Samuel Sprung, BSc

# In-Depth Security Testing of modern vehicles

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

**Graz University of Technology**

**Supervisors**

Roderick Bloem, Univ.-Prof. Dr.

Institute of Applied Information Processing and Communications

Graz, December 2023

## AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

---

Date, Signature

# Acknowledgements

I want to thank several people who have been very supportive of me in this long journey of finishing my master's thesis. First of all, a major thanks to my supervisor Roderick Bloem from TU Graz, who was always supporting me with never-ending patience and who brought Syntax Guided Synthesis to my mind, which I could use to challenge the UDS protocol. Second, I want to thank Gerhard Griessnig from AVL, who introduced me to the automotive domain and gave me the chance to work on this great topic.

Also, a special thanks need to be delivered to Lisa Suppan, who was the supporting force to structure my mind and get the motivation needed to finish this work, and Martin Mandl, who was always there to listen endlessly about all the issues I was facing while getting the SyGuS interpreter to find any solution.

Last but not least I want to thank my mother, Elke Sprung, and father, Helmut Sprung, for making this whole adventure at the Graz University of Technology possible for me.

Samuel Sprung
Kirchbach in der Steiermark, 17.12.2023

# Abstract

In today's world, everything is getting more and more connected. This trend does not stop in front of the automotive industry. Besides enabling new business cases for the Automotive Original Equipment Manufacturers (OEM) these changing technologies bring in new threats that need to be considered and handled by security engineers [SSS+21]. In the last decades, a vehicle was considered an isolated environment and so was the attack surface. However, with more and more connectivity, the attack surface changes and it becomes a bigger challenge to secure legacy technologies within a vehicle against advanced attacks. Therefore, we analyze successful attacks that have already been carried out in the past, build a threat model over a common vehicle architecture and identify software update functionality carried out over such a legacy technology called Unified Diagnostic Service (UDS) as one of the biggest risk factors. Utilizing Syntax Guided Synthesis (SyGuS) we successfully reconstruct the legacy authentication function of UDS of a real Electronic Control Unit (ECU), by only knowing some constants, including the key, in a men-in-the-middle attack manner. Further, we find a real attack, which can enable us to get root access to every vehicle using an ECU with this authentication algorithm. In the end, we create a benchmark for different mechanisms to break into the authentication mechanism and point out how SyGuS can be used in engineering to improve the security of important functions in legacy systems.

**Keywords:** Automotive Cyber Security · Syntax Guided Synthesis · UDS · Testing

# Kurzfassung

In der modernen Welt ist alles immer mehr vernetzt. Dieser Trend erreich nun auch die Automobil Industrie. Daraus geben sich nicht nur neue Geschäftsmodelle für die Fahrzeug Hersteller, sondern auch neue Gefahren, verursacht durch die neuen Technologien, von den Sicherheits-Entwicklern mitigiert werden müssen [SSS+21]. In den letzten Jahrzehnten wurde das Fahrzeug und seine Angriff Vektoren als isoliertes System betrachtet. Durch mehr externe Schnittstellen im Fahrzeug haben sich diese Angriffsvektoren jedoch verändert und es wird eine immer größere Herausforderung Angriffe gegen diese veralteten Technologien ab zu wehren. Daher analysieren wir erfolgreiche Angriffe gegen Fahrzeuge der Vergangenheit, führen eine Gefahren Analyse über eine Fahrzeug Architektur durch identifizieren das UDS Protokol als großen Risiko Faktor. Mittel SyGuS rekonstruieren wir erfolgreich die authentifizierungs Funktionion dieser Technologie eines echten Steuergerätes, indem wir nur ein paar Konstanten, inklusive geheimen Schlüssel, kennen. Außerdem können wir ein Angriffsszenario für die wirkliche Welt entwickeln, um administrativen Zugriff zum Steuergerät des Fahrzeuges zu erhalten. Zum Schluss erstellen wir noch eine Gegenüberstellung zu verschiedenen Möglichkeiten, um einen solchen authentifizierungs Algorithmus zu brachen und wie SyGuS in der Entwicklung verwendet werden kann um die Sicherheit in solchen veralteten Systemen zu verbessern.

**Schlagwörter:** Automobile Fahrzeug Cyber-Sicherheit · Syntax Guided Synthesis · UDS · Testen

# Contents


**1 Introduction** **1**
- 1.1 Background . . . 1
- 1.2 Problem and relation to AVL . . . 2
- 1.3 Solution . . . 3
- 1.4 Structure . . . 7

**2 Preliminaries** **8**
- 2.1 Architecture Overview . . . 8
  - 2.1.1 Electricial Control Units within a vehicle . . . 8
  - 2.1.2 Electrical Architecture . . . 11
  - 2.1.3 In vehicle communication . . . 14
- 2.2 Unified Diagnostic Service . . . 19
- 2.3 Security standards in the automotive domain . . . 23
  - 2.3.1 ISO 21434 . . . 23
  - 2.3.2 UNECE R155 . . . 24
  - 2.3.3 ISO 21434 and UNECE R155 . . . 24
- 2.4 Syntax Guided Synthesis . . . 25
  - 2.4.1 SyGuS input specification . . . 25

**3 Related Work** **27**
- 3.1 The 2014 Jeep Hack . . . 27
  - 3.1.1 Remote attack surface . . . 28
  - 3.1.2 Uconnect System . . . 28
  - 3.1.3 Exploitation of the D-Bus Service . . . 30
  - 3.1.4 Cellular Exploitation . . . 31
  - 3.1.5 Getting access to the CAN bus . . . 31
  - 3.1.6 Authors conclusion . . . 33
- 3.2 Vehicle hacks from public news . . . 33
- 3.3 Key generation using the Z3 SMT Solver . . . 34

**4 Vehicle Security Testing** **36**
- 4.1 Attack Paths . . . 36
- 4.2 In depth Security Testing UDS . . . 40
  - 4.2.1 Technical Setup . . . 40
  - 4.2.2 SyGuS into an ECU . . . 43
- 4.3 Benchmarks . . . 56

**5 Evaluation** **60**
5.1 Constructing a real attack scenario . . . . . 60
5.2 Synthesizer lessons learned . . . . . 62
5.3 Benchmark interpretation . . . . . 62

