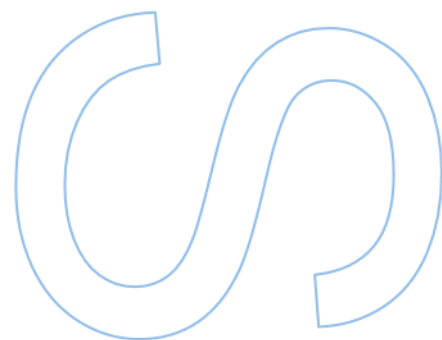
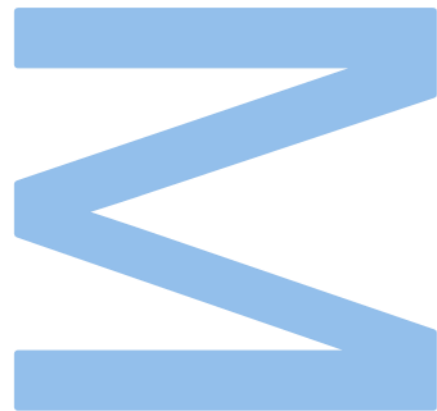


oCANada – Fuzzing Environment In Controller Area Networks



Thales Puget Santos

Master in Information Security
Department of Computer Science
Faculty of Sciences of the University of Porto
2024

oCANada – Fuzzing Environment In Controller Area Networks

Thales Puget Santos

Dissertation carried out as part of the Master in Information Security

Department of Computer Science
2024

Supervisor

Dr. Hugo Pacheco, Professor Auxiliar, FCUP

External Host Supervisors

Dr. Reza Parsamehr, Manager and Team Lead Security and Privacy, Continental Engineering Services

Patrick Grüner, Lead CyberSecurity Specialist, Continental Engineering Services

UNIVERSIDADE DO PORTO

MASTERS THESIS

oCANada - Fuzzing Environment in Controller Area Networks

Author:

Thales Puget SANTOS

Supervisor:

Dr. Hugo PACHECO

*A thesis submitted in fulfilment of the requirements
for the degree of MSc. Information Security*

at the

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

June 11, 2025

"Deeds will not be less valiant because they are unpraised."

JRR Tolkien, The Lord of the Rings

Acknowledgements

I'd like to acknowledge my parents for their love and support over the years.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto

Departamento de Ciência de Computadores

MSc. Information Security

oCANada - Fuzzing Environment in Controller Area Networks

by [Thales Puget SANTOS](#)

The number of cyber attacks against the automotive industry has been on the rise. As such, it is important to develop tools that can help security researchers find security vulnerabilities as quickly as possible. Fuzzing is a popular tool which can help find security vulnerabilities in different kinds of applications. This same technique can be utilized to fuzz the embedded devices in cars known as Electronic Control Units (ECUs) over the Controller Area Network (CAN). Many CAN fuzzers today are simple mutational fuzzers that change the bits in the CAN payload at random. Instead, this dissertation presents a novel generation-based fuzzing technique both utilizing CAN database files (DBCs) in order to generate valid messages, as well as a State-of-the-Art CAN fuzzing techniques which allows more efficient fuzzing. Since the format of the payload for CAN messages is essentially a formatted binary sequence, this paper also utilizes a State-of-the-Art fuzzer known as FormatFuzzer to fuzz CAN. This fuzzer can take in a formatted binary and generate syntactically correct messages. Furthermore, there is a discussion of the techniques used as the test oracle for fuzzing CAN within a greybox/blackbox environment.

Contents

Acknowledgements	v
Abstract	vii
Contents	ix
List of Figures	xiii
Glossary	xv
1 Introduction and Motivation	1
1.1 Introduction	1
1.2 Motivation	2
1.3 Industrial Context	4
2 Background	5
2.1 Automotive Industry Concepts	5
2.1.1 Automotive Security Processes	5
2.1.2 Penetration Testing	6
2.1.3 Vehicle Internals - ECUs, CAN, CAN DBC, and UDS	7
2.2 Fuzzing Concepts and Background	9
2.2.1 Fuzzing Strategies	10
2.2.1.1 Fuzzing Strategy - Mutation	10
2.2.1.2 Fuzzing Strategy - Coverage Guidance	10
2.2.1.3 Fuzzing Strategy - Generation	10
2.2.1.4 Fuzzing Strategy - Grammars	11
2.2.1.5 Fuzzing Strategy - Symbolic Execution and Concolic Execution	11
2.2.1.6 Fuzzing Strategy - Static Analysis	12
2.2.1.7 Fuzzing Strategy - Search-based Software Testing and Genetic Algorithms	12
2.2.2 Seeds and Seed Selection	12
2.2.3 Test Oracle	13
3 State of the Art	15
3.0.1 Relevant SoTA Fuzzers	15

3.0.1.1	AFL	15
3.0.1.2	AFL++	17
3.0.1.3	CaringCaribou	17
3.1	SoTA Generation Based Fuzzers	17
3.1.1	F1 Fuzzer	18
3.1.2	Grammarinator	20
3.1.3	Dharma	21
3.1.4	FormatFuzzer	21
3.2	CAN-based Fuzzers	23
3.2.1	EffCAN	23
3.2.2	autoFuzz - CAN Automatic Sensor Fuzzer Module	24
3.2.3	Structure-Aware Fuzzing	25
3.2.4	CAN-FT	27
3.2.5	CAN Fuzzer Relevance for the Dissertation's Fuzzing Solution	28
3.3	Test Oracles	28
3.3.1	Physical Harness Test Oracle	29
3.3.2	Test Oracle Discussion Fowler et al.	29
3.3.3	Timeout and Test Present Service Test Oracle	29
3.3.4	CAN Monitoring and IMU Sensor Monitoring Test Oracle	30
3.3.5	Image-Based Test Oracle	30
3.3.6	CAN-FT's Test Oracle	31
3.3.7	Test Oracle Relevance for the Dissertation's Fuzzing Solution	31
4	oCANada Fuzzing Environment	33
4.1	Fuzzing Environment Introduction	33
4.2	Lab Setup	34
4.3	oCANada Fuzzing Strategies	36
4.3.1	oCANada DBC Brute Force Mode	38
4.3.2	oCANada DBC FF Mode	38
4.3.2.1	btCompiler (Binary Template Compiler)	38
4.3.2.2	Compiling to Binary	41
4.3.2.3	oCANada fuzzing usage	42
4.3.3	oCANada Reverse FF Mode	43
4.3.4	Test Oracle	48
4.4	Testing Plan	48
5	Results	51
5.1	oCANada Results	51
5.1.1	Fuzzer Runtime Speed	51
5.1.2	Reverse Fuzzer and Evil bit	52
5.1.3	Memory Errors Found	53
5.1.4	Test Oracle utilized with Gateway ECU	54
5.1.5	Fuzzer Accuracy	55
6	Conclusion and Future Work	57
6.1	Conclusion	57
6.2	Future work	58

A Extra Binary Templates	61
---------------------------------	-----------

Bibliography	65
---------------------	-----------

List of Figures

1.1	Instances of Remote vs Physical Attacks Against Vehicles	2
1.2	Reported Automotive CVEs from 2019 to 2023	2
1.3	Common Attacks against automobiles.	2
2.1	Example Database CAN (DBC) message.	8
2.2	Diagram of Traditional Mutational Fuzzer.	10
4.1	Image demonstrating the fuzzing pipeline for the oCANada fuzzing environment.	34
4.2	Simple lab setup for purely simulation.	35
4.3	Lab setup to test real ECUs with the rest of the vehicle simulated.	36
4.4	Figure demonstrating the three modes of operation for oCANada.	37
4.5	Flowchart demonstrating the steps required for oCANada's brute force mode.	39
4.6	Figure showing the phases of the READ algorithm.	44
4.7	Flowchart demonstrating the steps required for oCANada's DBC Format-Fuzzer mode.	47
4.8	Proposed Steps for testing the different fuzzers.	49
5.1	The results from fuzzing the easy simulation using the oCANada BF Mode	53
5.2	The results from fuzzing the easy simulation using CaringCaribou	54
5.3	The results from fuzzing the easy simulation using the oCANada FF DBC Mode	55
5.4	Normal operations of vehicle captured by the canlog	56
5.5	Comparison of the evil bit set (top) and evil bit unset (bottom) in the easy simulation.	56

Glossary

AFL	American Fuzzy Lop
AFL++	American Fuzzy Lop Plus Plus
AFL+FFMut	American Fuzzy Lop Plus Format Fuzzer Mutator
AFL+FFGen	American Fuzzy Lop Plus Format Fuzzer Generator
AI	Artificial Intellegience
ANTLR4	ANother Tool for Language Recognition 4
AST	Abstract Syntax Tree
AUTOSAR	AUTomotive Open System ARchitecture
BFR	Bit-flip Rate
CAN	Controller Area Network
CFG	Control-Flow Graph
CPU	Central Processing Unit
DBC	Database CAN
DLC	Data Length Code
ECU	Electronic Control Unit
FF	FormatFuzzer
FPGA	Field-programmable gate array
GAN	Generative Adverserial Networks
IMU	Inertial Measurement Unit
IDE	Identifier Extension
IoT	Internet Of Things

LED	Light-emitting Diode
MAC	Message Authentication Code
MQTT	Message Queuing Telemetry Transport
OBD	On-board Diagnostics
PID	Parameter ID
PRNG	Pseudorandom Number Generator
READ	Reverse Engineering of Automotive Data
SecOC	Secure Onboard Communication
SDLC	Software Development Lifecycle
SoTA	State of the Art
SUT	System Under Test
TARA	Threat and Risk Assessment
UDS	Unified Diagnostic Services
WGAN-GP	Wasserstein GAN (Gradient Penalty)
BF	Brute-Force

Chapter 1

Introduction and Motivation

1.1 Introduction

Over the past two decades, vehicles have been growing more and more vulnerable to cybersecurity threats. This presents a major problem in the automotive industry, as historically, security was not a primary concern for car manufacturers past physical security (such as car hotwiring). In fact, as of 2023, 97% of automotive related cyber attacks were remote, while only 3% were physical [1]. The number of cybersecurity related vulnerabilities for automobiles has also been growing a lot in recent years (Figure 1.2): in 2019 there were 24 reported vulnerabilities for vehicles and there were 378 reported vulnerabilities in 2023 [2]. This demonstrates the ever growing danger to the automotive sector via cyber threats [1]. Common attacks against vehicles can be seen in Figure 1.3, which can range from anything from Denial of Service (DoS) to Replay Attacks. The reason these cyberthreats can be so prevalent within vehicles today is due to the rapid digitization of automobiles via embedded devices. These embedded devices are called Electronic Control Units (ECUs) and control various critical systems within the car such as airbags, fuel injection, and brakes [3]. While the usage of ECUs have improved both the performance and maintainability of vehicles, this rapid digitization also presents new attack vectors. These new attack vectors also present new opportunities for cybercriminals, as anything that contains code can be hacked. Famously, in 2015, two independent security researchers named Charlie Miller and Chris Valasek remotely hacked a 2014 Jeep Cherokee. These researchers were able to reverse engineer Controller Area Network (CAN) messages that caused dangerous behavior (such as changing the turn signals or locks), and even deadly behavior (such as killing the engine, disabling the brakes, and controlling the vehicle's

steering) [3]. With new cybersecurity challenges being given to the automotive industry, new techniques must be applied to ensure the defense of safety critical systems within vehicles. To this end, techniques such as penetration testing must be implemented to ensure the safety of vehicles. The next section explains the motivation behind the work of this dissertation.

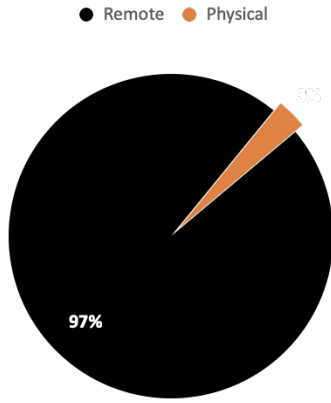


FIGURE 1.1: Instances of Remote vs Physical Attacks Against Vehicles

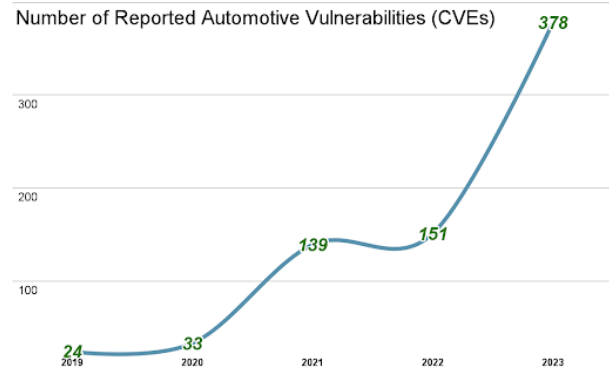


FIGURE 1.2: Reported Automotive CVEs from 2019 to 2023

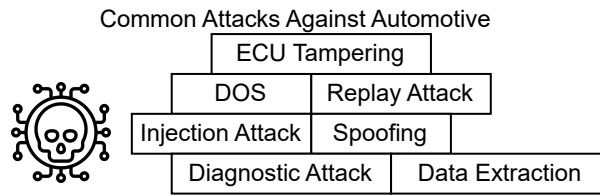


FIGURE 1.3: Common Attacks against automobiles.

1.2 Motivation

As can be seen in the previous section, the digitization of vehicles means that they are more vulnerable to attacks. To this end, this dissertation is focused on presenting a fuzzing solution to help with the penetration testing process. Three research questions will be presented that this dissertation attempts to answer:

1. What intelligent fuzzing techniques can be applied in order to fuzz Controller Area Networks (CAN) more efficiently in both a blackbox and greybox environment?
2. What are ways to define the test oracle for fuzzing a CAN system?
3. What metrics can be chosen to compare different CAN fuzzers?

A brief explanation of why these three questions were chosen are explained here. However, for more information on how fuzzing and test oracles work, please check the background (Section 2).

The first question was chosen in order to yield more complex behavior from a System Under Test (SUT) in both a blackbox and greybox environments. A blackbox environment is where no information is known about a SUT, whereas in a greybox environment some information is known. The most basic form of fuzzing is by using simple mutational fuzzers (Section 2.2.1.1). However, it is fairly easy to make and run a mutational fuzzer. It can also be the case that whatever is accepting the inputs from a mutational fuzzer, is simply rejecting the random mutations due to invalid formats. To this end, more intelligent methods of fuzzing exist to try and fuzz different systems. Therefore, this dissertation tries to apply intelligent fuzzing methods in order to fuzz ECUs over CAN.

The second question deals with the important aspect of how to validate fuzz tests during fuzzing. This process can be quite complex and difficult as it is not always clear if the fuzz tests themselves yielded any meaningful results [4] [5]. The mechanism that observes the system to see if a fuzz test was valid is known as a test oracle. These test oracles may be observing different behavior in a system. One common example is where a binary is available, where the test oracle will check for segmentation faults while the application runs dynamically. However, if a test oracle (Section 2.2.3) only tests for system crashes, but other bugs are still occurring during fuzzing that don't result in system crashes, then critical security bugs can be missed. Furthermore, it might not be the case that we can monitor the state of the binary while it is running. Therefore, good methods that can validate the results of the fuzz tests over the Controller Area Network must be selected. This can prove especially challenging in blackbox and greybox environments where no internal state changes of the system can be observed.

The third question deals with how we can take the different results and compare them among different fuzzers. This is important because it demonstrates how to effectively evaluate the efficiency of any created fuzzing solution as well as to compare their efficiency with any available State of the Art solutions.

In order to answer these questions, a fuzzing solution is required to be set up, and a fuzzer is chosen. A test oracle also needs to be chosen which will allow for fuzz test validation. The setup and results for this fuzzer and fuzzing environment will be explained in Section 4. Before this environment is explained, however, the background needed to

understand fuzzing and fuzz tests are explained in the next section (Section 2), and then the State of the Art for the fuzz tests are explained in Section 3.

1.3 Industrial Context

This dissertation was done with Continental Engineering Services under the supervision of both Dr. Reza Parsamehr and Patrick Grüner. The author of the thesis worked on-premise in Porto at the Continental Engineering Services office as part of an internship to get his master's thesis done in collaboration with the industry. The goal of this collaboration was to develop a fuzzing tool which could help with penetration testing in the automotive area. Some limitations existed with what types of ECUs were available at the time as well as the simulations available. So while the tool was initially developed for the dissertation, it can continue to be tested and refined over time. Furthermore, it should also be mentioned that this work was presented at the 2025 IEEE Vehicular Networking Conference and a paper based off of it is set to be published titled "oCANada: a Generation-Based Fuzzer for ECUs over CAN".

Chapter 2

Background

The ideas presented here aim to give an understanding of both automotive and fuzzing concepts. These ideas are necessary to understand both the State of the Art, and the solution presented in later sections. To this effect, first the important topics for the automotive area are explained (Section 2.1) followed by the most important fuzzing concepts (Section 2.2). The automotive industry concepts are divided into four sections: automotive security processes (see 2.1.1), the basics on automotive vehicle internals (see 2.1.3), and Penetration Testing (see 2.1.2). Then, fuzzing concepts are divided into three sections: common fuzzing strategies (see 2.2), how seeds are normally chosen for a fuzzer (see 2.2.2), and an explanation on test oracles (see 2.2.3).

