KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Automated Reverse Engineering and Fuzzing of the CAN Bus

Timothy Werquin

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Artificiële intelligentie

Promotoren:

Prof. dr. ir. F. Piessens
Dr. J.T. Mühlberg

Assessoren:

Prof. dr. B. Jacobs
J. Van Bulck

Begeleiders:

Dr. M. Vanhoef
A. Thangarajan

Academiejaar 2018 – 2019

Preface

I would like to express my sincere gratitude to anyone that helped me with any part of my thesis. In particular my promoters and assistants, who provided continuous support and guidance throughout my work. I would also like to thank the jury for reading the text. Finally, I would like to thank my parents and Toon Willemot for proof reading my work.

Timothy Werquin

Contents

Abstract

Modern vehicles are becoming more and more computerised and interconnected. They contain a network of Electronic Control Units (ECUs) which are responsible for taking inputs from the environment and the driver, process these inputs, and control the actuators that regulate the driving behaviour of the vehicle. These ECUs communicate with each other over vehicle control networks such as the Controller Area Network (CAN). These protocols were developed in the 80s without any security consideration. Increasingly, these ECUs also connect to the outside world through vehicle-to-vehicle and vehicle-to-infrastructure channels. Over the past few years, it has been shown that these communication channels can be manipulated and reverse engineered. These manipulations allow attackers to control the various subsystems of modern vehicles remotely. A significant part of these activities involves manually reverse engineering the communication traffic over the CAN bus, which can be a time-consuming task.

In this thesis, we present a method to reverse engineer this traffic automatically by applying fuzzing techniques. Over the past few years, fuzzing has become a growing part of the security tester's toolbox. However, in the embedded sector, fuzzing has seen limited use as defining a bug oracle for embedded systems is difficult. The method presented in this thesis uses a sensor harness to receive feedback from the ECUs under test. This sensor harness is attached to the ECU under test and detects physical responses, which are then used in a bug oracle function to inform the fuzzing process. First, we detail the construction of the sensor harness and the various fuzzing methods and bug oracles that we used. We then evaluate our approach on three instrument clusters, one virtual and two physical. These tests show that the fuzzer is capable of automatically identifying interesting CAN messages. Finally, some recommendations for potential future work are made.

Abstract

Moderne voertuigen zijn steeds meer gecomputeriseerd and geconcerteerd. Ze bevatten een network van *Electronic Control Units (ECU's)* die verantwoordelijk zijn om invoer van de omgeving en de bestuurder te nemen, deze invoer te verwerken en het regelen van de actuators die het rijgedrag van het voertuig regelen. Deze ECU's communiceren met elkaar over voertuigcontrolenetwerken zoals het *Controller Area Network* (CAN). Deze protocollen zijn in de jaren '80 ontwikkeled zonder beveiligingsoverwegingen. In toenemende mate zijn deze ECU's ook met de buitenwereld verbonden via voertuig-naar-voertuig en voertuig-naar-infrastructuur kanalen. In de afgelopen jaren is gebleken dat deze communicatiekanalen kunnen worden ontcijferd en gemanipuleerd. Dit stelt aanvallers in staat om de verschillende subsystemen van moderne voertuigen te besturen van op afstand. Een belangrijk deel van deze activiteiten bestaat uit de handmatige reverse engineering van het communicatieverkeer over de CAN-bus, wat een tijdrovende taak kan zijn.

In deze masterproef wordt een methode gepresenteerd om dit verkeer automatisch te reverse-engineeren door het toepassen van fuzzingtechnieken. De laatste jaren is fuzzing een groeiend deel van de toolbox van de veilgheidstester geworden. In de embedded sector heeft fuzzing echter een beperkt nut aangezien het definiëren van een bug orakel voor embedded systemen moeilijk is. De methode die in deze masterproef wordt gepresenteerd past bestaande fuzzing technieken toe op ECU's met een sensor harnas om feedback te ontvangen. Dit sensorharnas wordt bevestigd aan de te testen ECU en detecteert fysieke reacties, die vervolgens worden gebruikt in een orakelfunctie om het fuzzingproces te informeren. Eerst wordt de constructie van het sensorharnas en de verschillende gebruikte fuzzingmethoden gedetailleerd beschreven. Vervolgens evalueren we onze aanpak op drie instrumentenclusters, één virtuele en twee fysieke. Deze tests tonen aan dat de fuzzer in staat is om automatisch interessante CAN-berichten te identificeren. Tot slot worden enkele mogelijke toekomstige uitbreidingen voorgesteld.

List of Figures and Tables

List of Figures

List of Tables

Chapter 1

Introduction

The world is becoming more computerised and interconnected. This trend is also present in vehicles where more and more mechanical systems are replaced with ECUs (electronic control units). A modern car has more than 100 of these ECUs managing various systems such as the engine, ABS and infotainment systems [11]. In recent years these controllers are also performing more tasks without user input, most cars have adaptive cruise control which will maintain a set distance from the vehicle in front and lane assist which will steer the car between road markings. These vehicles will often contain systems that connect to the outside world. Examples of these connections are the WiFi and cellular internet present in the infotainment system or the link to roadside units and other vehicles using the Vehicle-to-Everything (V2X) technologies.

As these features become more and more common, and the amount of code present in modern vehicles increases the security of these systems becomes more critical to guarantee safety. There are a number of published attacks on modern cars such as a Tesla S model [23], Nissan LEAF [18] and Chevrolet Corvette [13]. One such attack, performed on a Jeep Cherokee, only required the IP address of the car [27]. This IP address could be found by scanning the internet, which allowed the attackers to hack every model of that car from their home computer. After connecting to the car using this IP address, the attackers could control the brakes, shut down the engine and control the media centre. The fact that this is possible is a consequence of modern cars becoming more connected, meaning that cyber-security is a new threat that car manufacturers have to deal with.

The local interconnect used to connect these different ECUs is a vital part of the vehicles electronic subsystem. A vehicle typically has a limited number of these control networks to which every ECU is connected. The protocols these busses use such as CAN (Controller Area Network) were developed in the 80s and security was not part of their design. Testing these networks is now more important than ever, as they are often also connected to the outside world through infotainment systems, for example. The goal of this thesis is to apply an automated testing method (fuzzing) to these control networks to both test their security and to aid reverse-engineering them.

Fuzz testing or fuzzing is an automated testing technique used to find vulnerabilities in software by sending intentionally malformed or unexpected input to the software. A fuzzer consists of three major components, an input generator which creates program inputs following some scheme, an evaluator which executes the program with the generated input and finally a bug oracle which detects whether the input is the cause of a bug. These three components are run in a loop to allow vulnerabilities and bugs to be found without any human input.

The techniques developed for fuzz testing can also be applied to reverse engineering, in which case the oracle would not check for vulnerabilities but for activation of a signal that is being reverse engineered. Reverse engineering the traffic present on the vehicle control networks is often a significant part of penetration tests on cars. This reverse engineering is a laborious task that requires intimate knowledge of the vehicle's systems. In the Jeep Cherokee hack, for example, the attackers decompiled diagnostic tools to reverse engineer CAN bus traffic [27]. Automating this process using fuzzing can speed up these penetration tests, making them less labour intensive.

Applying fuzzing on these ECUs and control networks requires some form of feedback. In an embedded network, it is difficult to use a debugger to monitor program execution, which is used when testing regular applications. In this thesis, a sensor harness is used to monitor the ECUs under testing. This allows the fuzzer to determine whether the ECU responds correctly to any generated input. This sensor harness can contain any number of sensors of many different types. Currently, the sensor harness only uses light sensors as we evaluated the fuzzer on instrument clusters, which mainly have visual feedback.

The results presented in this thesis were also submitted as part of a paper on CAN fuzzing to the Workshop on Cyber Security for Intelligent Transportation Systems (CSITS). This paper is included in appendix A.

1.1 Thesis Goals

The goal of this thesis is to apply fuzz testing to a specific type of vehicle control network, namely the CAN bus. First, a selection of sensors for the sensor harness is made, and a driver program is written to interact with the harness. To apply fuzzing to the CAN bus using the sensor harness, we develop some new fuzzing techniques and evaluate them. Finally, we apply the fuzzer to a virtual instrument cluster as a proof of concept and to two physical instrument clusters, one from a consumer car and one from a modern truck. We expect that results from these tests will show that fuzz testing is a promising method to automate both reverse engineering and vulnerability discovery in the automotive sector.

1.2 Limitations

As the tested ECUs came from consumer cars without any specification, the majority of the work in this thesis focuses on reverse engineering. Most methods can be directly

adapted from reverse engineering to vulnerability discovery when a specification is provided as the bug oracle would use this specification to detect unspecified outputs.

The sensor harness only contains one type of light sensor, since the tested ECUs were instrument clusters. These have mainly visual feedback for which light sensors are sufficient to detect most of the interesting signals. When testing other types of ECUs, other types of sensors would need to be integrated into the sensor harness.

1.3 Thesis Outline

- **Chapter 1** - The introduction provides a short overview of the goals and motivations for this thesis.
- **Chapter 2** - The second chapter gives an overview of the relevant background information. It contains an overview of fuzz testing and the various control networks and their protocols used in a modern vehicle.
- **Chapter 3** - Chapter three contains information about all hardware used in this thesis. It mainly focuses on the sensor harness.
- **Chapter 4** - In chapter four the fuzzing software is explained in detail. It explains how the Caring Caribou framework was adapted to use the sensor harness and which additional fuzzing methods were implemented.
- **Chapter 5** - In this chapter the fuzzer is applied to a virtual instrument cluster.
- **Chapter 6** - Chapter six applies the fuzzing techniques to a dashboard taken from a consumer vehicle.
- **Chapter 7** - This chapter explains how the fuzzer was applied to an instrument cluster taken out of a modern truck.
- **Chapter 8** - The related work chapter discusses other published research related to our work and reasons why our work extends this research.
- **Chapter 9** - In the conclusion the most important results are summarised and some potential future work is outlined.

Chapter 2

Background

This chapter gives an overview of various topics that are required to understand the rest of this thesis. First, an overview of fuzzing and the different fuzzing techniques is given. Next, the different automotive control network protocols used in modern vehicles are touched upon and the CAN protocol is explained in detail. Finally, the application of fuzzing to the CAN protocol is discussed.

2.1 Fuzzing

Fuzzing or fuzz-testing is an automated software testing technique. Fuzzing works by repeatedly running a program with random, unexpected or malformed input to discover vulnerabilities and other faults. As software is becoming more complex and challenging to reverse engineer, it is becoming the number one technique for finding vulnerabilities used by hackers [41]. In response, developers have also started using fuzzing to find vulnerabilities before attackers do. For example, the Google Chrome browser is continually tested using fuzzing in the cloud [28].

2.1.1 Fuzzer Taxonomy

A fuzzer can be categorised in three classes based on how much of the internal state of the program under test it can observe [25]. The groups are black-, grey- and white-box fuzzers. The classification of a fuzzer is not always clear and is often done on a best judgement basis. In [25] a broad set of fuzzers is classified according to this categorisation.

Back-box Fuzzer

A black-box fuzzer does not see the internals of the program it is testing. This means it only observes the input/output behaviour of the program and makes the program under test a black-box. Black-box fuzzing is the most difficult form of fuzzing as the execution of the program cannot influence the generated inputs. In this thesis, no information about the internal working of an ECU is used, which makes our fuzzer a black-box fuzzer.

White-box Fuzzer

A white-box fuzzer will generate inputs using the full knowledge of the internals of the program it is testing. This knowledge includes both static information such as analysis of the source code and dynamic analysis gathered while the program is running. A common technique used is symbolic execution, which analyses a program to determine which parts of the input influence the execution path of the program. Another technique is taint analysis, which marks input data and tracks this mark (taint) throughout execution. White-box fuzzing is usually much slower than black-box fuzzing.

Grey-box Fuzzer

A grey-box fuzzer situates itself between white-box and black-box. It will observe some internal state of the program and its executions. However, unlike white-box fuzzers, it does not statically analyse the full program under test. Grey-box fuzzers usually employ some form of instrumentation to aid the fuzzing progress. This instrumentation can track which branches are taken, for example. Grey-box fuzzing is the most common technique used when finding vulnerabilities in programs.

2.1.2 Fuzzer Components

A model algorithm fuzzing is given in listing 2.1, it is a simplified version of the model used in [25]. The model consist of three main components: the input generator (`GenerateInput` on line 3), the input evaluator (`InputEval` on line 4) and the bug oracle (`BugOracle` on line 5).

Input Generator

The input generator generates the input for the program under test [41]. It does this by selecting a program input from the input space of the fuzzer. As this input space is quite large, it cannot merely be enumerated. Thus a smarter selection method is needed. This selection can be completely random, based on a predefined model (such as a grammar) or be a mutation of an existing correct input. Random input generation is quite often very ineffective, as the probability of generating valid input is usually very low. Moreover, even if a valid input is generated the probability of exploring interesting execution paths is even lower. Using a model is very effective and can generate a lot of accepted yet unexpected inputs. However, a model of the valid input space is often not available or difficult to specify. Mutation based input generation uses an existing input (the seed) which is modified in some (random) way. This modification can be as simple as flipping some random bits, but can also include adding or removing data. Mutation based input generation is used very often when fuzzing files as it can generate a large set of input files from a limited set of seed files.

Listing 2.1: A model fuzzing algorithm

```
1 bugs := []
2 while Running():
3     input := GenerateInput()
4     exec := InputEval(input)
5     bug := BugOracle(exec)
6     bugs.append(bug)
7 end
```

Input Evaluator

The input evaluator will run the program under test on the generated input [41]. When fuzzing a file, the evaluator will run the program on the generated file. When fuzzing a network protocol, the evaluator will need to emulate either the client or server and provide the generated input messages to the program under test. Fuzzing user interfaces is also possible; in this case, the input evaluator will interact with the UI based on the generated input. Sometimes a mechanical device is used when fuzzing smartphones, for example [17].

Bug Oracle

The bug oracle is responsible for determining whether the given execution of a program is indicative of a bug [41]. It does this by determining whether the given execution violated a security policy. A commonly used technique is to use instrumentation to detect spatial memory errors. Another technique works by using a compiler plugin to detect undefined behaviour. Both of these techniques do not work for black-box fuzzers as they require internal information from the program. When using black-box fuzzing to fuzz a protocol, a common bug oracle is a valid case oracle. This oracle will check whether the program has crashed by sending a known valid case which has a known response to it. If the program does not respond correctly, the oracle knows it has crashed.

2.2 Automotive Control Network Protocols

As cars have gotten more and more complex, more ECUs are needed to control various aspects of the vehicle such as emergency braking, cruise control. Modern vehicles can contain more than 100 independent ECUs [11]. As most of these ECUs need to communicate with each other, various control networks are present in modern vehicles. This chapter will explore the different kinds of control networks there exist in current vehicles and focus on one particular protocol (CAN) that is used in almost all modern cars.

2.2.1 An Overview

There exist several different bus protocols which may be used by a modern vehicle. Some of these protocols are made for low-speed information such as air conditioning control, while other protocols are made for high-speed communications such as engine speed and ABS information.

CAN

The Controller Area Network (CAN) is the most common bus protocol used in automotive control networks. The first version of the protocol was released in 1984 by Bosch [3]. It was standardised by the International Organisation for Standardisation (ISO) in 1994 by ISO 11898 [20]. The on-board diagnostics connector (ODB), mandatory on all US and European vehicles, exposes the CAN bus. Also, as of 2008, the CAN bus is mandatory for all vehicles sold in the US. The CAN bus is used for both critical communications such as ABS and less critical communications such as media control. It is a serial multi-master bus, allowing multiple nodes to communicate with each other using a single two-wire bus. The bus supports bit rates of up to 1 Mbit/S for distances less than 40 meters. Recently a new standard, CAN-FD, has been released. This standard allows a flexible data-rate to be used and is viewed as the next-generation protocol for automotive control networks. Nevertheless, it does not include any increased security considerations. The detailed workings of the CAN 2.0 bus will be explained in section 2.2.2.

LIN

The Local Interconnect Network (LIN) is a cheaper alternative to CAN [32]. The first fully implemented version of the protocol was released in 2003. It is a single master network and supports up to 15 slave nodes. The latest version of the standard specifies a bit rate of 20 kbit/s for distances less than 40 meters. It uses a single wire and can also be used over the 12 volt DC battery bus.

MOST

The Media Oriented System Transport (MOST) bus is a serial bus used for transporting media in a vehicle. The network uses a ring topology with a single timing master. However, other network topologies are also supported. A MOST network can support up to 64 nodes. The network is mostly used to transport media such as video or audio streams. It uses an optical physical layer, but newer versions of the protocol also support Ethernet as a physical layer. The latest version of the protocol (MOST150) supports a data-rate up to 150 Mbit/s.

FlexRay

The FlexRay bus is a more expensive competitor to the CAN standard [30]. It is designed to be both faster and more reliable, supporting data rates up to 10 Mbit/s. It supports both a star topology and a single bus connecting multiple nodes. As it is

more expensive than the commonly used CAN bus, it is used mostly in premium vehicles such as the Audi A5 [2] or the Mercedes S-class [26].

Ethernet

In recent years the use of Ethernet within a vehicle has increased [24]. Ethernet offers a widely supported protocol with a high bandwidth (100 Mbit/s for 100Base-T1). As Ethernet is a point-to-point protocol, a switch needs to be used to connect multiple nodes. This increases the cost and the size of the wiring harness. As an example, the Tesla Model S uses Ethernet extensively [23] in addition to CAN.

2.2.2 The CAN Protocol

This section will give an overview of the CAN 2.0 bus protocol [20]. First, the physical layer is explained. Next, the transfer layer and frame formats are detailed. Finally, a brief overview of how the object layer of the CAN protocol is used by the applications.

Physical Layer

For its physical layer, the can protocol uses a two-wire bus. These two wires carry a differential signal, which means each wire carries the complement of the other. These wires are terminated with a 120Ω characteristic impedance. The signals carried over these wires are either dominant or recessive. A 'one' bit is called recessive as the bus is not driven when transmitting it. The 'zero' bit is dominant as the bus is driven (high wire towards 5V and low wire towards 0V) when transmitting it. This serves as a bus arbitration mechanism, when two nodes transmit a different bit at the same time only the zero bit will be received by the other nodes. When transmitting a node needs to monitor the bus constantly, when it detects that the bus state differs from its transmitted state, a collision has occurred and it will stop transmitting. This means the node that transmits the first recessive (1) bit will have to stop transmitting.