**6 Conclusion** **64**

**Acronyms** **65**

**Bibliography** **68**

# List of Figures


1.1 The C implementation of the function used for seed/key generation. In UDS, the seed is taken and transformed with the function. In the end, when the seed in the function gets returned, it is the key that gets sent back to the ECU via the Controller Area Network (CAN) bus. . . . . 4
1.2 This image shows the input grammar for the SyGuS synthesizer. . . . . 5

2.1 [VM14] Legend of ECU Symbols which are represented in Figure 2.2 and Figure 2.3 . . . . . 11
2.2 [VM14] Audi A8 2014 Architecture . . . . . 12
2.3 [VM14] Jeep Cherokee 2014 . . . . . 13
2.4 CAN bus frame taken from Wikipedia . . . . . 15
2.5 Screenshot of an example dbc file in the kvaser dbc editor . . . . . 17
2.6 Image taken from [AUT21] Figure 3. and outlines how the data flow works in an AUTOSAR setup using Secure Onboard Communication (SecOC) . 18
2.7 Picture of a connected On-Board Diagnostics (OBD) plug in a conventional vehicle . . . . . 20
2.8 UDS setup and communication flaw . . . . . 21
2.9 Example mapping for UDS CAN IDs within a vehicle . . . . . 21
2.10 Some examples of UDS commands and their service IDs . . . . . 21
2.11 UDS Authentication flow example for successful authentication. 8A0 is the CAN ID on which the ECU is listening. On CAN ID 8A8 it is answering, here the Tester is listening on for this specific ECU. . . . . 22

3.1 This image from [VM15b] lists all open ports found in the QNX operating system. . . . . 29
3.2 This image from [VM15b] shows the short exploit written in Python to open a remote root shell using the D-Bus service . . . . . 30
3.3 This image from [VM15b] shows the decrypted file containing the SecurityAccess function used to calculate a key from the seed provided by the ECU . . . . . 32
3.4 This image from [Yur15] shows how the author transformed the operations applied to the password into a constraint set for the Z3 solver in order to find the correct password. . . . . 35

4.1 This Figure shows a simplified version of the current state-of-the-art vehicle infrastructure. . . . . 37

4.2 This Figure shows a simplified version of the current state-of-the-art vehicle infrastructure, highlighting in red the CAN paths that can be infected if we manage to break the UDS authentication mechanism. . . . . 39
4.3 The reference setup with the ECU under attack, the self-made cabling to power it, and to bring the CAN line to the PEAK CAN USB adapter. . . 41
4.4 This image visualizes in the threat model our reference setup in blue and how it can be mapped into a vehicle. . . . . 42
4.5 The first communication recorded by the ECU . . . . . 42
4.6 A C implementation of the function used inside the ECU/tester to calculate the seed/key value within the UDS authentication process. . . . . 43
4.7 This image shows the very first implementation try for the synthesizer to synthesize the target authentication function. . . . . 45
4.8 As the calculation continued the RAM of the notebook was full - resulting in the notebook putting more work into swapping between the disk and the RAM . . . . . 45
4.9 The second draft of the input alphabet. Here the if-then-else clause was removed . . . . . 46
4.10 Here we replaced the old development key 0xFFFFFFFF with a real key. The real key is here however not displayed directly. The keys calculated out of the seeds are also updated accordingly. . . . . 48
4.11 This Figure shows all adaptions done for the synthesizer to synthesize only the 16-bit key used within the function. . . . . 49
4.12 This Figure shows all adaptions done for the synthesizer to synthesize not only the key but also the function. . . . . 50
4.13 This Figure shows the 6 outputs of the synthesizer which have been generated after 17 seconds. . . . . 50
4.14 This shows the output of the synthesizer which has been generated after 17 seconds. . . . . 51
4.15 This shows the reduced 15-bit version of the initial 16-bit file. . . . . 52
4.16 Reducing the space to 15 bits resulted in several blue screens at executing - as well as strange system behavior right before the blue screen. . . . . 53
4.17 This shows the working solution in which the 32 iterations for-loop have been excluded from synthesizing. The synthesizer only needs to synthesize the operations. . . . . 53
4.18 This is the synthesis function returned from the synthesizer. The real value for the key is replaced with KEY here. The variable x is the input seed 54
4.19 This is the input to the z3 constraint solver. The two defined functions are the same as in Figure 4.20. Line 19, the definition of the KEY to be determined is different due to the syntax of the constraint solver. . . . . 57
4.20 This is the input to the SyGuS synthesizer. The function to be synthesized f only contains a call to the encrypt function and a variable KEY where the synthesizer has to choose a constant value to solve all constraints. . . 58

5.1 To apply the men-in-the-middle attack a modification is done at the OBD port. Here, the UDS requests will be intercepted, and a modified payload will be sent to the computer of the car workshop technician. . . . . 61

# List of Tables

1.1 benchmarks of brute force attempts . . . . . 6

2.1 CAN message id arbitration phase . . . . . 16
2.2 reading CAN communication with candump. We see the timestamp on when the message was received, the CAN line, and the message ID and message, separated by a # . . . . . 16

4.1 benchmarks of brute force attempts . . . . . 57

# Chapter 1

# Introduction

This chapter serves as a shortened version of the complete master thesis. In the 1.1 Background section a vehicle and its architecture is introduced and discussed. In section 1.2 we discuss the security issues within modern vehicles and identify the firmware update protocol UDS as one of the most security-critical features. 1.3 outlines how we can attack the authentication mechanism of UDS with SyGuS in a men-in-the-middle-attack-kind of attack.

## 1.1 Background

With the Information and Communications Technologies for Safe and Intelligent Vehicles [cE03] communication document in 2003, an emergency e-call assistant for new vehicles was introduced by the European Government. With this, the first regulatory bases were created to have connectivity in a vehicle within Europe. This enforced OEMs to implement units within their vehicles that communicate actively with the outside world. Further, when Tesla entered the automotive sector in 2009, new technologies and new processes hit the old-fashioned automotive industry [Edw20]. In this article, it is mentioned, that not only the electrified powertrain is new, but also the methodology: "That investment gave Mercedes engineers an inside view of how Musk was willing to launch technology that wasn't perfect, and then repeatedly upgrade it, using smartphone style over-the- air updates, paying little regard to early profitability." These examples show, how technical possibilities and methodology have changed over the last decades in the automotive industry. But did the architecture and security of vehicles adapt to these changes?

