 1. Values in-between count how many messages possess the particular bit set to 1.

Figure 6.8 compares the heatmaps generated by the two strategies. In the bit flip example, a total of 41 messages has been sent (the original one, and one message for every bit being flipped), of these 37 received an error as a response. In the randomized strategy, about 2,000 messages were sent and 115 of them received an error as a response.

Figure 6.9 shows how the introduction of a CRC in the last byte of the message complicates error detection. By using the bit flip strategy, only the original message before mutation is able to cause an error to be reported, thus no additional insights on the message can be gathered from this mutator. By using the randomized strategy, about 57 thousand messages have been sent to the ECU, of which only 18 caused an error, this is sufficient to observe the same pattern as figure 6.8.

If the CRC calculation algorithm is known, the mutation generator can be adapted accordingly. However, in numerous instances, knowledge of the CRC algorithm may not be readily available. As an alternative, the bit flip generator can be modified to use a brute-force approach for the CRC. Although this still requires awareness of the CRC's position and size, it proves effective in revealing the bit pattern with a reduced number of messages compared to the random approach.

Figure 6.10 illustrates a bit flip generator implementing a brute-force technique

for the CRC byte. With this strategy, only 256 messages per bit flip are necessary, resulting in a total of 8,448 messages sent.

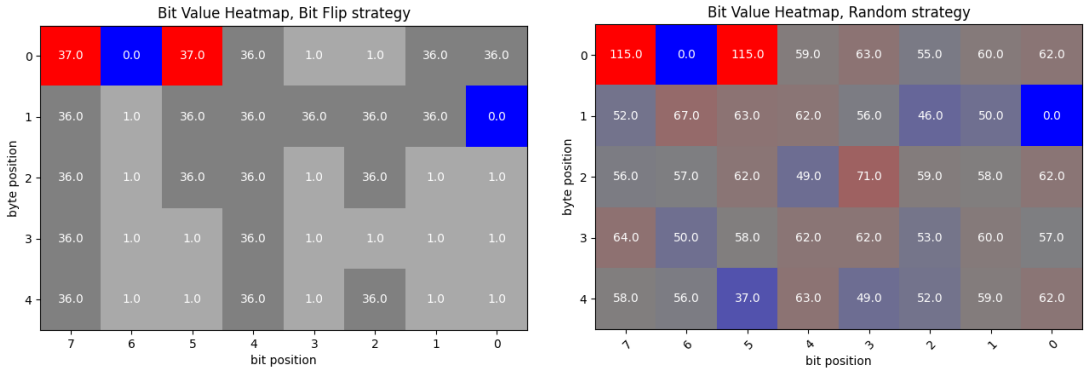


Figure 6.8: Bit Flip vs Random Fuzzing. The sum of bit values for all error-causing messages is plotted in this heatmap. Both strategies were able to successfully identify the patterns in bits with indexes 5-8. The random strategy sent about 2,000 messages to achieve this result.

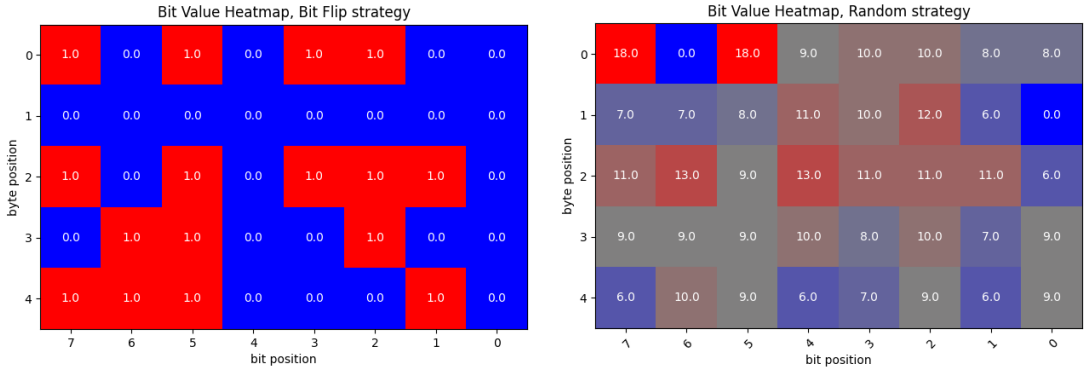


Figure 6.9: Bit Flip vs Random Fuzzing, on an ECU requiring a CRC in the last byte (4). The bit flip strategy proved ineffective, while the random strategy was able to identify the pattern. The random strategy sent about 57 thousand messages to achieve this result.

