Evaluating the use of black-box fuzzing for security testing the automotive UDS protocol

Alexander Nutt

2165043

MSci Computer Science
60 Credits

Supervised by Marius Muench

Word Count: 13,919

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
April 2024

Abstract

Electronic Control Units (ECUs) are embedded devices that are found within almost every modern automotive vehicle, they are responsible for controlling vital systems within the vehicle. ECUs communicate over the Controller Area Network (CAN) bus and have a multitude of different functionalities. One of the most prevalent protocols that runs on most ECUs is the Unified Diagnostic Services (UDS) protocol, its main use is to help testers diagnose issues with the ECU as well as update and test software. By design the CAN bus and many of the protocols running on ECUs do not have many (if any at all) security features, this has caused a spike in the research of their security in the past decade which has shown the different effects that cyber attacks on these ECUs can have.

Fuzz testing (or fuzzing) is a software testing technique, commonly used in security, that involves sending malformed or random data to the system under test. This project aims to evaluate the effectiveness of the use of black-box fuzzing for testing the security of the UDS protocol on automotive ECUs. We analysed the UDS protocol and the functionality of the ECU to design a custom fuzzing tool. We fuzzed our ECU using our different methods for test generation and feedback and analysed and evaluated our findings. We found several bugs within the implementation of the UDS protocol and concluded that black-box fuzzing is effective in testing the UDS protocol but there is more work that could be done in finding potential vulnerabilities in the protocol.

Acknowledgments

I would first like to thank Marius Muench for supervising this project and for his support and advice throughout the year. I would also like to thank my inspector Tom Chothia for his helpful feedback and advice during my demonstration regarding what certain points to focus on in this report. I would also like to thank Matthew Leeke for holding final year project support lectures that proved to be extremely helpful in structuring this report and preparing for my demonstration. Finally, I would like to thank the reader for taking the time to read this report, however brief.

Code Repository

The code for this project can be found in the following GitLab repository `https://git.cs.bham.ac.uk/projects-2023-24/axn043`

Contents

1 Introduction

Automotive vehicles have become more complex, containing more ECUs (Electronic Control Units), in the past two decades. These ECUs control the functionality of many different systems within a vehicle including many safety critical systems such as the anti-lock braking system and the airbag system. The security of such ECUs is extremely important as the malfunction of them due to malicious attackers can compromise the safety of the vehicle and put the driver and passengers at risk of harm. This is why the testing of ECUs during their development as well as the research into their security is important, so that any vulnerabilities can be found and patched before they are exploited by attackers for malicious intent. This is the main motivation for this project which will use certain security testing methods to try and find vulnerabilities within a given ECU.

Many techniques can be used to test the security of ECUs, many of which use the analysis of the firmware running on the ECU. The firmware of ECUs is most commonly obtained via its extraction from the device itself, which can be done in numerous ways [Van den Herrewegen, 2021]. These methods are beyond the scope of this project and therefore we will focus on black and grey-box testing methods. The main testing method that we shall utilise is fuzzing which has been shown to be very effective in finding bugs and vulnerabilities within a wide range of applications and systems [Liang et al., 2018]. Fuzzing involves sending random and malformed data to the system to discover edge cases and bugs that could potentially lead to an exploitable vulnerability.

ECUs have many different attack vectors that can be used to compromise the security of automotive vehicles [Sommer et al., 2019], the most common ones use access to the CAN (Controller Area Network) bus, which is how the ECUs within a vehicle communicate with each other. In real-world situations, it is extremely easy for attackers to connect to the CAN bus via the OBD (On-board Diagnostics) port which is normally accessible under the dashboard. One prominent protocol that is accessible on the CAN bus is the UDS (Unified Diagnostic Services) protocol. This protocol is used by manufacturers and mechanics to test and diagnose the function of ECUs as well as update firmware and access the internal memory of the ECU. This makes it a good target for fuzz testing as the implementation of the protocol could contain bugs that could be exposed via fuzzing.

1.1 Project goals

The overall goal of this project is to use fuzzing against the UDS protocol to find any unexpected changes in the operation of an ECU. We aim to use our knowledge of the CAN bus and the UDS protocol to develop a fuzzer that can effectively fuzz the UDS implementation running on an ECU. Through fuzzing, we hope to find messages that cause crashes or changes in the ECU that could lead to further vulnerabilities and further analyse these messages to potentially find causes of the crash. We will further build on previous research and demonstrate that black-box fuzzing is an effective security testing method against the UDS protocol.

1.2 Project Overview

The rest of this project will be structured as follows: Section 2 - Literature review - will explore the previous research and work related to the CAN bus, UDS, fuzzing and automotive security. Section 3 - Methodology - will explore the methodology used throughout the project and how we approached the design of the fuzzer. Section 4 - Set up and initial communication - will explore the hardware and basic software used to set up the system and how we achieved initial communication with the ECU. Section 5 - Implementing a UDS Fuzzer - will explore the design and implementation of the main fuzzer that we use. Section 6 - Using an industrial Fuzzer - will explore the use of a more industrial fuzzer for the input generation that has been developed over several years and how it differs from our methods. Section 7 - Project management - will explore how this project has been managed and describe any issues we have

had as well as how we overcame them. Section 8 - Results & Evaluation - will discuss the results from both the fuzzers and evaluate how our fuzzer performed. Finally, Section 9 - Conclusion & Future work - will conclude the report and discuss any advancements that could be made on the project and what some future work in the area might be.

2 Literature Review

2.1 The CAN Bus

The Controller Area Network (CAN) bus is a widely used network bus within automotive vehicles. There are two commonly used versions of CAN: CAN 2.0 and CAN FD (Flexible Data-Rate) [ISO 11898-1:2015]. The CAN bus is a decentralized multi-master communication protocol where each Electronic Control Unit (ECU), also known as a node, can send and receive data to one another without having to have a central node that deals with the transmission of all the traffic. This is why CAN is commonly used within the automotive industry as real-time performance is essential.

The CAN bus is physically implemented using two wires, one for the HIGH signal and one for the LOW signal, these are used to transmit the differential signal that represents the transmitted data. The HIGH wire's voltage ranges from 2.5v to 3.75v and the LOW wire's voltage ranges from 2.5v to 1.25v, when they are both 2.5v this is a recessive signal (logical bit 1) and when they are 3.75v and 1.25v respectively this represents a dominant signal (logical bit 0). This seems contradictory to standard practices having the dominant signal be a logical bit 0 but because of how the protocol is implemented it looks for 0 values over the 1 values.

CAN 2.0 has a transmission rate from 125Kb/s to 1Mb/s with a standard rate of 500kb/s whilst CAN FD can transmit data up to 5Mb/s - 8Mb/s. The increase in the transmission rate as well as the increase of the data frame (from 8 bytes to 64 bytes) are the main two differences between CAN 2.0 and CAN FD. However, CAN FD is backwards compatible, so an ECU that supports CAN FD can still receive CAN 2.0 packets. Both CAN 2.0 and FD support the two kinds of data frame formats, that being standard frames or extended frames, the main difference between the two data frame formats is the length of the CAN ID that they use, standard frames use an 11-bit ID whilst extended frames use a 29 bit ID. CAN IDs (also known as arbitration IDs) are used to determine the priority of messages sent on the bus, with lower IDs having higher priority than higher IDs (because 0 bits are dominant and 1 bits are recessive), different higher-level protocols (such as UDS) will have set CAN IDs that they use but these are generally specific to each ECU. An ECU may receive many messages at once so it'll perform a bitwise comparison on the IDs of the received messages, if one ID has a 0 bit in a position and another has a 1 bit in that same position the ECU will prioritize the first message since on the CAN bus 0 bits overwrite 1 bits. The data frame will also include up to 8 bytes of data (64 for CAN FD) and then several other fields, these include the Data Length Code (DLC) which denotes the length of the data field in bytes, the Cyclic Redundancy Check (CRC) which is a 15-bit code used to check for any accidental errors that may occur during transmission, this code does not, however, provide any form of security and is a fairly basic error detection method. CAN also supports three other types of frames, those are: error frame, overload frame and remote frame. These are used for detecting any errors, delaying data or remote frames and requesting the transmission of a specific ID respectively, For this project we will only focus on the data frame and only focus on the ID and data fields of the data frame.

When the CAN protocol was developed not much thought was put into its security for two main reasons, manufacturers did not think it was very necessary and that adding any form of security would have added extra complexity to the protocol which would have not made it ideal for fast real time processing within the automotive context. The CAN bus is very easily accessible via the On-Board Diagnostics (OBD) port, it's important that it is accessible so that people such as mechanics can connect to the bus to troubleshoot the different ECUs within a vehicle, however, this does also mean an attacker can easily

connect to the bus and read and manipulate CAN traffic. This can have several potential impacts on the operation of the vehicle and can potentially be dangerous to users of the vehicle and others on the road. This is why it is important that when ECUs are being designed and developed they are being tested for security and this is one of the main purposes of this project.

2.2 The UDS protocol

The UDS (Unified Diagnostic Services) protocol is a diagnostics communication protocol that is widely used in ECUs within automotive vehicles. The protocol is server-client based where the client is typically a piece of testing equipment used by mechanics and/or engineers to diagnose issues with ECUs on the CAN bus, the client is therefore colloquially referred to as the tester. The protocol is an international standard rather than differing between manufacturers [ISO 14229-1:2020]. The protocol consists of a set of services that each provide a purpose that can help testers diagnose any issues, test certain functionalities or reprogram an ECU. The CAN ID for the UDS protocol is not standard, however, ISO 15765-4:2021 specifies that the CAN IDs for UDS on OBD (On-board Diagnostics) should be in the range of 0x7DF – 0x7EF [ISO 15765-4:2021]. Most ECUs support UDS however sometimes an ECU will not support the protocol. Below we go more in-depth into the different aspects of the protocol.