According to Charlie Miller and Chris Valasek [VM15b] security research in the automotive world began in 2009, when the University of Washington and the University of California San Diego successfully showed that it is possible to inject messages onto the CAN bus and influence the behavior of a vehicle. Influencing the behavior is a problem because it influences the safety of the vehicle and all participants in traffic. However, the automotive industry did not care about this argument because the demonstrated attack required physical access to an vehicle. They argued, that with physical access one can also just cut off the braking wires and by that influence the safety of the vehicle and its passengers. To support the research community in demonstrating the relevance of

security in vehicles Miller and Valasek continued the work and developed several tools[1] to support security engineers in diving into vehicles and the way they work. As a next step, they released a white paper about the remote attack surface of modern vehicles in 2014 [VM14] outlining the architecture, network topology and remote attack surface of different vehicles from different vendors. As mentioned in [VM15b] this analysis made them pick the 2014 Jeep Cherokee as their target for the remote attack with one goal: "Hopefully this additional remote attack research can pave the road for more secure connected cars in our future by providing this detailed information to security researchers, automotive manufacturers, automotive suppliers, and consumers." With this hack, they clearly outlined that remote hacks to vehicles are possible, which impact the physical safety of a driver and every participant in public transport. The impact of this release was, that 1.4 million vehicles have been recalled by FCA[2] to fix the vulnerabilities found. The complete attack is explained in more detail later in the thesis in chapter3.1.

## 1.2 Problem and relation to AVL

This section describes the problem analyzed within the thesis and the relation to AVL.

Building secure vehicles is a very challenging task for automotive engineers. Many functions and connectivity have been added to vehicles in the last years, without big changes in the architecture or components of vehicles. These so-called legacy[3] technology in vehicles and the high pressure of cost savings, makes it very hard to secure vehicles. The risk of overlooking huge security issues within old technologies in a vehicle is a big threat. Also, technical limitations of very well-established technologies (e.g. BUS systems inside a vehicle) are one of numerous challenges in the automotive domain.

Anstalt für Verbrennungskraftmaschinen List (AVL) is one of the largest, privately owned, engineering company for the automotive domain. The company is well known for powertrain engineering, for the design and development of combustion engines, as well as for the development of testing equipment that is used all over the world. Therefore cyber security, in every part of a vehicle, is of greatest importance to AVL. To improve the security knowledge within these legacy technologies, the topic of this master thesis was chosen, to analyze techniques and find new methods to execute security tests within a modern vehicle. Further, it should be easier to estimate the risk of reusing legacy technologies in a vehicle. AVL targets to not only build safe and reliable systems but also to secure them. In addition, new standards and regulations, such as the ISO21434 (see chapter 2.3.1 and the UNECE R155 (see chapter 2.3.2) give more strict requirements on how to develop a secure vehicle and what controls and testing needs to be in place. These requirements do not only apply to AVL but to the complete automotive sector.

[1] http://illmatics.com/content.zip
[2] Fiat Chrysler Automobiles (FCA) is the seventh biggest OEM worldwide.
[3] An old technology, which is probably outdated is generally called legacy

This being said, especially protecting the firmware of ECUs and access to administrative functions in a vehicle is crucial. Chris Valasek and Charlie Miller also released another whitepaper [VM15a] where they extend their work previously done and leading to the famous Jeep Hack (see Chapter 3.1 for a detailed explanation of this hack) and show different other ways of influencing a vehicle, having access to the communication bus of the vehicle. In this paper, they outline that it is easily possible to reverse engineer the UDS[4] authentication algorithm if you have physical access to the ECU and can extract it. UDS is a protocol spoken on the CAN[5] bus within a vehicle to execute administrative tasks, like a firmware update or change of parameters. Further, they note that this authentication algorithm in general is very weak and sometimes even can be brute forced. As these authentication algorithms are crucial for a vehicle, we are targeting to show their weaknesses in this master thesis by constructing a men-in-the-middle[6] attack capable of reconstructing this algorithm to unlock a real ECU, by having restricted knowledge about the function and the secrets.
For doing so, AVL provided an exemplary ECU, a tool to flash software onto the ECU, and some documentation related to UDS. Exemplary means, that this ECU is not use in real world, but behaving exactly the same including having the same level of Security-Access algorithm, as a ECU in the vehicle would have. With this, a setup was created to communicate with the ECU and to perform a men-in-the-middle attack to break the authentication algorithm of the ECU.

The goal of this master thesis is to show that it is feasible to break into the authentication mechanism of a legacy protocol on the CAN bus (a justification on why UDS is chosen is provided in section 4.1). It should be shown that SyGuS is capable of reconstructing (synthesizing) the authentication algorithm, based on the recorded seed/key pairs with as little information as possible. Therefore we focused on two different approaches: only the function being known and synthesizing the key and synthesizing the function by having as little knowledge as possible (for example: only knowing the key and constants used). We compare the synthesizer with other methods of finding a key to a secret function.
With this ways of validating the security level of this legacy protocol should be enabled, to make sure that it still provides enough security. This is important because legacy software and protocols can't always be replaced immediately, but still, security needs to be proven as required by the new standards.

## 1.3 Solution

In this section, the solution implemented to solve the problem and the findings are described. It is required to have read the preliminaries, especially chapters related to

[4] See chapter 2.2 for an explanation of the UDS protocol.
[5] See chapter 2.1.3 to read more details about the CAN bus
[6] A men-in-the-middle attack is an attack, where a malicious actor is acting in the middle of a connection between two parties by manipulating the data send between each of them.

CAN bus (chapter 2.1.3), the UDS protocol (chapter 2.2) and SyGuS (chapter 2.4).

At the beginning of the work, we focused on creating a good input grammar for the CVC4 synthesizer. Our goal is to enable SyGuS to synthesize the function, given only the constants used within the function. Here, the secret key within the function is also provided as a constant to the synthesizer. The target function is visible in Figure 1.1. Even though the function does not look very complex, we had to deal with RAM issues, as the synthesizer consumed more RAM for synthesizing a solution than available on the notebook.

```
uint32_t calculateKey32(uint32_t seed, uint32_t key)
{
    for (int i = 0; i < 32; i++) {

        if (seed & 0x80000000) {

            seed = seed << 1;

            seed = seed ^ key;
        }
        else {
            seed = seed << 1;
        }
    }
    return seed;
}
```

Figure 1.1: The C implementation of the function used for seed/key generation. In UDS, the seed is taken and transformed with the function. In the end, when the seed in the function gets returned, it is the key that gets sent back to the ECU via the CAN bus.