2.1 Automotive Industry Concepts

2.1.1 Automotive Security Processes

Within the growing context of continued security threats towards software, there are processes that exist to try and prevent security breaches and secure these software and hardware products. In terms of this dissertation, these processes are important to understand as they are the processes that security professionals utilize in order to try and secure a product before testing occurs, and therefore will include the defenses that exist before a penetration test begins. The processes and documents discussed in this subsection are the Secure Development Lifecycle (SDL), the Threat Analysis and Risk Assessment (TARA), and ISO/SAE 21434 along with UNECE R155 and R156.

The secure development lifecycle (SDL), such as the Microsoft SDL [6] exists to help developers to integrate security while they are developing their code bases. These processes stretch all the way from designing and building to deploying code. These processes help developers to know they must chose security standards, perform security and design reviews, and more. By following the SDL, this helps to ensure that security is considered in a product. The SDL also considers penetration testing as can be explained in Section 2.1.2.

While the SDL is software-specific, at a high level, other automotive domain specific documents exist to ensure cybersecurity is maintained at every stage of a vehicle's life-cycle. These are the UNECE R155 and R156 certifications which have been created by the UN Economic Commission for Europe [1] as well as ISO/SAE 21434. R155 is about ensuring cybersecurity management from design to post-production while R156 ensures safe software updates through a vehicles life-cycle [1]. ISO/SAE 21434 provides a standard for cybersecurity engineering for road vehicles [7]. ISO/SAE 21434 also contains the description of the Threat Analysis and Risk Assessment (TARA) [7] which is used to describe how the user of a vehicle is impacted by a threat scenario and will aim to identify vulnerabilities and threats, as well as to assess and prioritize system security risks [8]. Like many ISO documents, ISO/SAE 21434 does not outline specific processes, but instead emphasizes the need for implementing high quality cybersecurity processes, management, and analysis in the automotive domain [1] [7]. ISO/SAE 21434 also calls for the practical testing of cyber systems by means of functional testing, vulnerability scanning, fuzz testing and penetration testing [9] [7]. Finally, R155 requires OEMs to have a risk management system, but does not explicitly mention the use of TARA like ISO/SAE 21434 does.

2.1.2 Penetration Testing

A vital part of software testing is known as penetration testing (pentesting), which deals with testing for security flaws within a system. Penetration testing allows the identification of security attacks which had not been considered in the design stage [9] [10]. Pentesting is a part of different security processes, including Microsoft's SDL as discussed in Section 2.1.1. Pentesting and ethical hacking is as much art as it is science, as a pen-tester will need need to think outside the box to be able to think like an attacker does in order to find security flaws within a system. Just like in fuzzing (see 2.2), a penetration tester may have to perform in a whitebox, greybox or blackbox environment depending

on how much information a penetration tester has at the beginning of their pentest [9]. Penetration testing therefore allows the mitigation of potential threats to a system, and to perform the validation part of the V model [9]. Finally, it should be mentioned that a valuable tool in the arsenal of pentesters is that of fuzzing, where by sending unexpected inputs, one can potentially find security flaws. Section 2.2 will concentrate on exploring the background surrounding the concept of fuzzing.

2.1.3 Vehicle Internals - ECUs, CAN, CAN DBC, and UDS

An Electronic Control Unit (ECU) is an embedded device inside a vehicle that is in charge of internal functionality. It is common nowadays for a vehicle to contain more than 70 ECUs [11], and a single ECU usually contains up to a million lines of code [1]. These ECUs can be in charge of any functionality from the airbags, to the pedal of a car, to the fuel injector [3]. Because of this, security of an ECU is paramount as any malfunction or unintended behavior can cause serious injury to either the vehicle or the vehicle's passengers. An ECU can communicate with any other ECU inside the network at any moment. There are multiple ways ECUs can communicate, but the most common way is via the Controller Area Network (CAN) [12].

The Controller Area Network (CAN) is one of the most popular ways by which a car can communicate. CAN wires are twisted pair wires that connect each ECU within the vehicle and communicate with CAN packets [13]. Furthermore, each device connected to the CAN network must be connected by a 120 Ohm resistor [12]. ECUs communicate by sending CAN packets over the CAN bus, and are designed to communicate with a speed up to 1MB/s [13]. There are two kinds of CAN messages, normal messages and diagnostic messages (these diagnostic UDS messages are explained in more detail below). Diagnostic messages are much more powerful, but usually they are only accepted by ECUs when the car is not above a certain speed threshold [3]. Each standard CAN packet has four key parts, they are as follows [12]:

- **Arbitration ID** - This is the ID of the device (usually an ECU) that the CAN message is being sent to. If two CAN packets are sent at the same time, the one with the lower ID is prioritized.
- **Identifier Extension (IDE)** - This bit is set to 0 for CAN communication.
- **Data Length Code (DLC)** - The size of the data, which ranges from 0 to 8 bytes

- **Data** - The size of the data from 0 to 8 bytes. Some systems force padding.

The exact data contained within CAN messages can be documented by the software developer beforehand, and can be defined by a CAN database (DBC) file [14]. These files will outline information on each message for each ECU, and each message is further subdivided into various signals. A signal can be further divided into such fields as where the bit starts, the bit length, the minimum and maximum values, etc. Figure 2.1 shows the format of an example DBC message (ExampleMessage) with three signals (AverageRadius, Enable, and Temperature). The figure also demonstrates how each message and signal can be divided into its component parts. It should also be noted that while DBC is proprietary, a lot of work has gone into reverse engineering the format of a DBC file as can be seen in open source information such as opendbc [15]. opendbc provides various DBC files, while also providing useful links to show how the DBC files are read. Furthermore, Python tools such as cantools [16] exist to parse through and read DBC files. DBCs help understand what valid inputs exist for CAN to communicate with the underlying ECUs via the CAN bus. Therefore, if the information inside a DBC gets leaked then this would help an attacker format valid CAN messages.

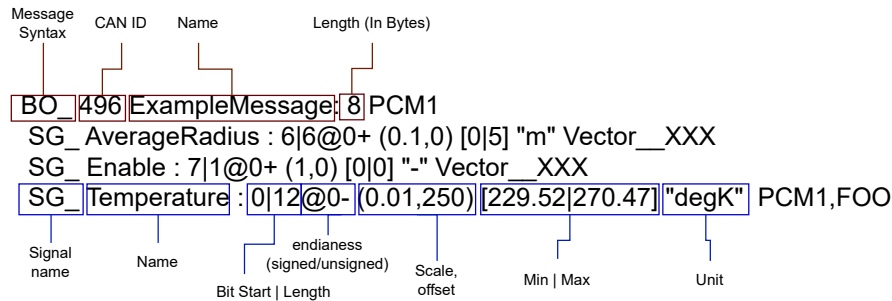


FIGURE 2.1: Example Database CAN (DBC) message.

Unified Diagnostic Service (UDS) is a diagnostic protocol based on the application layer [8]. UDS allows for mechanics to know what is going on in an automobile's internals without having to pay extravagant licensing fees for a manufacturer's CAN layout [12]. UDS allows a user to read information from the underlying ECU, as well as to perform diagnostic tests or underlying system firmware modification [12]. The above features are problematic for security if proper authentication is not enabled. Being allowed to read from the ECU allows the attacker to easily gather information. Being able to perform diagnostic tests can allow for an attacker to perform actions and gather information to reverse engineer CAN packets. For example, these diagnostic tests can request a particular

action, and then the system will generate the necessary CAN message to complete this test. An example of this could be generating a test to open a car door, and then this results in a component generating the signal for the car door to be unlocked [12]. Finally, having the ability to modify the underlying ECU firmware allows an attacker to tamper with the ECUs [8].

2.2 Fuzzing Concepts and Background

The following section is intended as a primer on how fuzzing works. Fuzzing, in the traditional sense, is the sending of random inputs to try and break a target program. Fuzzing can find source code bugs and vulnerabilities such as coding errors, buffer overflows, DoS, segmentation faults [5], and other bugs and security vulnerabilities. Fuzzing allows us to verify both the robustness and the security of a System Under Test (SUT) [8]. Fuzzers like American Fuzzy Lop (AFL) will take initial seed inputs and perform actions like random bit flips, and then tests if the input has done anything based off of system crashes (ex: segmentation faults). Part of the effectiveness of fuzzing is due to the vast array of inputs it can generate in a quick manner as compared to manual testing, as well as the fact that it usually requires a low level of knowledge to run the fuzzer [17].

Fuzzing can be divided into three main categories: Whitebox, greybox and blackbox fuzzing. Whitebox fuzzing is when the fuzzer completely knows the inner workings of a SUT (such as having complete access to the source code), and allows for the fuzzer to fully explore the internal states of a program [18]. These internal states generally involve knowledge gained from either code coverage or control flow graphs [18]. Greybox fuzzing is when a fuzzer has partial internal information of a System Under Test [19]. This partial information can be such things such as information which can be gathered from documentation, a reversed binary, or other sources which shed light on how the program functions and what is acceptable as input. Finally, blackbox fuzzing means the fuzzer assumes no knowledge of either the internal workings of the target program or any possible message structure.

This section will explain different strategies for fuzzing (see 2.2.1), what seeds are and how they may be selected (see 2.2.2), and what are test oracles and how they are important for the fuzzing process (see 2.2.3).

2.2.1 Fuzzing Strategies

2.2.1.1 Fuzzing Strategy - Mutation

This is the original form of fuzzing, where a stream of random bytes is sent to a target program [19] [8]. This black-box fuzzing strategy revolves around randomly mutating bytes in order to send to a SUT [8], as can be seen in Figure 2.2. This is the simplest possible type of fuzzer to construct since all it does is change bytes or bits at random [5]. If the parser for the input is efficient and has any kind of decent checking for invalid inputs, then most of the fuzz tests will be thrown out. It should also be noted that some fuzzers combine random mutational fuzzing with strategies such as code coverage (see 2.2.1.2 and 3.0.1.1)

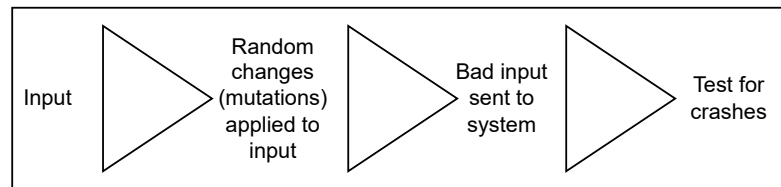


FIGURE 2.2: Diagram of Traditional Mutational Fuzzer.

2.2.1.2 Fuzzing Strategy - Coverage Guidance

Coverage-based fuzzers attempt to generate fuzzing testcases that execute over as much of the code of a target program as possible [4] [18]. The idea of code coverage is that if deeper code paths are being reached (meaning there is more code coverage), then the probability of finding new bugs is increased [18] [4]. In the case of well known fuzzers like AFL, instrumentation will be used to keep track if any new codes paths are reached during the fuzzing process. Then, the inputs which caused new code paths to be reached will then be used for further fuzzing [4].

2.2.1.3 Fuzzing Strategy - Generation

A generation-based fuzzer is a type of smart fuzzer that will create a template or model that describes the input or executions that the SUT can accept [19]. It takes into account the structure of an input and generates fuzz tests based off of that structure. Usually, a generation based fuzzer is highly specialized for a specific kind of program; for example, generation-based fuzzers might focus on fuzzing a specific data type like a Document

Object Model (DOM), a specific language like javascript, or specific network protocols like TLS [19]. There is also some work done on inferred models for generation-based fuzzers, where automated techniques combined with protocol reverse engineering techniques try to infer the structure of the data, rather than needing to have a predefined model [19].

2.2.1.4 Fuzzing Strategy - Grammars

Grammar-based fuzzing is a type of generation based fuzzing which involves feeding a grammar into the fuzzer in order to generate fuzz tests [19]. These grammars provide rules for valid and invalid inputs, which are then used to generate new test cases [5]. In general, grammar-based fuzzing is concerned with trying to solve the problem of fuzz tests failing because the parser is not accepting inputs generated by the fuzzer [17] [20]. A common type of grammar that is used for grammar-based fuzzing is a context-free grammar. A context-free grammar is a formal grammar which specifies how a set of strings can be generated [17]. The fuzzer will parse through the grammar and generate fuzz tests based off of it's rules (context-free or otherwise). Since many programs accept context-free grammars as input, this can prove to be an effective way to fuzz programs that utilize these kinds of parsers [17].

2.2.1.5 Fuzzing Strategy - Symbolic Execution and Concolic Execution

Another whitebox technique is symbolic execution. Symbolic execution involves keeping track of the various code paths to be executed, and symbolizes them to keep a set of constraints for each execution path [4]. Then, it utilizes a constraint solver to execute deeper into each code path to capture the control flow of the program [5] [4]. While a smart method for creating new inputs, symbolic execution can fall prey to the so-called path explosion problem, whereby trying to solve for a large number of possible branching statements causes any possible fuzzer to be very slow in execution [5].

Concolic execution, which is also called dynamic symbolic execution, regards the program variables as symbolic variables, tracks path constraints, and uses constraint solvers to generate an input for the specific path [18] [5] [19]. This is similar to symbolic execution, except for the fact that it utilizes concrete values for code execution in order to simplify the symbolic constraints [19]. Concolic execution will reduce the search space by directly solving the path constraints without needing to keep track of all the symbolic expressions as is done with symbolic execution which helps in mitigating the path explosion problem.

2.2.1.6 Fuzzing Strategy - Static Analysis

Static analysis is an analysis of a program without actually needing to execute that program itself. This is usually done with an analysis of the source code [4]. Static analysis techniques can be combined with fuzzing in such a way that a Control Flow Graph (CFG) is created to guide the creation of new fuzzing inputs [11] [4].

2.2.1.7 Fuzzing Strategy - Search-based Software Testing and Genetic Algorithms

Search-based Software Testing is a QA technique for test generation that can be applied to the fuzzing domain in order to generate better fuzz tests. Search-based test case generation relies on genetic (evolutionary) algorithms, as well as code coverage, in order to evolve an initial pool of randomly generated test cases. These algorithms will evolve the inputs and use given heuristics (ex: approach level and branch distance) in order to generate tests with both high coverage and fault detection [21].

These genetic algorithms from Search-based Software Testing can be utilized for fuzzing. AFL, for example, uses genetic algorithms such that each time it fuzzes, and a new input causes new code paths to be executed, that input is stored for further fuzzing [22] [4]. The power of the genetic algorithm can be shown in the case where AFL was able to generate valid JPEG input from mutational strategies alone. In his experiment [23], Michael Zaleswki used AFL to fuzz a jpeg reader with only the text file containing "hello" as input. AFL was then able to generate valid JPEG formats (SOFs, Huffman tables, quantization tables, SOS markers) within several hundred generations of fuzzing.

2.2.2 Seeds and Seed Selection

Test case quality is very important for fuzzing, as higher-quality test cases will explore more execution states and cover more code over a shorter period of time [4]. In terms of security, good test cases could target potentially vulnerable locations and bring a faster discovery of program bugs [4]. In mutational fuzzing, initial input data, also known as seeds, can be provided in order to begin fuzzing [19]. There are methods that exist to trim seeds, in order to take out potentially redundant configurations [19]. One such technique, known as corpus distillation, tries to gain a subset of interesting seeds from a large set of high-quality candidates, and then tries to fuzz them via mutational methods [22]. Tools such as AFL can take a corpus and will be able to differentiate one seed from the other if it reaches new code paths [22] [24]. Furthermore, good seeds help the fuzzing process by

ensuring that input is being provided that does not immediately get rejected by programs. If a program, for example, expects pdf files then it is very helpful to have valid pdf files that reach deep into different code paths and therefore speed up the fuzzing process. Otherwise, data will be rejected as invalid. However, generation-based fuzzers may not even require seeds, but instead generate new inputs from scratch based off of their fuzzing rules.

2.2.3 Test Oracle

The use of the test oracle is simply to answer the question: How does someone know that the fuzz tests that are sent result in unexpected behavior? A test oracle is simply the tool that is used to determine if a fuzz test was successful. In the case of famous fuzzers like AFL, the test oracle is the process which determines if the program has crashed via a segmentation fault [19] [24]. However, this type of testing is insufficient as not all memory type vulnerabilities will cause a segmentation fault [19], and many security vulnerabilities have nothing to do with memory type vulnerabilities (i.e., SQL injections, XSS, poor authentication, etc.). Test oracles can be defined to, for example, pick up on 500 internal server errors for web pentesting as is the case in Microsoft's RESTLER fuzzer [25]. Another example, is to use metamorphic testing [26] as a test oracle. In the case of metamorphic testing, two sets of tests are sent: one test is sent followed by the a mutated form of that same test. Then, if the output between them is unexpected (as defined by a set of metamorphic relationships between two outputs), then this is what the test oracle looks out for. Whether in the end, the test oracles checks for application crashes, a particular or a set of error messages, or general unexpected behavior, this is a highly important function during fuzz testing. Without the test oracle, there would be no way to know if a given fuzz test has worked as intended or not.