2.2.1 Request message structure

Every message that the tester sends to the ECU (also sometimes called the "server" in this context) must contain at least three things: the relevant UDS CAN ID so that the server knows that the tester is sending a UDS request, the service ID (SID) of the service that the tester would like to use and the data for the request. UDS requests also include a Protocol Control Info (PCI) field that is required by the CAN-TP standard that specifies how diagnostic messages are sent over the CAN bus [ISO 15765-2:2016], this field's main use is when a message is being sent that cannot fit within a single CAN frame i.e. are greater than 8 bytes, in the case that the request can fit within a single CAN frame the field simply tells the server the length of the request.

Every service that UDS provides has a standardised SID that the tester uses to request that service, these can range from 0x10 to 0x3E. After the SID the tester will define the sub-function that they would like to invoke. Sub-functions are supported by some services to enable further functionality for that specific service or to specify what you would like the service to do e.g. in the Diagnostics Session Control service the sub-function byte is used to determine which type of session the tester would like to enter. The sub-function byte has another use that services that don't support sub-functions can also use, the first bit is used for positive response suppression, if the first bit is set to 1 then the server will not send any positive responses for that request; negative responses, however, cannot be suppressed.

The final part of the request is the data that the tester wants to send with the request. The meaning of this data depends on what service is being requested, it may be the memory address the tester is requesting to read, or it may be the memory address followed by the data that the tester would like to write to said memory address. Sometimes data is not required, in these cases, the rest of the message is just padding. Here is an example request for the ECU Reset service:

0x7DF	0x02	0x11	0x01	0x00	0x00	0x00	0x00	0x00
CANID	PCI	SID	SubFunction	———	———	Padding	———	———

2.2.2 Response message structure

The structure of the response from the server is very similar to the request sent by the tester. The first part is the PCI, which in most cases will tell the tester how long the response is. The next parts depend on whether it is a positive or negative response. For positive responses, the next byte is the SID of the service request plus 0x40 e.g. for ECU reset (0x11) the positive response SID will be 0x11+0x40 = 0x51.

NRC	Description
0x10	General reject
0x11	Service not supported
0x12	Sub-function not supported
0x13	Invalid message length/format
0x14	Response too long
0x21	Busy- repeat request
0x22	Conditions not correct
0x24	Request sequence error
0x25	No response from subnet component
0x26	Failure prevents execution of requested action
0x31	Request out of range
0x33	Security access denied
0x35	Invalid key
0x36	Exceeded number of attempts
0x37	Required time delay has not expired
0x70	Upload/dowload not accepted
0x71	Transfer data suspended
0x72	Programming failure
0x73	Wrong block sequence counter
0x78	Request received - repsonse pending
0x7e	Sub funtion not support in active session
0x7f	Service not supported in active session

Figure 2.1: Negative Response Codes for UDS

After the SID the response will include the sub-function requested (if any) followed by the response data (again if any). For negative responses, the byte after the PCI will be 0x7F, this is the negative response SID and tells the tester the request was not accepted, this is followed by the rejected SID and then the Negative Response Code (NRC) which provides information regarding why the request was rejected. See Figure 2.1 for the list of NRCs. For both positive and negative responses padding is provided at the end to fill the CAN frame.

2.2.3 Services

Here is a summary of some of the key services that are typically made available on UDS.

Diagnostic session control: The diagnostics session control service has a SID of 0x10 and is used to switch between the different diagnostic sessions (described in section 2.2.4). The sub-function byte is used to denote the session the tester wants to switch to. An example request will look like 0x10 0x02 0x0 0x0 ...

ECU reset: The ECU reset service has a SID of 0x11 and is used to restart the ECU. The sub-function byte is used to denote the type of reset the tester wants to initiate. The types of reset are 0x01 = Hard reset, 0x02 = key off reset (simulates an ignition off-on with the key), and 0x03 = Soft reset.

Security Access: The security access service has a SID of 0x27 and is used to unlock the ECU to enable the use of certain services that are locked initially. The tester will request a seed from the server with sub-function 0x01 (requestSeed), the tester will then calculate the key and send it to the server with sub-function 0x02 (sendKey). If the key matches the key the server has then the server will respond with a positive response and unlock itself.

Communication control: The communication control service has a SID of 0x28 and can be used to turn on and off the ECU's ability to send and receive messages

Tester Present: The tester present service has a SID of 0x3E and is used to tell the server that a tester is still present to reset the timeout timer for sessions. Acts as a "heartbeat" to signal that the tester is still there.

Access timing parameters: The access timing parameters service has a SID of 0x83 and is used to view and change the timing parameters that the server uses when communicating with testers e.g. timeout parameter

Control DTC Settings: The control DTC settings service has a SID of 0x85 and is used to enable or disable the detection of errors within the ECU. DTC stands for Diagnostic Trouble Code and they are used to help diagnose errors within the ECU

Link Control: The link control service has a SID of 0x87 and is used to set the baud rate of the UDS implementation of the ECU.

Read Data by Identifier: The read data by identifier service has a SID of 0x22 and is used to read data values from the ECU based on a DID (Data IDentifier). Different DIDs will hold values like the software version, the VIN (Vehicle Identification Number) etc. DIDs range from 0 to 65535.

Read memory by address: The read memory by address has a SID of 0x23 and is used to read the physical memory at the provided address.

Write data by identifier: The write data by identifier service has a SID of 0x2E and is used to write data to a given DID.

Write memory by address: The write memory by address service has a SID of 0x3D and is used to write data to a given physical memory address on the ECU.

Clear diagnostic information: The clear diagnostic information service has a SID of 0x14 and is used to delete all stored DTCs (Diagnostic Trouble Codes).

Read DTC information: The Read DTC information service has a SID of 0x19 and is used to read the DTC (Diagnostic Trouble Codes). More details on DTCs are provided in section 2.2.5

Request download: The request download service has a SID of 0x34 and is used to download new data e.g. software into the ECU. The tester specifies the location and size of the data. The data is sent with the transfer data service.

Request upload: The request upload service has a SID of 0x35 and is the opposite of the request download service. The tester will specify the location and size of the data, the data is then transferred from the server to the tester via the transfer data service.

Transfer data: The transfer data service has a SID of 0x36 and is used for both uploading and downloading data. It is used after using either request upload or download services and is used for the data.

Request transfer exit: The request transfer exit service has a SID of 0x37 and is used to "exit" the transfer of data when data transmission is finished. A server will respond negatively to stop the data transfer request.

Request file transfer: The request file transfer service has a SID of 0x38 and is used to initiate a file transfer between the tester and the server in either direction.

2.2.4 Sessions

Diagnostic sessions in UDS help group services together into logical containers, certain services are allowed to run in one session but may not be able to run in another, for example, Request download is unable to run in the default session but can run in the Programming session. A tester can switch between these sessions by using the Diagnostic Session Control service and using the sub-function byte to determine the session type. There are 4 standard sessions defined in the protocol: 0x01 - default session, 0x02 - programming session, 0x03 - extended diagnostic session and 0x04 - safety system diagnostic session. The default session is as it sounds, it is the default session that the ECU is in when it boots up and/or after another session times out, the majority of services are available within this session however some that require security access or involve the alteration of the ECU require a non-default session. The programming session is related to services that involve the reprogramming of the ECU or the writing of certain memory locations. The extended diagnostic session allows for the same services as the programming session but is also reserved for sessions that alter the behaviour of the ECU itself, these tend to be the sessions that require security access because they could potentially break the ECU if used improperly. Finally, the safety system diagnostic session is used for services required to support safety-related functions e.g. the airbag; this service however tends to be ECU specific e.g. the ECU that controls the airbag system. The server will remain in its current session unless one of two things happens: the tester sends a diagnostic session control request to switch to a different session or no requests are sent and the ECU times out and returns to the default session. A tester can remain in its current session without timing out by sending a Tester Present request which tells the server that a tester is present and resets the timer. These sessions are not designed to act as security measures, services that require security should rely on the Security Access service.

2.2.5 Diagnostic Trouble Codes (DTCs)

Diagnostic Trouble Codes (DTCs) are a set of codes used by a vehicle's diagnostics system (sometimes a single ECU or multiple ECUs) to alert the driver or tester that the vehicle has experienced a malfunction of some form. When an error is detected it is normally sent to the instrument cluster which then lights up the corresponding light on the dashboard to indicate the specific error to the driver. DTCs can be split into three bytes: the first byte tells us where the fault occurred (powertrain, chassis, body or network), denoted by the first 4 bits, and whether the code is a standardised code or manufacturer specific, denoted by the last 4 bits. The second byte of the code tells us what system/component of the vehicle failed and the last byte indicates the specific type of failure that has occurred. DTCs can be retrieved by the tester via the UDS service 0x19 (Read DTC information) with several different sub-functions to allow the retrieval of DTCs that match a specific requirement such as DTCs that match a status mask which allows the tester to get DTCs that are confirmed, pending, test has failed etc. DTCs are extremely useful in

diagnosing issues within ECUs and will come in handy when fuzzing as we will be able to use them to detect whether a fault has occurred due to fuzzing UDS.