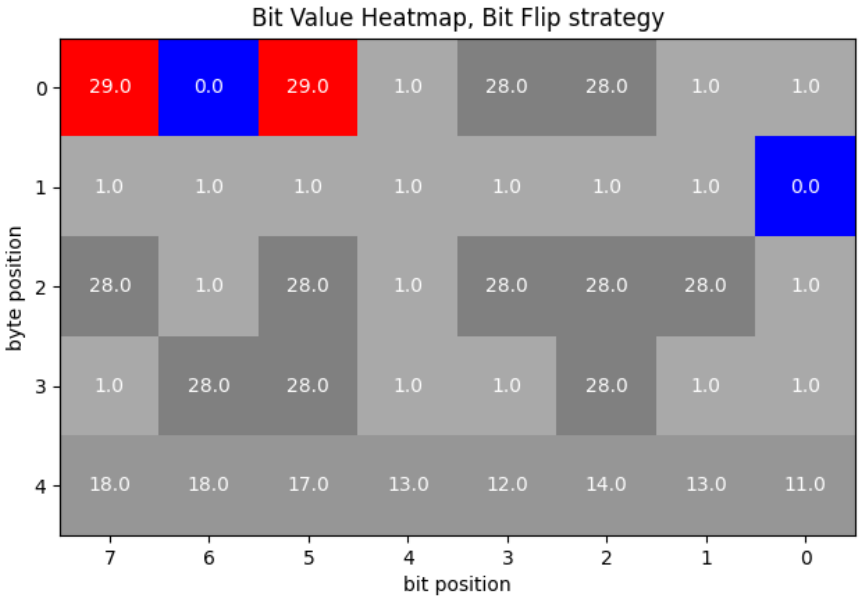


Figure 6.10: Bit Flip fuzzing, with CRC bruteforce. The resulting heatmap is similar to the case without the CRC, except that the bits corresponding to the CRC appear to be randomly distributed.

Chapter 7

Conclusions

This thesis has succeded in developing a CAN protocol fuzzer from scratch, focusing on the modularity and extensibily aspects of the project. By organizing the fuzzer into multiple modules and abstracting the CAN protocol, the project aimed to enhance ease of integration while providing a versatile tool for cybersecurity testing.

The project's contributions include the development of specialized modules for different network interfaces, effectively transforming them into plug-and-play components for the fuzzer. Though the focus was on network interfaces produced by Intrepid CS, the fuzzer can be easily extended to function with interfaces from other manufacturers.

Testing conducted on simulated virtual ECUs and physical test benches validated the fuzzer's effectiveness and highlighted the efficacy of various fuzzing techniques under different assumptions.

Moreover, post-processing techniques, such as analyzing heatmaps generated from message data and implementing mutation strategies, provided insights into error-inducing messages and enhanced the fuzzer's capabilities.

7.1 Future Work

The easiest way to improve the fuzzer would be to extend existing modules, adding new functionalities or refining existing ones based on feedback from testing and usage. For example creating a new CAN Interface to support different hardware devices would be trivial and would greately expand the fuzzer's compatibility.

Additionally, adding support for other CAN-based protocols, such as CANopen and SAE J1939, would be relatively straightforward, requiring only minor modifications to the *MessageEnvelope* class to support the additional features provided by those protocols.

Incorporating support for non CAN protocols, like WiFi or Bluetooth would require additional effort, as it involves different message structures. However, by maintaining the same basic structure and principles, the fuzzer can be adapted to accomodate these protocols, extending its applicability to a wider spectrum of automotive networks and technologies.

To further improve the developed solution, incorporating features of a smart fuzzer could significantly improve its effectiveness. For instance, integrating grammar or model based fuzz generation techniques would enable the fuzzer to generate test cases that adhere closely to the syntax and semantics of the underlying communication protocols. This approach has the potential to uncover more vulnerabilities and edge cases while potentially reducing the runtime needed to find them.

Furthermore, introducing a feedback mechanism within the fuzzer could enhance its adaptability and effectiveness. By collecting and analyzing feedback from the target system, the fuzzer could dynamically adjust its testing strategies and prioritize test cases that exhibit promising results or trigger interesting behavior.

Implementing such features would necessitate integration with database CAN files (DBC), which are widely used in automotive development. These files serve as structured databases containing information about the message, signals, and the encoding scheme used by an ECU. By integrating DBC files in the fuzzer, valuable insights can be gathered from the fuzzed data.

Bibliography

[1] Accenture. can_dlc_fuzzer. URL https://github.com/Accenture/can_dlc_fuzzer.

[2] AUTOSAR. Automotive open system architecture, . URL https://www.autosar.org/.

[3] AUTOSAR. Classic platform, . URL https://www.autosar.org/standards/classic-platform.

[4] Anirban Basu. *Software Quality Assurance, Testing and Metrics*. PHI Learning, June 2015.

[5] Hammerschmidt Christoph. Number of automotive ecus continues to rise. *eeNews Automotive*, May 2019. URL https://www.eenewseurope.com/en/number-of-automotive-ecus-continues-to-rise/.

[6] Cloudflare. What is penetration testing? URL https://www.cloudflare.com/en-gb/learning/security/glossary/what-is-penetration-testing/.

[7] Multiple Contributors. python-can, . URL https://pypi.org/project/python-can/.

[8] Multiple Contributors. python-can – can.io.blf, . URL https://python-can.readthedocs.io/en/stable/_modules/can/io/blf.html.

[9] Multiple Contributors. python-can – hardware interfaces, . URL https://python-can.readthedocs.io/en/stable/interfaces.html.