Chapter 3

State of the Art

The purpose of this section is to attribute the relevant research that is either trying to solve the same problem of fuzzing inside of a CAN network, or on fuzzing in a smart way. To this end, four categories were chosen as the focus for the research: relevant State of the Art fuzzers which are not generation-based (Section 3.0.1), generation-based and grammar-based fuzzers (Section 3.1), CAN-based fuzzers (Section 3.2), and test oracles (Section 3.3). The first section describes the widely used fuzzer, AFL, it's offshoot AFL++, as well as the well-known automotive specific fuzzer CaringCaribou. The second section provides an overview of grammar and generation-based fuzzers, which is a type of smart template based-fuzzing. The section on CAN-based fuzzing provides an overview of CAN-specific fuzzers. The final category, that of test oracles, was chosen to get a better understanding of how the fuzz-based techniques work in practice. While there were not many papers found specifically on how test oracles may be used in a CAN-specific environment, the CAN-based fuzzers always include techniques which could be used to determine the validity of the fuzz tests.

3.0.1 Relevant SoTA Fuzzers

This section briefly summarizes State of the Art fuzzers which are relevant to understand for this dissertation. This dissertation will look at AFL, AFL++ and the automotive fuzzer CaringCaribou.

3.0.1.1 AFL

AFL is a well known fuzzer which gained widespread popularity in the past decade. AFL was devised as a coverage-based, instrumentation-based fuzzer [19]. AFL requires source

code and will recompile a binary in order to inject instrumentation. During fuzzing, this instrumentation checks if a test achieves further code coverage while using a genetic algorithm to fuzz from an initial test corpus [22]. This tool has been highly used over the years, and has found numerous bugs across different applications as can be seen in AFL's bug-o-rama trophy case [24]. AFL will perform the black box mutation fuzzing in a way that has been tested over several binaries [27]. AFL will first perform deterministic strategies (determinism means that these strategies will yield the same outputs given the same inputs) before passing into nondeterministic strategies (nondeterministic strategies will yield different outputs given the same inputs). The author then explains that AFL uses the following strategies for deterministic fuzzing [27]:

- **Walking bit flips** - AFL performs bit flips (flip a single bit to the opposite value) with a stepover starting at 1 and going to 4. Since this strategy is expensive, AFL stops after 3 passes.
- **Walking byte flips** - this strategy performs byte flips (flip the entire byte to the opposite values) with 8,16, or 32 wide bit flips with a stepover of one byte.
- **Simple arithmetic** - AFL will try to simply increment and decrement known integers.
- **Known Integers** - AFL tries a set of interesting values that are known to possibly cause interesting behavior. These are, for example: -1, 256, 1024, MAX_INT-1, MAX_INT.

Then AFL will pass into a non-deterministic mode and will fuzz in the following "Stacked Tweaks" continuously until stopped by the user [27]:

- Single-bit flips
- Setting "interesting" bytes, words, or dwords
- Addition or subtraction of small integers to bytes, words, or dwords
- Completely random single-byte sets
- Block duplication via overwrite or insertion
- Block memset

3.0.1.2 AFL++

AFL++ is simply an extension of AFL, which is still updated to this day. New features of AFL++ include power schedules, an assortment of different mutators, and the ability to fuzz android native libraries [28].

3.0.1.3 CaringCaribou

CaringCaribou is a well known tool within the automotive industry and it tries to be like the nmap* for automotive. As per it's documentation [29], it contains many modules including UDS fuzzing and discovery, XCP fuzzing, secure UDS bypass, CAN fuzzing, and more. In terms of the CAN fuzzing module, it has a number of different functions. They are as follows:

- **Random** - a function whereby CaringCaribou sends random CAN messages.
- **Brute Force** - a function which brute forces all possible messages matching a given bit mask.
- **Mutate** - a function which mutates selected nibbles of a given message.
- **Replay** - a function which can replay log files from a previous fuzzing session.
- **Identify** - a function which replays a log file, and then identifies the message or set of messages that caused a specific event.

All of these functions provide a powerful way to gather information and bypass security of an ECU via CAN.

3.1 SoTA Generation Based Fuzzers

This section focuses on the State of the Art of generation-based fuzzers. These fuzzers both analyze the format of the input and then use this input to generate fuzz tests via various techniques. This section is important in order to understand how other researchers deal with the idea of generating smart fuzzing inputs rather than relying on just simple mutational fuzzing. From this understanding, it may be possible to apply the same or similar techniques to CAN-based fuzzing. Studying these novel generation and grammar

*That is, a tool which can perform scanning to understand the underlying system as much as possible.

fuzzers will also help in understanding their possible limitations when applied to the automotive area. The inputs provided by grammar fuzzers or generation-based fuzzers are superior to random mutational fuzzers because they can provide valid fuzz tests that pass through input validators [17][30]. The main purpose of the research in this section is to better understand generation fuzzers, as well as to find one that can be repurposed for CAN fuzzing.

Generation based fuzzing is a more intelligent method of fuzzing as compared to simple mutational fuzzers. It allows for fuzzers to take in a template, and then to generate a fuzz test with a valid format. A big issue with random mutational fuzzing is that a lot of test cases will simply be invalid if there is some kind of parser, and then this means that the fuzz testing will be more about testing the parser of the input rather than about testing the underlying system. In the case where we have speed limits, such as CAN where we can only send 1MB/s, then it is beneficial to try to get valid fuzz tests that reach deeper code paths than with mutational testing. For generation-based fuzzing, a fuzzer which utilizes a specific binary format language called FormatFuzzer (Section 3.1.4) was selected as oCANada's fuzzing engine.

As elaborated in the background (Section 2.2.1.4), grammar-based fuzzing is an offshoot of generation-based fuzzing. It accepts a context-free grammar and uses this to try and generate fuzz tests. A context-free grammar is in essence a set of rules to formulate a string-based output [17]. It can then translate these rules into something like using an Abstract Syntax Tree (AST) to then form variations of the edge nodes of the tree. Three grammar-based fuzzers are explained in this section: F1 Fuzzer (section 3.1.1), Grammarinator (Section 3.1.2), and Dharma (section 3.1.3).

3.1.1 F1 Fuzzer

Rahul Gopinath and Andreas Zeller proposed the F1 Fuzzer. The main concern with this fuzzer is about improving the speed by which fuzz tests get generated. Normally, mutation-based fuzzers can generate large number of test cases quickly but generate a lot of invalid tests cases, while template-based fuzzers will generate valid test cases but are much slower when generating (for example, the fuzzer Dharma produces inputs thousand times slower than the fastest random mutational fuzzer) [17]. Their research provides methods to make grammar-based fuzzing faster. The important and relevant ideas

in this paper are the ideas of a grammar compiler, compiling to Native Code, Supercompilation, System Optimizations, and Production Machines. These ideas are summarized in this subsection, as well as a description of the comparison of the F1 Fuzzer with other similar grammar-based fuzzers [17].

First, Gopinath and Zeller present their idea of grammar compilers, which allow the "compilation" of input grammars to code. That is, a given grammar the fuzzer will generate new code that can then be run to generate fuzzing inputs, similar to how a compiler will generate binary code to run human written code. This removes the overhead of always needing to parse the grammar each time new fuzz tests need to be generated [17]. In technical terms, this technique essentially boils down to transforming each nonterminal ending into a function which produces either more nonterminal ending (more functions), or produces an output function (such as print) which produces the terminal output [17].

In terms of compiling to native code, the authors explain that this technique involves compiling the grammar to a faster language such as C or C++. To this end, they change their usage of Python to C, and then they transform the grammar into a series of recursive C functions [17]. This in turn will avoid computationally heavy python operations like dictionary lookups, and will replace them with faster C array lookups [17].

Another technique the authors explore is Supercompilation in order to improve the performance of functional languages. This technique basically trims redundant calculations by inlining them (the functions get placed directly in the code rather than becoming named functions). This process involves inlining nonterminal expressions that only have a single rule and expansions that have deterministic terminations, leaving only the recursive portions of the program as named functions. Therefore, redundant calculations are trimmed and only recursive steps are kept as functions[17].

The authors are also concerned with system-level optimizations techniques, which are techniques that are concerned with how to improve the environment in order to improve the fuzzer's speed. With these they try to optimize two functionalities: random number generation that determines expansion, and system output to send to a produced string[17]. These steps simply involve using more efficient Pseudorandom number generators that avoid using the modulus operation, and about writing to memory instead of writing chunks directly to a large file [17].

Using these combined techniques they provide a grammar-based fuzzer which allows

them to be faster than other grammar based fuzzers such as Dharma or Grammarinator [17]. To this effect, the F1 prototype has a production speed of 100-300 times more than Dharma, and F1 fuzzer can save CPU cycles for fuzzing, and also produce larger inputs for areas for which high throughput is required (CPUs, FPGAs, Emulators, and IOT devices) [17].

Gopinath and Zeller present many techniques in order to make fuzzing faster. Techniques such as compiling to a native language like C instead of using python, are something that is considered by this dissertation. However, the F1 fuzzer does not implement all of these techniques and it instead presents them as more of a discussion. For example, while it mentions writing the fuzzer in C, the F1 fuzzer is written in python. Furthermore, because the F1 fuzzer itself is grammar based, and focuses on string fuzzing, it was not chosen to be repurposed for CAN fuzzing.

3.1.2 Grammarinator

Grammarinator aims to be a generic, out of the box, grammar-based fuzzer for simple inputs. Grammarinator provides the capabilities of both generation and mutation based fuzzers by the use of grammars [20]. In order to do so, it parses through an ANTLRv4 grammar, which are widely available grammars for various different formats. However, if this grammar is unavailable, an ANTLRv4 grammar can be created by following its format [20]. This subsection will explore how inputs are fed into Grammarinator using ANTLRv4 Grammars, as well as how Grammarinator tries to leverage the concept of parser and lexers from traditional generators[20].

Grammarinator tries to leverage similar concepts to traditional parser generators, which given a grammar will generate both a parser and a lexer. The lexer will tokenize the character input streams and the parser will generate an abstract syntax tree (AST). Grammarinator follows this concept by also generating two software components in order to generate fuzzing inputs which the authors of Grammarinator call an unlexer and an unparser. The combined results of the unlexer and the unparser can be used to produce an AST which can then be serialized into a test case. Grammarinator also attempts to avoid the problem of creating excessively big content by pre-calculating the minimum heights of the sub-trees that the unlexer and unparser can generate. Finally, Grammarinator also uses weighted choices, such that through the fuzzing process it can select various paths which were not previously selected for thereby making the fuzz coverage much wider[20].

Despite the interesting techniques presented here, this being a grammar based fuzzer makes it more difficult when trying to fuzz strict binary payloads like in CAN and there were also no ANTLRv4 grammars available for CAN for the previous reason. Therefore this technique was considered it was not chosen to be repurposed for CAN fuzzing.

3.1.3 Dharma

Dharma [31] is a grammar based fuzzer built by Mozilla. It takes in inputs in it's specific dg (dharma grammar) format, and then starts to give random output based on the grammar. In Dharma, the grammar file must take in a values, variables and variance. Dharma allows for ranges to be specified, as well as minimum and maximum values to be used [31]. This fuzzer is interesting due to it's popularity and it's development from Mozilla. However, there is not much documentation on it's internal structure and so was not chosen as a tool that could be modified to test with CAN.

3.1.4 FormatFuzzer

FormatFuzzer (FF) is a generation based fuzzer whose main concern is in taking a binary format and then applying smart fuzzing methods, as proposed by Dutra et al. Similar to F1 Fuzzer, FormatFuzzer is also a generator compiler which parses through a binary template file and then "compiles" it into C++ code which generates fuzz tests. FormatFuzzer attempts to improve the State of the Art by providing the following: a novel way to describe the format and fuzz binary inputs, an implementation of the concept of decision seeds (which are used to synchronize generators and parsers), and by making format agnostic fuzzers (like AFL++) input aware. This subsection will describe how FormatFuzzer functions in terms of basic functionality, it will summarize the concept of decision seeds, and it will explain how FF can be integrated with AFL++ [30].

As mentioned above, FormatFuzzer uses the binary template language which can generate any binary format [32]. The syntax is very similar to C, but mainly describes data formats. It can define not only variables, but also basic programming concepts such as conditional statements, loops, custom functions, error handling, and more [30]. Therefore, this allows a programmer to specify how a binary format is being read and can use coding syntax to apply certain rules and conditions to the data format. This allows for a lot of versatility when describing data formats. For example, one can specify if a template

needs to be read in little or big endian, and branching statements can be used to specify that certain data needs to be included if the data includes certain headers.

FormatFuzzer also has the additional functionality of being able to be loaded into other fuzzers after applying it's own fuzzing strategies. Particularly, the authors have spent a lot of effort in making it compatible with AFL++ and allows for fuzzing in both big and little endian modes [30].

FormatFuzzer's primary source of randomness is in it's decision seeds. A decision seed is essentially an array of bytes. Every time the generator needs to make a random decision, such as what to use for a variable, it reads from the decision seed itself. For each variable, one byte is kept to in order to decide if it will make an "evil" decision, meaning that it will fuzz with a completely random value rather than with the constraints defined for this given variable. Practically, this array of random bytes is generated from `/dev/random`. FormatFuzzer can also automatically generate this decision seed when parsing the `.bt` file. In practice, when a generator needs to make a random decision, it will read the required number of bytes from the decision seeds. This also implies that the generation process is completely deterministic, as the generation process will fuzz in the same way given the same input file [30].

It should also be mentioned that the authors of FormatFuzzer have also created the option for FormatFuzzer to be integrated with AFL++. They provide two methods to integrate FormatFuzzer with AFL++. The first is `AFL+FFGen`, when AFL is used to mutate and evolve the decision seeds which are then used as input for FormatFuzzer. In this way, AFL++ actually just concerns itself with fuzzing the decision seed, rather than needing to be aware of the input structure at all. In a whitebox system, AFL++ can also use coverage feedback to help guide the decision seed mutation. The second option they provide is `AFL+FFMut`, in this case AFL++ will mutate the files for the target program as usual, but also add the smart mutation operations which are introduced by FormatFuzzer. As stated in the github documentation for their version of AFL++, there is a blackbox version for `FFMut` and `FFGen` modes that will still allow AFL++ to do mutations while not needing the coverage feedback [30].

The authors of FormatFuzzer evaluated FormatFuzzer's strategies over a number of different popular binaries. Importantly, they do not compare their generation-based fuzzer against grammar-based fuzzers as the grammars based ones are text based and do not account for binary formats. They then evaluated over language coverage (which features

of the binary template are actually present in the generated files). This was at least 94 % coverage for most of the formats that they tested [30]. According to their results, utilizing both their fuzzing techniques with FormatFuzzer along with AFL++, leads to the best fuzzing approach [30].

This fuzzer was chosen to be repurposed for CAN. It appears to be perfect for the purposes of fuzz testing CAN as it is specifically takes in a binary format and formats it, and it's granularity allows for bit level changes. This would allow for greater control when analyzing the signals within a DBC, as opposed to just random mutations which many CAN fuzzers do.

3.2 CAN-based Fuzzers

This section now focuses on fuzzers which were created specifically to fuzz Controller Area Networks (CAN). As such, any smart fuzzing techniques will be geared towards CAN based fuzzing techniques, and their strategies might only be applicable in this context rather than in the case of general system-agnostic fuzzers. In this section, the first CAN-based fuzzer described is one that relies on the static analysis of the underlying firmware in order to generate control flow graphs for fuzzing (Section 3.2.1), the second uses a blackbox methodology with the CaringCaribou tool (Section 3.2.2), the third CAN-based fuzzer utilizes reverse engineering techniques to learn about the CAN message before fuzzing (Section 3.2.3), and the final CAN-based fuzzer discussed uses reverse engineering and machine learning techniques (Section 3.2.4) in order to generate the fuzz tests.

3.2.1 EffCAN

Radu and Garcia created a greybox tool using static analysis [11]. This subsection will explain how they were able to generate a control flow graph (CFG), how they were able to conduct fuzzing with that CFG, and briefly mention the results. Section 3.3.3 elaborates on the test oracle techniques which they also present in their research [11].

For their work, the authors utilized techniques which involved flashing an ECUs firmware, loading it into IDApro, and then extracting the entire control flow graph (CFG) using static analysis methods[11]. They then presented their fuzzer, EffCAN, which would fuzz the ECU via the CAN network based on the generated CFG. In terms of their efforts

to prepare a CFG for fuzzing, analyzing the firmware proved to be very difficult as it required intimate knowledge of the given ECUs. However, each ECU architecture can differ a lot, and there is usually a lack of available documentation[11] which increases the time that it takes to run this. Information such as the reset vectors need to be supplied which function as entry points in the code [11]. Interestingly, the authors note that even when Miller and Valasek reverse engineered the firmware from a V850 Chip, it took them several weeks.[11]. Therefore, while this method is powerful, it requires both a lot of time and know-how due to the need to reverse engineer the ECUs[11].