2.3 Fuzzing

Fuzzing is a software testing technique that is commonly used to find bugs and vulnerabilities within software and systems [Liang et al., 2018]. The software, or "fuzzer", will generate malformed inputs and automatically send them to the system, the inputs are generated by a message generator and can either be randomised data or generated based on some set of rules. There are three main challenges when it comes to fuzzing, they are message generation, fail detection and reset. We will go over some of the common methods for message generation below. The fuzzer must also be able to detect when the system under test (SUT) has failed. The solution to this challenge is normally problem specific and we will go over how we plan to tackle it along with test generation in section 5. As for the resetting of the SUT we can use the ECU Reset service.

2.3.1 Types of fuzzing

Fuzzing methods can be categorised into three main groups: black-box, grey-box and white-box. These terms are commonly used in the context of software testing [Luo, 2001]. In black-box fuzzing the fuzzer assumes no knowledge of the SUT, such as how the input should be structured or how it processes the input, and views it as a "black-box", input goes in and then it monitors the result [Takanen et al., 2018]. White-box fuzzing is the opposite, here the fuzzer assumes all knowledge of the SUT such as how the input is structured and how it's used, the fuzzer also has access to the source code and can use symbolic execution to its advantage [Godefroid et al., 2008]. But for this project, we will be focusing on black and grey box fuzzing as we do not have access to the source code or binary of the ECU and gaining access to those and analysing them is out of the scope of this project. Grey-box fuzzing is in the middle of black and white box fuzzing, the fuzzer will have some knowledge of how the SUT works. Normally the fuzzer will know how the input is structured, and how a protocol works and/or will have a model for the SUT and can use methods like mutational fuzzing to get the best code coverage [Böhme et al., 2016].

2.3.2 Mutational fuzzing

Mutational fuzzing is a method of fuzzing that builds on simply generating random data for the SUT. A mutational fuzzer will take one or more valid inputs, known as seeds, for the SUT and use these valid inputs to make more inputs by applying different mutations to them, these new inputs may be valid or may be invalid. The chances of the new inputs being valid are greater than the chance of some random data being valid because it takes an already valid input and changes it slightly. There are many different ways a fuzzer can mutate a valid input, they may be randomising a section of the valid input, flipping a set number of bits, changing bits based on a model or performing some computation on some bits based on some predefined or uniformly chosen variables [Woo et al., 2013]. Mutational fuzzing improves on using random data because the inputs have a higher chance of being valid, therefore, there is a greater chance of more code coverage. A common and popular fuzzer that uses mutations is the American Fuzzy Lop (AFL) fuzzer from Google [Google, 2013] and its successor AFL++ [Advanced Fuzzing League++ , 2017]. Mutational fuzzers can be improved by using feedback from the SUT to help decide how the inputs are mutated. This is a form of directed fuzzing and allows for even more code coverage, it is more commonly used with white-box methodologies like symbolic analysis [Cha et al., 2015] since black-box fuzzing doesn't provide as much feedback from the SUT.

2.3.3 Directed fuzzing

Directed fuzzing is a method of fuzzing that uses its knowledge of the SUT or feedback from the SUT to generate inputs to cover specific paths of the code. By doing this the fuzzer can target certain parts of the SUT that may allow it to trigger more vulnerabilities. For example, the fuzzer may know that one section

of the code doesn't have much functionality but can see that another section has more functionality, and therefore a higher probability of having a bug. The fuzzer can then use its knowledge to generate an input to reach that section of the code. There are many different ways that a fuzzer can be directed, the most common method of directed fuzzing is to use symbolic execution which is a white-box method that analyses the program and uses constraint solving to create inputs that execute different paths within the program. An example of directed fuzzing that doesn't require symbolic analysis involves using simulated annealing, a simple machine learning optimization algorithm, to help generate test inputs that are "closer" to a target within the program structure [Böhme et al., 2017]. Directed fuzzers can also use feedback from the SUT to change the way that it generates inputs, one example of this was shown in [Mathis et al., 2019] where they used feedback from the input parser to effectively cover the input space and produce a high number of quality inputs for the SUT. This method allowed them to avoid the issues like lack of code coverage and low quality inputs that come with traditional and constraint based fuzzing methods for non-trivial languages. Directed fuzzing can be used by itself but it can also be combined with other techniques such as binary analysis to ensure overall code coverage.

2.3.4 Grammar-based fuzzing

Grammar-based fuzzing is a method of fuzzing that uses the knowledge of the structure of the input for the SUT to systematically generate inputs to achieve greater code coverage. The fuzzer will use a grammar, such as a BNF, as a predefined set of rules that it will use to generate a wide range of test inputs for the SUT. It has been shown that using a grammar to produce inputs can lead to improved code coverage [Sargsyan et al., 2018]. The use of a grammar allows the fuzzer to generate inputs that match the syntax of the SUT meaning there is a greater chance of the input being valid and being accepted by the SUT. Grammar fuzzing can be further extended by allowing the fuzzer to prioritise certain values for the input with probabilistic grammar fuzzing. This allows us to assign probabilities to the different values within the grammar so that the fuzzer will generate inputs that prioritise the values in the grammar that have higher probabilities. This will allow us to get more specific inputs from our fuzzer and target certain parts of the SUT. Using more specific inputs is a form of directed fuzzing and we can expand on this more by optimising the probabilities associated with our grammar by using machine learning techniques [Eberlein et al., 2020]. By using machine learning techniques alongside fuzzing we can generate more sophisticated inputs that have a higher chance of triggering bugs and vulnerabilities within the SUT.

2.4 CAN and UDS Security

As previously mentioned the CAN bus and automotive technology in general were not designed with security in mind and it has only been in the last decade that the topic of automotive security has become more prevalent. With [Koscher et al., 2010] and [Checkoway et al., 2011] showing that the CAN bus was vulnerable to attacks that could cause changes to the car including to safety critical systems, it became clear that more work needed to be done in securing the CAN bus and subsequently other parts of the automotive technology stack such as higher level protocols.

2.4.1 CAN fuzzing

In the past decade, there has been a lot of research into the security of automotive technology, and one of the most popular approaches has been fuzzing. Since the details of how specific cars and ECUs work are very proprietary, and it is easy to connect to the CAN bus, fuzzing becomes one of the best options for quick security testing of these ECUs as we can treat them as a black-box and still get good results from a black-box based fuzzer. [Wong, 2023] showed that black-box fuzzing was an effective way to test the security and start to reverse engineer ECUs, the approach they used for test generation was to use random bytes,= so using a different technique like grammars or mutations could lead to further results. It was also shown by [Fowler et al., 2018] that fuzzing can lead to the breaching of certain security measures within the SUT, they showed that fuzzing was effective in unlocking the doors to a car in no more than

1 hour. The fuzzing of the CAN bus doesn't have to be black-box however, as discussed fuzzing can be combined with other reverse engineering techniques to get better results. [Radu and Garcia, 2020] demonstrated that they could use the control flow data from the firmware extracted from an ECU to guide the fuzzer and showed that this method would produce better results than traditional fuzzing with random data. A lot of work has been done with regards to fuzzing the CAN bus but this project will delve deeper by looking at the effectiveness of fuzzing the UDS protocol.

2.4.2 UDS security

The security access service (0x27) that UDS has is known for having poor cyber security as the cryptographic cipher that the ECU uses is not standardised, it is up to the manufacturer what cipher they use to create a key from a seed. This means that some ECUs will be less secure than others and could be vulnerable to common attacks such as timing and different side channel attacks [Standaert, 2010]. In the 2020 revision of the UDS standard, they introduced a new service, Authorisation 0x29, to improve on service 0x27. This service uses certificate-based asymmetric authentication, however, for more legacy ECUs using this service isn't feasible. [Thompson, 2022] describes a design that legacy ECUs can use to implement a practical UDS Security Access service. Having a secure cipher can protect from attackers that only have communication access with the ECU, however, if an attacker can obtain the firmware from the ECU they can analyse the cipher and reverse engineer it. This idea is shown by [Van den Herrewegen and Garcia, 2018], they demonstrated that they could reverse engineer the ciphers from the extracted firmware of several major car manufacturers, once authenticated they could perform remote code execution and write new firmware. This shows that if an attacker can reverse engineer the firmware of an ECU it can be easy to gain full control of the ECU.

To conclude, the current research shows that fuzzing is effective against the CAN bus as a whole and that UDS is vulnerable to different attacks. This project will further this research by investigating whether fuzzing is an effective method for finding bugs within a given UDS implementation.

3 Methodology

3.1 Challenges with Fuzzing Embedded Devices

Fuzzing embedded devices, such as an automotive ECU, comes with a set of challenges that aren't present when fuzzing typical desktop-based software. [Muench et al., 2018] explains three of the main challenges that we are faced with when fuzzing embedded devices, the most prevalent of them being the issue of fault detection. Faults, such as memory corruption, in embedded systems do not present themselves in the same way that they do in typical desktop systems. Desktop systems have mechanisms such as stack canaries and reporting segmentation faults to the user to deal with such faults, however, embedded devices don't tend to have such mechanisms due to constraints such as performance or code size. This means faults tend to go unnoticed as "silent faults" which makes fuzzing particularly hard as fuzzing relies heavily on observable faults.

[Muench et al., 2018] also describes several possible solutions to some of the challenges that come with fuzzing embedded devices. One of which is firmware re-hosting. This involves hosting the firmware of the embedded device on a separate piece of hardware that allows us to take advantage of mechanisms such as memory protection units and alike. It also solves the issue of scalability as we can re-host the firmware multiple times to allow us to have multiple instances of the fuzzer running at once. Re-hosting however, has a set of challenges as explained by [Wright et al., 2021] and it is unfortunately out of the scope of this project. Furthermore, all of the potential solutions proposed by [Muench et al., 2018] require the firmware image of the embedded device which we unfortunately do not possess, therefore we will have to devise our methods of detecting potential changes in the operation of the ECU.