After overcoming the RAM limitation, in which we run if the input grammar to the synthesizer allows too many operations (in this case if-then-else increases the RAM utilization drastically), we already make two observations. First, the function contains a weakness, that allows one to return the secret key used in the function, if 0x01 is provided as input (seed) to the function. This is happening, because only in the last iteration of the function the if-clause evaluates to true. With the shift operation, the seed variable then gets set to zero and then set to the key through the xor operation (see chapter 4.2.2 for a detailed explanation). The second observation is, that if the key is 0xFFFFFFFF (as it was for the first tries in our setup), the synthesizer can find a 2-line program, doing the same as the function in Figure 1.1:

```
uint32_t f(uint32_t seed){
   uint32_t tmp = seed & 1 ? 0xFFFFFFFF : 0;
```

```
    return ((seed >> 1)^tmp);
}
```

In the next step, the key is changed to a real one and the synthesizer again hits the RAM limitation. To help the synthesizer reduce the complexity, we additionally model the for loop in the grammar for the synthesizer as seen in Figure 1.2.

```
1  ( set-logic BV )
2
3  ( synth-fun f (( x ( _ BitVec 32))) ( _ BitVec 32)
4  (( BV32 ( _ BitVec 32)) )
5  (( BV32 ( _ BitVec 32) (#x00000000 #x00000001 #x[real key] #xffffffff #x80000000
6      x
7      ( bvand BV32 BV32 )
8      ( bvor BV32 BV32 )
9      ( bvnot BV32 )
10     (bvxor BV32 BV32)
11     (bvshl BV32 BV32)
12     (bvlshr BV32 BV32)
13     ))
14 ))
15
16 ( define-fun encrypt (( x ( _ BitVec 32))) ( _ BitVec 32)
17  ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f
18     ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f
19        ( f ( f ( f ( f x ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
20     ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
21 )
22
23 (constraint (= ( encrypt #x00000000 ) #x00000000 ))
24 (constraint (= ( encrypt #x00000001 ) #x[redacted] ))
25 (constraint (= ( encrypt #x4ad88327 ) #x[redacted] ))
26 ;(constraint (= ( encrypt #x90729a8a ) #x[redacted] ))
27
28 ( check-synth )
29
```

Figure 1.2: This image shows the input grammar for the SyGuS synthesizer.

The input grammar gives a function call to *encrypt*, with the seed as the function parameter, together with the resulting session key as a constraint to the synthesizer. the function *encrypt* is then called 32 times on the function to be synthesized: *f*. Within this function, we have defined some constants, including the real secret key, and the operations which can be used by the synthesizer. After 37 minutes and with usage of 12GB or RAM the synthesizer returns a function. A deep analysis of the returned function shows (see chapter 4.2.2 for the complete result of the synthesizer and how we verified that it is the same function), that the synthesizer synthesized the same function as visible in Figure 1.1. This means, that the synthesized function is valid for any seed and can be used to unlock the programming session of the ECU.

For the next goal, we take the synthesizer and compare it in a brute-force manner against a Z3 constraint solver and a classical brute-force approach, represented by a program written in C. Z3 and the C program received both the same 3 seed/key pairs to

brute-force the secret key against. SyGuS receives in addition a seed/key pair where 0x01 is the seed, to help it improve the synthesizing speed. Initially, the Z3 solver used 0x01 as seed, but as it is smartly testing constraints at the beginning it finds the correct secret key in under one second because it tries the key which results from the seed 0x01 as input (as mentioned, due to the weakness in the function, the resulting key for the seed 0x01 is the secret key from the function). Table 1.1 shows the results of the benchmarking.

| **ID** | **Key** | **Bruteforce 0** | **Bruteforce random** | **Wrong keys found** | **Z3** | **SyGuS** |
|---|---|---|---|---|---|---|
| 1 | real key | 00:00:37 | 00:12:23 | 0,95 | 00:00:54 | 32:20:00* |
| 2 | 0x01010101 | 00:00:01 | 00:15:42 | 57,55 | 00:02:52 | |
| 3 | 0x10101010 | 00:00:18 | 00:13:46 | 2,85 | 00:45:11 | |
| 4 | 0xDeadbeef | 00:04:16 | 00:07:57 | 2,8 | 05:27:34 | |
| 5 | 0xFac239af | 00:04:47 | 00:15:18 | 23,05 | 04:57:36 | |
| 6 | 0xff99ff99 | 00:04:59 | 00:14:30 | 24,35 | 01:22:26 | |
| 7 | 0xFfffffff | 00:04:53 | 00:11:12 | 8,2 | 19:36:45 | |
| **Mean over all keys** | | *00:02:50* | *00:12:58* | *17,11* | *04:36:11* | |

Table 1.1: benchmarks of brute force attempts

It is worth noting, that SyGuS only found a key one time, after one and a half days of calculation. This result could neither be reproduced for the real key nor could a result be brute forces for the other keys. Not even with 3 weeks of calculation. In a restricted version, where we only use 16-bits it works in around 23 minutes. Even with setting a high level of verbosity to the CVC4 synthesizer, we could not Figure out where the issues were arising.
Even thus it is as we expected, that brute force with a mean of 12 minutes and 58 seconds is the fastest approach, the detail analysis shows some interesting outcomes. For key number 4 brute force takes only half the time compared to the others, while the Z3 solver is even faster than brute force for the real key and key 2. Both variants show behavior for certain keys, which is not expected and not clear (e.g. the time delta for Z3 between keys 5,6 and 7 as well as the brute force time difference for keys 4 and 5). In addition, the column 'wrong keys found' displays the average number of wrong keys found while using the random brute force approach. A key is classified as wrong during brute force, if it can be used to calculate the provided seed/key pair, but not for the other 2 seed/key pairs. Here it is again very interesting that not only the performance for certain keys is unexpectedly fast, but also the number of wrong keys found while brute forcing is lower.

To conclude, we are surprised that SyGuS was able to synthesize the correct function. We would have expected that it would find a function, valid for some seeds, but not for all. Also, all the other findings done while working on the input for the CVC4 synthesizer make the point clear, that SyGuS brings great support to define such security critical function, for testing and validation. In this work, SyGuS helped us to identify a critical vulnerability in the function, which we could even use to construct a real-world attack (see chapter 5.1 for an explanation of how the attack would work), to extract the secret key from the ECU of any vehicle which utilizes this function in its UDS implementation. Further, SyGuS might have the potential to break more UDS SecurityAccess functions if

the grammar is fed with more constants and smarter grammar. This might be possible because the authors of [VM15b] also mentioned, that they found similar functions and keys during their research across different vehicles. Even though SyGuS did not prove to be a good tool to brute force a key, the Z3 solver showed potential, especially if there is a weakness contained in the assertions provided to it as input.