After the firmware is reverse engineered, and the CFG is extracted, fuzzing can begin. For this, the authors present their fuzzer called EffCAN. When their fuzzer runs, the CFG is read and for each branch or jump instruction a random value is chosen that still respects the value given by the CFG [11]. This in turn allows for fuzzing internally valid inputs as defined by their CFG. Furthermore, when running the fuzzer, they also give a choice between a queue or payload option, where the program can either fuzz one byte for each payload in the queue or can fuzz all bytes in each payload [11]. The authors also provide a method for a test oracle which is explained in Section 3.3.3 [11].

Finally, in terms of results, the authors tested their fuzzer on three ECUs. They were able to get two of them to stop communicating over the network and both these crashed ECUs were able to return to normal functioning after a reboot. The authors stress the importance of these results, as stopping safety critical ECUs while they are operating can possibly cause a lot of damage to the vehicle and whoever is operating it [11].

3.2.2 autoFuzz - CAN Automatic Sensor Fuzzer Module

Werquin et al. expand on CaringCaribou with their autofuzz module. This subsection will explain how the autofuzz module functions, as well as their experimental results. Also, Werquin et al. utilize a sensor harness as test oracle which is briefly explained in this section but is further elaborated on in Section 3.3.1

In the autofuzz module, the strategies of CaringCaribou (Section 3.0.1.3) are still used (brute-force, mutation, and identify), but they take the input from physical hardware. Normally, CaringCaribou requires a human operator to determine if there is a state change, but the use of a sensor harness (elaborated on in 3.3.1) allows the system to automatically see if there are changes by attaching various sensors to an instrument cluster. The sensor observations from the harness (such as a LED turning on, or the speedometer moving),

are then used to guide the generation of fuzzing inputs, as well as to automatically identify message bits that lead to changes in the system [33]. They also test their approach using commercial instrument clusters and even experiment on a setup that tests an implementation of AUTOSAR-compliant message authentication[33].

In terms of experimental results, they tested this on a instrument cluster without security features. Werquin et al. tried to brute force the entire 11-bit arbitration ID space with the payload of 0xffffffff. This led to the LED indicators, the control of the speedometer, and the control of the RPM to be activated. They also elaborated on how they used mutation-based fuzzing, and were able to reverse engineer the semantics of most payload bits. They give an example where they are able to start from the messages that activated the left turning indicator, the fuzzer was able to identify the bit responsible for triggering the indicator but also able to identify the bit responsible for the correct turn signal [33].

Werquin et al. also tried to fuzz on the vulCAN framework, which tries to provide a generic design for CAN authentication. As the authors of the paper note, this is a difficult challenge because security is kept in mind which means that it is unlikely for a random or mutation-based fuzzer to get a valid payload, nonce or authentication tag triple [33]. Despite this challenge, they were able to detect unique vulnerabilities in the system in a short period of time. Among their vulnerabilities was one where the attacker was able to arbitrarily adjust the displays of the test bench without having to pass authenticity checks [33]. They also found a denial of service in the authentication protocol when it was flooded with a specific pattern, allowing attackers to desynchronize nonces and render trusted components unresponsive [33].

3.2.3 Structure-Aware Fuzzing

Kim et al. present a blackbox fuzzing solution that tries to reverse engineer the structure of the CAN packet, and then use this reverse engineered structure in order to generate fuzz tests [34]. This subsection will present Kim et al's methodology as well as their results. Section 3.3.4 describes the implementation of their test oracle.

The methodology of the authors involves using reverse engineering techniques to better understand CAN packets and then use these results to fuzz in a way that the structure of the CAN packet is taken into account. Their proposed methodology for structure aware CAN fuzzing involves 3 steps: CAN message analysis, running a fuzzing input

value generator, and then monitoring (as per 3.3.4). Fuzz tests are based off of the following techniques: Bit-flip rate analysis, OBD-II PID analysis, correlation analysis [34]. The most important of these for their fuzzing efforts is Bit-flip rate analysis.

Bit-flip rate analysis was initially introduced in the paper on Reverse Engineering Automotive Data Frames (READ)[35] and an alternative implementation of this is found in LibreCAN [36]. Essentially, this methodology involves reverse engineering the message by checking the occurrence of the bit-flips and then guessing signal boundaries and types of signals that can be found in a given message [35]. In bit-flip rate analysis, the signal boundaries are identified by calculating the number of times a bit flips from 0 to 1, or vice-versa, given a consecutive number of CAN messages with the same CAN ID. When parsing a payload bit by bit, when the BFR is at the highest this indicates that a bit was flipped the most, and if the next BFR of a bit is higher than the BFR then the next bit, this can indicate that the signal boundary has ended. In READ, in addition to signal boundaries it can also find CRC checks as well as counter values (values that slowly increment) [35]. Kim et al. mention that maps the signal boundaries to seven signal types. These are unused fields, constant fields, multi-value fields, sensor fields, counter fields, checksum fields and undefined fields[34]. Knowing these fields then allows for them to use their fuzzing methodology keeping in mind the constraints of the signals within the CAN messages.

In terms of experimental results, the authors monitored CAN traffic with a tool called RAD Galaxy, and they monitored IMU information (Inertial Measurement Unit, a device that is used to integrate accelerometers and gyroscopes) using a raspberry pi 3. They have CAN logs of various states (stop, driving, driving with various vehicle operations functioning) in order to do their bit-flip rate analysis. After sending their fuzz test they were able to increase the engine's acceleration as well as able to change the RPM of the dashboard. When they compared the normal state with the fuzzing state, they found a periodic change of IMU sensor values [34]. This was the result of a steering control change which actuated by 5 to 10 degrees to the right. They were also able to find out that the autonomous emergency brake actuated [34]. They concluded that they provide a valid method to fuzz ECUs via the CAN bus without needing physical access to the ECUs[34].

This CAN fuzzer is of great interest to this dissertation. While no source code is available, the information presented by Kim et al. along with the underlying technique presented in the paper on READ[35], were inspirations for the reverse engineering techniques (see 4.3.3) that are part of the oCANada fuzzing solution.

3.2.4 CAN-FT

Zhang et al. focus on a way to use machine learning techniques for both fuzzing and monitoring the fuzz tests. In order to generate fuzz tests, they used both a bit-flip rate analysis and generative adversarial network, and to monitor the results of the fuzz tests it uses AdaptiveBoosting (see section 3.3.6).

Zhang et al. present two methods for generating fuzz tests. The first is the bit-flip rate analysis. This idea is presented in the previous paper (see 3.2.3). The bit flip rate analysis simply determines the frequency of change of a given bit in the CAN payload in messages with the same ID. This paper then divides the 64 bit payload of the CAN messages into signals, multi-values, constants, counters and check codes [37]. The payload can then be segmented given the BFR analysis. This allows the determining of variable and immutable regions, the boundaries between signals, division inside a given segment of check code, counter and multi-value segments. Only once the payload is segmented like this, can fuzzing strategies be applied to mutate the data[37].

Besides bit flip-rate analysis, another method that they use is to use Generative Adversarial Analysis. GAN uses a generator and discriminator that generally adopts a neural network format [37]. The difference between them is what is used for the training data. The generator selects random noise as input for training, and the discriminator uses real data and the fuzzed data for training. They attempt to enhance this by the usage of WGAN-GP, to try and mitigate some of the issues with GAN, namely an improved gradient penalty and the removed weight constraint to accelerate the training progress [37]. They use the messages from the OBD interface of the vehicle to generate the training dataset[37].

When they attempted their fuzzing, the random fuzzing methodology produced the most abnormal messages while the fuzz method based on WGAN-GP induced the fewest abnormal messages. This is because the methods based on WGAN-GP is closer to real CAN messages and therefore was less likely to cause an anomaly[37].

This research by Zhang et al. is mentioned here because there is ongoing research on ML techniques for CAN fuzzing. However, this research is not available and the other techniques seem to be more mature for the purpose of fuzzing. There is also the weakness that the machine learning techniques require data sets which adds a layer of difficulty. As such, no machine learning techniques are considered in this dissertation.

3.2.5 CAN Fuzzer Relevance for the Dissertation's Fuzzing Solution

This subsection gives a summary of whether the fuzzers described in this section are considered or not in this dissertation. The EffCAN [11] is a very interesting and advanced way of fuzzing as it generates a CFG, but it had two main drawbacks for this dissertation. The first is that it requires expertise for generating the CFG after the firmware is uploaded to IDAPro, and the second problem is that the code is not available. The autofuzz [33] module is available on github, but this dissertation was not focused on creating a physical test harness in order to utilize this method. CAN-FT [37] is not really considered as this dissertation was not concerned in utilizing ML techniques for fuzzing. The structure-aware [34] CAN fuzzer is considered in this dissertation as it is the only presented SoTA fuzzer that is generation based and takes the structure of the CAN message into account. However, this fuzzer itself is not considered as the code also is not available, but the underlying technique using the READ [35] algorithm for reversing is considered.

3.3 Test Oracles

In this section different test oracles that have been used for State of the Art fuzzers will be summarized. A test oracle is a method by which we can validate our fuzz tests, as explained in Section 2.2.3. Importantly, this section will focus on test oracles for Controller Area Networks in particular and will present methods by which CAN fuzzing can be validated. This section elaborates on different test oracles and expands on the ideas from the following papers: Werquin et al.'s hardware-based test oracle which relies on physical cues to determine interesting outputs (3.3.1), Fowler et al.'s suggestions on different possible test oracles (Section 3.3.2), Radu et al.'s test oracle which relies on ECU timeout and a UDS service (Section 3.3.3), CAN message monitoring and side information monitoring (Section 3.3.4), a test oracle which uses a physical camera (Section 3.3.5), and finally a machine learning technique using AdaBoost as a test oracle (Section 3.3.6) is summarized.

It should be noted that there are no specific works on test oracles with relation to testing CAN, but that the information on test oracles are always part of papers that propose some kind of CAN fuzzing methodology. As such, most of the papers that are in this section are also part of the CAN-based Fuzzers section (Section 3.2).

3.3.1 Physical Harness Test Oracle

Werquin et al. explain how they devised a physical test oracle which reads an instrument cluster. This physical test oracle is equipped with sensors which reads the changes of lights and indicators (such as a change in the speedometer) on an instrument cluster. Therefore, when a fuzzer sends its inputs, the authors rely on physical hardware as the test oracle to see physical changes in the instrument cluster. For Werquin et al. these physical changes on the instrument cluster means that there was a change in the state of the ECU. Depending on the fuzzing strategy which has been chosen, the test oracle can be used to guide the generation of fuzzing inputs as well as to identify which bits caused the state change [33]. The benefit of this physical test oracle is that manual approaches tend to rely on reverse engineering, which require hours or days or even weeks of trial and error [33]. However, the use of their sensor harness meant that this reverse engineering process didn't need to occur, allowing for the pentesting process to speed up [33].

3.3.2 Test Oracle Discussion Fowler et al.

Fowler et al. present several ideas in order to address the test oracle problem. These ideas included network communication monitoring, monitoring through a component debug interface, direct and indirect monitoring of system signal available in a simulator, the use of XCP that allows remote access to the internals of an ECU, and the monitoring of the physical responses of the system with external sensors [38].

3.3.3 Timeout and Test Present Service Test Oracle

Radu et al. presented two methods for detecting crashes. The first method was message timeout and the second was the UDS Tester Present service [11]. In the case of message timeout, they consider that the ECU has crashed if the ECU stopped sending messages over a given time period. In the case of the UDS tester service present, the CAN ID for UDS must be known and after each fuzzed payload, they sent UDS requests to the tester present service (the payload to use this service is 0x3e). Of course, this technique assumes

that the tester present service exists for a given ECU under test. If the ECU is still active and the fuzz test has not caused any behavior change, a response will be given by the ECU. This response may either be negative or positive. However, if a response is not given, or if there is a noticeable delay, this could mean that the ECU is has either crashed or is in an unexpected state[11].

3.3.4 CAN Monitoring and IMU Sensor Monitoring Test Oracle

As explained in the section on blackbox CAN fuzzer, Kim et al devised a fuzzing method whereby they analyzed live CAN packets and presented two methods for validating their fuzz tests: CAN message monitoring and side information monitoring (IMU sensors). Both these methods are discussed in this subsection[34].

Their test oracle works first through CAN message monitoring which involves four strategies: CAN ID monitoring, CAN DLC monitoring, sensor violation monitoring, and correlation monitoring. CAN ID monitoring involves checking for the existence of invalid CAN IDs; CAN DLC monitoring involves checking for changes in Data Length Code (DLC) values of any CAN ID; sensor violation monitoring involves checking whether or not the values of the Sensor field are outside the max or minimum values; finally, correlation monitoring involves checking for changes in the correlation between the sensor field values that are above a given threshold. It should be noted that despite these strategies, in their results section the authors mention that they found no results when monitoring with these strategies[34].

The authors also implemented a strategy which involved side information monitoring, whereby they obtained sensor values that occurred without CAN fuzzing (that is, during the normal operation of the vehicle) and compared them with the state of the sensor during CAN fuzzing. Utilizing this technique to monitor sensors, they were able to identify steering control misbehavior, engine acceleration in one vehicle they tested, and brake control in another vehicle that they tested[34].

3.3.5 Image-Based Test Oracle

These authors presented a test algorithm that uses a simple camera to check for physical changes in an instrument cluster (IC) while their fuzzing payloads consisted simply of randomly mutation. For their test oracle, they took an initial photo of the cluster that was used for baseline comparisons. They then utilized a motion detection algorithm in order

to process the image and check for events. These events were basically changes to the IC Examples given are moving needles, steady or blinking lights. A 200 millisecond waiting time was allowed for the cluster to react after each transmission. Then, 3 pictures were taken in burst mode and their gresycsale version was compared with the baseline image. If there were any changes, the image was saved [39].

3.3.6 CAN-FT's Test Oracle

This paper presents a test oracle via a machine learning technique. AdaBoost is a boosting method that combines multiple weak classifiers into a strong classifier. In machine learning a weak classifier has a stronger chance of guessing the result based on input data, rather than a random guess. They then utilized AdaBoost to train their classifier to detect anomalous CAN messages within the network. However, it should be noted that this technique was better able to pick up traffic from the mutational fuzzing rather than from the fuzzing when they use the Generative Adversarial Network, due to the fact the later method was better at crafting messages that were not anomalous [37].

3.3.7 Test Oracle Relevance for the Dissertation's Fuzzing Solution

The discussed test oracles above highlight various properties that can be monitored. Werquin [33] monitors for specifically physical signals in hardware (lights, speedometer changes), which is not applicable to this dissertation because the scope of ECU testing goes beyond just testing instrumentation clusters. This same issue exists for the paper [39] that took an image of an instrument cluster with a camera and checked for visual changes. CAN-FT [37] presents an interesting solution for ML based fuzzers, but since this dissertation is not using ML techniques their techniques were not considered. In Kim et al.'s paper [34], they utilized some CAN monitoring techniques, but since they did not find anything with these techniques they were not considered for this dissertation. They did find some vulnerabilities when they utilized IMU sensors, but our lab setup did not include such sensors this technique was also not considered for this dissertation. Fowler et al. [38] mention the idea of network communication monitoring. While the details of this are not discussed in detail, something similar is proposed in this dissertation for a network monitoring based test oracle. Finally, Radu and Garcia [11] utilize a timeout and the UDS tester present service to test for ECU activity, and these two ideas are considered as part of the solution for a test oracle in this dissertation.

Chapter 4

oCANada Fuzzing Environment

This section will present our fuzzing solution, including the lab setup as well as the particulars of how the fuzzer itself works and how the fuzz tests will take place. In [Section 4.1](#) there is an introduction to how the tool can be used in a real world environment. In [Section 4.2](#) there is a description of how the lab setup was done for testing. [Section 4.3](#) explains how the fuzzer itself functions and the three modes of operation are explained in [4.3.1](#), [4.3.2](#) and [4.3.3](#), as well as how the test oracle functions in [4.3.4](#). Finally, the testplan is explained in [Section 4.4](#).

4.1 Fuzzing Environment Introduction

The ultimate goal of this fuzzing solution is to allow CAN fuzzing in real world environments. The whole purpose of the fuzzer can be seen in [Figure 4.1](#). It shows that the fuzzer itself can run on the Raspberri Pi, and this Rasberry Pi can be connected to a real vehicle via CAN through the OBD-II port. Specific ECUs can be targeted with our fuzzed messages by specifying an ECU's arbitration ID. Messages from the CAN bus are then picked back up with the Raspberry Pi and sent to the test oracle.