3.2 Analysing the UDS protocol

To fuzz the UDS protocol we first needed to analyse how the protocol works and how the messages are structured so that we could effectively fuzz its input space rather than wasting time fuzzing inputs that would not do anything in the context of the UDS protocol. This required finding good sources of information on the protocol, with international standard protocols we can find the ISO document, luckily we were able to find a freely available version online. The document was extremely useful in understanding the structure of the protocol and how different messages are interpreted for the different available services. However, because the standard only defines the functional requirements of the protocol and not any requirements relating to the implementation we cannot analyse how the protocol is implemented on the ECU and use this to guide the development of our fuzzer.

When analysing the structure of the protocol we focused on the structure of the requests as that is what will guide our development of the fuzzer. We also made sure to make note of the different "states" that the ECU can enter during UDS sessions such as an authenticated "state" or a data transfer "state" and what responses the ECU could potentially respond with. Only a couple of bytes within the response will potentially be used by the fuzzer, the rest of the response can be analysed after the fuzzer has run, if the fuzzer deems the response interesting. We found that all the different services follow the same standard structure for requests and that the main difference is the context of the data. We could use this information to more efficiently fuzz specific services provided by UDS, however, this is out of the scope of this project but could potentially be further explored in future work.

4 Set up and initial communication

4.1 Hardware and Software

4.1.1 Hardware

Figure 4.1 shows the ECU that we will be using within this project, this specific ECU is a Body Control Module (BCM) for a Ford Fiesta. The purpose of the BCM is to monitor, control, and coordinate various electrical systems within the vehicle such as the locks on the doors and windows, the lighting, and power distribution. Because the BCM has no indicators/does not change physically during its operation it makes it a good ECU to use for black-box fuzzing, but we are still able to get some form of feedback by monitoring the traffic and checking for error codes, etc. so we can use these to develop a black-box fuzzer with lightweight feedback methods. More on these methods in section 5. The ECU has the power pins as well as the pins for CAN High and CAN Low, we connect the power pins to an external power supply and provide 12V and around 0.53A. We can interface with the ECU using a laptop by connecting a PeakCAN USB connector which allows us to use the USB port on a laptop to connect to the CAN pins on the ECU.

4.1.2 Software

We connect to the ECU with a laptop running a 64-bit version of Ubuntu. Using Linux, compared to Windows, allows us to use SocketCAN and can-utils. SocketCAN is an open-source implementation of the CAN protocol built into most modern Linux kernel implementations, it uses the socket API and Linux network stack and has enough abstraction that it allows communication with a range of different ECUs. We also use can-utils which is a Linux package that provides multiple different utilities for communicating over the CAN bus. The three most notable ones are cansend, cangen and cansniffer, these can be used to send, generate CAN data and display current CAN messages being received. We also used Wireshark to easily display and save batches of messages that are sent and received, Wireshark was very useful due to the ability to filter packets as the ECU responds with a lot of CAN packets for every message it receives, so being able to filter the packets made the development of the fuzzer easier as we could see if certain packets were being sent correctly. Finally for the development of the fuzzer we used the Python library python-can which provides us with the functionality to send, receive and process CAN packets in Python.

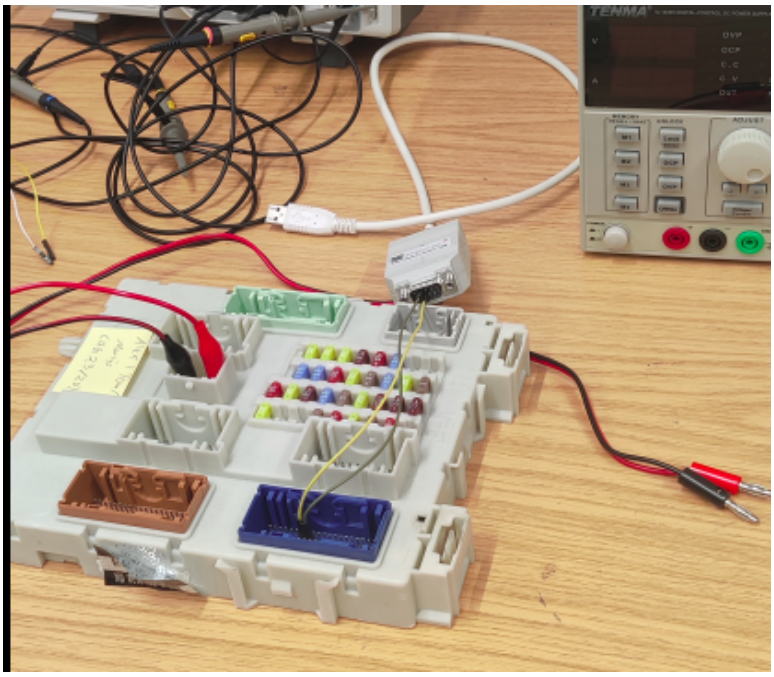

Figure 4.1: Fort Fiesta BCM

4.2 Setting up communication

To connect to and communicate with the ECU we first need to plug in the PCAN-USB device and then set up the SocketCAN interface with the Linux command

```
$ sudo ip link set up can0 type can bitrate 500000
```

This will set up a CAN interface called can0 with a bitrate of 500kb/s which is the standard bitrate for CAN 2.0. Once this is done we can send and receive CAN messages. We can next set up Wireshark to be able to capture and save the CAN messages that are sent and received by the ECU. Wireshark allows us to select the can0 interface that we just created and we can then use a filter can.flags.err == false to make sure it will only display successful messages otherwise, we get a constant stream of ERR packets that make it difficult to read and understand the packets that we are capturing.

Once everything is set up correctly we can start sending CAN messages to the ECU, we can do this quickly and easily via the terminal using the cansend command, which is part of the canutils package. With cansend, we specify the interface we wish to send messages over, in our case it is can0, and then we specify the CAN ID and data we wish to send. An example message being sent via cansend looks like this.

```
$ cansend can0 101#083df18c02a000fe
```

where 101 is the CAN ID we are sending to and the data following the # is the data we are sending in the CAN packet. All values here are in hexadecimal form.

When we send messages to the ECU we see that the ECU responds with a stream of messages from the following IDs 0x04, 0x030, 0x17e, 0x260, 0x380, 0x3b4, 0x420, 0x435, 0x0c8, 0x310, 0x360, 0x150, 0x290, 0x40a, 0x405, 0x400, 0x581. We later noticed that these IDs were also used in sending messages from the ECU when the ECU first powers on. We saved the captures from Wireshark to multiple text files and then used the Linux diff command to see what differences there were between them. We found that

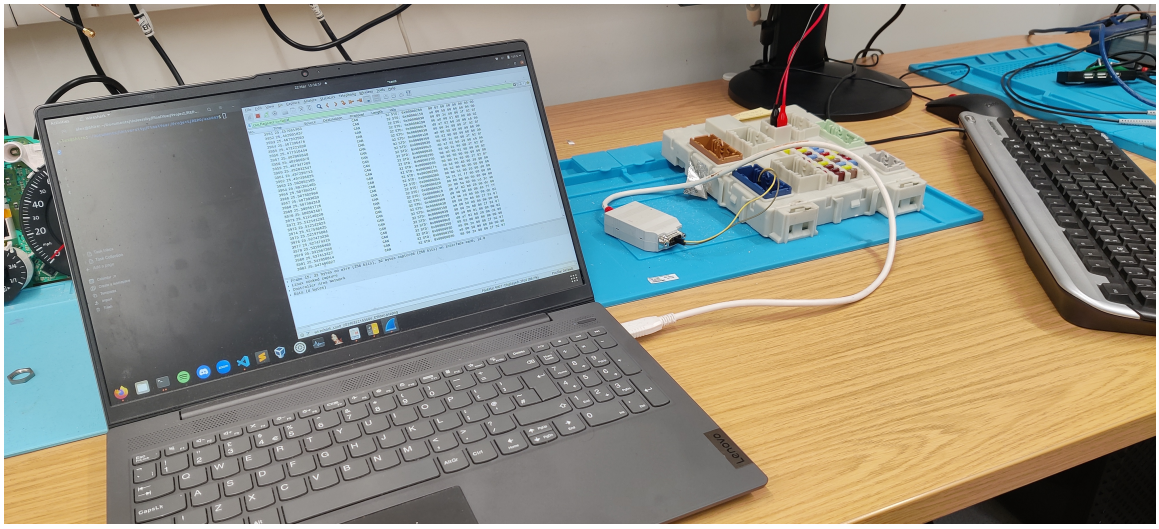

Figure 4.2: BCM connected to the power supply and the laptop

there were few differences between them and that all the IDs mentioned were used every time the ECU powered on or received messages, there were only a small number of differences in the sequence of the messages and the data being sent. The ECU would cease transmitting messages around 10 seconds after booting up or receiving a message and would not transmit anything else until it received a new message.

After the initial investigation of the ECUs' communication with cansend, we moved on to use Python so that we could start to write scripts that would send messages automatically rather than having to execute singular commands manually. We used the python-can library which provides the ability to send, receive, listen for and filter CAN messages with enough abstraction that the library will work with any device using the CAN protocol. At this point, we just focused on sending and receiving CAN messages but we knew that the ability to filter would come in handy later in the project. We wrote a small script that would send a couple of messages to the ECU and receive the first CAN message that was transmitted back, this was purely to test the library's functionality to confirm it worked how we expected it to work.