## 1.4 Structure

This thesis is structured in the following way:

- Chapter 2 provides all the background information needed to understand this work. The architecture of vehicles and how the communication on the CAN bus within a vehicle works. Also, the UDS protocol is discussed in detail and will be the main focus of this work. Relevant standards, recently introduced to the automotive domain are discussed as they underline the need of this work also from a legislative point of view. Last but not least SyGuS is explained, as it is the main tool used within this thesis.
- Chapter 3 presents the work of Charlie Miller and Chris Valasek, which performed the famous Jeep Hack in 2014. They spend much work in reverse engineering firmware of ECUs and bring big motivation into this work, with which we want to provide additional ways of attacking a vehicle.
- Chapter 4 Outlines the attack path and why we have chosen UDS as the most interesting protocol to be attacked. This chapter contains all the steps and pitfalls encountered to reach the goal of this thesis, to synthesize the authentication function.
- Chapter 5 Contains the evaluation of the outcomes of the SyGuS and benchmarking work. It also contains a detailed description of how we could construct a real-world attack scenario with the findings from the work, enabling us to get root access to the ECU in a vehicle.
- Chapter 6 provides an outlook on open questions arising from this work as well as on further possibilities on how to further enhance SyGuS grammar to break more UDS implementations within a vehicle.

# Chapter 2

# Preliminaries

In this chapter, we provide an overview of the (electrical) architecture of vehicles (Section 2.1) and how communication in a vehicle works. We will discuss all the preliminaries that are needed to understand the main work of this thesis. This chapter will end with a short introduction on how SyGuS works.

## 2.1 Architecture Overview

To understand, where the verification of security is needed, we need to understand the remote attack surface of a vehicle and what the goal of an attacker is. This goal could differ related to what the motivation of an attacker [CDKT15] is. It could be the breach of privacy of the vehicle owner, or in the worst case, the launch of a cyber-physical attack, which means starting a remote attack that manipulates safety functions and causes physical impact. We will focus on attack vectors that allow us to impact the physical behavior of a vehicle.
If one tries to compare vehicles from different OEMs, he will hardly find two vehicles that are equal to each other. This is not only measurable in how a vehicle is designed and how it looks from the outside but also in the electrical architecture of how a vehicle is built. In 2014 Miller & Valasek [VM14] already started with a comprehensive whitepaper analyzing the remote attack surface of vehicles from different vendors. Besides different levels of security, that make a car easy or hard to hack, they discovered that automotive topologies are built through the concept of security by obscurity. As the number of ECUs within a vehicle ranges from a very few up to more than 100, it is not very easy to define which ECU is relevant for cyber security testing and which is not. A good starting point could be an ECU, which has a direct or indirect safety impact on the vehicle, due to the purpose of its functionality. If so, one might take this ECU as a target to launch a so-called cyber-physical attack.

### 2.1.1 Electricial Control Units within a vehicle

As many ECUs are in a vehicle, as different their purpose is different. Some ECUs are safety-relevant, which means they are controlling functionalities that have a direct impact on the driver, like the Electronic Stability Program (ESP) which is responsible for keeping the car on track. Others offer a remote interface (e.g. Bluetooth, WiFi, LTE) and some are just for internal functionality and not visible to the driver. This Chapter

gives an overview of different ECUs and the purpose they are used for.
As the architectures differ between vehicles, we can't say if an ECU is safety-relevant, but based on the pure functionality of an ECU we can assume it. However, what most of them have in common is the following: almost all ECUs are based on proprietary software, which means that there is no common operating system present and software is directly interacting with the hardware. Usually, the only exception to this is the Infotainment System (INSYS) which usually runs Blackberry OS, Linux, or something similar.

As a vehicle can have multiple ECUs, each ECU may have a different purpose. Some ECUs have a direct impact on the driver's safety, such as the ESP, which is responsible for keeping the car on the road. Other ECUs offer a remote interface (e.g. Bluetooth, WiFi, LTE), while some are for internal functionality that isn't visible to the driver. This chapter provides an overview of the different ECUs and their intended functions. While it is difficult to say which ECUs are safety-critical, we can assume that some are safety-critical because of their name and the functionality within the vehicle. However, most ECUs share a common characteristic: they are based on proprietary software that interacts directly with the hardware. The INSYS is typically the only exception, running on operating systems like BlackBerry OS, Linux, or similar systems. The following list names several ECUs which can be found in most vehicles across different OEMs.

- Engine Control Module (ECM)
  - The ECM controls the combustion engine
  - It uses several sensors and communication via CAN as input
  - Several OEMs provide the possibility to turn on a vehicle's engine with a remote command issued by e.g. a mobile application on the smartphone
  - Because of controlling the engine this ECU is safety-relevant
- Body Control Module (BCM)
  - Is responsible for managing so-called convenient functions e.g. the electrical windows, seat heating functionality, or the parking heater
  - Often it is connected to the locking system of a vehicle – also to the keyless entry system
- Tire Pressure Monitoring System (TPMS)
  - monitors the tire pressure of the vehicle and issues warnings to the driver in case the pressure is lost rapidly.
  - information is transmitted using short range radio[Wen05] to the TPMS.
  - is safety-relevant and subject to active research [Mik18] to prove security vulnerabilities.
- Remote Keyless Entry (RKE)
  - provides the ability to unlock a vehicle without pressing a button on the key

  - In addition it enables the driver to start the engine without plugin the key into the vehicle physically
  - the key is replaced by a key fob which transmits short-range radio signals to the vehicle
  - opens up a wide range of relay attacks [FDC11] which are already heavily used to steal vehicles all over the world

- INSYS
  - The Infotainment System is for most car owners the heart of a vehicle
  - It supports many different applications e.g. a web browser, music apps, and servers as an interface to connect a smartphone with a vehicle and is used for most interactions with the vehicle
  - In contradiction to most of the other ECUs this ECU is usually not based on proprietary software, but on Linux or very close to Linux, operating systems
  - Additionally remote interfaces such as Bluetooth and a connection to the internet as well as to the WiFi settings (if built into the vehicle) are included
  - last but not least the INSYS also includes Apps which can be installed and used within a vehicle as well as a web browser - which are generally known to provide several exploits and need to be updated very often