Transfer Layer

Over this two-wire bus, the CAN protocol transmits frames. The CAN protocol defines four types of frames: data frame, remote frame, error frame and overload frame. There are two types of data frames, a standard data frame with an 11-bit arbitration ID and an extended data frame with a 29-bit arbitration ID. The structure of the standard data frame is given in figure 2.1.

The most important fields from this data frame are:

- **Aribtration ID** - the ID of the packet, often represents its type (for example engine speed message, brake light message, etc.). As the bits of the arbitration ID are the first bits transmitted they also serve as a priority. Due to the built-in bus arbitration, the message with a lower arbitration ID will have more

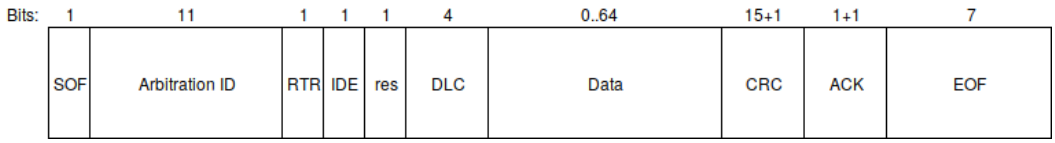


Figure 2.1: The standard CAN frame.

dominant bits. This causes a lower arbitration ID to have precedence over another message when a collision occurs.

- **DLC** - The data length code, a four-bit integer representing the number of bytes in the message. A message can contain a maximum of eight bytes.
- **Data** - The actual payload data of the message, containing up to eight bytes (64 bits) of data.
- **CRC** - A 15-bit cyclic redundancy check, used for error detection.

If the transmitter wants to verify that its message is received, it can use the acknowledgement bit. This bit is left recessive (1) by the transmitter when a receiver acknowledges the message it sets this bit to the dominant (0) state. The transmitter can then monitor the bus to verify whether the acknowledgement bit was set to the dominant state. When the identifier extension bit (IDE) is set to one, the frame is an extended data frame. The extended data frame will contain an additional 18-bit arbitration ID extension, which allows a 29-bit arbitration ID to be used.

The remote frame is used to request data from another node. It sets the remote transmission request (RTR) bit to one and contains no data. The data length code will contain the number of bytes requested. The overload frame is not used anymore in modern CAN controllers. It historically signalled that the CAN controller was overloaded and could not keep up, allowing the other nodes to pause.

The error frame is transmitted by a node when it detects an error. The errors can be caused by a failed CRC check, wrong frame format, wrong bit detected while transmitting or a failed acknowledgement. Each node keeps two counters: a transmission error counter and a receive error counter. When either counter reaches a specific value, the node will first stop transmitting active error frames. This means the error frames will not overwrite the other traffic on the bus. When the counters reach an even higher value, the node will enter the bus-off state and stops transmitting any frames. The error counters are decreased when a frame is successfully transmitted or received.

Object Layer

The object layer defines which messages are to be transmitted and which received messages are to be used. Together with the transfer layer, the object layer defines the data link layer of the OSI network model. The CAN protocol does not specify any standard protocols for arbitration ID assignment or any standard arbitration

ID interpretation. It also does not contain any transport protocol which would allow transmitting more than eight bytes of data over the bus. This means different manufacturers assign different interpretations to a CAN message. A common method is to use the arbitration ID to denote the message type, assigning the lowest arbitration IDs to the most critical messages. Several protocols built on top of CAN aim to improve the interoperability between components. There is, for example, the ISO-TP transport protocol. This protocol allows sending data over the CAN bus that is longer than eight bytes. The ISO-TP protocol is used by the Unified Diagnostic Services (UDS), which is a diagnostic service present on almost any vehicle. The AUTOSAR standard, which aims to improve the interoperability between applications running on ECUs within a vehicle, provides another standard [4]. The layers provided by AUTOSAR include a CAN abstraction stack, which includes network management in addition to some other services. Another protocol suite which builds on top of the CAN protocol is the J1939 standard [35]. This standard is mostly used in trucks and heavy-duty vehicles. The J1939 standard will be explained in the next section.

2.2.3 The J1939 Protocol Suite

The SAE J1939 protocol suite is used in trucks and heavy-duty vehicles [35]. It provides several features on top of the CAN bus such as peer-to-peer and broadcast messages, transport protocol for sending larger payloads, network management and the definition of standard parameter groups. A variation of the J1939 standard is also used for communication between the towing and towed vehicle in the ISO11992 standard. In this section, an overview of the J1939 protocol suite is given as one of the instrument clusters tested uses this protocol for all its communications.

Arbitration ID Interpretation

J1939 only uses the extended data frame format from the CAN protocol. It divides the 29-bit arbitration ID in a number of fields as seen in figure 2.2. Some of these fields define the parameter group number (PGN), which identifies how the message should be interpreted. Many standard PGNs are defined by the standard for common messages such as engine speed, engine temperature, etc.

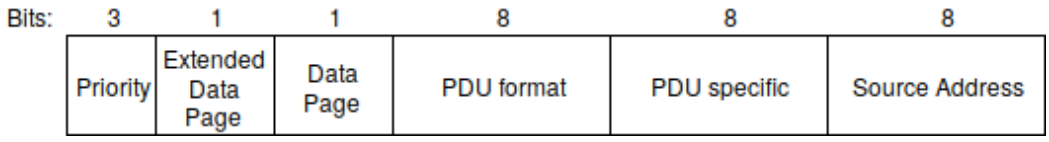


Figure 2.2: The J1939 arbitration ID.

The first field is the priority field, a lower priority message will be more likely to survive a transmission collision. The next two bits define the data page. The data page defines how the PGN should be interpreted. There are four data pages of which two are defined. The only data page of interest is page 0 as this page is used in the automotive sector, page 1 is used in the maritime industry. The next field,

the PDU format, is the first part of the PGN. If the PDU format is less than 240, then the message is a peer-to-peer message, and the PDU specific will contain the target address. If the PDU format is greater than or equal to 240, then the message is a broadcast message, and the PGN is constructed of the PDU format and the PDU specific. The last field is the source address of the node. This source address is defined by the network management protocol.

Parameter Group Number

The PGN is constructed from the data page number, the PDU format and the PDU specific (if the message is not peer-to-peer). The J1939 standard defines many standard PGNs, each with a standard data interpretation. For example, the PGN 65,262 defines the engine temperature, whose data contains several temperatures such as coolant temperature, oil temperature, etc. In addition to these standard PGNs, J1939 also defines one proprietary peer-to-peer PGN (61184) and 256 proprietary broadcast PGNs (65280 to 65535). These PGNs can be used by manufacturers to transmit custom messages.

Network Management

Each application running on an ECU needs to have a J1939 address. It has to claim this address on start-up and either choose a new address on collision or emit a fault. There are 256 addresses, some of which have special meaning. The address 254 is the null address, and the address 255 is the global address. The standard also defines some standard addresses for common applications such as the engine, gearbox, etc.

Transport Protocols

The J1939 standard also defines two transport protocols, one for broadcast messages and one for peer-to-peer messages. These protocols allow up to 1785 data bytes to be transmitted. For broadcast messages, the sender transmits an announce message followed by the message parts. A minimum interval of 50ms separates these messages. For peer-to-peer messages, the sender first sends a "request to send" message. The receiver will respond with a "clear to send" message, to which the sender responds with the first message part. The receiver will keep responding with a "clear to send" until all parts are received.

2.3 Fuzzing the CAN Protocol

In this thesis, fuzzing is applied to the CAN network of various instrument clusters. The goal is to use fuzzing not only for vulnerability discovery but also to reverse engineer unknown CAN communications. This poses several unique problems which are not encountered when fuzzing non-embedded software and protocols. The main problem is that the ECUs are separated which makes it difficult to receive feedback for crash detection. This makes it difficult to define bug oracles which work efficiently,

up till now the most commonly used bug oracle when fuzzing CAN is a human observer. We solve this by using a sensor harness which will be attached to the ECU under test. This harness can monitor the observable state of the ECU, allowing us to apply black-box fuzzing methods to the CAN network. The harness hardware is detailed in chapter 3.

When using fuzzing to reverse engineer unknown CAN message a simple sensor bug oracle can be used. This bug oracle will trigger as soon a sensor changes state, allowing the fuzzer to identify which message caused which state change. When trying to find vulnerabilities and bugs a valid case bug oracle can be used. This bug oracle will check if the ECU under test is still responsive by sending a CAN message which is known to trigger a detectable response. An example message would be one that triggers an indicator on an instrument cluster, which can be detected by the sensor harness. A detailed explanation of the implemented fuzzing algorithms is given in chapter 4.

2.4 Conclusion

This chapter provides an explanation of fuzzing and the automotive control network protocols used in this thesis. Fuzzing has become a valuable tool when testing software. It is used both by developers to find bugs and by reverse engineers to discover exploitable vulnerabilities. The CAN protocol is one of the most widely used bus protocols in consumer vehicles. The fact that the CAN protocol has no built-in message authentication or message source verification makes it a prime target to apply fuzzing.

Chapter 3

The Hardware

The hardware used by the fuzzer consists of a USB CAN interface and a sensor harness. The CAN interface allows the fuzzer to send CAN messages to the ECU(s) under testing while the sensor harness provides feedback about which responses these messages trigger. The sensor harness is needed as an ECU usually does not respond on the CAN bus to all received messages. This chapter gives an explanation of the used CAN interface and the components used in the harness and how they are connected.

3.1 The USBtin CAN Interface

The USBtin is a low-cost USB CAN interface which allows a regular PC to connect to a CAN bus using a USB port [12]. The USBtin uses a PIC microcontroller in addition to a CAN controller and transceiver to provide a simple serial interface over which a PC can send CAN messages. As the Caring Caribou framework uses the socketCAN interface to communicate, an additional driver is needed to connect the serial CAN interface to a socketCAN interface. This driver is provided by the *CAN utils* package which also includes some utilities to send and receive raw CAN messages.

3.2 The Sensor Harness

The sensor harness is used to provide feedback to the fuzzer. It connects multiple sensors and provides a simple USB interface to the PC. As we mostly fuzzed dashboards, the sensor harness only uses light sensors. Placing these light sensors over various indicators on the dashboard allows the fuzzer to respond to changes in these indicators.

The sensor harness uses no external microcontroller which allows all code to be written in the Python programming language and run on a single PC. The light sensors are connected using an I2C bus as this interface is present on many low-cost sensors. The use of I2C makes the sensor harness extendable with various other

sensors such as a sound sensor to monitor auditory alerts, a motion sensor to monitor steering wheel movement or a current sensor to detect an engine start.

As the I2C bus supports multiplexing, multiple sensors can be connected to the same bus as long as they have a different I2C address. The low-cost sensors used in the harness often have a fixed address, which means an I2C multiplexer must be used to connect multiple sensors of the same type in the harness. The harness uses a USB to I2C adapter to connect the I2C bus to the PC.

3.2.1 Light Sensors

The ISL29125 colour sensor is used as a light sensor, this allows the fuzzer to detect not only light levels but also colours [19]. The sensor provides a simple RGB light level readout and has some configuration options which determine the sensitivity and precision of the sensor. The sensitivity can be configured between 10K lux or 375 lux and the sensor has a built-in infrared light filter which can be configured separately.

The precision of the sensor can be configured to be either 12 or 16 bits. Changing the precision also changes the integration time, meaning a higher precision (16 bits) will require the ADC to sample the sensor longer, which results in slower measurements.

In addition to the I2C interface, the sensor has an interrupt pin which can be triggered by a configurable light level on either of the red, green or blue channel. Using the interrupt pin in the sensor harness would be difficult as this would require an additional wire per sensor. Currently, the harness works by polling each sensor individually. This does not require an interrupt pin but requires more computation time, and if the polling does not happen frequently enough, some events may be missed.

As is the case with many low-cost I2C devices, the ISL colour sensor has a single fixed I2C address, this means a multiplexer must be used to connect multiple sensors on the same I2C bus. The sensor harness uses the *SparkFun RGB Light Sensor* breakout for the ISL29125 sensor [38] to make experimenting on a breadboard easier. This board breaks out the I2C and interrupt pins from the tiny ISL chip and provides the required pull-up resistors on these lines.

3.2.2 Multiplexer

As the ISL colour sensor has a fixed I2C address, the TCA9548A multiplexer is used to connect multiple colour sensors in the harness [43]. The TCA9548A multiplexer has eight I2C channels which allow eight colour sensors to be connected on the same bus. As the multiplexer has a three-bit configurable address, multiple multiplexers can be daisy-chained, allowing up to 64 I2C busses on a single connection. A breakout for the TCA9548A is also available from *SparkFun* which makes experimenting easier and again provides the necessary pull-up resistors on each I2C channel [40].

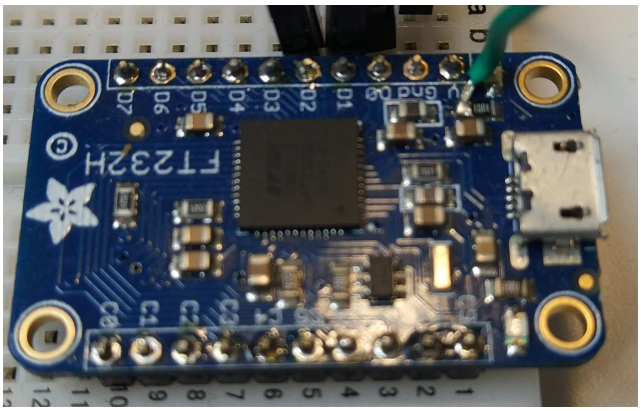


Figure 3.1: The USB to I2C adapter. The green wire has been soldered to a 3.3 volt trace as the adapter does not expose this trace.

3.2.3 USB Adapter

The harness uses the FTDI FT232H [15] adapter to connect the I2C bus to a PC. This adapter supports several different bus protocols such as UART, SPI and I2C in addition to a GPIO interface which can be used to write to eight digital IO pins. Some extra pull-up resistors need to be connected between the I2C bus lines and the power bus to use the I2C port.

In the harness a breakout board from *Adafruit* was used, this allows the adapter to be put on a breadboard for experimenting [1]. The ISL29125 colour sensor requires a 3.3-volt power supply which the FT232H chip provides but is not exposed on the breakout board. To not have to use an external 3.3-volt power supply, a wire was soldered to a trace which powers the power on led of the breakout board with 3.3 volts as seen in figure 3.1.

3.2.4 Connecting the Components

The I2C bus uses two wires to communicate (the SCL clock line and SDA data line) in addition to a ground and power wire. This means the connections from the USB adapter to the multiplexer and from the multiplexer to each sensor use just four wires. The USB adapter also requires some pull up resistors between the power wire and I2C wires which are put on a breadboard. The complete connection diagram with the adapter (1), the multiplexer (2) and two colour sensors (3) can be seen in figure 3.2. Additional sensors can be connected to each of the ports of the multiplexer or directly to the USB adapter if they have a unique I2C address.

3.2.5 Python Sensor Interface

The Arduino library for reading ISL29125 sensors [39] was adapted to Python to read out the light sensors from Python. The resulting library uses the Adafruit library for communicating with the FT232H adapter. As the Adafruit library only works using

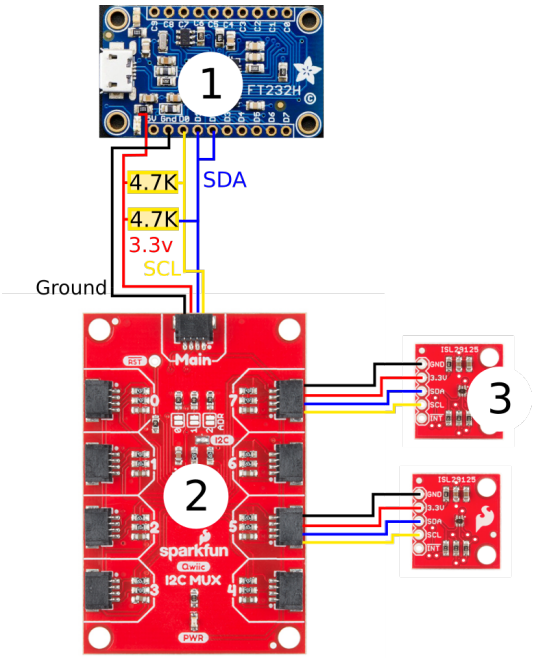


Figure 3.2: The connection diagram of the sensor harness. Contains the USB to I2C adaptor (1), the I2C multiplexer (2) and the colour light sensors (3).

version 2 of the Python programming language, the whole project needs a Python 2 environment instead of the newer Python 3 environment. The light sensor library exposes a few functions for initialising a new sensor and reading its red, green and blue light value.

The initialisation function resets the sensor and configures it in RGB mode, which enables all three colour channels, puts it in 10k lux mode for bright environments and enables high IR light adjustment. Reading a colour value is done by reading from the relevant device register, which returns a colour value of either 12 or 16 bits depending on the chosen precision. The time the sensor needs to take a reading varies depending on the ADC integration time which in turn depends on the chosen precision, at 16-bit precision each reading takes about 110ms while at 12 bit each reading takes 7 ms. As the sensor's output registers are double-buffered, reading out the sensor before it has finished returns the old values.

To use multiple light sensors, a library that interfaces with the TCA9548A multiplexer was created, this library allows virtual I2C ports to be created for each channel of the multiplexer. These virtual I2C ports can then be used in the sensor library instead of the default FT232H I2C port. To switch I2C channels, the library writes the number of the requested channel before any commands. This additional I2C command adds a small delay before every I2C command. This delay is insignificant when compared to the integration time of the light sensor, so it does not impact fuzzing.