4.3 Finding CAN ID and UDS Services

4.3.1 Finding CAN ID for UDS

Before we can start to work with UDS and start fuzzing it first we need to find the CAN ID that the ECU uses for UDS, the CAN ID is not standardized with the rest of the protocol however as mentioned before the specified range of the CAN ID for UDS on OBD is 0x7df - 0x7ef. We decided to expand the range of our search for the CAN ID to 0x700 - 0x7FF. We started with 0x7ff and worked down as diagnostics protocols tend to have a lower priority and therefore will have a higher CAN ID. We looped through each CAN ID sending a TesterPresent request (UDS SID 0x3e) and looked for the appropriate reply, which would be 0x7e 0x00. We discovered that UDS on this ECU has the CAN IDs of 0x726 and 0x7df with an RX (response) CAN ID of 0x72e. With this information, we can start work on communicating with UDS and use the RX CAN ID to filter messages that are just from UDS, which will be helpful for our next step: finding what services are available on this instance of UDS. This script was written using the udsoncan library rather than python-can because initially we thought using this library specifically for UDS would provide a better development experience, however, we soon realised that the further abstraction that this library provides would cause hindrance and that it would also not allow us to monitor the CAN bus in general, so after the development of this initial script we switched to just using python-can.

4.3.2 Finding available UDS services

Before we start to fuzz UDS on the ECU we need to find out what services are available on this instance of UDS as not all ECUs running UDS have all the specified services implemented for various reasons such as the purpose of the ECU doesn't require said service. It's best to find out what services are available otherwise we may waste time fuzzing a service that isn't implemented on the ECU. Our method for finding out what services are available is to loop through the list of SIDs set out in the standard and send varying requests to them to ensure that we get a response from the ECU. Even negative responses can mean that the service is available, for example, we may get the NRC 0x13 which tells us our message was incorrectly formatted but this still means the service is available. We will know if the service is unavailable if we get the NRC 0x11: Service not supported, or if after multiple tries of communicating with the service, we get no response at all. We start by sending blank data requests and then move on to send requests that are all ones i.e. FFFF..., we also use data with a single byte 0x01. We repeat this at least 5 times as sometimes the ECU will not respond the first time. If after 5 attempts we have no response we assume that the service is not supported, if we get a response we check to see if it was a negative response with the NRC 0x11, if it was we remove the SID from the list, otherwise we keep it. We save all supported SIDs to a file that will be used by our fuzzer to generate fuzz inputs.

Now that we've found the CAN ID for UDS and discovered what services are available on this instance of UDS we can start the development of a fuzzer to generate inputs and monitor the response and behaviour of the ECU to try and find interesting inputs that cause unexpected behaviour.

5 Implementing a UDS Fuzzer

5.1 Test Generation

There are many different ways that we can generate test cases for our fuzzer, some of which however, lend themselves more to white-box fuzzing as they require more information about the SUT such as the firmware or feedback from the SUT that provides more information than just the basic response. As we have to treat our ECU as a black-box we have to use test generation methods that work with black boxes. For our fuzzer, we will use two of the most common black/grey-box test generation methods, mutations and grammar-based test generation.

5.1.1 Grammar-based test generation

The first method of test case generation that we decided upon was grammar-based test generation using a Backus Naur Form (BNF) grammar, see Figure 5.1. The grammar allows us to define the structure of our fuzzing input so we can generate a range of test inputs for our ECU. We define the grammar using a simple dictionary that defines the non-terminal symbols as the keys and the values that they can expand to as the values. Each non-terminal symbol can expand into 1 or more non-terminal symbols or terminal symbols, which will define the actual bytes that we will send to the ECU. By default the SIDs that the grammar can use are all the SIDs that are listed in the UDS specification, however, when the fuzzer is run it will use the list of available SIDs that we discovered in section 4. We also defined a separate "⟨hex-2⟩" symbol that is used by the subfunction byte in the request, this is because the UDS specification tells us that if the first bit in the subfunction byte is set then positive responses will be suppressed, and we don't want this to happen as it will reduce the quality of our results as we will potentially miss out on positive responses that could give us interesting information. Finally, we added a probability feature that allows the code that expands the BNF expression to potentially favour different expansions for a given non-terminal symbol. The main purpose of this is to allow for varying lengths of data, rather than the data being of static length. The probability feature could potentially be expanded to be used with the SIDs and allow for the favouring of certain SIDs that prove to provide more interesting results.

There is some room for improvement with our implementation of the grammar. One improvement

⟨*start*⟩ ::= ⟨*udsReq*⟩

⟨*udsReq*⟩ ::= ⟨*SID*⟩⟨*subfunction*⟩⟨*data*⟩ | ⟨*SID*⟩⟨*data*⟩

⟨*SID*⟩ ::= `10` | `11` | `27` | `28` | `29` | `3e` | `83` | `84` | `85` | `86` | `87` | `22` | `24` | `2a` | `2c` | `2e` | `3d` | `14` | `19` | `2f` | `31` | `34` | `35` | `36` | `37` | `38`

⟨*subfunction*⟩ ::= ⟨*hex-2*⟩⟨*hex*⟩

⟨*data*⟩ ::= ⟨*byte*⟩⟨*byte*⟩ | ⟨*data*⟩⟨*byte*⟩

⟨*byte*⟩ ::= ⟨*hex*⟩⟨*hex*⟩

⟨*hex*⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `a` | `b` | `c` | `d` | `e` | `f`

⟨*hex-2*⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

Figure 5.1: UDS Grammar

may be to change how the subfunction byte is used and defined, it would be potentially better to implement some form of condition within the grammar that will mean that the subfunction byte symbol is only used if the grammar expands to an SID that we know uses the subfunction byte. Another potential improvement is to change how the data is defined, at the moment the data has a minimum of two bytes, so we potentially miss single byte payloads if we only use the grammar and don't mutate the input. The probability of the length of the data could also be improved by having it change as the fuzzer runs and sees what lengths of input tend to be the most interesting. These improvements and more will be discussed further in section 9.

5.1.2 Mutation-based test generation

The second method of test case generation that we decided upon was mutation-based test generation using a range of mutation types to mutate the UDS request that we are sending to the ECU. Mutations allow us to generate different inputs from one input (or seed), changing small or large parts of an input can lead to different results and is, therefore, an efficient way of finding different behaviours within the ECU. We have five different mutation methods that we use within our fuzzer, they are bit flipping, bitwise logical mutations, byte shifting, byte deletion, and byte addition. The bit flipping mutation method is as it sounds, it converts the input byte array into a binary string and then randomly selects a bit to flip and then flips said bit. This can be performed multiple times within one mutation call, based on what settings have been used by the fuzzer. The bitwise logical mutations method implements logical OR, AND, XOR, and, NOT operations. For the first three operations, that require two inputs, the fuzzer will generate a random byte that will then be used with a random byte from the input in the logical operation. For the NOT operation we return the absolute value of the logical NOT of the randomly selected byte from the the input. The byte shift mutation will select a random byte within the input and randomly shift the byte to the left or right between 0 and 255, for example, if the byte from the input was 0x42, and it gets randomly shifted to the left by 12 it will output 0x42000, this isn't a valid byte value so we mod the output with 256 to get a valid byte. So in our example, we get a final output of 0x28. Byte deletion and byte addition work in very similar ways. They both find a random index in the list of bytes of our input, byte deletion will make sure that the length of the input isn't one (as we can't delete any more data) and byte addition will make sure that the length of the input isn't 7 as we can't add any more data as the CAN frame is limited to 8 bytes and the first byte will be the Program Control Information (PCI) header so our UDS request must be no more than 7 bytes. Byte deletion will then delete the byte at the chosen index and byte addition will generate a random byte and insert the byte into the chosen index. As we mentioned we have to make sure that all the bytes we return have to be valid, so after each mutation on a byte we have to perform modulo 256 to make sure that the value is between 0 and 255 and is therefore a valid byte.

One issue with some of these mutations is the way that Python deals with memory and the mutability of certain data types in Python. During some of the mutations python will create a deep copy of the data that we are mutating, this takes time and will potentially significantly increase the time the mutations take and overall slow down the performance of the fuzzer. In section 8 we will explore whether or not this causes a significant slowdown in the performance and whether it is a major bottleneck of the fuzzer.

5.2 Feedback methods

Since we are treating the ECU as a black-box we don't have many options for how to gather feedback to guide the direction of our fuzzer to produce different inputs to potentially trigger more interesting functionalities. So we need to implement our own methods of getting feedback from the ECU. One very rudimentary method is just to detect whether the ECU responded (with either a positive or negative response), the positive responses will provide more information to us and will also show that our fuzzer is capable of generating valid UDS messages, we can also further analyse what the responses mean in the context of the input after the fuzzer has run to give us more information about the what has happened. We can now go into some slightly more in-depth methods of feedback.

5.2.1 Diagnostic Trouble Codes

The first method of feedback that we decided to use was the functionality that is available to us on UDS, which is the ability to read the Diagnostic Trouble Code (DTC) information. As mentioned in section 2, DTCs give us the ability to see whether or not there has been a fault somewhere within the system, this is normally the entire automotive vehicle however in our context it just means within our specific ECU. We can use the service ID 0x19 to call the service Read DTC Information, this service gives us the ability to use different subfunctions to query the DTCs in different ways. The most common way to query the DTCs is to use the subfunction 0x01 which returns the number of DTCs by a status mask. DTCs can have different statuses and we can get the DTCs with certain statuses by using a specific mask. In our fuzzer we used a mask of 0x05 which combines the statuses of pendingDTC (0x04) and testFailed (0x01), this means that when we call the service we can get the number of DTCs that have either fully "matured" and are confirmed to have failed or are pending but have still caused a fault. If the number of DTCs that it returns is greater than the number that we saw the last time we called the service we know that one of the inputs we used caused some kind of fault.