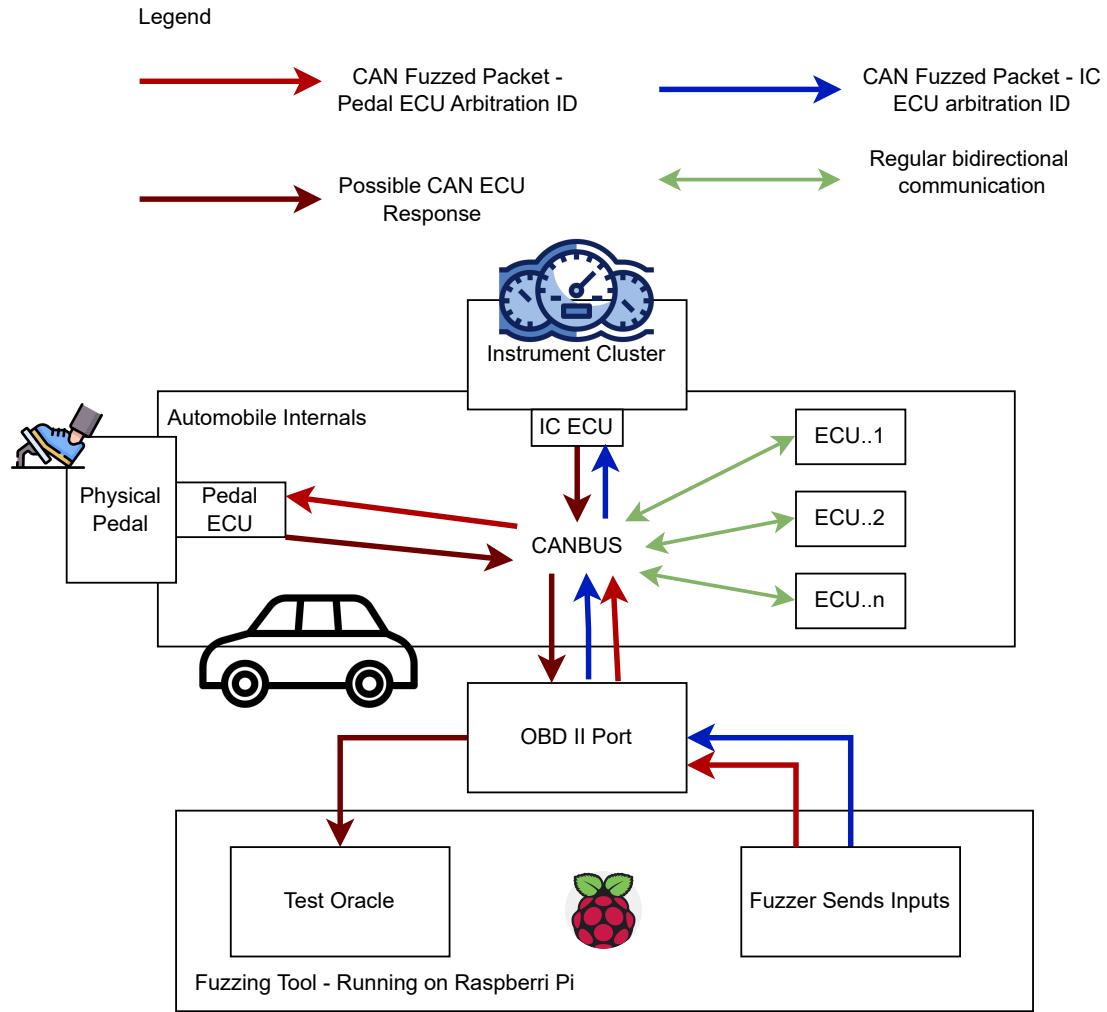


FIGURE 4.1: Image demonstrating the fuzzing pipeline for the oCANada fuzzing environment.

4.2 Lab Setup

This sections aims to explain how the lab was setup so that experiments could be run. The primary components involved were:

1. **CANoe Vectorbox VN5620** - This is a box that allows the simulation of CAN messages. VN5620 can run CANoe which is a specialized software that can be used to simulate ECUs. These CANoe simulations can be complex and ultimately simulate an entire vehicle architecture with ECUs that communicate with each other on a network. VN5620 contains slots for cables (such as CAN cables or CAN over USB)

which then allow outside devices to connect to the CAN network that is simulated by CANoe.

2. **Raspberry Pi with PiCAN** - This is a small and fairly cheap single board computer. A linux OS is installed on the raspberry pi which can run the fuzzer and the test oracle. A PiCAN is then attached to the Raspberry Pi, which allows the Raspberry Pi to be connected to CAN via CAN cables.
3. **CAN cables** - The CAN cables are the CAN bus wires themselves with the endpoints being connected with db9 connectors. These are required to make a connection between the Vectorbox and the Raspberry Pi, and any other ECU that is needed to connect with. These cables allow the bi-directional communication to and from the Vectorbox.
4. **Two 120 Ohm Resistors** - Each connected device in the CAN bus must have a 120 Ohm resistor. One resistor was connected before the CANoe box and another one was connected before the Raspberry Pi with PiCAN. The reason these are required is because there must be a 60 Ohm resistance for the CAN cables. Therefore, the easiest solution for this is to ensure that every device that is connected to the CAN network has a 120 Ohm resistor in front of said device.

With figure 4.2, it can be seen how the simple setup is done for the lab, such that the Vectorbox connects to the PiCAN sitting on top of the Raspberry Pi via the CAN high and CAN low wires. It should also be noted that before each component sits a 120 Ohm resistor. Then in figure 4.3 it can be seen how an ECU can be added to the lab setup to allow for testing of real ECUs.

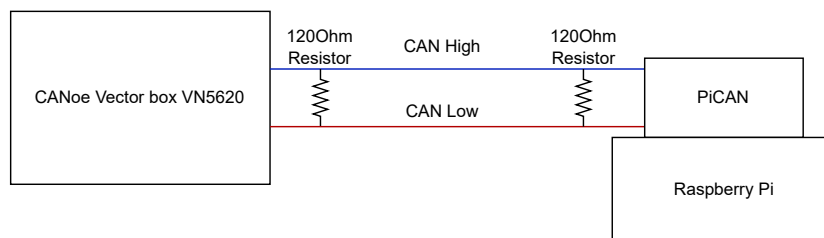


FIGURE 4.2: Simple lab setup for purely simulation.

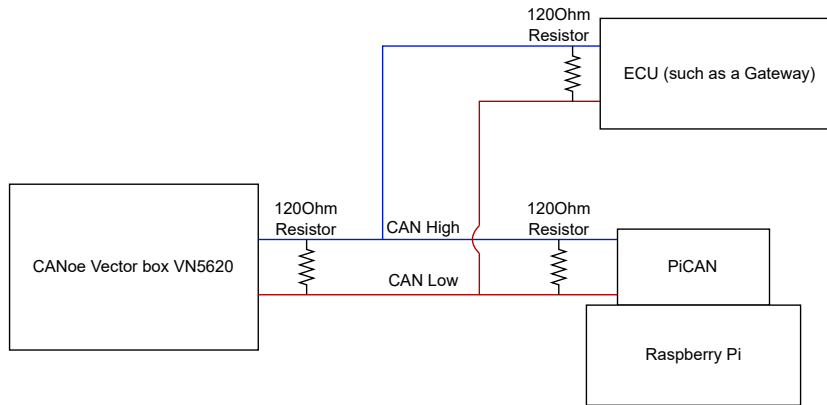


FIGURE 4.3: Lab setup to test real ECUs with the rest of the vehicle simulated.

4.3 oCANada Fuzzing Strategies

This section will detail how the entire fuzzing pipeline is set up. There are three fuzzing solutions that have been implemented in the oCANada environment that attempt fuzzing in a generation-based way. They are as follows:

DBC Brute-Force (DBC BF) Mode - This mode is a simple brute-force mode which attempts to generate all possible permutations of signals for a given message from a DBC file (see 4.3.1).

DBC FormatFuzzer (DBC FF) Mode - This mode will generate a template based off of a DBC file in order to feed it to FormatFuzzer (see 4.3.2).

Reverse FF Mode - The final mode of operation attempts to parse out signal boundaries from CAN via reverse engineering techniques to then fuzz in a generation-based way using FormatFuzzer (see 4.3.3).

It should also be mentioned that for FormatFuzzer, that for both of the FF modes, it grabs its source of randomness from `/dev/random`. How the test oracle is chosen is explained in Section 4.3.4. The three implemented fuzzing modes in combination with the test oracle provide in combination with the lab setup provide the fuzzing environment. Figure 4.4 shows the three possible modes of operation that can be chosen for the fuzzing tool. Both the FF mode and the BF mode require that a DBC be taken in. The fuzzing tool only natively generates fuzz tests by using the BF mode, the other modes require the generation of a binary template file. This binary template file is then fed into FormatFuzzer, which then generates a binary that can be used to generate fuzz tests. It should also be

emphasized that the reverse mode requires messages from the CAN bus (in the form of a canlog), while the FF and BF mode require a DBC and which describe the format of a can message. Then from this canlog, the reverse mode will figure out the necessary information in order to develop a binary template to then send to FormatFuzzer. As seen below, the oCANada fuzzing tool can be run from the command line and all the options can be shown with the -h flag.

```

1 $ python ./oCANada_driver.py -h
2 usage: oCANada [-h] {reverse,dbc_gen,fuzz,brute_force_fuzz} ...
3
4 options:
5   -h, --help            show this help message and exit
6
7 subcommands:
8   {reverse,bt_gen,fuzz,brute_force_fuzz}
9       oCANada operations
10  reverse                reverses a log and generates a binary template (bt)
11                          file
12  bt_gen                 generates a binary templates (bt) file from a dbc
13  fuzz                   fuzzes based off of a template
14  brute_force_fuzz       does brute-force fuzzing based off of a dbc

```

LISTING 4.1: Options from the oCANada tool

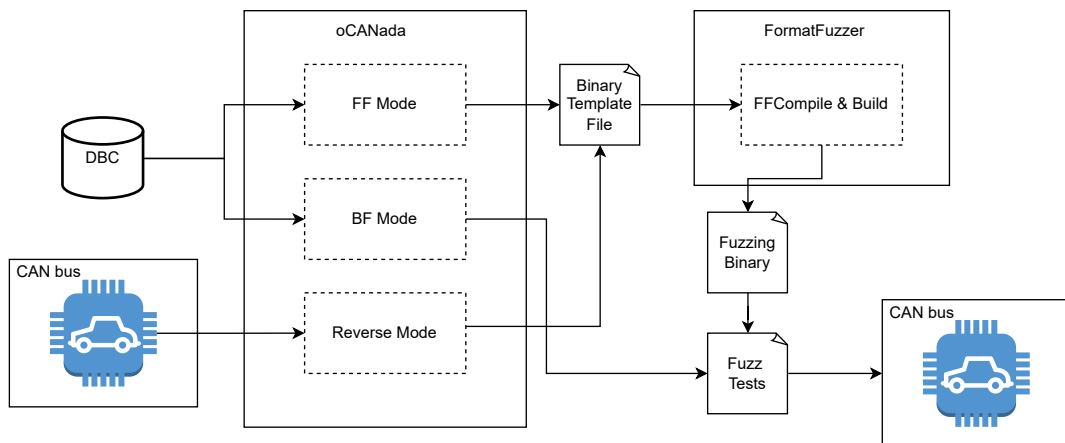


FIGURE 4.4: Figure demonstrating the three modes of operation for oCANada.

4.3.1 oCANada DBC Brute Force Mode

The brute force mode is the simplest of the three presented fuzzing modes. Figure 4.5 shows the steps that the DBC Brute Force mode takes. Here the fuzzer will parse through a DBC with the cantools library, and after doing that it will extract the CAN message information results to the internal FFGenerator class which contains the brute force function as can be seen in the "DBC information extraction" steps. This class will read the parsed messages and determine the important pieces of information as seen in the steps regarding the "Message Attribute Extraction". For the message the parser will determine the arbitration ID and the endianness of the message, and then for each signal the FFGenerator class will determine if the bits are signed or unsigned, the position in the payload, as well as possible minimum and maximum values. Once these pieces of information are determined, then a generator function is called, yielding each possible permutation of the brute force combination so that fuzzed message can be sent to the CAN bus. These fuzzing steps will occur until all possible values for signal permutations are generated for every message. The high level logic in the code for the brute force function can be seen in Listing 4.2.

```
1 def brute_force(dbc_file_in):  
2     #this function here will use the custom brute force function in a  
3     template fuzzing driven way  
4     dbc_reader = DbcReader()  
5     messages = dbc_reader.read(dbc_file_in)  
6     ff_gen = OCanadaFuzzGen()  
7     ff_gen.brute_force(messages)
```

LISTING 4.2: Function for running brute force code

4.3.2 oCANada DBC FF Mode

4.3.2.1 btCompiler (Binary Template Compiler)

The first thing that needs to be done in order to begin the DBC FF mode is the compilation to the binary template. In order to do this, a DBC is read and the message and signal information are extracted. This can be seen in Figure 4.7 in the "DBC Information Extraction" and "Message Attribute Extraction" steps. These steps are also similar to the steps of the brute force function in Figure 4.5. However, instead of feeding this information directly to the fuzzer, these values are fed into a function which generates a binary template file.

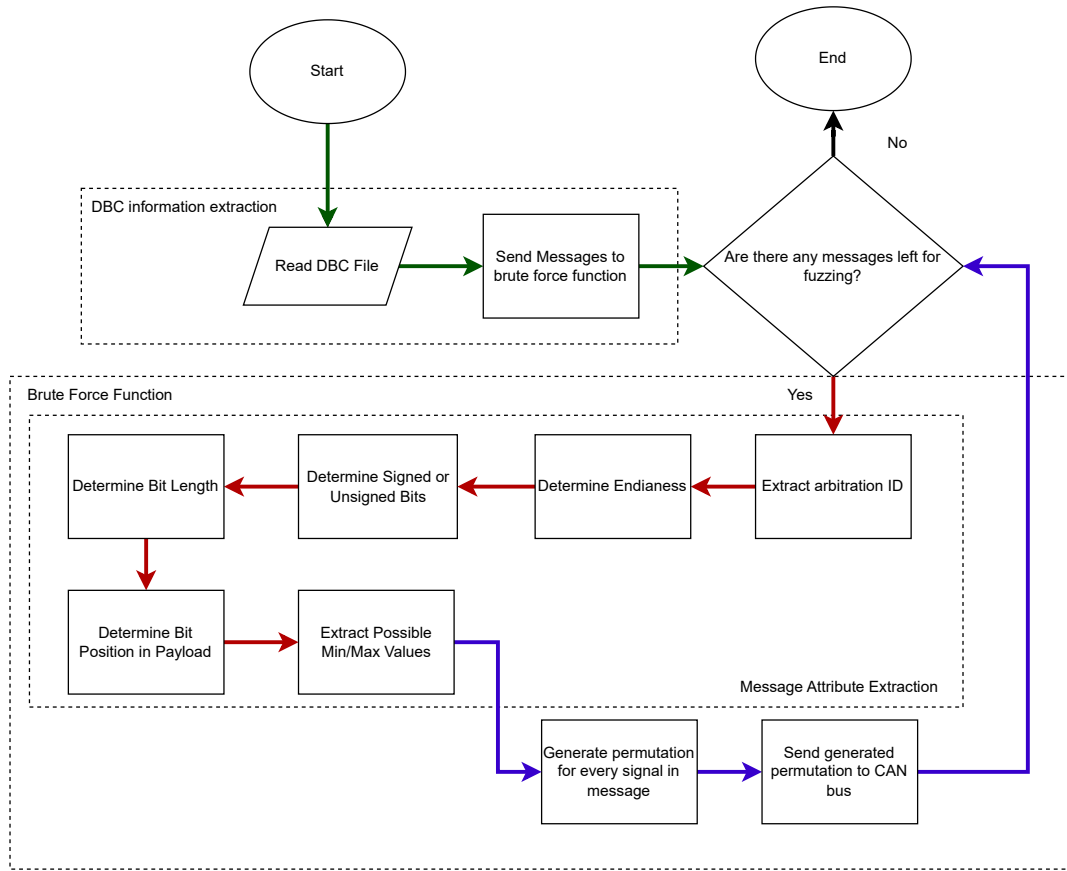


FIGURE 4.5: Flowchart demonstrating the steps required for oCANada's brute force mode.