3.2.6 Sensor Calibration

The colour light sensors are used to detect whether the various indicators on a dashboard are on or off, which means that the red, green and blue light levels need to be converted into a binary signal for the fuzzer. As the sensor is sensitive enough to detect a hand moving behind the sensor when attached to a dashboard in both sensitivity configurations (up to 375 lux and 10k lux) a simple threshold is not sufficient. An alternative calibration method involves taking a reading when the indicator is on and when the indicator is off. This gives two red, green and blue light level triples to which any new measurement can be compared if the new measurement is closer to the on-value the indicator is detected as on and vice versa. This method assures that a uniform increase or decrease in ambient light does not change the detected indicator value. However, it requires calibration with the indicator both on and off, which may not be possible when the indicator trigger is unknown. When the indicator cannot be triggered during calibration, a simple threshold can be used to detect the indicator state.

3.3 Conclusion

This chapter detailed the hardware used to run the fuzzer. The USB CAN interface used to connect the PC to the ECU under testing and the sensor harness used to receive feedback from the ECU. The USBtin adapter is a cheap and easy to use CAN interface, but any other CAN interface that is compatible with socketCAN can be used. The sensor harness currently only consists of light sensors but can be extended with any other I2C sensor. As the used light sensor has a fixed I2C address, an I2C multiplexer is needed to connect multiple sensors in the harness. This increases the complexity slightly, but as every component can be purchased on a breakout board, the wiring is still straight forward.

Chapter 4

Adapting Caring Caribou

The Caring Caribou car security exploration tool was extended to implement a fuzzer that uses external sensors. Caring Caribou provides a framework with various modules that make interacting with a CAN network straightforward. To this framework, various fuzzing methods were added which make use of the sensor harness.

4.1 Caring Caribou

Caring Caribou is a framework for car security exploitation that was created as part of the HEAVENS (HEAling Vulnerabilities to ENhance Software Security and Safety) research project [9]. The project provides various modules that interact with the CAN bus and provides the tools to write new modules in the python programming language. The modules included in the project are:

- *send*: Manually send CAN packets from the command line or a file.
- *dump*: Logs all traffic on the CAN bus to the command line or a file.
- *uds*: Interacts with the Universal Diagnostic Services (UDS) standard, it can discover UDS services and reset the ECU.
- *xcp*: Interacts with the XCP calibration protocol, it can discover ECUs that support XCP and dump their internal memory
- *fuzzer*: Provides sevral fuzzing techniques. This module was written by Mathijs Hubrechtsen at Distrinet.

The fuzzing module exposes a number of different fuzzing techniques such as *random fuzzing*, *brute-force fuzzing*, *mutation fuzzing* and *identify fuzzing*. These fuzzing methods use either monitor the CAN bus for feedback or require user input in the case of the *identify method*. This means that the user of the module needs to manually monitor the ECU under testing in order to observe the results. This is a time-intensive task that is automated using the sensor harness in the new *autoFuzz* module.

The *autoFuzz* module implements various fuzzing methods that use the sensor harness for feedback. The module was written from scratch as monitoring the sensor harness requires a significant amount of code changes which cannot be factored out easily when not using the harness. This means that the existing fuzzing module would need to be entirely rewritten to support the sensor harness.

4.2 Fuzzing Methods

The *autoFuzz* module implements various fuzzing methods that can be used to either automatically reverse engineer an ECU or to test an ECU. This section explains in detail how each method works and when it is best used. These methods can be chained together to accomplish more complex reverse engineering or testing tasks.

4.2.1 Brute-force Fuzzing

The *brute-force fuzzing* method sends a fixed payload (or payloads) to a user-defined subset of all arbitration ids. The fuzzer can send both standard CAN packets, or it can send J1939 packets to proprietary PGNs. As the address space of the non-extended CAN packet is just 11-bits it can be enumerated in a few minutes. Most signals can be triggered by a well-chosen payload, such as all ones if the signal is suspected to be triggered by a bit flag. Because the address space is so small and enumerating, it does not take much time multiple payloads can be tested.

Input Generation

After the user specifies the minimum and maximum arbitration ID the *brute-force fuzzer* will send the CAN packets with the user-specified payload one by one waiting for a user-defined delay in between. The simplified algorithm can be found in listing 4.1. The `bruteforce_fuzz` method is supplied by the user with a range of arbitration ids to test (`start_id` and `end_id`), the payload to send (`payload`) and the amount of time to wait between sending messages (`delay`). The method will then run over every arbitration id in the range and construct a CAN message which will be sent to the CAN bus and a log file (lines 14 to 16). After sending a CAN packet, the fuzzer will wait for the specified `delay`, during this period any state changes of the sensors will also be logged. This is done by the `delay_log` method, which will loop for the specified amount of time while monitoring the sensor harness state.

Oracle

After collecting the logs from the *brute-force fuzzer*, an analysis can be made. If a specification is available all sensor activations can be checked against this specification to identify anomalous behaviour. While reverse engineering and no specification is available, the log file can be used to identify which CAN message caused which sensor activation. When using the *brute-force method* for vulnerability discovery, a known message can be used as bug oracle. This known message causes a response

Listing 4.1: The *brute-force fuzzing* method

```
1  func delay_log(delay):
2      start_time := time()
3      old_state := sensors.states()
4
5      while time() - start_time < delay:
6          new_state = sensors.states()
7          if new_state != old_state:
8              log_state(new_state)
9      end
10 end
11
12 func bruteforce_fuzz(start_id, end_id, payload, delay):
13     for id in range(start_id, end_id):
14         message := CanMessage(id, payload)
15         send_message(message)
16         log_message(message)
17
18         delay_log(delay)
19     end
20 end
```

from the ECU under test, which can be detected by the sensors. If no response is observed the fuzzer has discovered a bug.

The state referred to by line two and six of the `delay_log` function can be either a binary on/off state for each sensor or a raw RGB value from the sensors depending on whether calibration is used. If the signal cannot be triggered the on/off calibration method cannot be used, so a simple threshold needs to be configured by the user. The value of this threshold is most likely unknown and setting a wrong threshold will cause the fuzzer to not detect any sensor state changes. For this reason, the fuzzer has an option to not log sensor changes but raw sensor values, which allow the log files to be filtered after fuzzing with an appropriate threshold value.

While using the *brute-force fuzzing* method we observed that the response to a message could be delayed by a certain period. During this period, other messages might have been sent by the fuzzer, which makes identifying the responsible CAN messages more difficult. One possible solution is to increase the delay between sending messages. However, this will increase the time required to enumerate the address space. Another option is the resend a portion of the messages preceding the sensor activation at a slower pace, which is what the *identify fuzzing* method does.

4.2.2 Identify Fuzzing

The *identify fuzzing* method is used when reverse engineering an ECU, it can identify the individual CAN message responsible for a change in the state of a single sensor. The method works by replaying recorded CAN traffic while observing the sensor for any state changes. This means a recording of CAN traffic, which is known to cause the state change, is needed. This recording can be obtained using the dump module provided by the Caring Caribou framework to record all messages on the CAN bus while the signal is manually triggered. If such a recording is not available, then this method cannot be used. As only a single sensor is used, this fuzzing method does not need an oracle, the activation of the sensor tells the fuzzer whether the target change is observed.

Input Generation

The *identify method* will try to find the single CAN message responsible for triggering the signal by repeatedly replaying parts of the recorded CAN traffic. This is done as there might be some delay between sending the message and the signal being observed, in which case just attributing the change to the last message sent will not work. To remedy this, the traffic is split into two parts, which are replayed after each other while the sensor is monitored. The *identify fuzzing* algorithm is listed in listing 4.2, the `identify_fuzz` method takes two arguments, `messages` which is a list of recorded CAN messages and `delay` which specifies the delay between messages. The algorithm will first split the messages into two equal parts on line 14. After this each part is replayed on the can bus, after sending a message, the fuzzer will wait for the specified delay time using the `wait_for_on` method. During this time, the fuzzer will monitor the sensor harness, when the sensor state changes to on, the fuzzer knows the responsible message must be in that part. If such a state change is observed, the fuzzing will continue with the messages from that part, which will be split until only one message remains.

This version of the *identify method* only works if the signal turns off automatically after a short delay. If the signal does not turn off automatically the fuzzer cannot start replaying other messages, the only option, in this case, is to find both the message that turns the signal on and the message that turns it off. This is done optionally by using the `dual` option of the *identify fuzzing* method, for which the algorithm is given in listing 4.3. When using this method a separate stack for the on and off messages are kept, the `on_stack` and `off_stack`. These are both initialised to contain all messages on line 3. Then on line 6 the method `fuzz_stack` is called with either the `on_stack` or `off_stack` depending on the current state of the sensor. This is done in a loop until both messages are identified.

The `fuzz_stack` method works similarly to the `identify_fuzz` method from listing 4.2. It will split the messages from the top of the stack and replay them until the sensor changes to the target state which is passed in the `target_state` argument. When the state changes the current part is added to the stack, and the function returns, the `identify_fuzz` method will now call it using the other stack

Listing 4.2: Initial *identify fuzzing* method

```
1  func wait_for_on(delay):
2      start_time := time()
3      result := False
4      while time() - start_time < delay:
5          if sensor.state() == ON:
6              result := True
7      end
8      return result
9  end
10
11 func identify_fuzz(messages, delay):
12     if len(messages) == 1:
13         return messages[0]
14     part_a, part_b = split(messages)
15
16     for message in part_a:
17         send_message(message)
18         if wait_for_on(delay):
19             return identify_fuzz(part_a)
20     end
21
22     for message in part_a:
23         send_message(message)
24         if wait_for_on(delay):
25             return identify_fuzz(part_b)
26     end
27
28     return identify_fuzz(messages)
29 end
```

to change the state again. If the top of the stack only contains a single message it will be sent, and the method returns true, indicating that fuzzer has identified the message responsible for the state change.

4.2.3 Omission Fuzzing

While testing the *identify fuzzing* we identified that some ECUs expect certain packets to be sent regularly on the CAN bus. If these packets are not sent the ECU will shut down or stop responding and indicate the failure. This prevents the *identify fuzzing* method from being used as sending the complete traffic log can keep the ECU responsive but sending parts of the recorded traffic will cause the ECU to fault. To prevent this from happening the *omission fuzzing* method was created, this method sends the complete recorded traffic but omits some messages to identify which message causes which changes. Again, as this fuzzer is used for reverse engineering, no bug oracle is used except the state of the sensors.

Input Generation

The method works by first creating a set of all unique arbitration IDs present in the recorded traffic, which is the set `arb_ids` on line 2 in listing 4.4. This set is usually limited as not many different message types are sent on a given CAN bus (on the order of 10-20 unique IDs). The loop on line 9 will then replay the filtered messages for each arbitration ID. The filtered messages contain all messages except those with an arbitration ID that matches the current ID from the loop. As almost all messages are still present in the transmitted traffic, the ECU should not fault.

The arbitration id of the message responsible for the state change can then be found by observing the sensor state while replaying as done on line 14. When the state has not changed after replaying the filtered messages, the responsible message must have been filtered out. If an arbitration ID is omitted that is required to keep the ECU alive, the ECU will fault, which may cause it not to change state anymore. This will make the fuzzer identify that ID as the ID responsible for the state change. To prevent this from happening a blacklist can be passed to the omission fuzzer, any arbitration ID that is known to cause a fault when omitted can be added to this blacklist. Any arbitration ID in the blacklist will not be omitted as they are removed from the filtered messages on line 7.

Disadvantages

As this method sends all recorded traffic multiple times it can be slower than the *identify fuzzing* method, but it is often the only option as the ECU would fault when using the *identify fuzzing* method. The run time can be improved by trimming the recorded traffic only to include the messages that are necessary to cause the observed change and keep the ECU responsive. The number of unique arbitration IDs can also increase the run time as for each ID the traffic needs to be replayed. If there are certain IDs which are known to not interfere with the signal, they can be added to a

Listing 4.3: The dual stack *identify fuzzing* method

```
1  func identify_fuzz(messages, delay):
2      found_on := found_off := False
3      on_stack := off_stack := [messages]
4
5      while not found_on and not found_off:
6          if sensor.status == ON:
7              found_on = fuzz_stack(on_stack, delay, ON)
8          else:
9              found_off = fuzz_stack(off_stack, delay, OFF)
10     end
11     return on_stack.end(), off_stack.end()
12 end
13
14 func fuzz_stack(stack, delay, target_state):
15     messages := stack.end()
16     if len(messages) == 1:
17         send_message(messages[0])
18         return True
19     part_a, part_b := messages.split()
20
21     for message in part_a:
22         send_message(message)
23         if wait_for_state(delay, target_state):
24             stack.append(part_a)
25             return False
26     end
27
28     for message in part_a:
29         send_message(message)
30         if wait_for_sate(delay, target_state):
31             stack.append(part_b)
32             return False
33     end
34
35     return False
36 end
```

Listing 4.4: The *omission fuzzing* method

```
1  func omission_fuzz(messages, black_list, delay):
2      arb_ids := set()
3      for msg in messages:
4          arb_ids.add(msg.arb_id)
5      end
6
7      arb_ids.remove(black_list)
8
9      for id in arb_ids:
10         filtered_msgs := messages.filter(id)
11         state_changed := False
12         for msg in filtered_msgs:
13             send_message(msg)
14             if wait_for_change(delay):
15                 state_change = True
16         end
17
18         if not state_changed:
19             return id
20     end
21
22     return None
23 end
```

blacklist that can be provided to the fuzzer. All messages with an arbitration ID in the blacklist will not be omitted during fuzzing.

4.2.4 Mutation Based Fuzzing

The *mutation fuzzing* method will send random small variations of a known packet to the CAN bus while observing the sensors. This method can be used after identifying a packet using the other techniques, to reverse engineer the exact meaning of the payload. It can also be used to test an ECU if a specification is known to verify that no other packets can cause the specified observable changes.

Input Generation

The *mutation fuzzing* method will mutate a packet that has a known effect, and observe if the mutated packet still causes the effect. The mutation can be done in several ways, of which the most straightforward method is by flipping bits in the packet. Flipping bits works very well when each bit in the payload signifies a flag. In this case the fuzzer can identify the meaning of each bit flag. When working with

Listing 4.5: The *mutation fuzzing* method

```
1  func mutation_fuzz(message, start_bit, end_bit, delay):
2      for bit in range(start_bit, end_bit):
3          copy := message
4          copy.data[bit] = not message.data[bit]
5
6          send_message(copy)
7          log_message(copy)
8
9          delay_log(delay)
10     end
11 end
```

a payload that contains a number (for example the speed value for the speedometer) the fuzzer can identify which bits in the packet represent the number. If the *mutation fuzzing* method is used to test an ECU, the bits that should not cause any changes can be flipped to detect whether there are any implementation errors.

A simple version of the bit flipping mutation method is given in listing 4.5. This method will flip each bit in the user-defined bit range consecutively and then logs and sends the message. At the same time, the method uses the `delay_log` method to log any sensor state changes to a file. If all bits of the eight-byte payload are flipped, then eight messages will be sent. If the delay is too short then just as with the *brute-force fuzzing* method the response can lag behind the right message. To remedy this, either the delay can be increased, or the *identify fuzzing* method can be used. Increasing the delay is more appropriate in this case as only eight messages are sent, which allows a considerable delay while keeping the total run-time small.

Bug Oracle

When using the *mutation fuzzing* method to test ECUs, the output log file needs to be verified by a specification. This verification would ensure that any activation happened in accordance with the specification. During this thesis, we did not have access to an actual specification but only to the J1939 PGN documentation. The truck instrument cluster uses the J1939 standard for most indicators. This documentation detailed which J1939 packets represent which values, allowing us to test whether other packets caused unspecified changes to the instrument cluster. Another possibility is to use a valid case oracle just as when using the brute-force method. The fuzzer will then periodically send out the known message to verify that the ECU is still responding.

4.3 Conclusion

This chapter presents four fuzzing methods that have been added to the Caring Caribou car exploitation framework. The methods are developed specifically to minimise user interaction by using the sensor harness from chapter 3. The *identify*, *omission*, and *brute-force fuzzing* methods are used to reverse engineer which packets cause which state changes as observed by the sensors. The *mutation fuzzing* and *brute-force* methods provide a way to either reverse engineer a specific packet or to verify a packet specification. These methods will be used in the next three chapters to reverse engineer and test various ECUs.

The methods are implemented in a separate Caring Caribou module as adapting the existing fuzzing module would require rewriting most of it. The methods all use the same log format which allows writing more complex tests by chaining several methods together. The output of the brute-force method can, for example, be used by the identify methods in order to identify one specific packet, after which the mutation fuzzer can reverse engineer the meaning of the payload of the identified packet.

Chapter 5

Fuzzing ICSim

As a first step in developing and testing the fuzzing module a virtual CAN bus and instrument cluster are used. This allows the framework to be developed and tested without requiring any automotive hardware. The virtual CAN bus is part of the standard *CAN-utils* Linux package and the virtual instrument cluster used is *ICSim*.

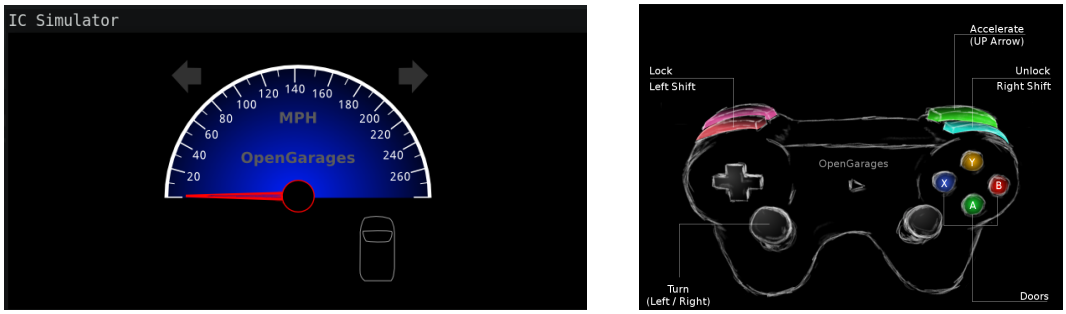
5.1 ICSim

ICSim is a virtual instrument cluster developed by Craig Smith as part of the Open Garages project [36]. It is intended to be used as a tool for learning to reverse engineer CAN messages. The ICSim project consists of two applications written in the C programming language, a virtual dashboard as seen in figure 5.1a and a controller application is shown in figure 5.1b. These two applications communicate over a virtual socket CAN interface which is part of the default *CAN-utils* Linux package. The virtual interface allows multiple applications to monitor it. This allows the fuzzer to both receive and transmit CAN messages to the dashboard while the controller is running.