- Telematics Unit (TCU)
  - serves with the connection to the internet.
  - OEMs are using e-Sims [Por21] within the TCU
  - it is used to retrieve data, e.g. traffic or weather data, and to provide the network interface to the WiFi router in the vehicle.
  - in addition, GPS and also the emergency call assistant is included in this ECU

- Adaptive Cruise Control (ACC)
  - is in charge of keeping the configured speed of a vehicle and taking care of other traffic participants
  - if a vehicle is coming up in front and driving slower, the ACC will slow down the vehicle and follow the other vehicle with the same speed
  - This feature is highly safety critical as it takes active intervention in the vehicle

- Gateway
  - Used as a gatekeeper between different bus lines
  - A secure gateway also applies special white-listing to messages to prohibit a participant on CAN line one from sending a message to line two, unless explicitly allowed

- Backend
  - not physically connected to the vehicle, but still an integral part is the OEMs backend and therefore listed for the sake of completeness.
  - Software Update Over the Air (SOTA) is a crucial part of modern vehicles to keep them safe and secure on the one hand and the other hand it is the promise for OEMs to end expensive callbacks if there are critical flaws, in the software of vehicles, detected
  - How the Backend is contacted is different from vehicle to vehicle. Some vehicles have a built-in GSM module (most likely in a TCU) for communication with the backend, some utilize the Smartphone which is connected to the INSYS of the vehicle.

Of course, there are many more ECUs built into a vehicle, with similar names or functionality, but the enumerated ones will be used within this thesis.

### 2.1.2 Electrical Architecture

In their whitepaper [VM14] list 22 vehicle architectures manufactured between 2006 and 2015. We picked two architectures for reference because they seem to be good starting points for further work in this thesis.
Figure 2.1 outlines the legend used by Miller and Valasek to model functionalities in a vehicle. Please note that the ECUs and functionalities in this reference architecture are dependent on the variant of the vehicle inspected. One might have ACC in place, another one might not - always dependent on which additional features have been bought with the vehicle. What most vehicles have in common is that they group ECUs into logical blocks. This has several reasons: efficiency, logical grouping by function, utilization of bandwidth on the BUS, saving costs (by reducing bus lines needed within the vehicle), or security.


Figure 2.1: [VM14] Legend of ECU Symbols which are represented in Figure 2.2 and Figure 2.3

### Audi A8

Figure 2.2 draws the architecture of the Audi A8 and shows 4 bus systems: 3 CAN lines with a separation of driving functionalities (Drivetrain CAN), convenience functions (Convenience CAN) and a CAN line for Advanced Driver Assistant Systems (ADAS) functions like ACC (Distance Control CAN). In addition, there is also an own bus for communication within the infotainment system: MOST bus.
The architecture is set up as star architecture and CAN lines are interconnected via a CAN gateway. The CAN gateways', respectively the connector, purpose is to separate the communication between the different buses. This functionality is considered a security functionality.
The remote interfaces that can be attacked are on the convenience CAN: RKE and TPMS and on the MOST bus: Bluetooth, the radio control module, TCU and the Audi Connect System (internet/apps).

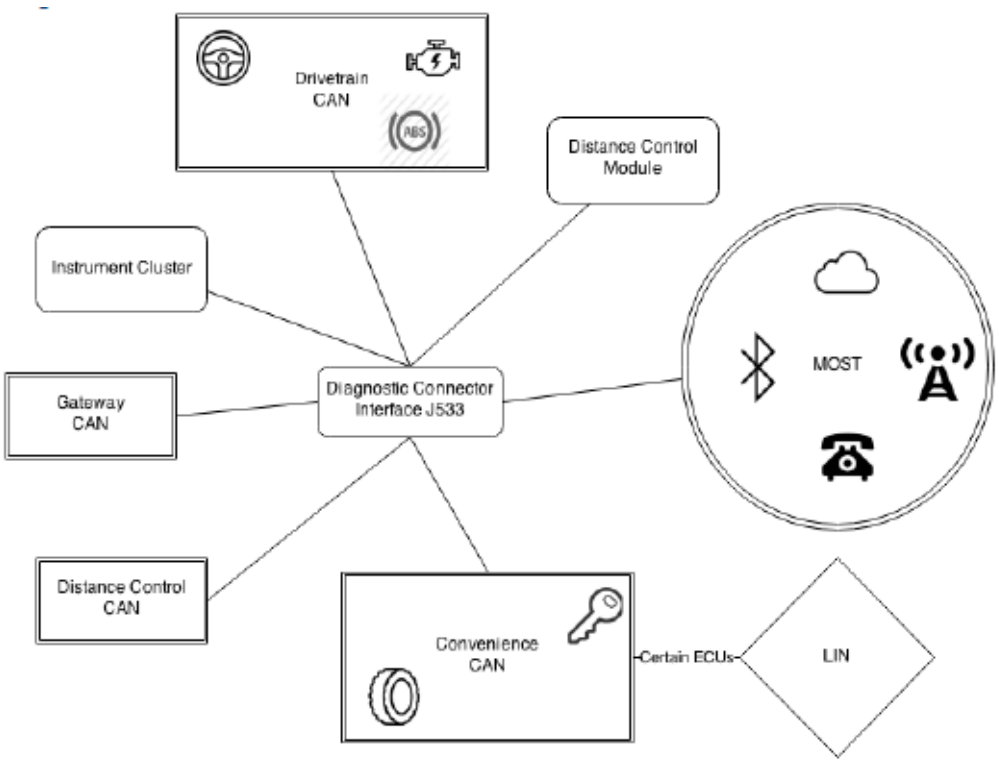


Figure 2.2: [VM14] Audi A8 2014 Architecture

### Jeep Cherokee

The Jeep Cherokee, shown in Figure 2.3, has two CAN buses and one Local Interconnect Network (LIN) bus for communication between the different ECUs. The CAN C bus

groups the ACC, Park Assist (PA), BCM, and the ECM. The IHS CAN groups the BCM (again), door controls, air conditioning system, and some other convenient functions. The LIN bus connects the BCM and many sensors used for functions e.g. humidity sensors, light and rain sensors as well as some sensors and modules related to the engine.
Remote interfaces in this vehicle are: RKE, TPMS on CAN C and Bluetooth, the radio control module, TCU and the INSYS including internet and apps via the CAN C and CAN IHS.