This brings us to the dilemma of how often do we call this service. The trade-off becomes whether we sacrifice accuracy for performance or performance for accuracy. The more often we call the service the more accurate our knowledge is about what input caused the fault since we have used fewer inputs since the last call, however, the more often we call the service the slower our performance will be as we will have to wait for the service to respond each time. We decided to call the service after every 10th fuzz input, this allowed us to balance performance with the ability to narrow down what inputs caused faults.

5.2.2 Timing

The second method of feedback that we decided to use was based on the timing of the response from the ECU. The idea behind this method was to see whether or not certain inputs caused prolonged processing from the ECU, we can measure the response time from the ECU and if the response time is longer than the average time that the ECU normally takes to respond by some kind of threshold then we can, with some degree of confidence, say that the input caused the ECU to take longer to process. The reasons for the ECU taking longer to process our request may be unknown but with further analysis, we may be able to come up with some theories as to why it may have taken longer to respond. Since each ECU is different we need to find the average response time first as different ECUs may take different amounts of time to respond for varying reasons.

Unfortunately, this method does come with some challenges, the main issue is that because the ECU

sometimes does not respond at all we can't just wait indefinitely for a response as this would cause the fuzzer to hang. So we have to implement some form of time out. If the time out is reached and we do not receive a response from the ECU then we mark the input as not useful otherwise we would get a lot of "useful" inputs from all the inputs that did not get a response. Also because of this caveat when we use this method, it will slow down the performance of the fuzzer slightly, this is because when we use this method the timeout is increased slightly (average response time * user decided threshold multiplier), because of this the fuzzer must wait longer before timing out when the ECU does not respond.

5.2.3 Traffic

The final method of feedback that we decided to use was based on the traffic from the rest of the CAN bus. We first record initial traffic from either the boot sequence or after the "wake up" message, this is saved to a file where we can analyse and compare the traffic we later receive. We implement the recording of the rest of the traffic by constantly listening to the CAN bus in a separate thread, we receive the messages from the CAN bus and see if we have "seen" them before, that being whether the CAN ID has transmitted that data before. If we have "seen" the message before then we log it, noting the number of times we have "seen" it, if not then we also log it but also log what the most recently used input was and log the fact that this input, with a high degree of confidence, caused this new traffic. When monitoring the CAN bus we ignore traffic coming from UDS (both transmitting and receiving) as well as messages from the CAN IDs 0x405 and 0x40a, we ignore these CAN IDs because we found that these CAN IDs transmit messages after every received message, and seemed to increment the data that they sent after each message. If we chose not to ignore these CAN IDs then the fuzzer would detect all these messages as new and we would get lots of useless data and feedback. Once the fuzzer finishes we save the new traffic to a file as well as all the traffic we recorded sorted by the number of times that we received that traffic. These can be used later to analyse the traffic and our results to see if anything particular happened to the traffic whilst we fuzzed the ECU. This method of feedback has room for expansion, we could potentially try to detect whether certain arbitrary services have been stopped or started and improve the way that we record the traffic. We will discuss these further in section 9.

5.3 Fuzzing harness

All the aforementioned parts are brought together by the main fuzzing program/harness; uds-fuzzer.py. The harness checks which options have been selected by the end tester and uses them accordingly, both test case generation methods can be used together or just one of them can be used, the same goes for the feedback methods, any combination of them can be used but at least one of them must be used. The fuzzer will check whether the file with the available SIDs is present, if not it will run the script to detect which services are available. Finally, before beginning the actual fuzzing the fuzzer will send a "wake up" message to the ECU, we found that the BCM will enter an idle state after having received no messages after a minute or so, so it requires a message to "wake it up" and get it ready to properly receive messages. To wake up the ECU we send a UDS Tester Present request in a loop until we receive a response to make sure that the ECU is no longer idle. We also send a Diagnostic Session Control request to put us in the Extended Diagnostic Session so that we can request more services and reduce the chance of getting a 0x7F (Service not available in the current session) negative response code, this message is also sent if we ever detect that we get said NRC in case for whatever reason we drop out of that specific session. The fuzzer will then run in a loop until stopped by the user (using a keyboard interrupt) and use the test case generation and feedback methods as described above. The fuzzer will save all the inputs that it uses and what inputs it has deemed as useful based on the feedback methods that it uses to two separate files, the used inputs file is used when generating new inputs to make sure that we have not used that given input before to maximise input space coverage. Both files are later used in analysing the performance and effectiveness of the fuzzer in section 8. When the fuzzer is ended by the user it will output how long it ran for, how many inputs it produced, and how many it deemed to be useful throughout the fuzzing.

6 Using an industrial Fuzzer

To expand on what we have already done we decided to use different fuzzing libraries that have had years of development by professionals in the topic to see if they could produce better or different results than our fuzzer. The two we decided to use were Radamsa and BooFuzz.

6.1 Radamsa

Radamsa [Helin, 2018] is a test case generator that takes in sample inputs and uses mutation to generate different and interesting outputs that can be used for fuzzing. It has been used to find many different bugs in a wide range of applications such as the Chrome browser and the 7-Zip program for Windows. By default it works as a command line program that reads input from stdin and then outputs to stdout, luckily there is a Python library called pyradamsa that provides an interface for calling libradamsa (a pre-compiled radamsa library) in python allowing us to easily use Radamsa's functionality within our fuzzer.

We implement Radamsa into our fuzzer by allowing the user to select the option to use Radamsa's mutation method instead of our default mutation methods. The rest of the fuzzer will work as previously mentioned, it will generate an initial UDS request either by grammar or randomly and then it will use this as the initial seed for Radamsa to mutate it and pass it to the ECU until the user stops the fuzzer. The feedback methods are the same as previously mentioned as Radamsa only provides test case generation functionality.

We found that Radamsa's mutation method was much faster and more efficient than our methods, this was to be expected since Radamsa has been under development for multiple years and by multiple professionals in the field. We will discuss the effectiveness of Radamsa compared with our methods in section 8.

6.2 BooFuzz

BooFuzz [Pereyda, 2016] is a fuzzing framework written in Python and is a fork and successor of the similar fuzzing framework Sulley [OpenRCE, 2012]. BooFuzz allows us to define our protocol definition using a request object and a set of primitives that it will use to generate inputs that fit the definition of our protocol, it is widely used for web application fuzzing such as fuzzing APIs. This framework would not work with our main fuzzer so it would have to be a stand-alone version. We first defined the structure of the CAN protocol, unlike python-can we have to establish the specifics of the CAN message ourselves such as the CAN ID, flags, and the Data Length Code (DLC), we could do this by using BooFuzz's static object as these variables will not change. Defining the rest of the message, the UDS request, was as simple as telling BooFuzz to use the set of SIDs that we have (using BozzFuzz's Group object) and then instructing BooFuzz to use a set of bytes that it can fuzz, we started with 7 as when setting it up we set the DLC to 8, so the SID plus the 7 bytes is the 8 bytes for the data frame. We could then instruct BooFuzz to use the SocketCAN interface. BooFuzz outputs its status and what inputs it's sending to the terminal as well as a web interface on the localhost.

One of the downsides to BooFuzz is that it is a bit more difficult to get feedback from the ECU, typically BooFuzz tries to detect whether the SUT has crashed or not based on monitoring the process, however, since we are dealing with black-box fuzzing it is difficult to monitor the process of the ECU to detect whether it has crashed, so BooFuzz won't be able to use its default detection and give us results on what inputs caused crashes. Despite this, we decided to see how well BooFuzz would be able to run initially before we integrated our aforementioned feedback methods. We ran BooFuzz, expecting it to run for quite some time since the input space would have been 2^{64} bits (8 bytes of data), however, it stopped running after just generating 659 inputs. We tried different combinations of the settings within BooFuzz to be able to enable it to run for longer and generate a better range of test cases, however, we

were unsuccessful and BooFuzz did not seem to work well with our task at hand.

7 Project Management

The management of this project was particularly important as most of the work had to be done within the lab which had certain opening hours, this meant that we had to plan each step of the project to maximise our time in the lab. Plans change and unexpected things happen whilst working, this is especially the case with a research project like ours, when we discover something isn't feasible or feasible within the time we have we have to change our approach. One example of this happening within our project was towards the beginning, our initial plan for the project was to retrieve the firmware from the ECU and use that to help construct our fuzzer using techniques like symbolic execution. However, after several weeks of being unsuccessful in finding a way to extract the firmware we decided to use black-box methods of fuzzing rather than spending too much time trying to extract the firmware and potentially being unsuccessful and having less time to design and implement a different fuzzer.

Once we concretely decided that our fuzzer was going to use black-box techniques we could lay out some milestones for our project. Those milestones were:

- Setting up basic CAN communication
- Understanding the UDS protocol and finding the CAN ID and services
- Deciding on input generation and feedback methods for the fuzzer
- Implementing and testing the components of the fuzzer
- Running the final fuzzer
- Analysing our results

We used the first term to concentrate on the first two milestones, setting up the ECU and communication with it, as described in section 4, as well as researching and understanding the operation of the CAN bus and the UDS protocol so that we may start to investigate it on the ECU, starting by finding the necessary information we need to effectively fuzz it i.e the CAN ID and the UDS services that are available on the ECU, also described in section 4.

Over the winter break, since we didn't have access to the lab, we spent the time researching fuzzing to get a better understanding of the field so that we may start to make some decisions for our third milestone, deciding on input generation and feedback methods. It was during this time we decided on our methods for the fuzzer and could start thinking about implementation after the break.