The virtual dashboard supports three different CAN messages each with its own arbitration ID:

- Door message: This message has a bit for each door which indicates whether it is locked or not.
- Signal message: This message has two bits, one for the left turn signal indicator and one for the right turn indicator.
- Speed message: This message contains a 16 bit integer representing the speed in ten meters per hour.

The specific arbitration ID of these messages can be randomised to increase the difficulty while reverse engineering. The randomisation option will not only randomise the arbitration ID but also the position of the various flags within a message. In addition to the randomisation option, there are two optional difficulty levels, at level



(a) The ICSim dashboard.

(b) The ICSim controller.

Figure 5.1: The ICSim virtual instrument cluster.

one, the length of the CAN messages are randomised, and at level two, the contents of the longer packets will be randomised.

The controller application will get user input from either the keyboard or an attached game-pad. It will then update its internal state according to this input and send this updated state to the dashboard application via the virtual CAN bus. This update loop runs at about 100Hz, which means any state updates injected by the fuzzer will be overwritten by the controller in at most five milliseconds. The controller also has an option to increase the difficulty of reverse engineering by injecting bogus traffic into the CAN bus. This traffic is read from a file and injected into the CAN bus, making the traffic consist of more than three different messages.

5.2 Test Setup

The sensor harness and fuzzer can be tested using ICSim without requiring any automotive hardware. This is done by attaching the light sensors from the harness to the computer screen on which the virtual dashboard is running. The setup can be seen in figure 5.2. The placement of the light sensors is important as the indicators are very small on the screen. For this reason, a logger utility was made that simply outputs the current detected colour from a light sensor, this allows the sensor to be placed while monitoring the current colour ensuring it can detect the right indicator.

5.3 Results

In this section each of the fuzzing methods discussed in chapter 4 will be applied to ICSim. The various methods can be run with and without the controller running, without the controller running there will be no interference and fuzzing will be easier. But with the controller running the state of the dashboard will reset itself, which is useful for the *identify fuzzing* method.

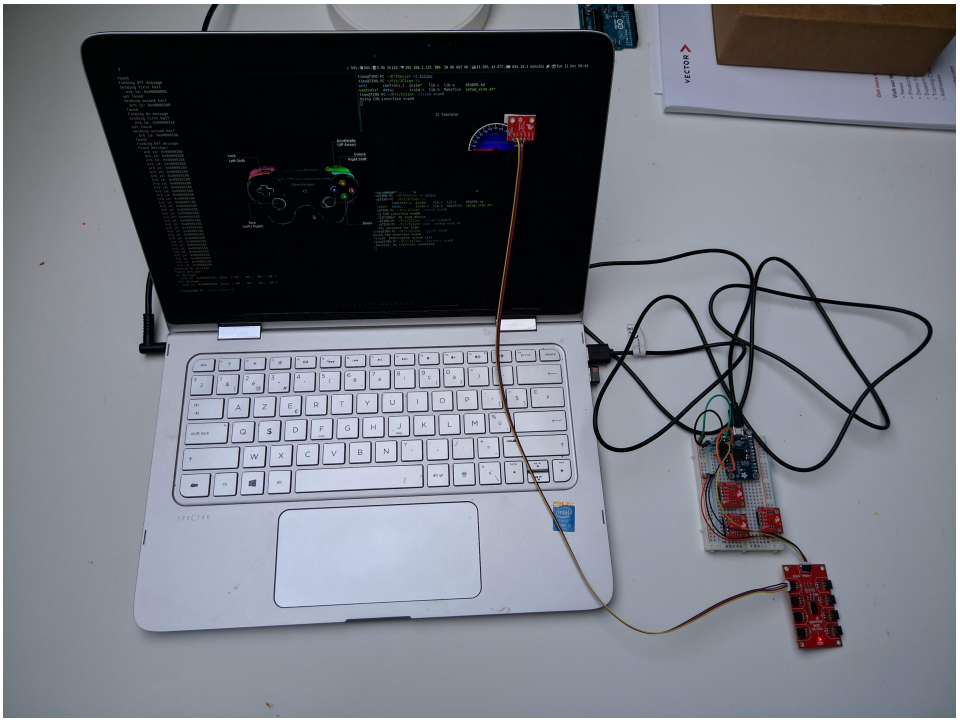


Figure 5.2: The setup for testing ICSim.

5.3.1 Brute-force Fuzzing

After placing the sensors over the right indicators and calibrating them by triggering them using the controller application the *brute-force fuzzing* method can be run. The fuzzer correctly detects all indicators (turn signals, speedometer and door unlock indicators) when using the right payload. A payload of eight bytes filled with ones will cause the turn signal indicators to turn on and will cause the speedometer to move to its maximum value. To trigger the door unlock indicators a payload consisting of all zeros is needed, this is because the doors are locked by default and will unlock with a zero bit in the right position.

By using the valid case bug oracle, the fuzzer can check whether the ICSim dashboard is still responsive. Unfortunately, we found no bugs while using the *brute-force* fuzzing method. However, we manually inserted bugs into the source code of ICSim. These bugs were successfully triggered by using the brute-force method, and the valid case oracle successfully identified that the ECU stopped responding.

The time it takes to enumerate the 11-bit address space depends on the delay between messages. With a delay of 10ms, it takes about 30 seconds and with a delay of 1ms, it takes about 15 seconds. When analysing the logs from the fuzzer it became clear that an indicator responds about 50ms after the CAN message was sent. Thus to identify which message is responsible for which indicator the messages preceding the activation will need to be replayed with a delay longer than 50ms. Another option is to use the *identify fuzzing* method which will automatically isolate

the responsible message.

5.3.2 Identify Fuzzing

The *identify fuzzing* method can be applied to ICSim after recording some traffic from the virtual CAN bus between the controller and dashboard application. This recording needs to contain the trigger for the indicator that needs to be identified, this can be done by simply pressing the relevant button in the controller application. After collecting the CAN traffic the controller can either be shut down when fuzzing, which will require the fuzzer to recover both the on and off messages. Or the controller can be left running, in which case the controller will interfere with the traffic sent by the fuzzer. This interference will cause the indicators to turn off automatically, allowing the fuzzer to only identify the message that turns an indicator on.

All three CAN messages were identified correctly by the *identify fuzzing* method with the controller running. The controller will update the state at about 100Hz, so any changes that the fuzzer causes will reset after about 10ms. If the activation of an indicator is too short, the fuzzer will miss it, which causes it to retry with the same messages. This interference is reflected in the large variation in time required to identify a message, which is summarised for the turn signals in table 5.1. In this table, it can be seen that the time to identify increases when more messages are present in the recording. The difficulty options of ICSim, such as randomisation and injecting bogus traffic, do not have a significant impact on the *identify fuzzing* method. Injecting bogus traffic will cause more messages to be present on the CAN bus, which will increase the identification time.

When the controller is not running, all three messages were also identified correctly, although requiring a bit more time on average as seen in table 5.2. As the indicators will not turn themselves off in this case, the *identify fuzzing* method is run with the dual option. Without interference from the controller application, the time required to identify a message is much more consistent. This is due to the indicators staying on as long as the fuzzer has not sent the message to turn it off, which means no messages need to be resent.

Another use for the *identify fuzzing* method is to run it directly after running the *brute-force fuzzing* method. The *identify fuzzing* method can then identify exactly which message caused the observed state change during *brute-force fuzzing*. To use this method the controller needs to be running as otherwise, the indicators will not turn off, the *brute-force fuzzing* method will often not produce a message that turns a signal off. Using the combination of the *brute-force fuzzing* method and the *identify fuzzing* method the message responsible for the turn signal could be identified in 40 seconds. This is faster than using the *identify fuzzing* method on its own as only the five messages preceding a state change are passed to the *identify fuzzing* method. Identifying from five messages takes about 10 seconds and enumerating the 11-bit address space using the *brute-force fuzzing* method takes 30 seconds.

The *brute-force fuzzing* method can also be used with the *identify fuzzing* method and without the controller running. In this case, two payloads need to be passed to the *brute-force fuzzing* method, one that will turn the indicator on (such as all

	Time (s)		
# messages	avg	min	max
1047	50	36	87
2076	69	36	104
4146	79	48	146
8292	81	59	122

Table 5.1: Time to identify the turn signal message with the controller running.

	Time (s)		
# messages	avg	min	max
1047	74	74	74
2076	71	71	71
4146	80	80	80
8292	93	92	93

Table 5.2: Time to identify the turn signal message without the controller running.

ones) and one that will turn it off (such as all zeros). When the *brute-force fuzzing* method detects a state change, it will pass the preceding messages to the *identify fuzzing* method with the dual option. The *identify fuzzing* method will then be able to identify both messages. As this method requires double the amount of messages to be sent and identified, it takes about double the amount of time, 100 seconds in case of the turn signal indicators.

5.3.3 Omission Fuzzing

The *omission fuzzing* method is not necessary in the case of ICSim as it will keep working even if it does not receive any CAN messages. Nevertheless, the method was still tested on ICSim to verify the implementation and record some timing benchmarks. The traffic recorded from the controller with bogus traffic enabled contains 36 different arbitration ids. This means in the worst case the complete recording will need to be replayed 36 times.

When applying the *omission fuzzing* method to identify the message responsible for the turn signals it can identify that message in 80 seconds. This is so fast because the arbitration ID of the message was the third omitted ID, which means the traffic only had to be replayed three times. When using it to identify the speed message it took about 40 minutes as the arbitration ID of the speed message was one of the last IDs omitted.

These results show how much worse the *omission fuzzing* method can perform than the *identify fuzzing* method. Using the *omission fuzzing* method should only be done when necessary such as in chapter 7. The fact that the fuzzer will replay

all recorded traffic several times means that trimming the recorded traffic is very important, if possible. When disabling bogus traffic the number of arbitration IDs in the recorded traffic drops to three, allowing the *omission fuzzing* method to recover the speed message in just 15 seconds.

5.3.4 Mutation Fuzzing

After using *identify* or *omission fuzzing* to identify which CAN message is responsible for which indicator, the *mutation fuzzing* method can be used. The *mutation fuzzing* method will identify which bits in the payload of the message are responsible for activating the indicator. The simple bit flipping mutator can recover which bits cause the left and right turn signal indicator to change. By supplying the fuzzer with a message containing all zeros it will flip each bit consecutively, which will sometimes trigger the turn signals. The door unlock indicators can also be recovered using this method in exactly the same way. The speed is more difficult as it is not a binary signal but a 16-bit number. By flipping bits in this number, the speedometer needle will move, sometimes triggering the sensor. The least significant bits, however, do not change the speed enough to trigger the sensor. The fact that the needle moves instantly also makes it difficult to detect any movement, this is not the case in a physical dashboard as in chapter 6.

5.4 Conclusion

In this chapter the fuzzing tools developed in chapter 4 and sensor harness developed in chapter 3 were applied to the ICSim simulator. Using these tools the various CAN messages used by the simulator were successfully reverse-engineered. Both the *identify* and *omission fuzzing* methods can use the traffic generated by the controller application to identify which message triggers which indicator on the dashboard. Although the *omission fuzzing* method can require significantly more time to identify a message. When no controller and thus no recorded traffic is available the random fuzzing method can be used to successfully identify all CAN messages. When used in combination with the *identify method* it can also recover which individual message triggers which indicator. The *mutation fuzzing* method was also applied and can recover the meaning of the various bits in two of the three CAN messages supported by the dashboard application. These successful tests using the virtual dashboard were a good first step to test the effectiveness of fuzzing the CAN network. These methods will be applied to a physical dashboard in chapter 6.

Chapter 6

Fuzzing an Instrument Cluster

After testing the fuzzer and sensor harness on a virtual instrument cluster in chapter 5, we applied the fuzzer to a real instrument cluster taken from a consumer vehicle. As this instrument cluster is disconnected from the vehicle there are no other ECUs connected on the CAN bus. This means that there is no traffic on the bus that can be recorded and replayed by the identify or *omission fuzzing* methods. So at first only the *brute-force* and *mutation fuzzing* methods were used to reverse engineer the various CAN messages. After identifying a number of messages we applied the various other fuzzing methods by writing a controller in software. This allowed us to test the identify and *omission fuzzing* methods on the instrument cluster.

6.1 The Instrument Cluster

The instrument cluster is fairly modern, it was produced in 2013 and contains a multi-functional monochrome LCD. The dashboard consists of two dials (one for vehicle speed and one for motor RPM) behind which various status indicators are located as seen in figure 6.1. Between the two dials, there is an LCD screen which contains various information such as fuel level, distance travelled and temperature. The cluster also contains two buttons, one to navigate the LCD screen's menu and one to reset the odometer. On the back, there is a single 32-pin connector which is used to provide power and exposes the CAN bus lines. The various other pins are used for functions that do not work over the CAN bus such as the immobiliser, oil sensors and fuel level sensor.

6.2 Test Setup

The test setup is similar to the setup with ICSim from chapter 5. A major difference is that the USBtin adapter used to connect the physical CAN bus to the PC. The sensors are placed on the dashboard in strategic locations. The placement is important as a transparent panel is placed over the dashboard, preventing the sensor from being placed close to the indicators. This makes calibration extra important as neighbouring indicators can trigger a sensor unintentionally. An example setup

Figure 6.1: The instrument cluster just after boot-up, the indicators that are lit-up are on by default after boot-up

can be seen in figure 6.2. In this setup four sensors are used, one over the left turn indicator, one over the low-oil indicator, one over the steering lock indicator and one over the handbrake indicator. Because the light sensors cannot differentiate the various changes of the text displayed on the LCD screen, it is not used for fuzzing.

To test the working of the CAN bus, we wrote a controller application that would send various fixed CAN messages to the dashboard. The CAN messages were adapted from a project that used a similar instrument cluster model as a display for a racing simulator [6]. This controller allowed us to test various indicators such as the turn signal indicators and speedometer. When using this controller application we observed several interesting behaviours, such as that the dashboard lights up as soon as it receives traffic. When the dashboard does not receive any traffic, the back-lights turn off and it will go in a kind of sleep mode. Sending any valid CAN message will cause it to respond and wake back up. Some indicators, however, did not respond to the messages specified in the racing simulator project. By using the fuzzer we were able to correct the arbitration ID of these messages and update the controller application.

6.3 Results

This section gives an overview of the results of applying all fuzzing methods to the dashboard. First, the *identify fuzzing* method was applied by recording traffic of the controller application. After this, some missing indicators were identified by using the brute force-fuzzing method in combination with the *mutation fuzzing* method. Finally, the *omission fuzzing* method, while not necessary, was applied to the dashboard. The time required to identify various indicators is summarised in table 6.1.

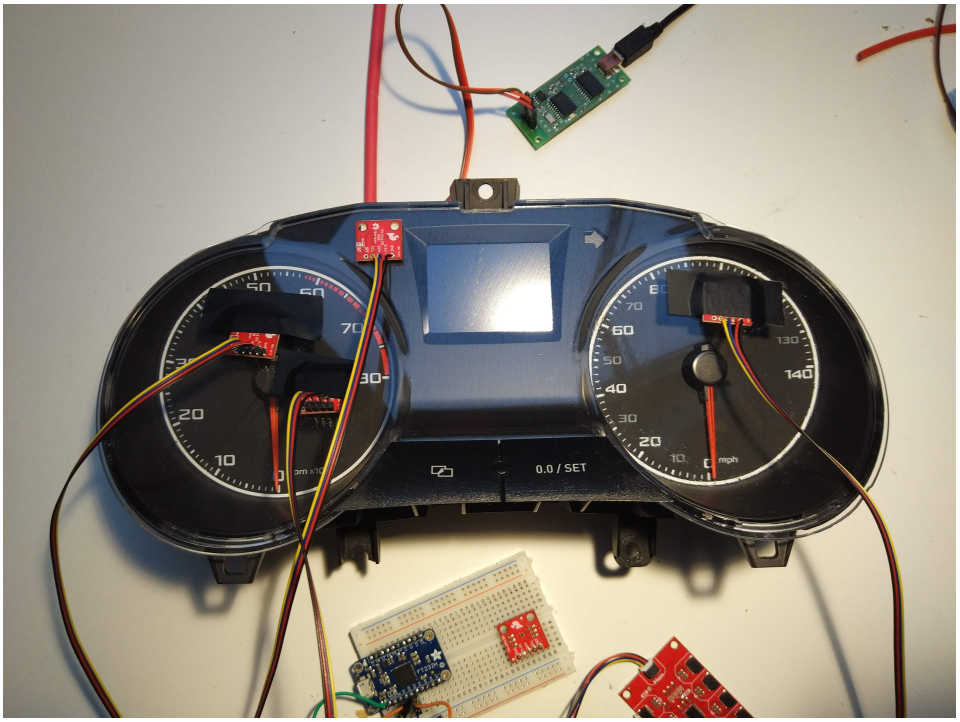


Figure 6.2: The instrument cluster under test, with sensors attached over various indicators. The USBtin adaptor connected to the CAN bus can be seen in the top of the figure.

6.3.1 Identify Fuzzing

By using the controller application adapted from the racing simulator project we can apply the *identify fuzzing* method. First, the traffic generated by the controller application is recorded. After this, the traffic can be replayed by the *identify fuzzing* method either with or without the controller application running. Using this method the various trigger messages (such as the turn signal indicators) were successfully identified. Of course, the messages that were not correctly implemented by the controller could not be identified.

Without the controller running, the dual method of the *identify fuzzing* method needs to be used. However, the *identify fuzzing* method has difficulty identifying messages that turn indicators off. This is caused by the dashboard going into the sleep mode when not receiving valid messages for a certain period. In this sleep mode, the dashboard turns off certain indicators such as the turn signals, causing the fuzzer to attribute the off message to a random unrelated message. This could be prevented by decreasing the time spent waiting after sending messages, but this causes some indicator activations to be incorrectly attributed. Another solution is to simply not use the dual method, as the indicators turn themselves off when the dashboard goes to sleep. This correctly identifies all on messages, taking about 41 seconds to identify the turn signal indicator for example.