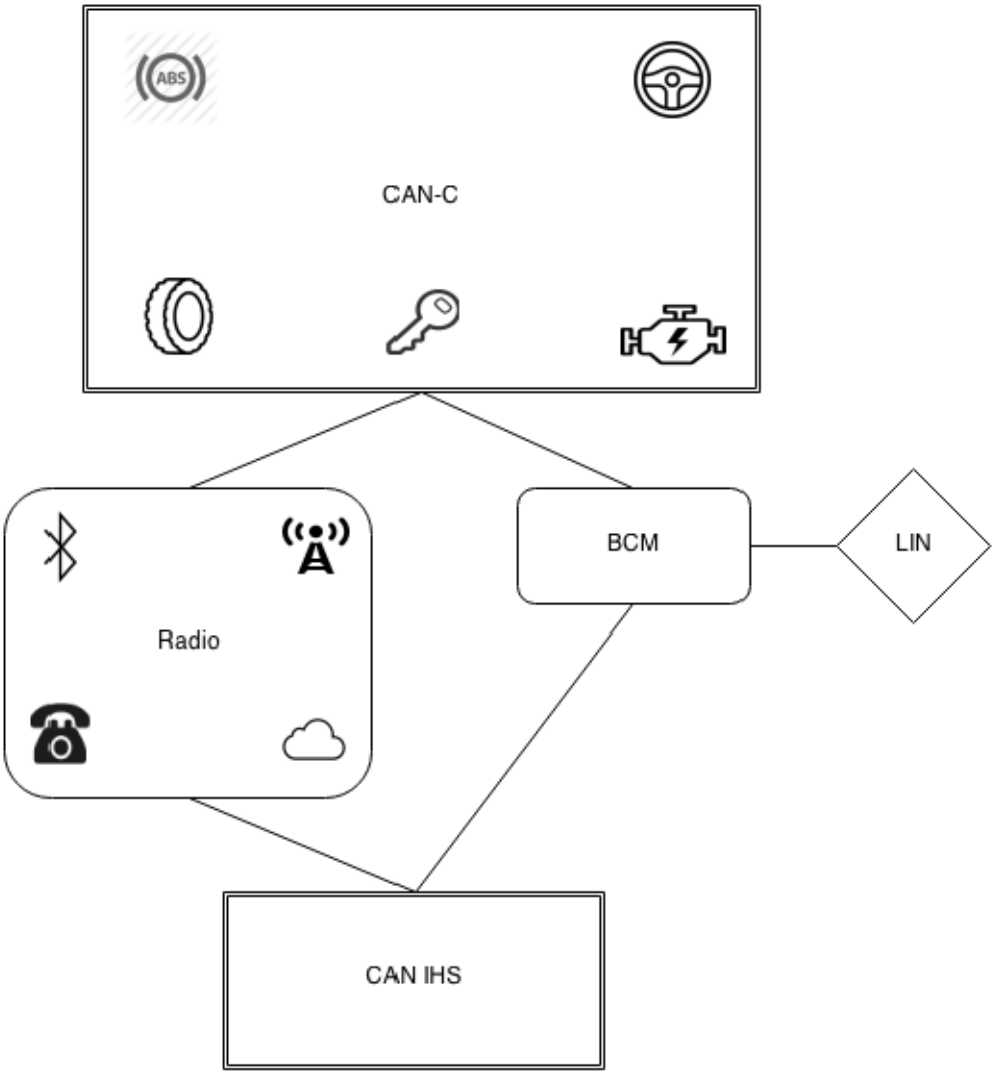


Figure 2.3: [VM14] Jeep Cherokee 2014

If these two architectures are compared with each other from a security point of view, it becomes clear that the architecture of the Audi might be a little more secure,

due to the separation of communication introduced by the gateway, which divides the communication buses into a star architecture. By this, only predefined messages can be routed within the different bus types. The same conclusion led the authors of [VM14] to their later work on the Jeep Hack discussed in Chapter 3. The reason for their choice is, that the INSYS, represented as Radio in the Jeep Cherokee architecture (figure 2.3), looks to be a valuable target for an attack, as it is directly connected to other communication buses and therefore with other ECUs. In the Audi, there would be the hurdle of the gateway which has to be overcome.

### 2.1.3 In vehicle communication

As already mentioned, the number of ECUs in a vehicle ranges from very few to several hundred. To work properly, these ECUs do not only need to be connected to external sensors and actuators but also to each other for communication. Communication is crucial because different ECUs need to interact with each other to trigger e.g. a safety reaction within the vehicle in milliseconds. In the automotive domain, we have several different bus systems to enable communication. Some of them are very similar to each other. Most of them are already very old but still very reliable. Other techniques are brand new in this domain and promise to solve the challenges that new techniques like ADAS bring with them e.g. more bandwidth needed or faster exchange of data.

#### CAN

The CAN is the widest-spread communication bus within the automotive domain. Initially CAN was developed by the Robert Bosch GmbH [Rob21] and is now standardized by the ISO group in the [Com15] standard. The CAN is a bus system allowing devices to communicate with each other without the need for a host computer, respectively without the need for a server. It is very robust and allows message prioritization which enables each participant to communicate at the same time while still ensuring that messages of high priority get sent first.
Classic CAN lines operate on bit-rates between 125bit/s, 500bit/s up to 1000bit/s and a message sent on the bus has a message identifier[1] of 11 bits and payload size of 8 bytes. However, there are also other types of CAN, with greater payload (so-called extended CAN with ids greater than 11 bits) or flexible data rates. But as the protocols on the CAN line remain the same we just stick to a simple setup with 8 bytes of payload and 500 bit/s.
Out of the box CAN provides also Cyclic Redundancy Code (CRC) check for each message which protects the data integrity while being sent on the CAN line. As it is not needed to understand all the details on how CAN works, we will only focus on the most important functionalities like how messages are prioritized and not how time synchronization and bit stuffing work.
the CAN is not only used within the automotive domain. Other industries are also using this bus because of its great reliability and fault tolerance.