After the break, we could work on the final three milestones and get a functional fuzzer for the UDS protocol. We started with developing the test generation methods described in section 5 as these required less time spent with the actual ECU since we could develop the module to generate inputs without needing to interact with the ECU. We could then move on to developing the feedback methods, also described in section 5, we needed to be able to connect to the ECU to develop and test these as they used messages and information from the ECU to work. Once the two main components of the fuzzer had been developed and tested individually we could bring them together by developing the main fuzzing harness that will use the different components of the fuzzer to fuzz the ECU and collect the results. Once the fuzzer had been run against the ECU multiple times we would analyse the results that the fuzzer gave us and start to evaluate our findings, as discussed in section 8.

8 Results & Evaluation

8.1 Results

After multiple instances of running our fuzzer, both using our test generation methods and Radamsa's mutation methods, we collected some varying results on what the fuzzer deemed to be "useful" inputs. Most of the "useful" inputs caused positive responses and provided proof that our fuzzer was capable of generating a range of valid messages for UDS, examples of which can be found in Table 8.1. Unfortunately, our fuzzer didn't find any inputs that either caused new DTCs or took a significant amount of time to get a response from the ECU. There isn't much room for improvement regarding these two methods of feedback, our fuzzer just didn't generate inputs that triggered these methods. We will talk about the fuzzer's code coverage in this section.

From the inputs that we deemed useful enough to investigate, we found some to be particularly interesting. Our first case of some interesting findings was found when we were going through the positive results from one of the fuzzer runs, early on in the run the fuzzer requested to read data from the DID 0x401c with the request 0x**22**26efbbbe**401c** and the ECU responded with a positive response of 0x62401c**00000000**, indicating that the DID 0x401c contained the data 0x00000000. However, further along, this run of the fuzzer the ECU once again ended up requesting to read from DID 0x401c with the request 0x**22**401c401c**401c**, but this time the ECU responded with the positive response of 0x62401c**00004000**, indicating that the DID 0x401c now contained the data 0x00004000. We can see that somewhere between these two requests the data in DID 0x401c changed, at first we thought the fuzzer may have sent a write request which was accepted, however after looking through the used inputs from this run of the fuzzer we did not find any DID write requests (SID 0x2e) for the DID 0x401c. This means that some other request must have somehow triggered the change in the DID, this is a clear bug in the ECU as we should only be able to write to a DID by using the write data to DID service. Unfortunately, we are unable to tell what request potentially caused this bug as the two read requests were quite far apart from each other in the run of the fuzzer, around 1000+ inputs were used between them. Whether or not this bug could lead to a vulnerability and exploit is yet to be seen, it is however unlikely, but despite this, the finding of this bug is a great outcome for our fuzzer.

Another interesting result that our fuzzer found was when using the service ID 0x23 (Read Memory By Address). The request our fuzzer sent was 0x230000e280adff, according to the UDS standard the first byte after the SID indicates the length of the memory address the request is using as well as the size of the memory the request is reading. In the case of our input here, the byte is 0x00, indicating that the size of the memory we are reading is 0 and the length of the address is also 0. We would assume that in this case the ECU would either respond with an error such as 0x13 (invalid length/format) or respond with all zero data. However, the fuzzer's request was met with the response of 0x63000000006000, and the ECU still responding with data even though the request indicated that the size of the data should be 0 as well as the fact that the memory address we were reading from should have had a length of 0 is very interesting and leads us to believe that there is another bug with the implementation of this ECUs UDS. Once again it is unlikely that this bug could lead to a vulnerability and exploit but it is still a great result for our fuzzer.

8.2 Evaluation of Performance

One of the main metrics for performance for a fuzzer is how many inputs it can test per time interval, the two main factors that impact this are the speed of the test case generation and the response time of the SUT. In the case of embedded fuzzing, and our case of car ECUs, the response time of the SUT may be a potential bottleneck when it comes to fuzzing as we have to wait for the ECU to respond within a certain timeout before we can go to the next test case, and we can't send multiple requests at once because the CAN bus only allows one message at a time. Typically when fuzzing embedded devices like

Request	Response	Interesting?
0x230000e280adff	0x63000000006000	Yes
0x27210000000000	0x6721fd74670000	No
0x2226efbbbe401c followed by 0x22401c401c401c	0x62401c00000000 followed by 0x62401c00004000	Yes
0x2ed3caf3a081ac	0x6e03003201f400	No
0x238f8c00000000	0x63000000006200	No
0x313ea5f3a0819d	0x7101ff00000000	No
0x22af5cf12d3133	0x62f12d00000000	No
0x85020000000000	0xc5020000000000	No
0x3e000000000000	0x7e000000000000	No
0x22eef3a081931c	0x62eef301000000	No

Table 8.1: Example Results

```
         86448107 function calls (86405505 primitive calls) in 1837.877 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    275/1    0.002    0.000 1837.878 1837.878 {built-in method builtins.exec}
        1    0.000    0.000 1837.878 1837.878 uds_fuzzer.py:1(<module>)
        1   12.042   12.042 1837.549 1837.549 uds_fuzzer.py:29(main)
    17956    0.693    0.000 1792.179    0.100 canCommunication.py:15(sendUDSReq)
    17972    0.186    0.000 1786.750    0.099 bus.py:78(recv)
    17972    0.269    0.000 1786.489    0.099 socketcan.py:710(_recv_internal)
    35944 1785.358    0.050 1785.358    0.050 {built-in method select.select}
     8978    0.082    0.000  576.591    0.064 ECUFeedback.py:11(readDTCInformation)
 40306731   12.877    0.000   23.287    0.000 uds_fuzzer.py:126(<lambda>)
 40397438   10.440    0.000   10.440    0.000 {method 'replace' of 'str' objects}
     8983    4.278    0.000    4.543    0.001 {method 'readlines' of '_io._IOBase' objects}
    11602    0.236    0.000    3.529    0.000 testGeneration_Mutation.py:4(mutateData)
    17956    0.100    0.000    3.155    0.000 <__array_function__ internals>:177(pad)
53870/17958    0.231    0.000    3.046    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
```

Figure 8.1: Mutation profiling results

```
         33006575 function calls (32977487 primitive calls) in 1959.241 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    275/1    0.002    0.000 1959.241 1959.241 {built-in method builtins.exec}
        1    0.000    0.000 1959.241 1959.241 uds_fuzzer.py:1(<module>)
        1    5.116    5.116 1958.936 1958.936 uds_fuzzer.py:29(main)
    11199    0.499    0.000 1931.110    0.172 canCommunication.py:15(sendUDSReq)
    11201    0.142    0.000 1927.367    0.172 bus.py:78(recv)
    11201    0.140    0.000 1927.185    0.172 socketcan.py:710(_recv_internal)
    22402 1926.972    0.086 1926.972    0.086 {built-in method select.select}
     5599    0.052    0.000 1102.194    0.197 ECUFeedback.py:11(readDTCInformation)
     6128    9.481    0.002    9.552    0.002 pyradamsa.py:44(fuzz)
 15682800    5.068    0.000    9.266    0.000 uds_fuzzer.py:126(<lambda>)
 15684042    4.198    0.000    4.198    0.000 {method 'replace' of 'str' objects}
    11199    0.077    0.000    2.236    0.000 <__array_function__ internals>:177(pad)
33599/11201    0.147    0.000    2.153    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
```

Figure 8.2: Radamsa profiling results

this we would try to re-host the firmware so that we could run multiple instances of our fuzzer to get better performance, unfortunately, this is out of scope for this project. After running our fuzzer using both our mutation methods and Radamsa's mutation methods along with the DTC feedback method we found that our fuzzer was able to test 5 inputs per second using Radamsa and 4 inputs per second using our mutation methods. So as expected Radamsa did outperform our mutation methods but not by too much. In the context of fuzzing this is still considerably slow, so we need to figure out whether the bottleneck is with our test generation or with the response of the ECU, if the bottleneck is our test generation methods then we know that there is much room for improvement but if the bottleneck is the ECU then there is not much we can do about it since we are limited to how fast the ECU processes our requests and responds to them.

To figure out what the bottleneck of our fuzzer is we need to see where our fuzzer spends the most time processing, if the fuzzer spends more time generating test cases than it does waiting for the ECU to respond then we know the bottleneck is our code and not the ECU. If the reverse is true then we know the bottleneck is the ECU. To find this out we used cProfile, a built-in Python module that provides profiling for Python scripts. cProfile allows us to see how much time our fuzzer spends in each function, so we can compare how much time on average the fuzzer spends in sendUDSReq compared with our test generation function (mutateData for our mutations and pyradamsa.fuzz for Radamsa's mutations). cProfile is easily used with our fuzzer by executing the following Python command

```
$ python3 -m cProfile -s cumtime uds_fuzzer.py -m -d
```

This Python command will run our fuzzer but will use cProfile to measure and record how much time is spent in each function, we sort by the cumulative time so that we can see which functions the fuzzer spends the most time in. As we can see in figures 8.1 and 8.2, both instances of the fuzzer spend more time in sendUDSReq throughout the fuzzing run than in the test generation functions mutateData and pyradamsa.fuzz. This shows that the bottleneck of our fuzzer is the ECU and not our test generation methods.

8.3 Evaluation of Code Coverage

Code coverage is one way that fuzzers are normally evaluated, a good fuzzer will cover more of the available code in a given SUT. White-box fuzzers have a greater chance of getting better code coverage as they generally have access to the code/firmware so they can make decisions based on them to produce inputs that will cover more code. Since our fuzzer is a black-box fuzzer we did not have this option and we had little options to use to maximise code coverage. Furthermore, because our fuzzer uses black-box fuzzing it is hard to tell whether or not it has good code coverage since we have very little information on what the

inputs that the fuzzer produces do. One way we can try to evaluate the potential code coverage of our fuzzer is to analyse the inputs that our fuzzer produced during its fuzzing runs.