With the controller application running the dashboard is simulated as if it was still integrated into a vehicle. The *identify fuzzing* method does still work but has a harder time as the controller interferes with the messages sent by the fuzzer. These messages, just as in chapter 5, cause some signal activations to be missed. But as opposed to the virtual dashboard, the physical dashboard sometimes does not even activate the indicators when two messages follow each other too closely. As these activations are missed the fuzzer will need to resend some traffic, increasing the time needed to identify a message. The time to identify the turn signal message, for example, is 66 seconds with the controller running.

6.3.2 Brute-force Fuzzing

The *brute-force fuzzing* method was applied to identify the missing messages from the controller. By enumerating the whole 11-bit arbitration ID space with a fixed payload of all-ones most indicators could be activated. Some indicators are on by default on startup, these indicators could still be fuzzed using a fixed payload of all zeros. Even the speedometer moved to its maximum value when fuzzing with an all-ones payload. The engine speed dial, however, did not move with either of those two payloads, probably because the value exceeded its maximum value. When tested again with a payload consisting of alternating ones and zeros, the engine speed dial did move.

By using the *brute-force fuzzing* method in combination with the *identify fuzzing* method, we were able to find the missing CAN messages that differed from the controller application. Most of the differing messages just had a different arbitration ID with the same payload, while others had missing or miss identified fields in the payload. By updating the controller application to use these arbitration IDs, signals that could not be activated before (such as the hand-break indicator) could now be activated. The *identify fuzzing* method could now also be applied to these signals. The fuzzer takes about 30 seconds to enumerate the 11-bit address space using a delay of 10ms between messages. Running the *identify fuzzing* method after finding a state changes takes an extra 30 seconds per activation to identify the message.

We also attempted to find any vulnerabilities which would make the instrument cluster unresponsive. We did this by using the *brute-force fuzzing* method with random payload and using the valid case oracle to detect if the instrument cluster is still responsive. However, after running the fuzzer for several hours, the instrument cluster was still responsive. This indicates that a smarter fuzzing approach would need to be used, possibly based on the specification of the ECU. We, however, did not have access to such a specification.

6.3.3 Mutation Fuzzing

Using the *mutation fuzzing* method on any of the messages identified in the previous two sections, we were able to reverse engineer the meaning of most bits. By for examples starting from the messages that activated the left turn signal indicator, the fuzzer was able to not only identify the bit responsible for the indicator. But it

Fuzzing method	Time to identify (s)
Brute-force, 1 sensor	60
Brute-force, 2 sensors	100
Brute-force, 4 sensors	241
Identify, no controller	41
Identify, controller	66
Omission	50
Mutation, 1 sensor	21
Mutation, 2 sensors	25

Table 6.1: Time to identify some indicators using the various fuzzing methods on the instrument cluster.

was also able to identify the bit responsible for the right turn signal, the headlights indicator and some various other indicators. Enumerating all eight bits took about 20 seconds with a delay of 3ms between messages and identifying each of them required an additional 5 seconds. The delay between messages is critical when using the *mutation fuzzing* method. If the delay is too short then the next message, which will have another bit flipped, will overwrite the previous message. This will cause more indicator activations to be missed or some indicators to not even activate. Just as when fuzzing ICSim, the speedometer and engine speed dial were more difficult. Activation some bits caused them to move and activate the sensors but activating other bits caused them to move too little to activate the sensors.

6.3.4 Omission Fuzzing

The omission based fuzzing method is not necessary for fuzzing the instrument cluster. Although the dashboard will go into a sleep mode when not receiving any messages for some period. It still responds to any correct message it receives, so sending them in a certain order does not matter. Nevertheless, the *omission fuzzing* method can successfully be applied to the instrument cluster. The traffic generated by the controller application is used, this traffic contains 21 unique arbitration IDs. Using the omission method to identify which arbitration ID is responsible for which state changes took 6 to 120 seconds. This variation is mainly dependent on the order of the arbitration IDs that are omitted. The turn signal indicators, for example, took about 50 seconds to be identified, they were the ninth arbitration ID that was omitted.

6.4 Conclusion

This chapter showed how the fuzzing techniques described in chapter 4 can be successfully applied to a physical instrument cluster. The fuzzing techniques can reverse engineer the trigger messages for most indicators, with some difficulties for others (such as the speedometer). To test the *identify* and *omission fuzzing* method,

a controller application was written. The messages sent by this application are a combination of messages taken from a racing simulator project, and the messages reverse engineered by the *brute-force fuzzing* method. Using traffic recorded from this application the *identify* and *omission* (although unnecessary) methods were also applied. The LCD screen present on the dashboard was not used, during fuzzing some messages popped up, but this was not further investigated. A future update to the fuzzer could use computer vision to detect messages on LCD screens. Using the *brute-force* fuzzing method to find vulnerabilities was unsuccessful. This indicates the need for a smarter fuzzing approach for vulnerability discovery.

The instrument cluster still consists of a single ECU, which means the CAN bus only communicates with the instrument cluster. Usually, there are multiple ECUs connected to a single CAN bus. These ECUs will all send competing traffic, which may make fuzzing more difficult. Up till now, only virtual applications running on the PC were used to simulate this extra traffic. The next step to testing the fuzzer will be to use an actual CAN network with multiple ECUs connected to it. This is done in chapter 7 as the truck instrument cluster consists of multiple ECUs.

Chapter 7

Fuzzing a Truck Instrument Cluster

As the last test, the fuzzer was applied to a truck instrument cluster. A European truck manufacturer provided this instrument cluster. As opposed to consumer cars, the truck uses the J1939 standard for all of it's CAN communications. This means the messages for common data such as engine speed and diagnostics are much more standardised. In addition to the standard J1939 messages, the cluster uses some proprietary messages to control the content of the screen and various indicators. By using the fuzzer we hope to reverse engineer these messages. To fuzz this instrument cluster, the fuzzer needs to generate valid J1939 packets. This is especially important when using the *brute-force method*, as this method does not send a prerecorded CAN packet. The instrument cluster consists of three separate ECUs and two CAN buses. By isolating some ECUs from the network the fuzzer can imitate that ECU to fuzz the rest of the network.

7.1 The Provided Instrument Cluster

A European truck manufacturer graciously provided us with a test-bench setup of an instrument cluster from one of their trucks. This instrument cluster consists of three ECUs connected via two CAN buses. The network diagram can be seen in figure 7.1. In addition to the two interconnecting CAN buses, there is an additional diagnostic CAN bus connected to the controller which is not used in our experiments. The controller relays any messages destined for the switches or dashboard on this diagnostic CAN bus to the CAN-2 bus.

The specific functions of these three ECUs are:

- **Controller** - The controller is the central ECU in the cluster. It connects to most CAN buses and relays information between them. It also connects directly to the fuel and Ad Blue level sensors and relays this information to the dashboard via the CAN-2 bus. The turn direction indicators are also relayed from a separate CAN bus to the CAN-2 bus by the controller. In the test-bench

setup, it runs a testing application which will emulate the rest of the vehicle with some defaults values.

- **Dashboard** - The dashboard displays the information it receives from the controller. It consists of a central colour LCD screen with a dial for the vehicle speed to the left and a dial for engine speed on the right. In addition to the central LCD, three smaller monochrome LCDs display additional information such as the current time, temperature, and distance driven. The menus on the colour LCD screen can be used to view information about the vehicle and change various settings. These menus are navigated using a rotary joystick which is attached directly to the controller.

- **Switches** - The switches are normally connected to the steering wheel, and allow the driver to navigate some menus without taking his hands away from the steering wheel. The switches contain twelve buttons, six on each side of the steering wheel. These can be used to increase and decrease the media volume, answer a call or change the cruise control driving distance. The ECU embedded in these switches will continuously send the state of these buttons to the dashboard via the CAN-2 bus.

The first CAN bus (CAN-1) is used throughout the vehicle for the most critical information. This bus carries critical information such as engine speed and engine temperature, and uses standard J1939 messages. The second CAN bus (CAN-2) is mainly used for communicating less important status information from the controller to the dashboard and from the switches to the dashboard. This information can be the current time or the display information for the current menu on the main LCD. The messages on the second bus are mainly using proprietary J1939 messages, which is useful for reverse engineering. The dashboard is also connected to the CAN-1 because when the controller faults, it can still display important information such as engine and vehicle speed.

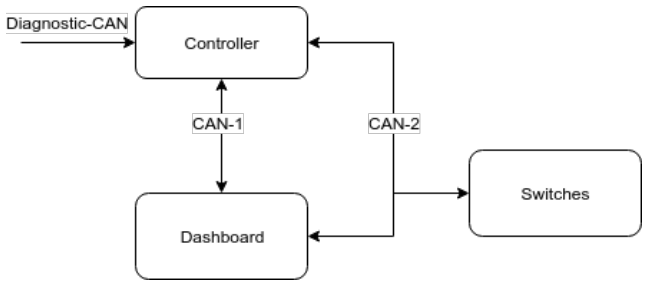


Figure 7.1: The network diagram of the instrument cluster. The first CAN bus connects the dashboard to the controller and the second CAN bus connects all three components.

7.2 Test Setup

Just as in chapter 6, the setup consists of the sensor harness taped to the dashboard in strategic places and the USBtin connected to one of the two CAN buses. In addition to the usual LED indicators, the colour LCD screen can also be used for fuzzing. This is due to its large size and the fact that changing a menu causes a significant change in the displayed colour. This allows the fuzzer to detect when a new menu screen is displayed. A choice needs to be made as to which CAN bus will be fuzzed as only one CAN bus can be fuzzed at a time. The CAN-2 bus is the most interesting to fuzz as it uses proprietary messages. The CAN-1 bus can, nevertheless, still be fuzzed to verify the response to standard J1939 messages and to try and find invalid messages to which the dashboard still responds.

In order to make fuzzing easier, some ECUs may be disconnected from the CAN bus to either let the fuzzer emulate its traffic or to record its traffic in isolation. When the dashboard is disconnected from the CAN-2 bus, it will enter a fault state. In this state, it will instruct the driver to stop on all of its LCD screens. Some traffic from the controller to the dashboard can be recorded to prevent the dashboard from entering this fault state. By replaying this traffic shortly before disconnecting it from the CAN bus it will respond to the recorded traffic, and then enter the fault state after the traffic ends. However, only sending a part of the traffic will cause the dashboard to enter the fault state, meaning that the *identify fuzzing* method can not be used on this bus. Instead, the *omission fuzzing* method was used as this method will replay the complete recorded traffic log. Disconnecting the switches from the CAN-2 bus will not cause any faults, allowing us to record its traffic in isolation and replay it.

7.3 Results

This section discusses the results of applying the fuzzer to the truck instrument cluster. These results are all concerning reverse engineering, as there were no vulnerabilities identified by the fuzzer. First, the *identify fuzzing* method is applied to the switches and the dashboard. However, the *identify fuzzing* method has difficulties when working with the dashboard, so the *omission fuzzing* method is applied. Next, the *brute-force fuzzing* method is used to reverse engineer the proprietary J1939 messages. Finally, the *mutation fuzzing* method is used to reverse engineer some proprietary messages. Table 7.1 contains an overview of the time needed to identify some messages using the various fuzzing methods.

7.3.1 Identify Fuzzing

As a first step to test the fuzzer when using multiple ECUs, the messages sent by the switches were recorded. When replaying the messages, the dashboard responded correctly, indicating that the *identify fuzzing* method could be used. The method also does not need to be adapted to the J1939 standard, as the recorded messages are all J1939 messages meaning replaying them without modification still works.

One button on the switches caused a warning message to be displayed on the colour LCD screen. This message disappears after a second, making it a prime candidate for using *identify fuzzing*. By attaching a colour sensor to the LCD screen and using the recorded traffic with the *identify fuzzing* method, the message that sends the button states to the controller was successfully identified. The J1939 PGN of the message was, as expected, in the range of proprietary group numbers.

The next step was to try and use the same method on the communication of the LCD state between the controller and the dashboard. When we replayed the traffic recorded from the controller, however, the dashboard did not respond. This is probably due to the fact that the controller sends the state updates for the dashboard on a fixed interval. When replaying the traffic while the controller is still connected, our traffic is ignored by the dashboard. However, when we disconnect the controller from the dashboard and start playing the recorded traffic at the same time, the dashboard does respond. Nonetheless, applying the identify method with this traffic and the controller disconnected causes the dashboard to enter its fault mode. It enters this fault mode as soon as only a part of the traffic is replayed, indicating that some messages are critical to keeping the dashboard functioning. As a result, the omission method must be used to reverse engineer the traffic between the controller and the dashboard.

7.3.2 Omission Fuzzing

The *omission fuzzing* method was used to prevent the dashboard from faulting. The recorded traffic only contained 16 unique arbitration IDs and 1567 messages. Replaying the traffic once takes about six seconds, meaning the *omission fuzzing* method can take up to 96 seconds. Using the omission method, we successfully identified the message responsible for sending the current colour LCD state to the dashboard. Additionally, when this message is left out of the traffic, the dashboard enters its fault state. Meaning the reason for using the *omission fuzzing* method was inherent to the message we were trying to reverse engineer. Moreover, the *omission fuzzing* method needed to be stopped after leaving out this arbitration ID, as it was not able to recover the dashboard by sending the traffic again.

To prevent the dashboard from faulting, the identified arbitration ID was added to the blacklist of the *omission fuzzing* method. By doing this, we were able to identify some additional messages. These include the messages for each of the three monochrome LCD screens and the messages that update the fuel level indicators. Some messages, however, are impossible to reverse engineer using the omission or identify method, as the controller does not send them. This includes the turn signal indicator messages, which are sent by a separate ECU to the controller over a different CAN bus not included in the test-bench. The controller will then relay this message to the dashboard over the CAN-2 bus. The *brute-force fuzzing* must be used to reverse-engineer this message.

7.3.3 Brute-force Fuzzing

As opposed to the omission and identify methods, the brute-force method does need to be adapted to the J1939 standard. This is because J1939 messages have a sender field. The sender is an 8-bit identifier of the ECU application that sent the message. In our tests, the sender is checked by the ECUs, and when it does not match the expected sender the message is ignored. To successfully use the brute-force method, it not only needs to enumerate the J1939 PGN but also the 256 senders. Luckily the input space is still small as the PGN is known to be proprietary. There are only 256 proprietary broadcast PGNs and one peer-to-peer proprietary PGN. This peer-to-peer PGN has an additional 8-bit receiver field that also would need to be enumerated if the identifier of the receiver was unknown. For our purposes, however, this PGN was already excluded as it is used by the message which updates the colour LCD identified in the previous section. This means the search space of the *brute-force fuzzing* method is increased from 11-bits to 16-bits when using the J1939 protocol.

To identify the proprietary turn signal message, some payloads were determined that would likely cause the blinkers to activate. The *brute-force fuzzing* method was then used to enumerate all proprietary broadcast PGNs and all 8-bit sender addresses. We visually verified that the turn signals are indeed activated, but in order for the fuzzer to detect them some calibration difficulties needed to be overcome. As the default calibration with an on and off state measurement cannot be done, the threshold calibration method was used. The threshold calibration method is very prone to errors due to changes in ambient lighting. For this reason, the fuzzing was very unreliable. Nevertheless, the message was successfully identified after several attempts, taking about 16 minutes when calibration is successful. The duration depends, of course, heavily on the number of different payloads, the order in which the senders are enumerated and the delay between sending messages.

7.3.4 Mutation Fuzzing

The payload of the identified messages from the previous sections can now be reverse engineered using the *mutation fuzzing* method. The first message tried was the switch state message, by mutating this message the bits responsible for each button was found. Although the sensors could not discern every button activation as some menus that were activated by the buttons had the same colours. A possible future addition could use computer vision to differentiate these menus. The turn signal indicator message was also successfully reverse engineered, after identifying it with the *brute-force fuzzing* method. This message has a bit for the left and right turn signals for both the tractor and trailer, which was unknown to us until we used the *mutation fuzzing* method.

The more complicated display messages between the controller and dashboard were more challenging to reverse-engineer using the *mutation fuzzing* method. The first problem with these messages is that they control the complicated menus on the colour LCD screen. These menus are difficult to differentiate with colour sensors, so again some kind of computer vision would be required to reverse engineer them

Fuzzing method	Time to identify (s)
Brute-force, 1 sensor	972
Identify, no controller	72
Identify, controller	85
Omission	96
Mutation, 1 sensor	23
Mutation, 2 sensors	27

Table 7.1: Time to identify some indicators using the various fuzzing methods on the truck instrument cluster.

successfully. The second problem is that these messages do not contain simple bit-fields. Thus randomly or consecutively flipping bits does not always lead to valid messages. A more intelligent mutation strategy would be needed actually to enumerate the input space efficiently. Lastly, the dashboard only expects these messages on a certain interval. Thus sending these mutated messages from the fuzzer with the controller still connected does not work as the dashboard ignores the injected messages. Also, disconnecting the dashboard requires very accurate emulation of the traffic that the dashboard expects from the controller. If the timing of the messages is off, or an invalid message is sent the dashboard will enter a fault state. Reconnecting the controller will get the dashboard out of the fault state. These problems make automated fuzzing of these messages difficult and currently unsuccessful.

7.4 Conclusion

In this chapter, the fuzzing framework was successfully used to reverse engineer a dashboard embedded in an instrument cluster network. The fact that the dashboard was connected to multiple ECUs posed some problems but also helped in some fuzzing applications. The ability to isolate some ECUs and record their traffic separately greatly aided when using *identify fuzzing*. However, the fact that some communication between the controller and dashboard was critical to keep the dashboard operating posed some problems when trying to apply the *identify fuzzing* method. To overcome these problems the *omission fuzzing* method was used. This allowed us to identify these critical messages successfully.

The instrument cluster uses J1939 for all its communications. This increases the search address space for the *brute-force fuzzing* method from 11 bits to 16 bits (8 bits for the address and 8 bits for the PGN), increasing the enumeration time from about 30 seconds to 16 minutes. Nevertheless, the brute-force method was used successfully to identify the proprietary J1939 messages which communicate the turn signal state. The *brute-force fuzzing* method was the only method which needed to be adapted to the J1939 protocol. All other methods either replay recorded messages or modify the payload, both of which result in valid J1939 packets.