It should also be mentioned that an important consideration when perform the "Message Attribute Extraction" steps is how the conversion from the numerical value to raw value works. That is, the numerical value as specified in the DBC to the raw bytes that are sent in the CAN packet. The following formula is followed in terms of raw byte generation, where offset and scale are values contained in the DBC file, and value is the numerical value that is to be sent.

$$raw_bytes = \frac{(value - offset)}{scale}$$

Listing 4.3 demonstrates a binary template file which has been generated after the DBC file was read. As shown below, a typedef struct is generated for each message and the variables inside the struct represent the signals and the padding. Each represented signal may contain restrictions on the possible represented values. The final struct is a value for the arbitration ID which is used in the main function. In the main function, first

the endianness is set, and then there is an if statement which tells FF what the arbitration IDs must be in order to adhere to the valid values. After this, in the main function there is a switch statement which randomly chooses one of the structs and populates returns valid bytes for the CAN payload. Within this file there is also the option to set the "evil bit". As explained in the section on FormatFuzzer (3.1.4), the evil bit specifies if FF can fuzz outside of the normal constraints of the given variable. The evil bit will always be a binary True/False variable and can be set or unset for any variable or group of variables in the binary template. In this example, the evil bit is just unset so that the arbitration id for the given message is always an accurate one.

```
1 typedef struct {
2     QWORD OnOff: 1;
3     QWORD EngineSpeed: 15;
4     QWORD Padding: 48 <min=0, max=0>;
5 } EngineState;
6
7 typedef struct {
8     QWORD HeadLight: 1;
9     QWORD FlashLight: 1;
10    QWORD Padding: 62 <min=0, max=0>;
11 } LightState;
12
13 typedef struct{
14     uint16 arbitration_id;
15 } ARBID;
16
17 struct {
18     BigEndian();
19     local int evil = SetEvilBit(false);
20     ARBID id;
21     SetEvilBit(evil);
22     if(id.arbitration_id != 291 && id.arbitration_id != 801){
23         Warning("Not valid arbid");
24         return -1;
25     }
26
27     switch(id.arbitration_id){
28         case 291:
29             EngineState message;
30             break;
```

```

31     case 801:
32         LightState message;
33         break;
34     }
35 } main;

```

LISTING 4.3: Example Binary Template

In practical terms, the oCANada tool itself needs to run the following command in order to generate the binary template file. This can be seen below in Listing 4.4.

```

1 python ./oCANada_driver.py bt_gen -d .<path_to_dbc> -o <path_to_bt_output>

```

LISTING 4.4: oCANada command to generate a Binary Template with the oCANada_driver.py

The main code which runs the listing from the driver can be seen below in Listing 4.5.

```

1 def generate_bt_from_dbc(args):
2     dbc_file_in = args.dbc_file
3     file_output = args.output
4
5
6     print("Reading files...")
7     dbc_reader = DbcReader()
8     messages = dbc_reader.read(dbc_file_in)
9     print("Generating grammars...")
10
11     #generate the .bt file for FormatFuzzer
12     ff_gen = OCanadaFuzzGen()
13     ff_gen.run(messages, output_filename=file_output)

```

LISTING 4.5: oCANada code to generate Binary Template in oCANada_driver.py

4.3.2.2 Compiling to Binary

Once a binary a template has been generated via the oCANada code, then the built-in function in FormatFuzzer can be run to compile the binary template to a binary. This binary is then called each time a fuzz test needs to be generated. This can then be sent to FormatFuzzer with the steps below. After this is in FF's ./templates/ folder, it can be compiled with ./build.sh.

```

1 $ git clone https://github.com/uds-se/FormatFuzzer.git

```

```
2 $ cd FormatFuzzer
3 $ cp ~/<path_to_bt_output> ./templates/
4 $ $ ./build.sh <name_of_bt_file_without_extension>
```

LISTING 4.6: copying the binary template to FF to generate an FF binary

4.3.2.3 oCANAda fuzzing usage

Now that the binary has been compiled, oCANAda may be utilized to run the fuzz tests. oCANAda will continuously call the FF compiled binary, reading the output and sending that same output on the can bus in the correct endianness. This fuzzing will occur until until a user ends the fuzzing process with a ctrl-c interrupt. The binary is called with the "fuzz" option, which reads from /dev/random as a source of randomness. It should also be reiterated that whether the evil bit is set or not depends on how the binary file was compiled. When running in FF mode, then either the little and big endian options must be selected through the -le or -be flag.

```
1 def ff_runner(args):
2     if args.little_endian:
3         endianness = "little_endian"
4     elif args.big_endian:
5         endianness = "big_endian"
6     else:
7         sys.exit("Endianness must be specified with -le or -be")
8
9     fuzzer_name = args.fuzzer
10    output_file_name = args.output
11    oracle = args.oracle
12    channel = args.channel
13
14    if oracle:
15        print("Oracle set")
16
17
18    fuzzer = FFRunner()
19    try:
20        while True:
21            fuzzer.fuzz(fuzzer_name, output_file_name, endianness, oracle, channel
22            )
23    except KeyboardInterrupt:
```

```

23     print("\nCtrl-c detected... ending!!")
24     print("cleanup...")
25     os.remove(output_file_name)
26     print("done!")

```

LISTING 4.7: Main Function for running FF code

Practically, the different options that the tool needs to run can be seen below.

```

1 $ python ./oCANada_driver.py fuzz -h
2 usage: oCANada fuzz [-h] -f FUZZER [-fo OUTPUT] [-orc] [-le] [-be] [-c
   CHANNEL]
3
4 options:
5   -h, --help            show this help message and exit
6   -f FUZZER, --fuzzer FUZZER
                           location of FormatFuzzer compiled fuzzer
7   -fo OUTPUT, --output OUTPUT
                           location of where the output file for the fuzzer is
8                           stored. Default is ./out1.txt
9   -orc, --oracle        set oracle to be true
10  -le, --little_endian   set little endian parsing
11  -be, --big_endian      set big endian parsing
12  -c CHANNEL, --channel CHANNEL
                           sets channel to listen to, i.e.: vcan0, can0
13
14

```

LISTING 4.8: Command for running the oCANada fuzzer

4.3.3 oCANada Reverse FF Mode

The reverse mode is a purely blackbox method which can be utilized when a DBC is not available. This methodology involves using the Reverse Engineering of Automotive Data (READ) [35] (also mentioned as the algorithm in Section 3.2.3) algorithm in order to sort out signal boundaries and then feed those boundaries to FormatFuzzer via a binary template file. The READ algorithm involves 3 different steps, which oCANada's reverse mode honors, these are briefly explained below as well as referenced in Figure 4.6 and then further elaborated in the paragraphs below:

- **Pre-process phase** - a bit-flip array and magnitude array is generated.
- **First phase** - signal boundaries are found.

- **Second phase** - Cyclic Redundancy Checks (CRC) and Counter values are found.

READ Algorithm

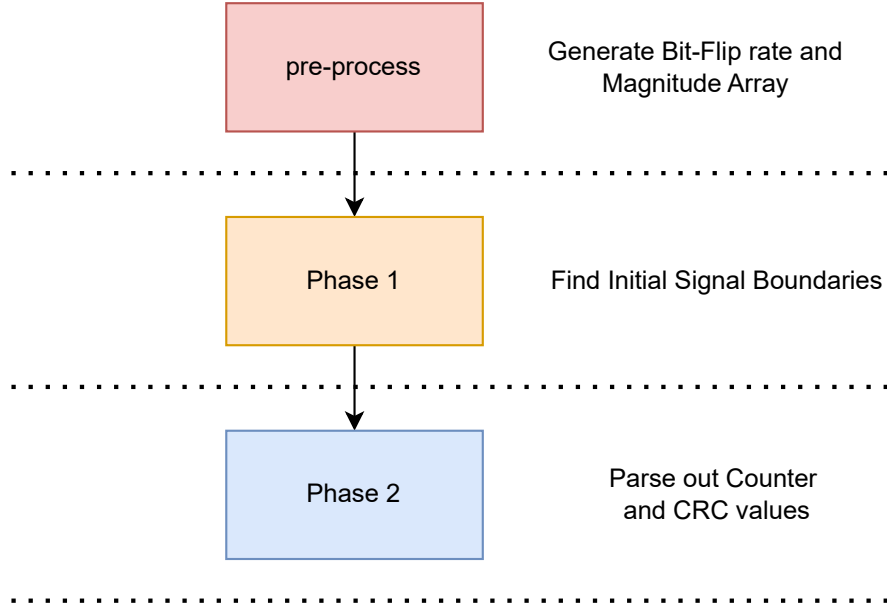


FIGURE 4.6: Figure showing the phases of the READ algorithm.

The first phase will generate a bit-flip array and a magnitude array. The bit flip array is calculated using the following equation as based off of the READ algorithm [35] [34]:

$$BFR_{c,k} = \frac{1}{n_c} \sum_{i=1}^{n_c} b_{c,k,i}$$

This formula calculates the bit flip rate of a given message for each bit position in the payload. n_c is the total number of recorded CAN messages, and this is multiplied over the sum of the recorded bit flip changes. However, while testing this formula was slightly changed for the work with oCANada. Instead of counting the total number of consecutive messages, it was decided to divide by the total number of possible bit flips. The reason for this decision was because by doing so the results were more accurate since dividing by the total of messages was not yielding the result of 1 for the BFR when it should have. The following formula was formulated by testing in order to arrive at more accurate results. So while the rest of the formula remains the same, we divide by t_{bf} where t_{bf} is the number of total possible bit flips:

$$BFR_{c,k} = \frac{1}{t_{bf}} \sum_{i=1}^{n_c} b_{c,k,i}$$

Then the following equation is used to generate a magnitude array also based off of the READ algorithm [35]:

$$M_i = \lceil \log_{10}(B_i) \rceil$$

Once the pre-process phase is complete as per the formulas above, the first-phase can begin. In this first phase, the initial signal boundaries are mapped out by going through the magnitude array. Once a value in the magnitude array is smaller than the previous value in the array, this means that a new signal boundary is found. The reason for this is because the final bit of a signal (least significant bit) should have the highest BFR in that message, and therefore if the BFR has suddenly decreased it means that a new signal should have started (most significant bit). In other words, as the entire payload is processed bit-by-bit, the magnitude of the BFR should steadily decrease indicating the next least significant bit, until finally a higher magnitude indicate that there is a new message and a new most significant bit. So once this is found, the start and end index for the preceding signal boundaries are saved. It should also be noted that this assumes that the endianness is big, and the reverse logic would have to be applied for little endian.

The following listing shows the code utilized in order to perform the bit-flip rate analysis itself:

```

1 def apply_BFR_analysis(self, parsed_logs_dict, endianness="little_endian"):
2     bfr_dict = {}
3     for arb_id, dlc_seperated_payloads in parsed_logs_dict.items():
4         for payloads in dlc_seperated_payloads:
5
6             if endianness=="little_endian":
7                 payloads = self.rearrange_payloads_as_little_endian(
8                     payloads)
9             elif endianness != "big_endian":
10                 raise Exception("Endianness not properly set")
11
12             DLC = int(len(payloads[0])/2)
13             BFR_array, magniture_arr = self.generate_BFR_and_mag_array(
14                 payloads, DLC)

```

```

14     print("Bit flip analysis array found:")
15     print(BFR_array)
16
17     print("magnitude array found: ")
18     print(magnitude_arr)
19
20     print("Finding signal boundaries!!!!")
21     signal_boundaries = self.find_signal_boundary(payloads,
22 magnitude_arr)
23
24     print("Finding counter and CRC values")
25     updated_signal_boundaries = self._find_Counter_and_CRC(
26 BFR_array, magnitude_arr, signal_boundaries)
27
28     print("Finding constant values when the BFR is 0")
29     self._check_for_constant_values(BFR_array,
30 updated_signal_boundaries)
31
32     print("Finding cosntant values")
33     print(updated_signal_boundaries)
34
35     if bfr_dict.get(arb_id):
36         bfr_dict[arb_id].append(updated_signal_boundaries)
37     else:
38         bfr_dict.update({arb_id:[updated_signal_boundaries]})
39
40     return bfr_dict

```

LISTING 4.9: Function which analyses the parsed CAN log in order to generate the BFR analysis

In the second and final phase, CRC and Counter values are found. Counters values can be found as the bit-flip rate of counters drop by 1/2 after each bit. For example, if there was a 4 bit counter the BFR would be: 1/8, 1/4, 1/2, 1. CRC values have a distribution a BFR distribution of 0.5 [35], although in their paper Kim et al. found that it was better to use a distribution between 0.3 and 0.6 [34].

Once the boundaries have been found, and the values have been sorted (into physical values, CRC and Counter values), then these different signals can be saved into a binary template file similar to how this was done in the regular FF mode 4.3.2. The same process for fuzzing can then be found when running the compiled FF binary as in Section 4.3.2.3.

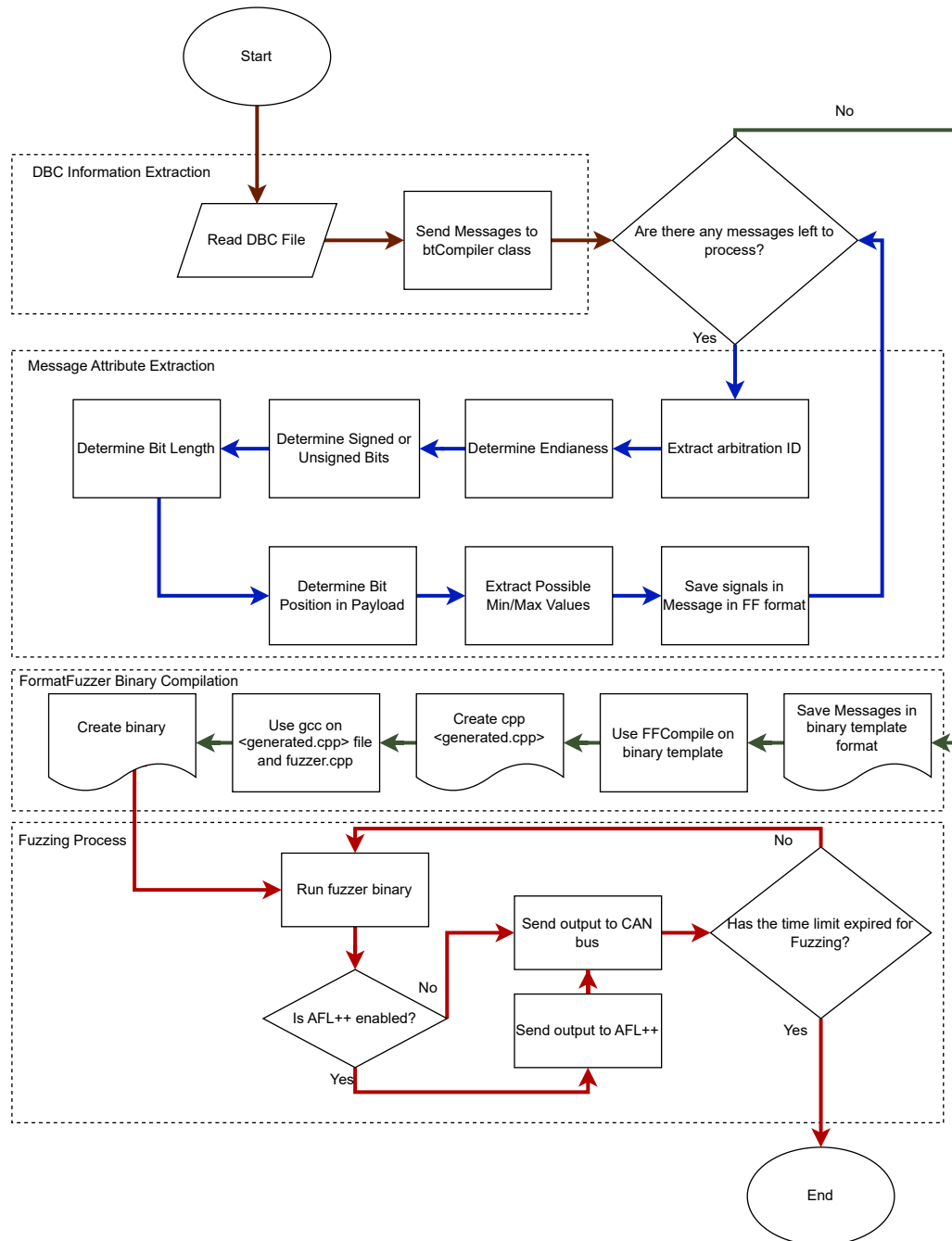


FIGURE 4.7: Flowchart demonstrating the steps required for oCANada's DBC Format-Fuzzer mode.

4.3.4 Test Oracle

The idea for the test oracle is to have three properties to test for. These properties are timeout based, using the tester-present service, and monitoring network traffic for messages outside a baseline. The first two properties are taken from Radu and Garcia's paper as described in section 3.3.3. In the timeout based approach, component behavior must be monitored to see if the component is sending messages within a network. If it stops sending messages in the network, then this could indicate that some fuzzed message has caused behavior change. Another thing that can be used is the tester present service, if the ECU has this feature. The UDS message 0x3e can continuously be sent, and it is expected that the ECU will give some kind of response (the response can either be positive or negative). If it stops giving a response then this can indicate that the component has crashed or is in a different internal state. Finally, in the network-based solution, baseline of CAN packets is recorded over the network for a set period of time. After the baseline is gathered, the fuzzing process begins. The test oracle will isolate for a specific ECU ID. When the ECU response sends a CAN message to the network and the packet is outside of the baseline, the response packet is flagged. While any of these properties do not guarantee that there is a behavior change, it is a step forward in a blackbox/greybox environment where information on the SUT's internal state is not so clear. In general the properties of what to test for in a blackbox/greybox are not as clear as in a whitebox setting, where a debugger could be set to monitor the internal state.