[10] Multiple Contributors. python-can – virtual interfaces, . URL https://python-can.readthedocs.io/en/stable/virtual-interfaces.html.

[11] Multiple Contributors. Qt for python, . URL https://wiki.qt.io/Qt_for_Python.

[12] Copper Horse. Threat modelling. URL https://copperhorse.co.uk/threat-modelling/.

[13] Rebbe David. python-can. URL https://pypi.org/project/python-ics/.

[14] CSS Electronics. Can bus explained - a simple intro, 2023. URL https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial.

[15] Eurostat. Eu people on the move: changes in a decade. URL https://ec.europa.eu/eurostat/web/products-eurostat-news/w/edn-20230918-1.

[16] Qt for Python Team. Pyside6. URL https://pypi.org/project/PySide6/.

[17] OWASP Foundation. Fuzzing, January 2020. URL https://owasp.org/www-community/Fuzzing.

[18] Python Software Foundation. Python. URL https://www.python.org/.

[19] FrostTusk. Can-fuzzer. URL https://github.com/FrostTusk/CAN-Fuzzer.

[20] Vector Informatik GmbH. Canalyzer, . URL https://www.vector.com/int/en/products/products-a-z/software/canalyzer.

[21] Vector Informatik GmbH. Vector vn1630, . URL https://www.vector.com/int/en/products/products-a-z/hardware/network-interfaces/vn16xx/.

[22] Google. American fuzzy lop. URL https://github.com/google/AFL.

[23] Google. Oss-fuzz – trophies, August 2023. URL https://github.com/google/oss-fuzz?tab=readme-ov-file#trophies.

[24] Thomas Huybrechts, Yon Vanommeslaeghe, Dries Blontrock, Gregory Van Barel, and Peter Hellinckx. *Automatic Reverse Engineering of CAN Bus Data Using Machine Learning Techniques*, page 751–761. Springer International Publishing, November 2017. ISBN 9783319698359. doi: 10.1007/978-3-319-69835-9_71. URL http://dx.doi.org/10.1007/978-3-319-69835-9_71.

[25] Python Institute. Python – the language of today and tomorrow. URL https://pythoninstitute.org/about-python.

[26] Intrepid CS. Vehicle spy. URL https://intrepidcs.com/products/software/vehicle-spy/.

[27] ISO 11898-1:2015. Road vehicles – Controller area network (CAN). Standard, International Organization for Standardization, December 2015.

[28] ISO 26262-1:2018. Road vehicles – Functional safety. Standard, International Organization for Standardization, December 2018.

[29] ISO/IEC 27000:2018. Information technology – Security techniques – Information security management systems. Standard, International Organization for Standardization, February 2018.

[30] ISO/SAE 21434:2021. Road vehicles – Cybersecurity engineering. Standard, International Organization for Standardization, August 2021.

[31] Neystadt John. Automated penetration testing with white-box fuzzing. URL https://learn.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10).

[32] jonese. Testbench setup – ross-tech forum. URL https://forums.ross-tech.com/index.php?threads/5918/page-10#post-226184.

[33] Kohnfelder Loren. The stride threat model. URL https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20).

[34] Mozilla. Browser fuzzing at mozilla, February 2021. URL https://hacks.mozilla.org/2021/02/browser-fuzzing-at-mozilla/.

[35] Mozilla. Bugs found while fuzzing with domino, December 2023. URL https://bugzilla.mozilla.org/show_bug.cgi?id=domino.

[36] Volkswagen Research. Socketcan - controller area network. URL https://www.kernel.org/doc/html/next/networking/can.html.

[37] Intepid Control Systems. Valuecan 4-2, . URL https://intrepidcs.com/products/vehicle-network-adapters/valuecan-4/valuecan-4-2-overview-low-cost-two-channel-can-fd-usb-interface/.

[38] Intrepid Control Systems. Intrepidcs, . URL https://intrepidcs.com/.

[39] TU Dortmund University. Learnlib. URL https://github.com/LearnLib/learnlib.

[40] U.S. Government Accountability Office. Vehicle Cybersecurity: DOT and Industry Have Efforts Under Way, but DOT Needs to Define Its Role in Responding to a Real-world Attack, March 2016. URL https://www.gao.gov/assets/gao-16-350.pdf.

[41] U.S. Government Publishing Office. 40 cfr 86.005-17(h)(3). URL https://www.govinfo.gov/content/pkg/CFR-2010-title40-vol18/pdf/CFR-2010-title40-vol18-sec86-005-17.pdf.

[42] Drake Victoria. Threat modeling, December 2019. URL https://owasp.org/www-community/Threat_Modeling.

[43] Wikipedia, the free encyclopedia. Can bus, 2024. URL https://en.wikipedia.org/wiki/CAN_bus. [Online; accessed February 01, 2024].

[44] Wikipedia, the free encyclopedia. Can bus – history, 2024. URL https://en.wikipedia.org/wiki/CAN_bus#History. [Online; accessed February 01, 2024].