Another problem when using the sensors on the modern dashboard was the colour LCD screen. The sensors can differentiate between some menus, which was useful when using the *identify fuzzing* method on the switches ECU. However, some other menus were not differentiable by the colour sensors. This made applying the *mutation fuzzing* method on the communication between the controller and dashboard difficult. A possible future extension to the fuzzing framework would be to use some kind of computer vision approach in addition to the sensor harness. This would allow the fuzzer to use a simple camera to not only detect changes on the LCD but also detect most other indicators.

Chapter 8

Related Work

Fuzzing has a lot of published work, mainly in the area of software fuzzing. In [33], a general introduction to fuzzing is given. Recently, in [25], an overview of the various fuzzing techniques is given. The paper also classifies the most commonly used fuzzers according to their own taxonomy, which is useful when selecting a fuzzer to work on a new project.

There is some interesting work being done on fuzzing protocols. The *SNOOZE* fuzzer [5] allows a tester to specify protocol states and the required messages in these states. The fuzzer will then automatically fuzz the protocol according to these definitions. The *AutoFuzz* fuzzer [16] tries to automatically construct a finite state machine (FSM) from recorded network traffic. It then uses this FSM to guide the fuzzing, but it still requires user-defined generalisation functions which are protocol specific. As noted by *BBuzz* [8], *AutoFuzz* assumes that the protocol is character-based, byte-aligned, and it is mainly focused on application layer protocols. *BBuzz* solves these problems by using a bit aware fuzzer which can fuzz network protocols. It uses limited feedback and is also more geared towards IP protocols. Nevertheless, it might be useful to integrate the *AutoFuzz* and *BBuzz* methods in the Caring Caribou fuzzing framework. The methods used by these fuzzers can be useful to fuzz protocols implemented on top of CAN.

Automotive security is also an active research topic as vehicles are becoming more interconnected, and the cyber-security of these vehicles becomes more and more critical. Recent work exposes many critical security vulnerabilities in several different vehicles. The work by Dr Charlie Miller and Chris Valasek [27] exposed a set of vulnerabilities in a 2014 Jeep Cherokee which allowed them to control most of the vehicle's systems remotely. A recent master thesis [21] explores the security of the AUTOSAR standard based on top of CAN by using data flow analysis. The data-flow within a vehicle is modelled using a computer program, which will then generate a threat report. This analysis found the same vulnerabilities used in this thesis, such as the spoofing of other ECUs by sending CAN packets with the right arbitration ID. A second thesis [42] applies various threat modelling techniques on the CAN protocol and again identifies the same vulnerabilities which allow any node to send any packet on the network.

In the Car Hacker's Handbook [37], Smith states that fuzzing has limited applicability on automotive control networks. This is reiterated in recent work [29] which states that fuzzing embedded systems poses certain difficulties not encountered when fuzzing applications. Nevertheless, there have been papers published which apply fuzzing to these networks. Some work applies fuzzing to automotive networks without any feedback ([22], [31]) or with limited feedback directly from the CAN bus [7]. All these works are successful in either triggering some response from the vehicle (such as indicator lights changing) or in finding vulnerabilities in the applications running on the ECUs. The work in this thesis extends that work by using a sensor harness for immediate feedback, this allows a large part of the manual work to be automated as shown in chapters 5, 6 and 7. There is also a proposal [14] to use the DBC files describing the CAN messages in vehicular networks as a base to generate fuzzing test cases automatically. This is a potential future addition to the fuzzing framework, which would be useful when fuzzing from a specification. In [10] the timing of messages is used to identify the ECUs present on a CAN network. Identifying ECUs can be useful when reverse engineering an unknown can network from scratch. After identifying the ECUs using their method, our fuzzing framework can be used to reverse engineer the communications on the network further.

Chapter 9

Conclusion

The cyber-security of modern vehicles is becoming more important as cars get more connected. This is proved by recent penetration tests which allowed attackers to control a vehicle's system remotely. Fuzzing is a widely used testing methodology that allows a tester to evaluate large input domains automatically without much effort. It can also be used to aid reverse engineering by automatically discovering vulnerabilities in applications. There is some literature on applying fuzzing on vehicular networks with success. This thesis extends this work by using a sensor harness as feedback during fuzzing. This allows us to apply black-box fuzzing techniques and to reverse engineer the traffic on the CAN bus automatically.

The current sensor harness consists of colour light sensors which can be attached to instrument clusters. These sensors can detect activations of the various indicators on these instrument clusters. As the sensor harness uses the common I2C protocol, it can be extended with other types of sensors in the future. These other sensors could, for example, measure the steering wheel angle or detect engine ignition. After creating the sensor harness, the fuzzer from the Caring Caribou car exploitation framework was extended to use the harness during fuzzing. By writing several oracle functions, the fuzzer can be used for both reverse engineering of CAN traffic and vulnerability discovery.

We then applied the fuzzer to three separate instrument clusters. As two out of the three instrument clusters had no specification available, we focused on reverse engineering their traffic. The first test on a virtual instrument cluster proved that the fuzzer and sensor harness worked. It also provided some benchmarks which allowed us to evaluate the amount of time required to fuzz a particular ECU. The second test was on a dashboard of which the communications were completely unknown. Nevertheless, the fuzzer was able to reverse engineer most of the indicators on the dashboard. This proved that the fuzzer had real-world applications. Finally, we applied the fuzzer to an instrument cluster provided by a European truck manufacturer. Since this instrument cluster uses the J1939 protocol for communication, the fuzzer needed to be adapted to support J1939 packets. This was successful, as the fuzzer was able to identify most of the proprietary communications in the instrument cluster network.

There were, however, some difficulties in reverse engineering certain signals.

When the indicators cannot be manually triggered for calibration, an error-prone threshold calibration technique needs to be used. As the light sensors are susceptible to ambient light, this can lead to incorrect state changes to be identified. Another difficulty was applying the light sensors to the LCDs of modern instrument clusters. The sensors can correctly differentiate some messages on the screens but were not able to identify the more complex menus. This prevented us from reverse-engineering the exact payload of some messages which updated these LCDs. A future extension to the fuzzer could use a webcam in combination with computer vision to recognise the content of the LCDs.

The current fuzzing methods and bug oracles are more aimed towards reverse engineering instead of vulnerability discovery. A future version of the fuzzer could extend this with smarter fuzzing methods and better bug oracles. When an ECU specification is available, the bug oracles can be extended using this information. As mentioned in [14], this specification could also be used to generate smarted fuzzing test cases. This would allow the fuzzer to explore the input space smarter and faster.

As evidenced by recent attacks, the media centres are a prime target for finding vulnerabilities. These media centres are connected to the outside world and contain a lot of code dealing with external protocols. The colour sensors that the harness currently uses would serve well, other sensors such as auditory could also help the fuzzing process. When available, using hardware debuggers to aid the fuzzing process would also be interesting. They can be used to verify the found vulnerabilities or to apply some grey-box fuzzing methods which could speed up the process.

The fuzzer currently only works with the CAN protocol. However, support for other control network protocols can be added by changing the input evaluator. Doing this might also require updating the input generator as it has some assumptions about payload and arbitration ID size. Updating the fuzzer to support J1939 messages has already shown that supporting protocols with a more complex addressing scheme is possible. Another area to find vulnerabilities in are in media files. The sensor harness can still be used for crash detection when fuzzing files into the media centre. However, the input delivery would be a network media delivery protocol such as RSS feeds. There are already reported cases of media centres which incorrectly parse RSS titles [34].

The techniques used by *AutoFuzz* [16] could potentially be used to fuzz stateful protocols. It is possible that the current fuzzer never enters some states of these protocols. Which means a significant amount of code paths are potentially unexplored. The most interesting protocols used by ECUs, such as firmware update delivery, are often stateful. These protocols would thus be an attractive target for fuzzing. Finally, there are is a limited amount of benchmarks to compare the effectiveness of CAN fuzzing methods. A potential benchmark could be created by implementing several artificial vulnerabilities in the ICSim simulator. This benchmark would allow different fuzzing methods to be evaluated and compared against each other.

Appendix A

Automated Fuzzing of Automotive Control Units

Please see the next page for the paper.

Automated Fuzzing of Automotive Control Units

Timothy Werquin, Mathijs Hubrechtsen, Ashok Thangarajan, Frank Piessens, and Jan Tobias Mühlberg

KU Leuven, Dept. of Computer Science, imec-DistriNet, B-3001 Leuven, Belgium
jantobias.muehlberg@cs.kuleuven.be

Abstract. Modern vehicles are governed by a network of Electronic Control Units (ECUs), which are programmed to sense inputs from the driver and the environment, to process these inputs, and to control actuators that, e.g., regulate the engine or even control the steering system. ECUs within a vehicle communicate via automotive bus systems such as the Controller Area Network (CAN), and beyond the vehicles boundaries through upcoming vehicle-to-vehicle and vehicle-to-infrastructure channels. Approaches to manipulate the communication between ECUs for the purpose of security testing and reverse-engineering of vehicular functions have been presented in the past, all of which struggle with automating the detection of system change in response to message injection. In this paper we present our findings with fuzzing CAN networks, in particular while observing individual ECUs with a sensor harness. The harness detects physical responses, which we then use in a oracle functions to inform the fuzzing process. We systematically define fuzzers, fuzzing configurations and oracle functions for testing ECUs. We evaluate our approach based on case studies of commercial instrument clusters and with an experimental framework for CAN authentication. Our results show that the approach is capable of identifying interesting ECU states with a high level of automation. Our approach is applicable in distributed cyber-physical systems beyond automotive computing.

1 Introduction

Modern cars are controlled by software. In 2016 the Ford Motor Company already reported that their latest models are running on 150 million lines of code. This software forms a distributed mixed-criticality system that executes on a number of interconnected Electronic Control Units (ECUs). Jointly, these ECUs govern the vehicle's behavior – from convenience functions to infotainment to safety-critical functionality. ECUs are connected via automotive bus systems that facilitate the exchange of messages, most of which communicate sensor readings and control instructions. The control software then interprets sensor readings and reacts to events by triggering the relevant actuators, e.g., brakes, airbags, or steering gear. Given the enormous complexity of these systems, they are notoriously hard to test, for safety as well as for security properties.

Since 2004, researchers have been expressing their concerns with respect to the security limitations of communication standards, including the widely used

Controller Area Network (CAN), which typically provide no security ([27], [12], [11]). Since about 2010, a series of high-profile attacks ([13], [6], [16], [17]) illustrate that with increased vehicular connectivity, even remote adversaries can take control of critical functions of a vehicle. These risks have been acknowledged by emerging industry standards [23,1] that encompass authentication and software security for control systems, and prototypes that showcase secure system designs for automotive computing based on software attestation and Trusted Computing primitives [26] have been proposed. Meanwhile, more and more low-level vulnerabilities of these communication systems are being revealed (e.g., [9], [22]) and guidance for the reverse-engineering and penetration testing of vehicular communications and control systems becomes readily available [24] and the need for advanced testing methodology for these systems is generally acknowledged.

Fuzz testing ([21], [15]) is a well established methodology to expose software and systems to unexpected conditions, for example by providing random input streams that may crash the target. Yet, the approach does not easily apply to embedded software [18] and few approaches have been made to fuzz embedded control systems or automotive ECUs in particular ([6], [14], [4]). A key difficulty to overcome here is the definition of oracle functions that define when a fuzzer has potentially triggered an "interesting" system state, and to automatically evaluate these functions.

Our Contributions. In this paper we discuss fuzz testing in the context of automotive control networks. Specifically, our research investigates the use of fuzzing so as to find vulnerabilities and to reverse engineer ECU functionality in CAN networks. We make the following contributions:

1. We systematically define fuzzers, fuzzing configurations and oracle functions for testing automotive ECUs through their CAN interface.
2. We develop a sensor harness to automatically evaluate fuzzing oracles for ECUs with physical outputs.
3. We evaluate our approach, taking commercial automotive instrument clusters and an experimental setup for testing AUTOSAR-compliant message authentication as case studies.

To the best of our knowledge, this paper is the first to largely automate a methodical fuzzing approach (e.g. following [15]) for automotive ECUs. Although our implementation is targeting CAN components, our approach can be generalised to cyber-physical systems with any underlying communication technology. Our fuzzer implementation, instructions to build the sensor harness and to repeat our experiments will be made available under an open-source license.

2 Background

In this section we briefly introduce the CAN bus, which is commonly used to facilitate communication between automotive ECUs but also in industrial control systems. We further introduce the CaringCaribou penetration testing framework, which our fuzzing toolchain is integrated with.

2.1 Controller Area Network (CAN) & Security

The CAN bus is the most commonly used broadcast network in modern cars. A CAN message consists of an 11-bit arbitration ID, followed by an optional 18-bit extended ID, and up to 8 bytes of data payload (cf. Fig. 1). Dedicated transceiver hardware implements a protocol for message acknowledgement and bus arbitration for sending/receiving data frames. CAN requires a fixed data transmission rate, and allows recessive bits (one) to be overwritten by dominant bits (zero) during transmission. Message acknowledgement can thus simply be implemented by overwriting the ACK bit at the end of the data frame in real-time. Likewise, to implement bus arbitration, CAN transceivers are required to continuously listen on the bus while sending the message ID at the beginning of the data frame, and to back off when one of their ID bits has been overwritten. This scheme ensures that messages with lower IDs effectively have higher priorities. Finally, each CAN frame features a 16-bit CRC field to detect transmission errors.

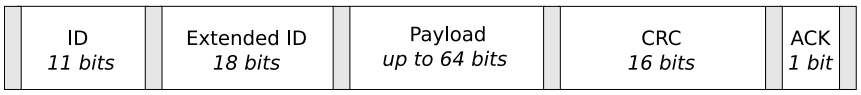


Fig. 1. Extended data frame standardised by CAN 2.0B.

CAN was originally developed in 1983, when cyber attacks were of no concern. Thus, the protocol does not provide any form of message authentication. Any ECU connected to the network can spoof messages with arbitrary sender ID and payload, which forms the basis of many attacks [13,6,16,17]. As a response, the AUTOSAR [1] standardisation body published industry guidelines for backwards-compatible message authentication in vehicular networks. The VulCAN framework [26], which we study in Sect. 5.1 is one implementation of these authentication extensions.

2.2 CaringCaribou and Automotive Penetration Testing

We built our implementations of CAN fuzzers as modules for the open-source penetration-testing framework CaringCaribou. Our fuzzing extensions are freely available for further experimentation and follow-up research. Since fuzzers are widely used as a means to perform black-box testing in the regular penetration testing industry, it is our belief that a similar tool could prove useful in the automotive penetration testing community.

CaringCaribou[1] is a tool developed for the purpose of being the "nmap" of automotive security. Cyber security research in the automotive industry is a new field that is rapidly expanding. Yet, it still lacks the mature tooling available to the mainstream security community. Tools like CaringCaribou aim to fill that gap.

[1] `https://github.com/CaringCaribou/caringcaribou`

CaringCaribou has a modular architecture that allows developers of penetration testing techniques for automotive systems to easily write new modules for their specific purpose, and deploy these modules using a unified tooling infrastructure. Thus, CaringCaribou provides the developer with a layer of abstraction which protects them from the specifics of CAN and other automotive communication protocols, allowing them to focus on writing the actual penetration testing tool instead of dealing with the lower-layer interactions. CaringCaribou is supposed to be a zero-knowledge tool that can be deployed on any CAN network regardless of its specific configuration.

3 Fuzzing CAN Networks

In this section we lay out our approach to define a fuzzing tool for CAN-based automotive control systems. We follow the approach of Manes et al. [15] and dissect the tool into an oracle component, the actual fuzzer and the run-time configuration for the fuzzer for a particular run.

3.1 Bug Oracles for ECUs

According to [15], a (bug) oracle is a program that determines whether a given execution of the target system violates a specific (security) policy. In Sect. 5 we outline two very different case studies for our system: In one of these we have a partial specification of security properties of the network available, and where we are looking for violations of this specification. In the second case study we have no reliable specification but we are interested in reverse-engineering such a specification. Both case studies are characterised by not being able to observe software interactions directly (as opposed to software fuzzing with code instrumentation in a simulator [15]). Instead, we are looking at black-box systems to which our fuzzer can provide an input stream, while any observable communication output, physical output (actuation of a LED or a relay) or even the timing or absence of such outputs (e.g., due to a software crash) may indicate that an "interesting" system state has been reached.

Recent related work in the field of intrusion detection for for industrial control system (e.g., [25] and [28]) suggests that machine-learning approaches can be used to train detectors that then report anomalies in the communication behaviour of vehicular networks. We have not implemented such oracle functions.

Physical outputs of control units can certainly be observed by human operators. They can also be sensed and electronically reported through sensor networks, or in our case a sensor harness that is attached to the target system. In the following sections we emphasise on this form fuzzing oracle, where a state change in the target is defined by a sensor (de-)activation or sensor threshold.

Most difficult is certainly the detection of system failure which results in the absence of an observable response from the target. Thus, inputs that lead to failures are easily misinterpreted by a fuzzing tool as inputs that have no effect. For example, our work deals with ECUs that need to regularly receive

certain messages or they will fault, effectively rendering the continuation of a fuzzing campaign ineffective. Detecting these messages and constructing traffic that satisfies the input requirements of ECUs in the absence of a specification, is difficult. An oracle to identify these kinds of system faults requires the use of recorded traffic that periodically triggers a known observable output. An oracle function will then fire when the periodic signal is absent due to, e.g., message omission. We apply this method in our *omission fuzzer* below.

3.2 Defining CAN Fuzzers

Fuzzing is the execution of the target system using input(s) sampled from an input space (the "fuzz input space") that protrudes the expected input space of the target system [15]. With fuzzing we aim to enumerate and exercise a large subset of this fuzz input space to find system behaviour that triggers an oracle function. ECUs that process CAN messages are an interesting target since the frame size of CAN messages is at most 110 bits. This fuzz input space is certainly huge, but much smaller than, e.g., Ethernet frames, WiFi frames, or multimedia streams. Still, even for CAN networks, this fuzz input space is prohibitively large for being exhaustively exercised. Furthermore, with a maximum bandwidth of 1 MBit, and most ECUs using 500 MBit as a fixed transfer rate, data transmission to a target network of ECUs represents a bottleneck.