4.4 Testing Plan

Now that the lab setup as well as the fuzzing strategies have been explained, a test plan is required in order to test the effectiveness of the fuzzer. Figure 4.8 shows the steps which the tool will be used and evaluated to help in the pentesting process. In the first step, a system under test will be chosen. There are three systems to test, a simple CANoe simulation called Easy, a medium complexity simulation called CANSys, and finally a real gateway ECU plugged into a simulation. In the second step, the different fuzzers are chosen to compare one against the other. The three modes of operation (as explained in the section on oCANada fuzzing strategies) are chosen, along with the well known CAN fuzzing tool CaringCaribou. CaringCaribou was specifically chosen because of its widespread use within the automotive area. Other CAN specific fuzzing tools that were

discussed in the SoTA were considered, but these were not freely available. While no SoTA paper had a solution that is compatible to the oCANada's DBC FF mode, there was a paper that utilized reverse engineering for format-based fuzzing [34].

All these strategies will be run against the different system and compared to each other. As these fuzz, the third step demonstrates that the test oracle is running. In the final step, the results are compared by the different criteria found by the test oracle. As can be seen in the figure and elaborated on in the section on the test oracle (see section 4.3.4), the criteria for the test oracle are the tester present service, system timeout, and unexpected network behavior. The next section will explain the results founds using this strategy.

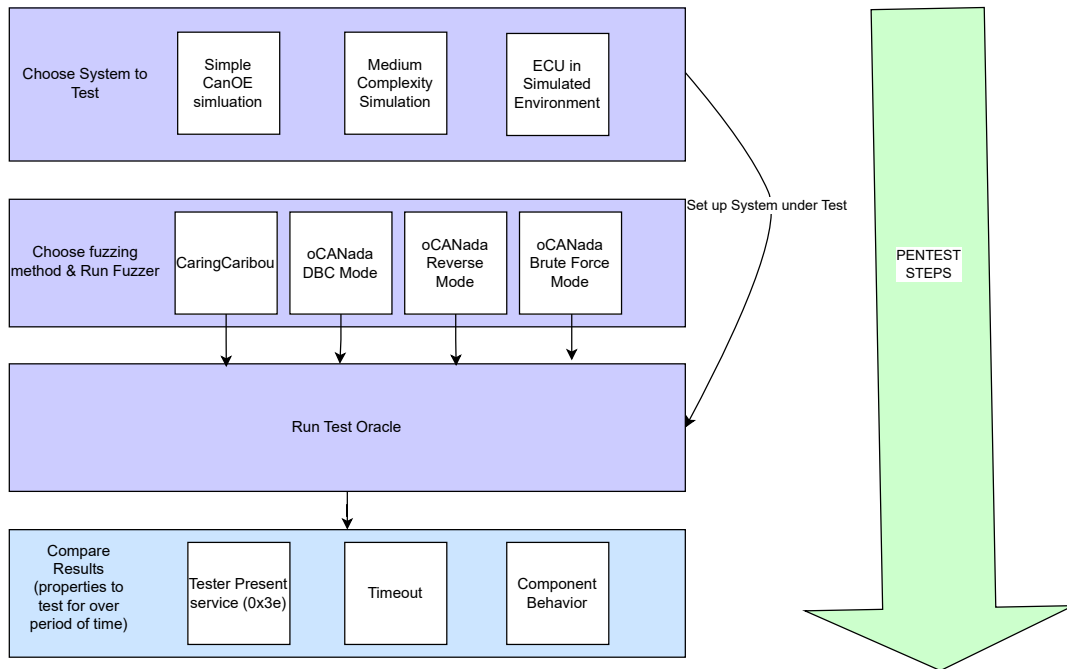


FIGURE 4.8: Proposed Steps for testing the different fuzzers.

Chapter 5

Results

5.1 oCANada Results

5.1.1 Fuzzer Runtime Speed

Tables 5.1, 5.2, and 5.3 show the speed at which the different fuzzers inject messages. The values represented in the following table 5.1 represent the speed of the fuzzer, where each value is the number of messages per second. These messages were recorded over a total number of messages, and then divided by how many seconds it took (which is why these numbers include fractions). However, table 5.1 shows us the total messages within a simulated environment, and this simulated environment can be polluted with messages from the simulation. Therefore, there is also Table 5.3 where the speed of the fuzzers in a pure environment without simulated or real ECUs, where the only thing that can send messages to CAN is the fuzzer itself. In this table, the only thing that was compared was FormatFuzzer vs CaringCaribou, it can be seen that CaringCaribou is slightly faster when generating messages.

The general conclusion from this diagram is that CaringCaribou provides a speedup of 1.2-4 times compared to FormatFuzzer depending on the scenario. The speedup appears to diminish when complexity is increased, particularly, the more signals that exist within a message. The reverse mode seems to generally follow the speed of the FormatFuzzer mode, which is expected as the reverse mode utilizes FormatFuzzer for fuzzing.

Fuzzing Strategy	<i>Easy</i>	<i>CANSystem</i> PowerTrain	<i>CANSystem</i> Comfort
CaringCaribou	113.7	224.0	224.0
DBC FF Mode	32.35	144.4	143.6
BF Mode	60.61	52.2	50.5
Reverse FF Mode	72.75	139	158

TABLE 5.1: The speed of the fuzzer, where each values is the speed calculated for messages/second.

Fuzzing Strategy	<i>Gateway ECU</i>
CaringCaribou	216.09
DBC FF Mode	174.63

TABLE 5.2: The speed of the fuzzer, where each values is the speed calculated for messages/second.

Fuzzing Strategy	Frequency in messages/second
CaringCaribou	94.4
DBC FF Mode with <i>Easy</i> compiled .bt	72.8
DBC FF Mode with <i>CANSystem</i> compiled .bt	74.6
DBC FF Mode with <i>Gateway</i> compiled .bt	71.2

TABLE 5.3: The speed comparison of fuzzing in a purely test environment

5.1.2 Reverse Fuzzer and Evil bit

Before the reverse mode was utilized a canlog needed to be captured. Figure 5.4 shows the canlog of the easy simulation that was captured when the vehicle was turned on and "driving" occurred. In this simulation, it can be seen that the RPM could only go from 0 to 4000. The oCANada reverse mode then generated a binary template from the canlog with both the evil bit set (so random mutations would occur), and with the evil bit unset (so no random mutations would occur) as can be seen in figure 5.5. From these two binary templates, an FF compiled binary could be generated and fuzzing could occur. When the evil bit was not set, the fuzzer only sent values from 0 to 4000, which was the same min and max range as the canlog. However, with the evil mode set, it could send random messages and capture new functionality that went from from -10000 to 10000 RPMs. Once the evil bit was set, then not only were most of the values still part of a "valid" range, but there was also the discovery of values outside of this range. This can demonstrate how useful it is to include the evil bit for fuzzing for further discovery and refinement

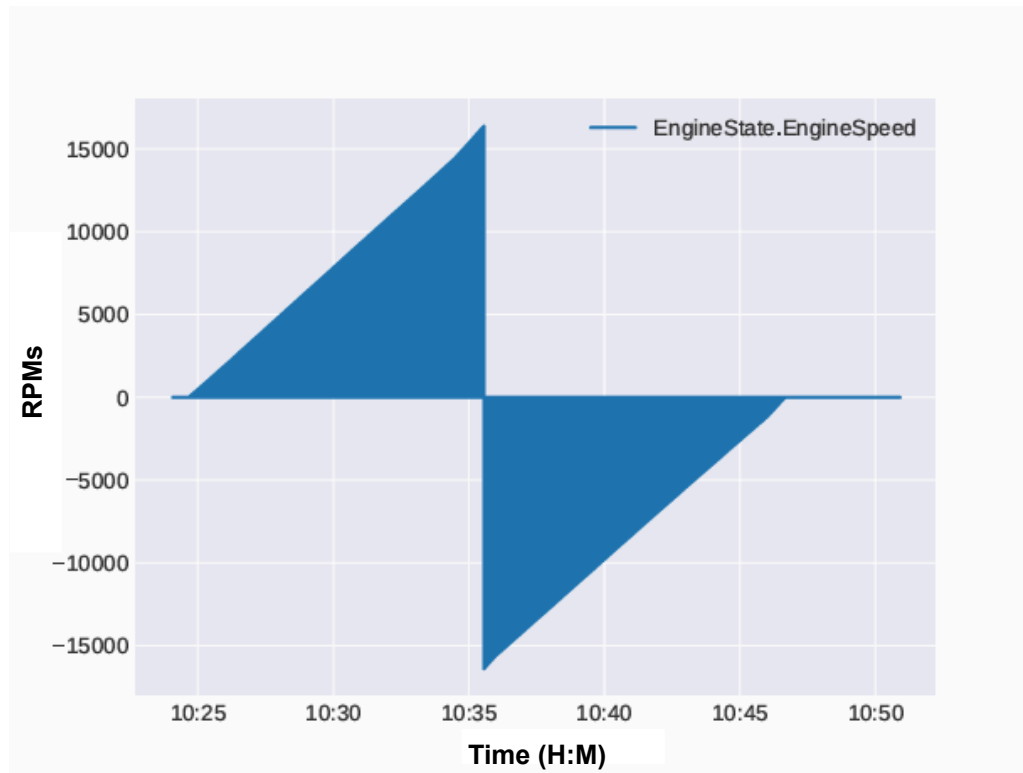


FIGURE 5.1: The results from fuzzing the easy simulation using the oCANada BF Mode

during the fuzzing process, since it found new functionality while also having most of the messages stay in the "valid" format of the original canlog.

5.1.3 Memory Errors Found

While fuzzing, memory errors were found in the CanSystem simulation. The benefit of running it in the simulation was that it made it possible to see where and how the code failed. In terms of results, Table 5.4 demonstrates what kind of errors were found when running CaringCaribou versus running oCANada utilizing the FormatFuzzer compiled binary. Different memory related errors were found by fuzzing, which were division by zero, invalid array range, and overflows. FormatFuzzer, in specific, found an array out of bounds having to do with the CAN transport protocol function when receiving a message. That are two conclusions that can be drawn from this section. First, the presented fuzzing strategies are able to find memory corruption errors by fuzzing ECUs via CAN. Second, it can be seen that the smart DBC fuzzing strategy using FormatFuzzer was able to find the memory errors that CaringCaribou find as well as one additional error.

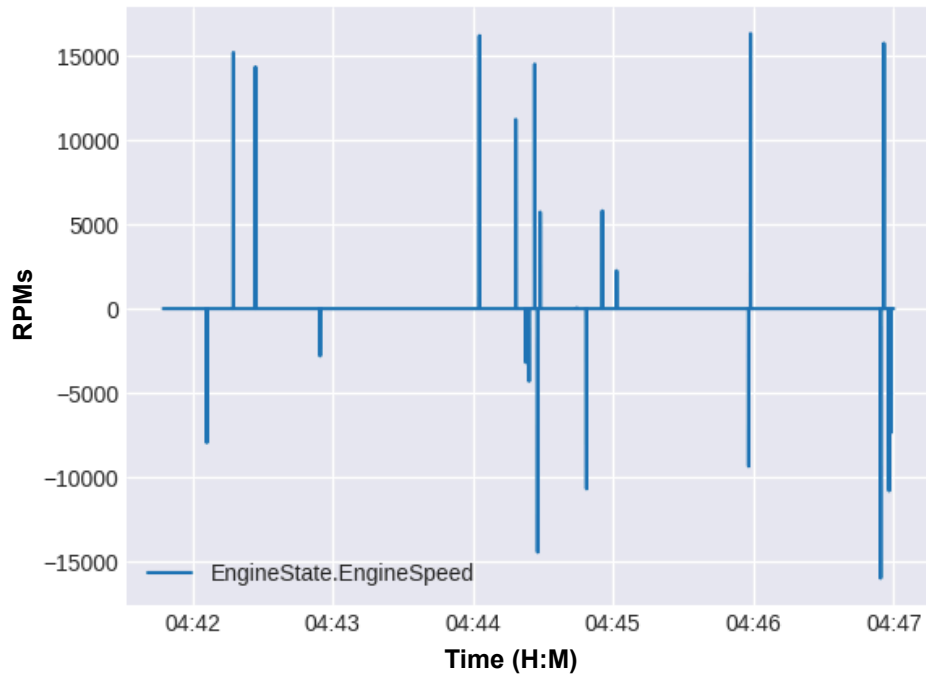


FIGURE 5.2: The results from fuzzing the easy simulation using CaringCaribou

Fuzzing Strategy	Division By Zero	Invalid Array Range	Overflow
CaringCaribou	1	3	1
oCANada FF DBC Mode	1	4	1

TABLE 5.4: Memory corruption errors found in the simulation when running oCANada's FF DBC Mode versus CaringCaribou

5.1.4 Test Oracle utilized with Gateway ECU

The test oracle was applied when fuzzing the gateway in specific. When this occurred there was a packet being reflected from the gateway to the APU. The test oracle was able to pick this up as this packet was new and generated from the gateway. However, since the gateway did not crash, the tester present and timeout aspects of the test oracle yielded no results. A larger and more varied amount of ECUs would be needed to further test the efficacy of the presented test oracle techniques.

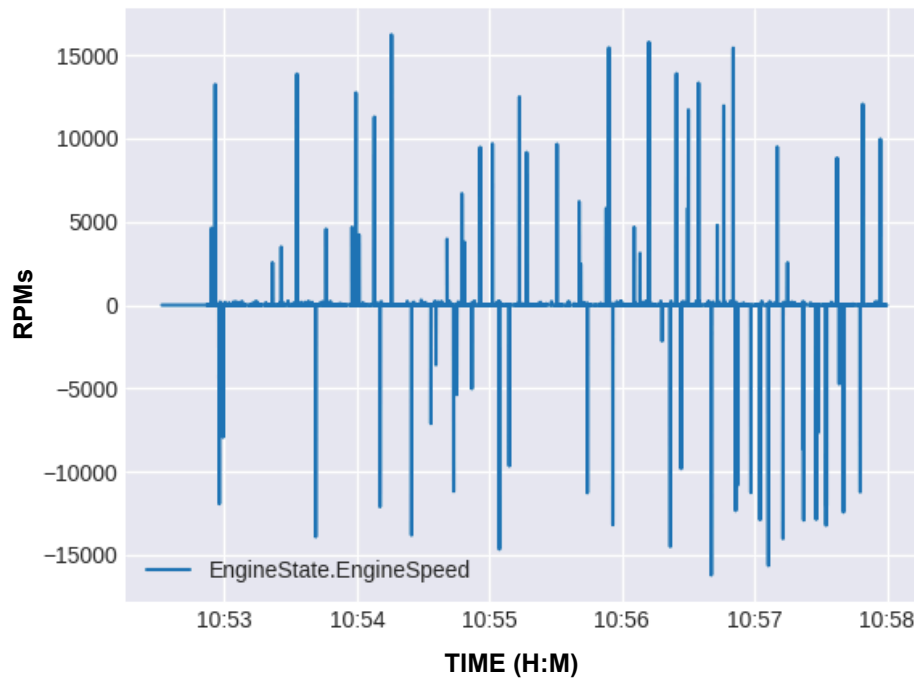


FIGURE 5.3: The results from fuzzing the easy simulation using the oCANada FF DBC Mode

5.1.5 Fuzzer Accuracy

Another metric for comparing fuzzers is the accuracy of the fuzz tests that are generated in terms of whether they'd be accepted by the input parser of the system. That is, how many generated messages actually have a valid syntax and will be accepted by the system. Since any system's input-parser would reject syntactically invalid messages and only accept syntactically valid ones, it'd greatly speed up the fuzzing process by generating valid messages to test the system. For our results, when CaringCaribou does not know what the right arbitration ID to use, it has an accuracy of 17.6%. However, when CaringCaribou knows the right ID, the accuracy jumps to 70%. The oCANada FF mode, because they take into account the format, always have a DBC accurate message and should always be accepted by the parser of the ECU under test.