After analysing the inputs that our fuzzer produced during the multiple runs we found that our fuzzer produced a wide range of inputs, ranging from single byte payloads (just the SID padded with zeros) to utilising the full 8 bytes of the CAN data frame, the average length of the inputs was around 5 bytes. There was also a good spread of the available SIDs being used in the inputs, we did find that most of the interesting results came from the read and write related SIDs so this could be an indication that the fuzzer could maybe limit itself to those SIDs to maximise the coverage of those services. It's hard to comment on the robustness of the UDS implementation on the ECU against fuzzing in general as our fuzzer wouldn't have covered the entire input space or even a large proportion of it. Furthermore, a large chunk of the functionality within UDS likely required authentication using the Security Access service (SID 0x27), so we would need to either somehow bypass the authentication or find how the key is derived so that we may authenticate ourselves with the ECU.

So to conclude our evaluation, whilst we may have found some bugs our fuzzer hasn't explored enough of the input space and states that the ECU can be in to certifiably say that the ECU is robust against fuzzing.

9 Conclusion & Future work

9.1 Conclusion

The security of automotive electronic control units has been an area of research that has grown largely in the past decade, many different techniques have been seen to be used to test the security of these ECUs, one common method has been fuzzing which has been shown to be effective against the CAN bus in general and we want to expand on this research.

This project aimed to develop a black-box fuzzer for the Unified Diagnostic Services protocol to find bugs and unexpected changes in the operation of the protocol and the ECU in general. We developed our knowledge of the UDS protocol to gain a better understanding of how we can fuzz the protocol and designed a fuzzer to fit the operation of the UDS protocol.

In this project, we developed a black-box fuzzer for the UDS protocol used in automotive electronic control units. We showed that black-box fuzzing is an effective method in finding bugs and potential vulnerabilities within implementations of the UDS protocol by fuzzing UDS on our Ford Fiesta Body Control Module (BCM) and finding several inputs that caused unexpected results and showed that there are some bugs within the implementation of UDS on this ECU.

9.2 Future work

Whilst we have shown that our fuzzer can find bugs within the implementation of the UDS protocol there is room for future works and improvements, not only within how our fuzzer operates but also with the approaches used towards the testing of the security of ECUs. Here we will discuss some of those possible future works that can expand on this project.

9.2.1 Test generation methods

There are several ways in which the test generation of our fuzzer could be built upon. The first is the grammar-based test generation, first of all, the grammar does not allow for single byte or zero byte (just the SID) data payloads so this is one way it can be improved as this would lead to new inputs being generated that were potentially not generated before and could lead to better code coverage. Furthermore, the grammar uses probabilities to allow for varying lengths of data, however, these probabilities are static

and do not change throughout the run of the fuzzer. Having the probabilities of the data length change as the fuzzer runs to allow it to favour different lengths of data based on the feedback from the ECU would allow for a more directed form of fuzzing and could lead to better results and potentially better code coverage. The probabilities could also be used with the SIDs, initially, they all have an equal chance of being used by the grammar, we could change this so as we get feedback from the ECU we could determine what SIDs cause more interesting results and change the grammar so these SIDs are favoured more.

Building upon the mutation side of the fuzzer, this could be further developed so that the fuzzer uses previously used inputs that it deemed interesting, as the new seeds. This would allow the fuzzer to potentially find even more interesting inputs by mutating already interesting inputs, it could also once again lead to more code coverage as a mutation could lead to a different branch being taken within the code and could lead to more interesting results.

We could also build on our concept of the UDS fuzzer and develop fuzzers for specific services, the advantage of this would be having the context of the data values whilst developing. For example, if we were developing a fuzzer specifically for the Read Memory by Address service we would know that the first byte denotes the size of the memory being read and the length of the memory address being used, this would allow us to fuzz these specific values in such ways that it could lead to better results. Furthermore, we would have the context of the responses immediately rather than analysing the responses after the fact. Whilst this could provide better results we would be limiting ourselves to only a few services and could potentially miss out on interesting results relating to other services.

9.2.2 Feedback methods

Given that this project has been black-box fuzzing, feedback methods are hard to design and tend to have to be very primitive as we don't have access to much information from the ECU. That being said there are a couple of bits of future work that could be done on the feedback methods whilst remaining in the realm of black-box fuzzing. The first of these is building upon the traffic method of feedback that we developed, during this project the traffic that we recorded had very little context which meant that we couldn't do that much analysis on it other than try to detect patterns within it. However, if we were to gain a better understanding of the meaning of the traffic and add some context to it we could use this information to better inform us on what the traffic that our fuzzer caused means. For example, we could detect whether or not different non-UDS services on the ECU start or stop based on the starting and stopping of transmission of different CAN IDs, this would require more research into the specific ECU being used but it could provide valuable information.

Another method that could be developed in future work is to use physical changes to the ECU, while the BCM doesn't have any indicators on it we did notice that it will tick/make noises from time to time. These ticks could potentially be nothing or not be related to anything important or be random but it could also be the case that certain messages trigger a specific part of the hardware that causes said ticking noises. We could set up a noise sensor to detect when these ticks are made and record what inputs potentially caused them. It may lead to us seeing that the ticks are indeed random but it could also lead to some interesting results and is an interesting method of potential feedback for our ECU.

9.2.3 Other approaches

As mentioned before other approaches besides black-box fuzzing have been shown to be effective in finding bugs. One notable one that could be used as future work for this project is the use of the firmware of the ECU, if we are able to extract the firmware from the ECU we could use it to analyse the implementation of the UDS protocol and potentially find bugs. The firmware can be used to perform symbolic execution which can in turn be used to create a white-box fuzzer which would be guaranteed to find more interesting results than our black-box fuzzer. Furthermore, we could use firmware re-hosting to not have to use the actual ECU at all in our investigations, this would mean we could fuzz the re-hosted firmware which

would mean we could have multiple instances running at once to maximise the effectiveness of the fuzzer. Having access to the firmware opens up many possibilities in the testing of the security of the ECU and would make a very interesting future project.

References

Advanced Fuzzing League++ . AFL++ (ALFPlusPlus). https://github.com/AFLplusplus/AFLplusplus, 2017.

M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX security symposium (USENIX Security 11)*, 2011.

M. Eberlein, Y. Noller, T. Vogel, and L. Grunske. Evolutionary grammar-based fuzzing. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*, pages 105–120. Springer, 2020.

D. S. Fowler, J. Bryans, S. A. Shaikh, and P. Wooderson. Fuzz testing for automotive cyber-security. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 239–246. IEEE, 2018.

P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

Google. American Fuzzy Lop (AFL). https://github.com/google/AFL, 2013.

A. Helin. Radamsa. https://gitlab.com/akihe/radamsa, 2018.

ISO 11898-1:2015. Road vehicles controller area network (can) part 1: Data link layer and physical signalling. Standard ISO 11898-1:2015, International Organization for Standardization, 2015. URL https://www.iso.org/standard/63648.html.

ISO 14229-1:2020. Road vehicles unified diagnostic services (uds) part 1: Application layer. Standard ISO 14229-1:2020, International Organization for Standardization, 2020. URL https://www.iso.org/standard/72439.html.

ISO 15765-2:2016. Road vehicles diagnostic communication over controller area network (docan) part 2: Transport protocol and network layer services. Standard ISO 15765-2:2016, International Organization for Standardization, 2016. URL https://www.iso.org/standard/66574.html.

ISO 15765-4:2021. Road vehicles diagnostic communication over controller area network (docan) part 4: Requirements for emissions-related systems. Standard ISO 15765-4:2021, International Organization for Standardization, 2021. URL https://www.iso.org/standard/66574.html.

K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. Experimental security analysis of a modern automobile. In *2010 IEEE symposium on security and privacy*, pages 447–462. IEEE, 2010.

H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on*

Reliability, 67(3):1199–1218, 2018.

L. Luo. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232(1-19):19, 2001.

B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller. Parser-directed fuzzing. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, pages 548–560, 2019.

M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

OpenRCE. Sulley. `https://github.com/OpenRCE/sulley`, 2012.

J. Pereyda. BooFuzz. `https://github.com/jtpereyda/boofuzz`, 2016.

A.-I. Radu and F. D. Garcia. Grey-box analysis and fuzzing of automotive electronic components via control-flow graph extraction. In *Proceedings of the 4th ACM Computer Science in Cars Symposium*, pages 1–11, 2020.

S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, and S. Asryan. Grammar-based fuzzing. In *2018 Ivannikov Memorial Workshop (IVMEM)*, pages 32–35. IEEE, 2018.

F. Sommer, J. Dürrwang, and R. Kriesten. Survey and classification of automotive security attacks. *Information*, 10(4):148, 2019.

F.-X. Standaert. Introduction to side-channel attacks. *Secure integrated circuits and systems*, pages 27–42, 2010.

A. Takanen, J. D. Demott, C. Miller, and A. Kettunen. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

M. Thompson. UDS Security Access for Constrained ECUs. Technical report, SAE Technical Paper, 2022.

J. Van den Herrewegen. *Automotive firmware extraction and analysis techniques*. PhD thesis, University of Birmingham, 2021.

J. Van den Herrewegen and F. D. Garcia. Beneath the bonnet: A breakdown of diagnostic security. In *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I 23*, pages 305–324. Springer, 2018.

C. L. Wong. Security testing & reverse engineering of automotive can by black-box fuzzing. Master's thesis, University of Birmingham, School of Computer Science, 2023.

M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 54(1):1–36, 2021.