Starting with the idea of *random fuzzing*, where arbitration IDs and message payloads are selected randomly, we devise three additional fuzzing strategies, *brute-force fuzzing*, *mutation fuzzing* and *identify fuzzing*, to narrow down the fuzz input space and explore interesting ECU behavior more efficiently. These strategies are based on the observation that an ECU typically accepts inputs on a relatively small number of IDs only, that also the number of payload bits that result in a observable state change is limited, and that several consecutive messages may be required to trigger an observable state change. We then integrate these approaches in an *automated exploration* mode, where inputs from a sensor harness (cf. Sect. 4), which is attached to a target ECU, guide input generation. We have implemented our approach in two modules for CaringCaribou, namely `fuzzer` and `autoFuzz`, which can be invoked as `./cc.py <module> <parameters> [-f <file>]`. Here, `./cc.py` refers to the CaringCaribou main script, `<module>` to a fuzzer module, and `<parameters>` to a fuzzer configuration which we discuss below. `-f <file>` can be used to store a message trail on disk. For example, `./cc.py fuzzer random` will generate entirely random messages and dispatch them over the configured CAN interface.

Brute-Force Fuzzing. This method aims to exhaustively enumerate selected hexadecimal digits in a message, specifically in the message's ID field and the payload. For example, the fuzzer can be invoked as `./cc.py fuzzer brute 0x123 12ab..78`, where the 5th and 6th octet of the message payload will be enumerated and sent, while the message ID `0x123` and all other payload octets remain constant.

Mutation Fuzzing. This strategy can be used to systematically explore a larger fuzz input space through mutating selected hex digits in arbitration ID and message by means of individual random bit flips. An example use for this strategy is `./cc.py fuzzer mutate 7f.. 12ab....`; the syntax follows the example given for *brute-force fuzzing* above.

Identify Fuzzing. Once a fuzzing run resulted in an event of interest, e.g., a change of an indicator LED on a target ECU, the *identify* method can be used to replay and identify a minimal set of messages that caused the event. The syntax for invoking this method is `./cc.py fuzzer identify log.txt`, where `log.txt` refers to a log file previously recorded with the `-f` parameter. The method relies on human input – i.e., key presses – to gather information about the timely occurrence of events, and aims to prune the set of recorded messages in `log.txt` so that the event still occurs when the pruned set is replayed.

Automated ECU Exploration. Our `autoFuzz` module implements the above strategies so that system change can be detected directly through our sensor harness (cf. Sect. 4). Sensor observations can then be used to guide the generation of the next inputs and to automatically identify message bits that lead to observable system change, depending on the fuzzing strategy. The module further features the generation of J1939-compliant messages and the fuzzing of J1939 function group addresses (PGNs and SPNs).

When fuzzing an ECU with a single sensor attached to one of the ECU's actuators, it is possible to immediately run the identify fuzzer when a change in the sensor state is detected without relying on recorded traffic. This requires that the actuator can be triggered with a predictable payload. When using multiple sensors, the log file can be filtered to keep a number of messages preceding the activation of a specific sensor which can then be used as input to the identify fuzzer.

While experimenting with fuzzing strategies, we observed that an ECU's response to a message is often delayed. During the delay period, other messages are being sent by the fuzzer, which makes identifying the CAN messages specifically responsible for an response more difficult. One possible solution is to increase the delay between sending messages. Yet, this will increase the time required to cover the fuzz input space. Another option is the resend only a subset of the messages preceding a sensor activation with increased delays, which we implemented in our identify method.

Our experiments further revealed that some ECUs expect certain messages to be received regularly. The absence of these messages will lead to a shut-down or render the ECU unresponsive and to indicate a failure. These behaviours prevent our identify method from working as sending the complete traffic log can keep the ECU responsive but sending parts of the recorded traffic will cause the ECU to fault. To address this, we developed an approach that we refer to as *omission fuzzing*. This strategy sends the complete recorded traffic but omits some messages in order to identify which message cause specific state changes.

The identified arbitration IDs or payloads can then be added to a "blacklist" to inform other methods. E.g., any arbitration ID in the blacklist will never be omitted during *identify fuzzing*.

After collecting logs from the fuzzer, various analysis can be applied. For example, if a specification of the target ECU is available, which would detail the expected actuator responses to specific groups of messages, sensor activations can be check against this specification to detect bugs or undocumented behaviour.

3.3 Target-Specific Fuzzer Configuration

A fuzz configuration of a fuzzer comprises the parameter value(s) that control(s) the fuzz algorithm [15]. In the context of our approach, these parameters involve *message generation*, *message timing*, *message omission*, and the configuration of the *sensor harness*. As outlined before fuzzing entire CAN messages makes little sense as it results in an extremely large fuzz input space. Thus, configurations will typically restrict the fuzz space to specific octets in (extended) arbitration IDs and message payload. Message timing is typically configured to schedule a new message every 3 ms to 20 ms to avoid message collisions and to leave enough time for actuators to be engaged and sensed. Message omission and baseline traffic are to be set up to simulate typical bus traffic in a car so as to make target ECUs function normally. The sensor harness offers a wide range of configuration options that involve the type of sensors, sampling rates, the number of sensors and their placement on the target ECU.

4 A Sensor Harness to Automate ECU Fuzzing

In this section we describe an inexpensive and extensible experimental sensor harness to automate the analysis of automotive ECUs. The intuition behind the setup is that fuzzing communication in an automotive control network, or in cyber-physical systems in general, can cause a range of interesting responses beyond network communication. Thus, to use these responses as inputs to fuzzing oracles (cf. Sect. 3.1) they must be automatically measured at an appropriate sampling rate. Previous approaches to consider these responses typically rely on human observation and human interaction during the fuzzing process. For example, a fuzzing tool may require the user to press a key if they observe a change in the system, e.g., a flashing indicator light on a control panel. Our work improves over this by detecting physical responses of ECUs automatically, with negligible delays, and at a configurable granularity.

Fig. 2 gives an overview of the sensor harness. The system in action is depicted in Sect. 5.2. The harness connects multiple sensors together and provides a Universal Serial Bus (USB) interface for a PC to control the setup. The depicted configuration contains only light and colour sensors which can be placed over the various indicators LEDs on a target ECU. No general-purpose microcontroller is used in the harness, which allows the entire setup to be programmed and configured from the PC in a higher-level language, Python in our case.

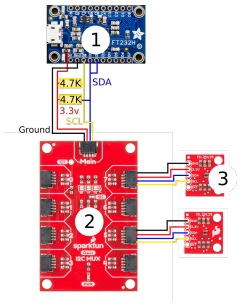


Fig. 2. Sensor harness connection schema.

The light sensors are connected through an I2C bus, a communication interface which is present on many low-cost sensors. The use of I2C make the harness extensible with various other sensors, such as sound sensors to monitor auditory alerts, motion sensors to monitor steering wheel movement, or current sensors to detect an engine start. As the I2C bus supports multiplexing, multiple sensors can be connected to the same bus as long as they have a different I2C address. The low-cost sensors used in the harness often have a fixed address, which implies that an I2C multiplexer must be used to connect multiple sensors of the same type in the harness. In order to interface with the sensor harness from a PC an USB to I2C adapter is used.

Light & Colour Sensors: ISL29125. The ISL29125 colour sensor is used to measure the status of visual indicators on automotive equipment. The sensor provides a simple RGB light level readout and has a number of configuration options which determine the sensitivity and precision of the sensor. The sensitivity can be configured between 10k lux or 375 lux and the sensor has a built in infrared light filter which can be configured separately. The precision of the sensor can be configured to be either 12 or 16 bits. Changing the precision also changes the integration time, meaning a higher precision (16 bits) will require the Analogue-Digital Converter (ADC) to sample the sensor longer resulting in slower measurements. In addition to the I2C interface the sensor has an interrupt pin which can be triggered by a configurable light level on either of the red, green or blue channel. Currently the harness does not use the interrupt functionality but works by polling each sensor individually. Our colour sensors have a single fixed I2C address, requiring a multiplexer to connect multiple sensors on the same I2C bus.

I2C Multiplexer: TCA9548A. As the colour sensor has a fixed I2C address, the TCA9548A multiplexer is used to connect multiple colour sensors in the harness. The TCA9548A multiplexer has eight I2C channels which allows eight colour sensors to be connected on the same bus. As the multiplexer has a three bit configurable address multiple multiplexers can be daisy chained allowing up to 64 I2C busses on a single connection.

I2C-to-USB: FT232H. In order to connect the I2C bus to a PC the FTDI FT232H adapter is used. This adapter supports a number of different bus protocols such as UART, SPI and I2C in addition to a GPIO interface which can be used to write to eight digital IO pins.

Programming, Calibration & Use. To obtain readings from the light sensors, the Arduino library[2] for reading ISL29125 sensors was adapted. The resulting library uses the Adafruit[3] library for communicating with the FT232H adapter. Our current sensor library exposes functions for initialising a new sensor, and for reading the sensor's red, green and blue values.

The initialisation function resets the sensor and configures it in RGB mode which enables all three colour channels, puts it in 10k-lux mode for bright environments and enables high IR light adjustment. Reading a colour value is done by reading from the relevant device register, which returns a colour value of either 12 or 16 bits depending on the chosen precision. The time the sensor needs to take a reading varies depending on the ADC integration time which in turn depends on the chosen precision: At 16 bit precision, each reading takes about 110 ms while at 12 bit each reading takes only 7 ms. As the sensor's output registers are double-buffered, reading out these registers between sensing operations will result in outdated readings.

In order to use multiple light sensors a library that interfaces with the TCA9548A multiplexer was created, this library allows virtual I2C ports to be created for each channel of the multiplexer. These virtual I2C ports can then be used in the sensor library instead of the default FT232H I2C port. In order to switch I2C channels the number of the requested channel is written before any commands, this adds a delay before every I2C command, which is negligible in comparison with the integration time of the light sensor and does not impact fuzzing performance.

In initial experiments we use the colour light sensors to detect whether the various indicators on an automotive dashboard are changing state, effectively converting the red, green and blue light levels into a binary input signal for the fuzzer. As the sensors are sensitive enough to detect (even reflected) movements behind the sensor while duct-taped to a dashboard in both sensitivity configurations (up to 375 lux and 10k lux) a simple threshold is not sufficient to distinguish state change. We devise a calibration method that involves taking a reading when the indicator is on and when the indicator is off. This results in red, green and blue light level triples to which any new measurement can be compared, if the new measurement is closer to the on-value the indicator is detected as on and vice versa. This method assures that a uniform increase or decrease in ambient light does not change the detected indicator value. The method further requires a calibration with the indicator both on and off, which may not be feasible when the indicator trigger is unknown. When the indicator cannot be triggered during calibration, a simple threshold may be used to detect the indicator state. More

[2] `https://github.com/sparkfun/SparkFun_ISL29125_Breakout_Arduino_Library`
[3] `https://github.com/adafruit/Adafruit-FT232H-Breakout-PCB`

elaborate calibration methods may be required to operate the sensor harness in noisy environments.

5 Evaluation and Discussion

We have applied our fuzzer implementation and the sensor harness to a number of case studies that include the ICSim automotive instrument cluster simulator[4], a demo setup for illustrating and implementing message authentication in CAN networks with the VulCAN [26] framework, as well as real instrument clusters. In this section we focus on our experience and lessons learned from the latter two case studies. We compare our findings with earlier manual approaches to discover bugs and explore proprietary functionality in these scenarios.

5.1 Case Study 1: VulCAN

We evaluated the effectiveness of our fuzzer to find implementation bugs and security vulnerabilities on a demo implementation of VulCAN [26], a generic design for CAN message authentication. VulCAN provides efficient and AUTOSAR-compliant [1] authentication plus software component attestation based on lightweight trusted computing technology. We used the same test bench as described in [26] to test the abilities of the fuzzer.

In brief, the demo consists of a number of ECUs with keypads as input devices and LED displays as actuators. A distributed control application which simulates a traction control system is executing on the ECUs. Application components communicate via cryptographically authenticated CAN messages with freshness guarantees: only messages that are successfully validated to be fresh and to originate from unmodified and integrity-protected remote component should ever be able to trigger output events. The application communicates only a few valid payloads at fixed intervals. Thus, deviation from expected behaviour would be easy to detect. Yet, since it is unlikely for a random or mutation-based fuzzer to "guess" a valid payload, nonce, and authentication tag triple, and since the system was designed with security in mind, we did not expect the security properties of the system to be broken easily. This part of the evaluation is conducted without using the sensor harness but by relying on visual observation on the demo's LED displays. The fuzzer is executing on desktop PC, which is connected to the demo setup via a USB to CAN interface.

To our surprise, with the help of the fuzzer, we detected and traced several unique vulnerabilities in the system in a fairly short period of time. Below we focus on two particularly subtle discoveries.

The first vulnerability was discovered nearly instantaneously in a fuzzer configuration where messages with extended CAN arbitration IDs are generated. Such messages resulted in system states where the injected messages could lead to actual display outputs, breaking the security properties of VulCAN entirely.

[4] `https://github.com/zombieCraig/ICSim`

Extended CAN IDs are not being used in the VulCAN demo, and thus, the components were not tested in environments where these messages occur. Most likely, a misconfigured driver for the CAN controller on an ECUs – "untrusted" software in VulCAN's attacker model – together with an incomplete rejection condition in a secure application module, allowed an attacker to arbitrarily adjust the displays of the test bench without having to pass authenticity checks.

The second vulnerability was found within the implementation of one of the authentication protocols in VulCAN, specifically VatiCAN [20]. This implementation turned out to be particularly vulnerable to denial-of-service attacks when being flooded with specific traffic patterns, allowing an attacker to desynchronise nonces and render trusted components unresponsive even to dedicated re-synchronisation messages. The bug was discovered in a timespan of several minutes when fuzzing the test bench in a configuration where both, the fuzzer as well as an ECU, are simultaneously attempting to send messages to a target ECU. Interestingly, due to the configuration error in CAN drivers described above, messages with extended CAN IDs are effectively interpreted as broadcast messages. Application components are thus subject to receiving a mix of fuzzer-generated payloads and authenticated messages which results in denial-of-service.

5.2 Case Study 2: Instrument Clusters

In the context of automotive security research and for building demos such as the VulCAN [26] setup, instrument clusters are commonly used as easily accessible off-the-shelf components with many visible indicators (speed needle, turning indicators, display, etc.), most of which can be controlled through CAN messages. Yet, the specific arbitration IDs and payloads to control these functions are not publicly documented. Literature on car hacking (e.g., [24]) suggests manual approaches to reverse-engineer these details, which may require hours or even days of try-and-error, even for a skilled engineer. By using our sensor harness, we expect a substantial speed-up of these processes, on top of being able to largely automate the process.

We have been experimenting with a number of clusters from passenger cars and commercial vehicles. As illustrated in Fig. 3, components of our sensor harness are duct-taped to indicators of the cluster. The instrument cluster is connected to a desktop PC with the fuzzer via a USB to CAN interface and there are no other ECUs present on the CAN.

In order to test the instrument cluster's basic functions, we developed a controller application that would send a number of documented [3] CAN messages to the dashboard. Only some these control messages worked in combination with our dashboard. We then applied our fuzzer in *identify* mode to filter the traffic data for messages that trigger physical functions, then applied *brute-force* and *mutation* mode to explore arbitration IDs and payloads to trigger further functionality.

By *brute-forcing* the entire 11-bit arbitration ID fuzz-space with a fixed payload `0xffffffff`, most indicators LEDs in the dashboard could be activated. Some of these indicators are switched on by default when the instrument cluster

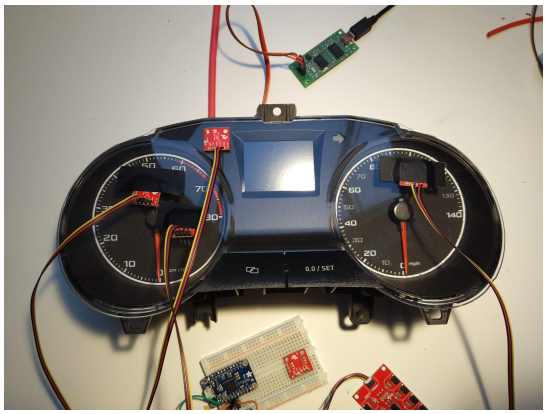


Fig. 3. An automotive instrument cluster with (part of) our sensor harness attached. The cluster originates from a 2014 Seat Ibiza model.

is powered up. These indicators could be triggered using a fixed payload of `0x00000000`. Control of the speedometer and engine RPM needles could also be triggered. The fuzzer takes about 30 s to enumerate the 11-bit address space using a delay of 10 ms between messages. Running the *identify* method after finding a state changes takes an extra 30 s per activation in order to identify the message responsible for an indicator activation.

Using *mutation-based* fuzzing on any of the messages identified with the brute-force method, we were able to reverse engineer the semantics of most payload bits. For example, by starting from the messages that activated the left turning indicator, the fuzzer was able to not only identify the bit responsible for triggering the indicator. It was also able to identify the bit responsible for the right turn signal, the headlights indicator and a number of other status LEDs. Enumerating eight payload bits takes about 20 s with a delay of 3 ms between messages. Using the *identify* method requires an additional 5 s per indicator. The delay between messages is critical when using mutation-based fuzzing. If the delay is too short, the next message, which will have another bit flipped, will overwrite the previous message. This will cause more indicator activations to be missed or some indicators to not even activate.

Omission fuzzing is not necessary for analysing this particular instrument cluster. Yet, we heavily relied on this method method when working with more modern clusters from commercial vehicles.

5.3 Discussion & Lessons Learned

The two case studies outlined above show that our fuzzers can efficiently reveal undocumented functionality, intricate bugs and security vulnerabilities in ECUs connected to CAN networks. Ultimately, our experiments provide further evidence for fuzzing to be a useful tool in testing and reverse-engineering, which is due to

the technique's ability to cover an enormous range of possible combinations of system inputs, in our case arbitration IDs and payloads. Many of these inputs may not even occur in normal and benign operation, and are difficult to consider in static test cases.