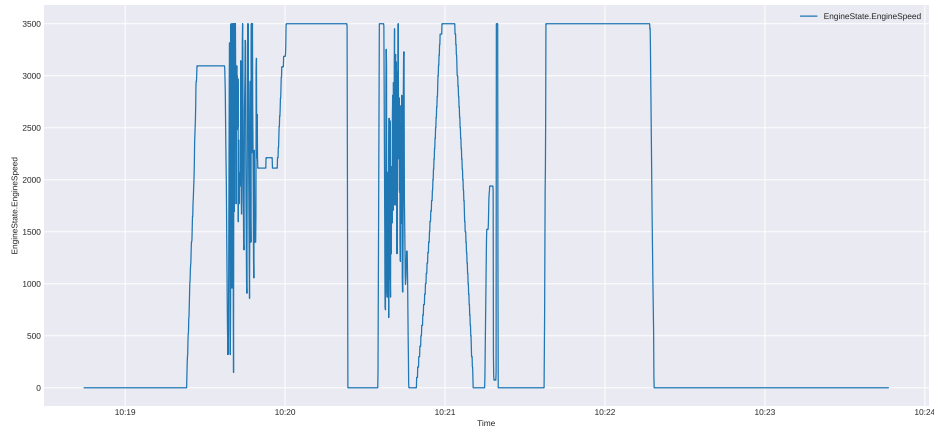


FIGURE 5.4: Normal operations of vehicle captured by the canlog

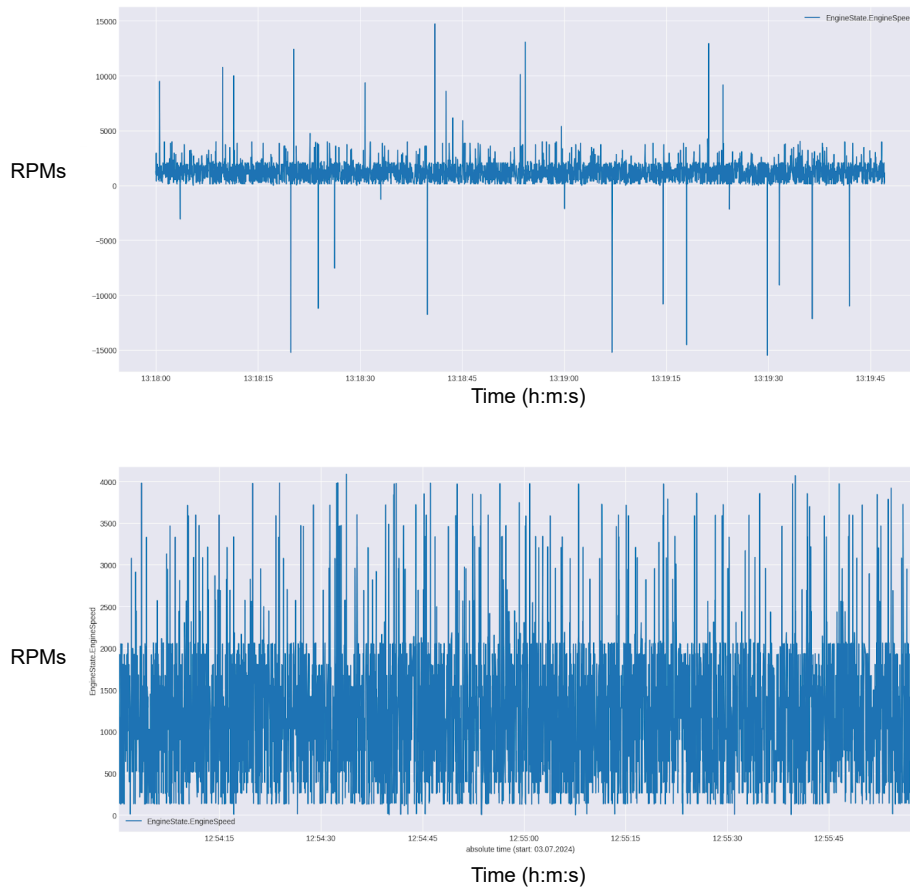


FIGURE 5.5: Comparison of the evil bit set (top) and evil bit unset (bottom) in the easy simulation.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

For this dissertation, a fuzzing solution was built in order to resolve three questions which were asked in the section discussing the motivation (Section 1.2). These research questions were the following:

1. What intelligent fuzzing techniques can be applied in order to fuzz Controller Area Networks (CAN) more efficiently in both a blackbox and greybox environment?
2. What are ways to define the test oracle for fuzzing a CAN system?
3. What metrics can be chosen to compare different CAN fuzzers?

The first question asked what intelligent fuzzing techniques could be applied in order to fuzz Controller Area Networks (CAN) more efficiently in both a blackbox and greybox environment. For this, intelligent fuzzing techniques that moved beyond simple mutational fuzzing strategies were explored and a solution was presented in Chapter 4. oCANada's solution relied on a generation-based fuzzer, utilizing FormatFuzzer on the backend, which would fuzz based off the structure provided in a DBC or utilizing CAN reverse engineering methods as opposed to utilizing mutational fuzzing such as what is used in the CaringCaribou tool.

The second question had to do with ways to define the test oracle when fuzzing a CAN system. A big challenge for this question was that there couldn't be more reliance for more conventional methods of a test oracle (e.g. checking for segmentation faults with AFL) due to the fact that the fuzzing occurred over the CAN network and there was

no way to monitor the internal states of the ECU. Any monitoring of the ECU had to be physical or over the CAN network. Three techniques were chosen which could help with CAN specific fuzzing and were presented in Section 4.3.4 of Chapter 4. The first technique was that of utilizing a network-based test oracle, that would gather a baseline of packets and then would monitor for packets that existed outside of that baseline. The next technique was that of utilizing the UDS Tester Present service (0x3e), which could be utilized in order to ping the ECU for a response and assume that there is some kind of internal state change if there was no response. And finally there was the technique of checking for a timeout in the ECU, such that if the ECU would send messages consistently and then suddenly stops during fuzzing, then this could indicate a further state change. It should be noted that while these methods can be utilized as a test oracle, they do fall under the limitation of not being as solid as something like checking for segmentation faults if that was available.

The final question had to do with what metrics could be chosen in order to compare different CAN fuzzers. This was presented mainly in Chapter 5. Here, CaringCaribou and oCANada's fuzzing solution was ran over two distinct CANoe simulations as well as a real Gateway ECU. We demonstrated results which showed the speed of the fuzzers during runtime, the effectiveness of having some more randomness while mostly fuzzing within bounds utilizing FormatFuzzer's evil bit, and we demonstrated some memory corruption errors which were found while fuzzing.

All of the research explained above contributes to the state of the art around fuzzing ECUs over CAN and in order to help move the knowledge base forward in order to help secure the automotive world.

6.2 Future work

While the previous work was successful in developing a tool that fuzzed in a smart way, there exist a few areas of research that could be continued. There is more work to be done in terms of applying reverse engineering techniques within the fuzzing environment. New techniques are always being developed for reverse engineering CAN messages. Importantly, these works have not yet been applied to fuzzing. There is also more work to be done for trying to apply machine learning techniques to fuzzing, this could be in relation to the previous point about reverse engineering techniques with machine

learning, or other techniques for fuzzing CAN that utilize machine learning. A big problem with this is ensuring that there is the availability of high quality data sets which can be challenging.

In terms of future work for the oCANada solution in itself, said future work could include further testing with the fuzzing tool on more real world ECUs. Because of the restriction where there was only one real world ECU to test on, it was difficult to prove any real world bugs despite proving that it is possible to find memory corruption errors with the simulated ECUs. More research is also needed for the test oracle, both in terms of improving the techniques themselves, as well as testing those techniques on real ECUs. The issues with the current techniques can be summed up by saying that the solution provided is better than simply having someone observe the behavior of a vehicle, but the solutions provided are not enough to prove that there is a security flaw. Rather, the current test oracle solution are indications of possible flaws. Admittedly, a test oracle within a black box or grey box environment is much more likely to not only find flaws or find false positives than a test oracle which is working in a white box environment. More research on this area is required, as it is possible that newer and better techniques could provide a test oracle which better finds security vulnerabilities activated by the fuzzer.

Appendix A

Extra Binary Templates

```
1 typedef struct {
2
3     WORD sig_2: 1;
4     WORD sig_1: 12;
5     WORD sig_0: 3<min=0, max=0>;
6 } msg_123;
7
8 typedef struct {
9
10     UBYTE sig_3: 1;
11     UBYTE sig_2: 1<min=0, max=0>;
12     UBYTE sig_1: 1;
13     UBYTE sig_0: 5<min=0, max=0>;
14 } msg_321;
15
16
17 typedef struct{
18     uint16 arbitration_id;
19 } ARBID;
20
21 struct {
22     LittleEndian();
23     local int evil = SetEvilBit(false);
24     ARBID id;
25     SetEvilBit(evil);
26     if(id.arbitration_id != 291 && id.arbitration_id != 801){
27         Warning("Not valid arbid");
28         return -1;
```

```
29 }
30
31 switch(id.arbitration_id){
32     case 291:
33         msg_123 message;
34         break;
35     case 801:
36         msg_321 message;
37         break;
38 }
39 } main;
```

LISTING A.1: Little Endian Binary Template Generated Through Reversing

```
1 typedef struct {
2
3     WORD sig_0: 7;
4     WORD sig_1: 1;
5     WORD sig_2: 3<min=0, max=0>;
6     WORD sig_3: 5;
7 } msg_123;
8
9 typedef struct {
10
11     UBYTE sig_0: 5<min=0, max=0>;
12     UBYTE sig_1: 1;
13     UBYTE sig_2: 1<min=0, max=0>;
14     UBYTE sig_3: 1;
15 } msg_321;
16
17
18 typedef struct{
19     uint16 arbitration_id;
20 } ARBID;
21
22 struct {
23     BigEndian();
24     local int evil = SetEvilBit(false);
25     ARBID id;
26     SetEvilBit(evil);
27     if(id.arbitration_id != 291 && id.arbitration_id != 801){
28         Warning("Not valid arbid");
```

```
29         return -1;
30     }
31
32     switch(id.arbitration_id){
33         case 291:
34             msg_123 message;
35             break;
36         case 801:
37             msg_321 message;
38             break;
39     }
40 } main;
```

LISTING A.2: Big Endian Binary Template Generated Through Reversing

Bibliography

- [1] U. S. LTD., “Upstream 2023 global automotive cybersecurity report,” 06 2023. [Online]. Available: <https://upstream.auto/reports/2023report/> [Cited on pages 1, 6, and 7.]
- [2] —, “Upstream 2024 global automotive cybersecurity report,” 06 2024. [Online]. Available: <https://upstream.auto/reports/global-automotive-cybersecurity-report/> [Cited on page 1.]
- [3] C. Valasek and C. Miller, “Remote exploitation of an unaltered passenger vehicle,” 06 2015. [Online]. Available: https://ioactive.com/pdfs/IOActive_Remote_Car_Hacking.pdf [Cited on pages 1, 2, and 7.]
- [4] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” 2018. [Online]. Available: <https://doi.org/10.1186/s42400-018-0002-y> [Cited on pages 3, 10, 11, and 12.]
- [5] S. Mallisery and Y.-S. Wu, “Demystify the fuzzing methods: A comprehensive survey,” *ACM Comput. Surv.*, vol. 56, no. 3, oct 2023. [Online]. Available: <https://doi.org/10.1145/3623375> [Cited on pages 3, 9, 10, and 11.]
- [6] Microsoft, “Security development lifecycle practices.” [Online]. Available: <https://www.microsoft.com/en-us/securityengineering/sdl/practices> [Cited on page 6.]
- [7] “Road vehicles — cybersecurity engineering,” International Organization for Standardization, Geneva, CH, Standard, Mar. 2021. [Cited on page 6.]
- [8] F. Luo, X. Zhang, Z. Yang, Y. Jiang, J. Wang, M. Wu, and W. Feng, “Cybersecurity testing for automotive domain: A survey,” *Sensors*, vol. 22, no. 23, 2022. [Online]. Available: <https://www.mdpi.com/1424-8220/22/23/9211> [Cited on pages 6, 8, 9, and 10.]

- [9] C. Ebert and R. Ray, "Penetration testing for automotive cybersecurity," vol. 16, no. 2524-8804, 2021, pp. 516–527. [Online]. Available: <https://doi.org/10.1007/s38314-021-0629-4> [Cited on pages 6 and 7.]
- [10] F. Sommer and R. Kriesten, "Attack path generation based on attack and penetration testing knowledge," 01 2023. [Cited on page 6.]
- [11] A.-I. Radu and F. D. Garcia, "Grey-box analysis and fuzzing of automotive electronic components via control-flow graph extraction," in *Proceedings of the 4th ACM Computer Science in Cars Symposium*, ser. CSCS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3385958.3430480> [Cited on pages 7, 12, 23, 24, 28, 29, 30, and 31.]
- [12] C. Smith, *The car hacker's handbook: a guide for the penetration tester*. no starch press, 2016. [Cited on pages 7, 8, and 9.]
- [13] D. Fowler, J. Bryans, M. Cheah, P. Wooderson, and S. Shaikh, "A method for constructing automotive cybersecurity tests, a can fuzz testing example," 07 2019. [Cited on page 7.]
- [14] D. S. Fowler, J. Bryans, and S. A. Shaikh, "Automating fuzz test generation to improve the security of the controller area network," 2017. [Online]. Available: <https://api.semanticscholar.org/CorpusID:218595214> [Cited on page 8.]
- [15] "opendbc." [Online]. Available: <https://github.com/commaai/opendbc> [Cited on page 8.]
- [16] "cantools." [Online]. Available: <https://github.com/cantools/cantools> [Cited on page 8.]
- [17] R. Gopinath and A. Zeller, "Building fast fuzzers," 2019. [Cited on pages 9, 11, 18, 19, and 20.]
- [18] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, sep 2022. [Online]. Available: <https://doi.org/10.1145/3512345> [Cited on pages 9, 10, and 11.]

- [19] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021. [Cited on pages 9, 10, 11, 12, 13, and 15.]
- [20] R. Hodován, A. Kiss, and T. Gyimóthy, “Grammarinator: a grammar-based open source fuzzer,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 45–48. [Online]. Available: <https://doi.org/10.1145/3278186.3278193> [Cited on pages 11 and 20.]
- [21] M. Olsthorn, A. van Deursen, and A. Panichella, “Generating highly-structured input data by combining search-based testing and grammar-based fuzzing,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’20. New York, NY, USA: Association for Computing Machinery, 2021, p. 1224–1228. [Online]. Available: <https://doi.org/10.1145/3324884.3418930> [Cited on page 12.]
- [22] “American fuzzy lop.” [Online]. Available: <https://github.com/google/AFL> [Cited on pages 12 and 16.]
- [23] M. Zalewski, “Pulling jpegs out of thin air,” 2014. [Online]. Available: <https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html> [Cited on page 12.]
- [24] “American fuzzy lop.” [Online]. Available: <https://lcamtuf.coredump.cx/afl/s> [Cited on pages 12, 13, and 16.]
- [25] V. Atlidakis, P. Godefroid, and M. Polishchuk, “Restler: Stateful rest api fuzzing,” in *ICSE 2019*, November 2019. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/restler-stateful-rest-api-fuzzing/> [Cited on page 13.]
- [26] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, “Metamorphic testing: A review of challenges and opportunities,” *ACM Comput. Surv.*, vol. 51, no. 1, jan 2018. [Online]. Available: <https://doi.org/10.1145/3143561> [Cited on page 13.]

- [27] M. Zalewski, “Binary fuzzing strategies: what works, what doesn’t,” 2014. [Online]. Available: <https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html> [Cited on page 16.]
- [28] “Aflplusplus.” [Online]. Available: <https://github.com/AFLplusplus/AFLplusplus> [Cited on page 17.]
- [29] “Caringcaribou.” [Online]. Available: <https://github.com/CaringCaribou/caringcaribou> [Cited on page 17.]
- [30] R. Dutra, R. Gopinath, and A. Zeller, “Formatfuzzer: Effective fuzzing of binary file formats,” 2023. [Cited on pages 18, 21, 22, and 23.]
- [31] “Dharma.” [Online]. Available: <https://github.com/MozillaSecurity/dharma> [Cited on page 21.]
- [32] R. Dutra, R. Gopinath, and A. Zeller, “Format fuzzer.” [Online]. Available: <https://github.com/uds-se/FormatFuzzer> [Cited on page 21.]
- [33] T. Werquin, M. Hubrechtsen, A. Thangarajan, F. Piessens, and J. T. Mühlberg, “Automated fuzzing of automotive control units,” in *2019 International Workshop on Secure Internet of Things (SIOT)*, 2019, pp. 1–8. [Cited on pages 25, 28, 29, and 31.]
- [34] H. Kim, Y. Jeong, W. Choi, D. H. Lee, and H. J. Jo, “Efficient ecu analysis technology through structure-aware can fuzzing,” *IEEE Access*, vol. 10, pp. 23 259–23 271, 2022. [Cited on pages 25, 26, 28, 30, 31, 44, 46, and 49.]
- [35] M. Marchetti and D. Stabili, “Read: Reverse engineering of automotive data frames,” *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 4, pp. 1083–1097, 2019. [Cited on pages 26, 27, 28, 43, 44, 45, and 46.]
- [36] M. D. Pesé, T. Stacer, C. A. Campos, E. Newberry, D. Chen, and K. G. Shin, “LibreCAN: Automated CAN message translator,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2283–2300. [Online]. Available: <https://doi.org/10.1145/3319535.3363190> [Cited on page 26.]

- [37] H. Zhang, K. Huang, J. Wang, and Z. Liu, "Can-ft: A fuzz testing method for automotive controller area network bus," in *2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI)*, 2021, pp. 225–231. [Cited on pages [27](#), [28](#), and [31](#).]
- [38] D. S. Fowler, J. Bryans, S. A. Shaikh, and P. Wooderson, "Fuzz testing for automotive cyber-security," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2018, pp. 239–246. [Cited on pages [29](#) and [31](#).]
- [39] A. Anistoroaei, B. Groza, P.- Murvay, and H. Gurban, "Security analysis of vehicle instrument clusters by automatic fuzzing and image acquisition," in *2022 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 2022, pp. 1–6. [Cited on page [31](#).]