Above we describe two critical security vulnerabilities in an experimental system design, which are based on intricate implementation and configuration bugs. Previously undetected, these vulnerabilities became apparent within minutes with the use of a fuzzer, even without relying on a sensor harness. The harness could be used to reduce human interaction and to improve the duration for detecting and tracing these bugs, but non-trivial extensions of the harness would be required to sense the state of actuators such as the attached displays. Alternatively, the demo setup could be modified to feature simpler actuators (i.e., individual LEDs or relays) that allow for an easier detection of conditions that satisfy our bug oracles. We used our fuzzer, specifically the identify and replay functionality, to trace bugs in source code and fix vulnerabilities. The resulting fixes are not straight-forward as they require consideration of rather involved network states. Our findings highlight the need for thorough testing and verification on top of strong cryptographic primitives and Trusted Computing technology when designing distributed control systems that are potentially exposed to malicious interactions.

We further described how the sensor harness in combination with our fuzzing techniques can be used to largely automate the process of reverse-engineering communication protocols of proprietary ECUs. Manually reversing a substantial subset of the functionality of, e.g., an instrument cluster, can easily be an effort of several days or even weeks. With our approach, this can be achieved within hours. Additional sensors (e.g., audio, power consumption, vibration) could further extend the harness' abilities. We believe that our approach can be used to identify bugs and unintended functionality when being applied to components for which a specification is available. This specification could be integrated in a bug oracle such that responses outside of the specified behaviour are detected as errors. In this context, our approach may be useful to automate activities such as integration testing and to achieve a high input-space coverage in these activities.

Fuzzing ECUs under realistic conditions, i.e., while being connected to a vehicle's CAN network with many other "noisy" ECUs, may also be feasible but requires fuzzing strategies that aim at noise reduction by exploring the effects of individual messages or sequences of messages in different system states. In this context it may be useful to also consider CAN responses of ECUs. We may borrow from recent approaches in anomaly detection in control systems (e.g., [25] and [28]) to define oracle functions that detect changes in the response stream of an individual ECU, or even to detect state change throughout the network.

6 Related Work

Fuzzing has a long history and is still actively developed, in particular in the domain of security- and penetration testing of software systems ([21], [15]). Recent

work [18] elaborates on the difficulties of employing fuzzing in embedded systems. Specifically for automotive systems, Smith states in [24] that, while fuzzing can certainly be useful in discovering undocumented services or crashes, it is rarely useful beyond that, e.g., to find and exploit vulnerabilities. Our experience report disagrees slightly with this observation: We discovered that fuzzing is more efficient in finding subtle vulnerabilities and configuration errors than monitoring or reverse engineering the firmware and communications. Fuzzing exposes substantially more of the system's unintended states than what one would be able to explore manually, due to the sheer amount of pseudo-random message combinations that are generated and dispatched by the fuzzer. This allows testers to focus on tracking down and responding to vulnerability reports instead of having to manually probe the system. With automated oracle function, as discussed in Sect. 3.1 and Sect. 4, fuzzing becomes even more efficient. While our approach mostly relies on black-box fuzzing where very little knowledge of the system is assumed and oracle functions must rely on system outputs rather than observing the system's internal state, our approach can certainly be combined with more advanced reverse-engineering and firmware inspection tools. This would lead to more powerful and also much more intricate oracle functions.

Related research investigates the extent to which fuzzing can be applied be in automotive systems ([14], [5], [19], [8], [4]). Our work aims to improve over this state of the art by not only defining a fuzzer for CAN networks, but by developing an entire methodology that defines fuzzing objectives, oracle functions and fuzzing strategies, and substantially improves the automation of testing Cyber-Physical Systems. We report on experiments and lay out our experience from applying this methodology to two realistic systems, one of which being a prototype for an automotive security system. While related work reports mixed results on the usefulness of fuzzing automotive networks, we judge our results as largely positive since we were able to identify a few subtle vulnerabilities and dramatically speed up reverse-engineering activities.

Further related research investigates the use of fuzzing to explore stateful communication protocols ([2], [10]), also in the context of Cyber-Physical Systems such as smart-grid communications [7]. Our toolchain does currently not employ such techniques. We believe that these could be a sensible addition to the current stateless exploration approach. Specifically, these techniques may be used to brute-force intricate CAN protocols that trigger, e.g., software updates on ECUs.

7 Summary & Conclusions

Automotive control networks are highly complex safety-critical and security-critical systems which have been shown to be vulnerable to adversarial interactions. In this paper we devise a largely automated approach for fuzz-testing these systems. We discuss how bug oracles for automotive ECUs can be described, define a number of fuzzing strategies, and develop a sensor harness to allow oracle functions to detect interesting system behaviour in an automated fashion. We have implemented our fuzzing approach in CaringCaribou, an open-source automotive

penetration-testing toolkit, and we report on two sets of experiments where we apply our fuzzer to find vulnerabilities and to reverse-engineer proprietary ECU functions. To the best of our knowledge, our approach is the first to achieve a high degree of automation for these activities, and we see future applications of our fuzzing approach in, e.g., penetration testing but also in integration- and compliance testing. While we have been focusing on CAN networks, we believe that the approach is applicable to other types of control networks and beyond the domain of automotive computing.

Acknowledgements. This research is partially funded by the Research Fund KU Leuven. This research is partially funded under SErVO, "Secure and Economically Viable V2X Solutions", by the Flemish Agentschap Innoveren & Ondernemen. We thank the developers of CaringCaribou for their sport and ideas, and for integrating parts of our fuzzer into their platform.

References

1. AUTOSAR Specification 4.3. Specification of module secure onboard communication. `https://www.autosar.org/standards/classic-platform/release-43/software-architecture/safety-and-security/`, 2016.
2. Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., and Vigna, G. Snooze: toward a stateful network protocol fuzzer. In *International Conference on Information Security*, pp. 343–358. Springer, 2006.
3. Bataille, L. Volkswagen can bus gaming. URL: `https://hackaday.io/project/6288-volkswagen-can-bus-gaming`.
4. Bayer, S., Enderle, T., Oka, D.-K., and Wolf, M. Automotive security testing – the digital crash test. In *Energy Consumption and Autonomous Driving*, pp. 13–22. Springer, 2016.
5. Bayer, S. and Ptok, A. Don't fuss about fuzzing: Fuzzing controllers in vehicular networks. *13th escar Europe*, p. 88, 2015.
6. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. San Francisco, 2011.
7. Dantas, H., Erkin, Z., Doerr, C., Hallie, R., and Bij, G. v. d. efuzz: A fuzzer for dlms/cosem electricity meters. In *Proceedings of the 2nd Workshop on Smart Energy Grid Security*, pp. 31–38. ACM, 2014.
8. Fowler, D. S., Bryans, J., and Shaikh, S. Automating fuzz test generation to improve the security of the controller area network.
9. Fröschle, S. and Stühring, A. Analyzing the capabilities of the CAN attacker. In *ESORICS '17*, vol. 10492 of *LNCS*, pp. 464–482, Heidelberg, 2017. Springer.
10. Gorbunov, S. and Rosenbloom, A. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.
11. Henniger, O., Apvrille, L., Fuchs, A., Roudier, Y., Ruddle, A., and Weyl, B. Security requirements for automotive on-board networks. In *9th International Conference on Intelligent Transport Systems Telecommunications, (ITST)*, pp. 641–646, 2009.

12. Hoppe, T., Kiltz, S., and Dittmann, J. Security threats to automotive CAN networks – practical examples and selected short-term countermeasures. In *Computer Safety, Reliability, and Security (SAFECOMP '08)*, pp. 235–248, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
13. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., et al. Experimental security analysis of a modern automobile. In *Security and Privacy, 2010 IEEE Symposium on*, pp. 447–462. IEEE, 2010.
14. Lee, H., Choi, K., Chung, K., Kim, J., and Yim, K. Fuzzing can packets into automobiles. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pp. 817–821. IEEE, 2015.
15. Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140, 2018.
16. Miller, C. and Valasek, C. A survey of remote automotive attack surfaces. *Black Hat USA*, 2014.
17. Miller, C. and Valasek, C. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.
18. Muench, M., Stijohann, J., Kargl, F., Francillon, A., and Balzarotti, D. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
19. Nishimura, R., Kurachi, R., Ito, K., Miyasaka, T., Yamamoto, M., and Mishima, M. Implementation of the can-fd protocol in the fuzzing tool bestorm. In *2016 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, pp. 1–6. IEEE, 2016.
20. Nürnberger, S. and Rossow, C. – vatiCAN – Vetted, authenticated CAN bus. In *Cryptographic Hardware and Embedded Systems – CHES '16: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pp. 106–124, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
21. Oehlert, P. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.
22. Palanca, A., Evenchick, E., Maggi, F., and Zanero, S. A stealth, selective, link-layer denial-of-service attack against automotive networks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 185–206. Springer, 2017.
23. SAE International. J3061: Cybersecurity guidebook for cyber-physical vehicle systems, 2016. `http://standards.sae.org/j3061_201601/`.
24. Smith, C. *The car hacker's handbook: a guide for the penetration tester*. No Starch Press, 2016.
25. Taylor, A., Leblanc, S., and Japkowicz, N. Anomaly detection in automobile control network data with long short-term memory networks. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 130–139, 2016.
26. Van Bulck, J., Mühlberg, J. T., and Piessens, F. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *ACSAC '17*, pp. 225–237. ACM, 2017.
27. Wolf, M., Weimerskirch, A., and Paar, C. Security in automotive bus systems. In *Workshop on Embedded Security in Cars*, 2004.
28. Wressnegger, C., Kellner, A., and Rieck, K. Zoe: Content-based anomaly detection for industrial control systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 127–138, 2018.

Bibliography

[1] Adafruit. Adafruit ft232h breakout - general purpose usb to gpio+spi+i2c. URL: https://www.adafruit.com/product/2264, last checked on 2019-03-10.

[2] Audi. Driver assistance systems. URL: https://www.audi-mediacenter.com/en/the-new-audi-a5-and-audi-s5-coupe-6269/driver-assistance-systems-6281/, last checked on 2019-05-28.

[3] C. I. Automation. History of can technology. URL: https://www.can-cia.org/can-knowledge/can/can-history/, last checked on 2019-06-1.

[4] AUTOSAR. Classic platform. URL: https://www.autosar.org/standards/classic-platform/, last checked on 2019-06-1.

[5] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. Snooze: toward a stateful network protocol fuzzer. In *International Conference on Information Security*, pages 343–358. Springer, 2006.

[6] L. Bataille. Volkswagen can bus gaming. URL: https://hackaday.io/project/6288-volkswagen-can-bus-gaming, last checked on 2019-05-28.

[7] S. Bayer and A. Ptok. Don't fuss about fuzzing: Fuzzing controllers in vehicular networks. *13th escar Europe*, page 88, 2015.

[8] B. Blumbergs and R. Vaarandi. Bbuzz: A bit-aware fuzzing framework for network protocol systematic reverse engineering and analysis. In *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, pages 707–712, Oct 2017.

[9] Caring Caribou. Caring caribou. URL: https://github.com/CaringCaribou/caringcaribou, last checked on 2019-03-10.

[10] W. Choi, H. J. Jo, S. Woo, J. Y. Chun, J. Park, and D. H. Lee. Identifying ecus using inimitable characteristics of signals in controller area networks. *IEEE Transactions on Vehicular Technology*, 67(6):4757–4770, June 2018.

[11] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009.

[12] T. Fischl. Usbtin - usb to can interface. URL: https://www.fischl.de/usbtin/, last checked on 2019-06-1.

[13] I. Foster, A. Prudhomme, K. Koscher, and S. Savage. Fast and vulnerable: A story of telematic failures. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., 2015. USENIX Association.

[14] D. S. Fowler, J. Bryans, and S. Shaikh. Automating fuzz test generation to improve the security of the controller area network.

[15] FTDI Chip. *FT232H Single Channel HiSpeed USB to Multipurpose UART/FIFO IC*, 05 2018. FT_000288.

[16] S. Gorbunov and A. Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.

[17] Guntis. How we built a robot for automated manual mobile testing. URL: https://www.testdevlab.com/blog/2017/07/how-we-built-a-robot-for-automated-manual-mobile-testing/, last checked on 2019-05-28.

[18] T. Hunt. Controlling vehicle features of nissan leafs across the globe via vulnerable apis. URL: https://www.troyhunt.com/controlling-vehicle-features-of-nissan/, last checked on 2019-06-1.

[19] Intersil. *Digital Red, Green and Blue Color Light Sensor with IR Blocking Filter*, 1 2014. FN8424.2.

[20] ISO 11898:1993. Road vehicles – Interchange of digital information – Controller area network (CAN) for high-speed communication. Standard, International Organization for Standardization, Geneva, CH, Nov. 1993.

[21] A. Karahasanovic. Threat modeling of the autosar standard, 2016.

[22] H. Lee, K. Choi, K. Chung, J. Kim, and K. Yim. Fuzzing can packets into automobiles. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 817–821, March 2015.

[23] K. Mahaffey. Hacking a tesla model s: What we found and what we learned. URL: https://blog.lookout.com/hacking-a-tesla, last checked on 2019-05-28.

[24] D. Maliniak. An inside look at the automotive ethernet protocol. URL: https://www.ecnmag.com/article/2018/08/inside-look-automotive-ethernet-protocol, last checked on 2019-05-28.

[25] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140, 2018.

[26] Mercedes. The world's first suspension system with "eyes". URL: https://media.daimler.com/marsMediaSite/de/instance/print/2322446-83-Fahrwerk-S-Klasse-endoc.xhtml?oid=9259968, last checked on 2019-05-28.

[27] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. 2015.

[28] M. Moroz. Guided in-process fuzzing of chrome components. URL: https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html, last checked on 2019-05-28.

[29] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. 01 2018.

[30] National Instruments. Flexray automotive communication bus overview. URL: https://www.ni.com/en-us/innovations/white-papers/06/flexray-automotive-communication-bus-overview.html, last checked on 2019-06-1.

[31] R. Nishimura, R. Kurachi, K. Ito, T. Miyasaka, M. Yamamoto, and M. Mishima. Implementation of the can-fd protocol in the fuzzing tool bestorm. In *2016 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, pages 1–6. IEEE, 2016.

[32] NXP. The lin short story. URL: https://www.nxp.com/files-static/training_pdf/29021_S08_SLIN_WBT.pdf, last checked on 2019-06-1.

[33] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security Privacy*, 3(2):58–62, March 2005.

[34] PJ Vogt and A. Goldman. Reply all #140 - the roman mars mazda virus. URL: https://gimletmedia.com/shows/reply-all/brh8jm, last checked on 2019-06-3.

[35] SAE. Sae j1939 standards collection on the web: Content. URL: https://www.sae.org/standardsdev/groundvehicle/j1939a.htm, last checked on 2019-06-1.

[36] C. Smith. Instrument cluster simulator. URL: https://github.com/zombieCraig/ICSim, last checked on 2019-05-22.

[37] C. Smith. *The car hacker's handbook: a guide for the penetration tester*. No Starch Press, 2016.

[38] SparkFun. Rgb light sensor. URL: https://www.sparkfun.com/products/12829, last checked on 2019-03-10.

[39] Sparkfun. Sparkfun isl29125 breakout arduino library. URL: https://github.com/sparkfun/SparkFun_ISL29125_Breakout_Arduino_Library, last checked on 2019-03-10.

[40] SparkFun. Sparkfun qwiic mux breakout - 8 channel (tca9548a). URL: https://www.sparkfun.com/products/14685, last checked on 2019-03-10.

[41] Synopsys. What is fuzzing: The poet, the courier, and the oracle, 2017.

[42] S. Talebi. A security evaluation and internal penetration testing of the can-bus, 2014.

[43] Texas Instruments. *TCA9548A Low-Voltage 8-Channel I2C Switch with Reset*, 11 2016. SCPS207F.

Fiche masterproef

Student: Timothy Werquin

Titel: Automated Reverse Engineering and Fuzzing of the CAN Bus

Nederlandse titel: Automatisch Reverse Engineeren en Fuzzen van de CAN Bus

UDC: 681.3

Korte inhoud:
Moderne voertuigen zijn steeds meer gecomputeriseerd and geconcerteerd. Ze bevatten een network van *Electronic Control Units (ECU's)* die verantwoordelijk zijn om invoer van de omgeving en de bestuurder te nemen, deze invoer te verwerken en het regelen van de actuators die het rijgedrag van het voertuig regelen. Deze ECU's communiceren met elkaar over voertuigcontrolenetwerken zoals het *Controller Area Network* (CAN). Deze protocollen zijn in de jaren '80 ontwikkeled zonder beveiligingsoverwegingen. In toenemende mate zijn deze ECU's ook met de buitenwereld verbonden via voertuig-naar-voertuig en voertuig-naar-infrastructuur kanalen. In de afgelopen jaren is gebleken dat deze communicatiekanalen kunnen worden ontcijferd en gemanipuleerd. Dit stelt aanvallers in staat om de verschillende subsystemen van moderne voertuigen te besturen van op afstand. Een belangrijk deel van deze activiteiten bestaat uit de handmatige reverse engineering van het communicatieverkeer over de CAN-bus, wat een tijdrovende taak kan zijn. In deze masterproef wordt een methode gepresenteerd om dit verkeer automatisch te reverse-engineeren door het toepassen van fuzzingtechnieken. De laatste jaren is fuzzing een groeiend deel van de toolbox van de veilgheidstester geworden. In de embedded sector heeft fuzzing echter een beperkt nut aangezien het definiëren van een bug orakel voor deze systemen moeilijk is. De methode die in deze masterproef wordt gepresenteerd past bestaande fuzzing technieken toe op ECU's met een sensor harnas om feedback te ontvangen. Dit sensorharnas wordt bevestigd aan de te testen ECU en detecteert fysieke reacties, die vervolgens worden gebruikt in een orakelfunctie om het fuzzingproces te informeren. We evalueren onze aanpak op drie instrumentenclusters, één virtuele en twee fysieke. Deze tests tonen aan dat de fuzzer in staat is om automatisch interessante CAN-berichten te identificeren.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Artificiële intelligentie

Promotoren: Prof. dr. ir. F. Piessens
Dr. J.T. Mühlberg

Assessoren: Prof. dr. B. Jacobs
J. Van Bulck

Begeleiders: Dr. M. Vanhoef
A. Thangarajan