[1] CAN message IDs are also often referred to as arbitration id

### Message structure and prioritization

Big parts of the CAN frame are used for synchronization and error correction. There are just a few simple rules to be aware of. Figure 2.4 shows a complete example of a CAN message. It explains what the structure on the CAN bus looks like. In our later work, the Arbitration ID will be referenced as message ID, and the data section will be the actual CAN message we want to send. All the other fields are abstracted by the used software to read and write on the CAN bus.

**Arbitration (green)** Arbitration ID or Message ID of the CAN message.

**Control (yellow)** Responsible for managing the data length.

**Data (red)** This section contains the actual payload of the message. A message in standard CAN can have 0-8 bytes.

**CRC** The CRC is responsible for providing evidence that a message was transmitted/received correctly. The sender calculates the CRC and sends it with the message. All recipients receive the CRC and verify it as well. If any receiver fails to verify the CRC he drops the message and raises an error on the CAN bus.

**Stuffing Bit (violet)** Stuffing bits are inserted automatically in the frame if there are 4 bits following each other with the same value. the inserted bit will be the inverse of the previous 4 bits. A stuffing bit is not part of the data sent and will be removed automatically after receiving it.

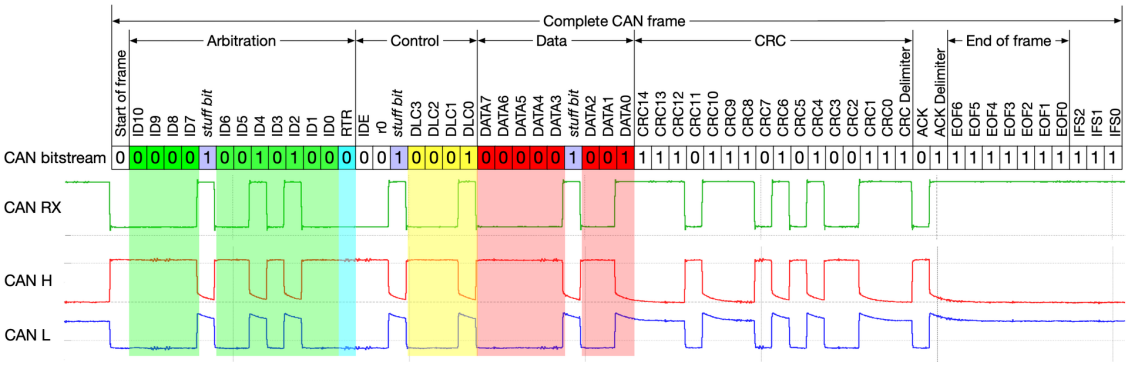


Figure 2.4: CAN bus frame taken from Wikipedia

The most important information related to the CAN bus is, that each message ID (often also referred to as arbitration ID) must be only sent by the sender and all other participants are only allowed to read the message but not to modify it or send it. If two ECUs write on the same ID with different payloads this would lead to an error state on the bus, which means that in the best case, a vehicle would switch into a safe state, indicate several issues to the driver, or in the worst case, undefined behavior will be triggered in the vehicle.

As already mentioned, a logic for prioritization is needed, to allow an ECU to send a message in time and to allow each participant to send a message at all. To allow this, the message ID is used in the so-called arbitration phase to decide on an order as to who is allowed to send a message on the bus now. Each ECU operates synchronized, which means that each ecu tries to put a bit on the bus at the same time. Due to this synchronization mechanism, the arbitration phase starts at the same time. To decide which participant on the bus is allowed to write, there is a dominant bit and a non-dominant bit. Because of the electric properties of the CANbus the 0-bit is the dominant bit. If now two ECUs want to write at the same time, the one with the lower message id wins. Table 2.1 shows an example of such an arbitration phase.
For example, we have two ECUs trying to write a message: *ECU A* and *ECU B*. *ECU A* uses the message id 80 to write a message, *ECU B* uses message id 89. As Table 2.1

| | **Message ID** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** |
| **ECU A** | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **ECU B** | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | | | |
| **ID written on CAN** | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

Table 2.1: CAN message id arbitration phase

shows, both ECUs start to write at the same time (at bit position 11). While writing the first bits of their message id they do not notice each other, as the id 80 (in binary form: 000 0101 0000) and id 89 (in binary form: 000 0101 0001) have the first 7 bits in common. At the point in time when bit 4 is written, *ECU B* notices that there is a dominant bit written by someone else and stops the transmission.

| Timestsamp | CAN line | CAN message id # payload |
|---|---|---|
| (1625466622.926576) | can0 | 121#708FCD0019000000 |
| (1625466622.926807) | can0 | 126#1388D70270BA9FAF |
| (1625466622.927052) | can0 | 130#0000000352FF0FB5 |
| (1625466622.927297) | can0 | 135#0000000000F00000 |
| (1625466622.936116) | can0 | 15D#1E00F00000400054 |
| (1625466622.936365) | can0 | 12B#FF8FF80007800055 |
| (1625466622.936570) | can0 | 126#1388D70270BA9035 |
| (1625466622.936814) | can0 | 130#0000000352FF002F |

Table 2.2: reading CAN communication with candump. We see the timestamp on when the message was received, the CAN line, and the message ID and message, separated by a #

Table 2.2 shows how real CAN communication looks like[2]. Observing the time delta between each CAN message shows, that the messages are sent very fast. further, we can see that not all messages are sent in the correct order: on the CAN-bus different messages are sent in different frequencies. Also, the frequency is not fixed. For example: if the ESP is taking over active intervention, the CAN communication frequency gets higher and messages might be sent each 20ms, as it starts to provide information to regulate or instrument several functions in the vehicles like the ECM. Still, this hexadecimal representation is not very readable to human eyes and we also do not know which bits of the message are used to calculate which data point (either if it is a float or integer). This information is contained in the Database CAN (DBC) file.

### DBC file

As a standard CAN message has a size of 8 bytes, each bit can be used to transfer information. A boolean value can be transferred as one bit, 0 for false, 1 for true. An integer can for example be transferred as 4 bit representation. To finally convert the bits of a message into real values, the DBC file is used. OEMs try to keep them secret because with being able to read the CAN communication competitors might be able to get a better understanding of the functionalities built into a vehicle. However, out on the internet several DBC files can be found. Reverse engineering of CAN messages is very time-consuming but still possible.
With the help of the dbc file a CAN message can be decoded into several different signals. The DBC file contains the exact description for the CAN message including information for byte order, data type (integer, float), length of signal, offset and multiplication factors as well as minimum and maximum value.

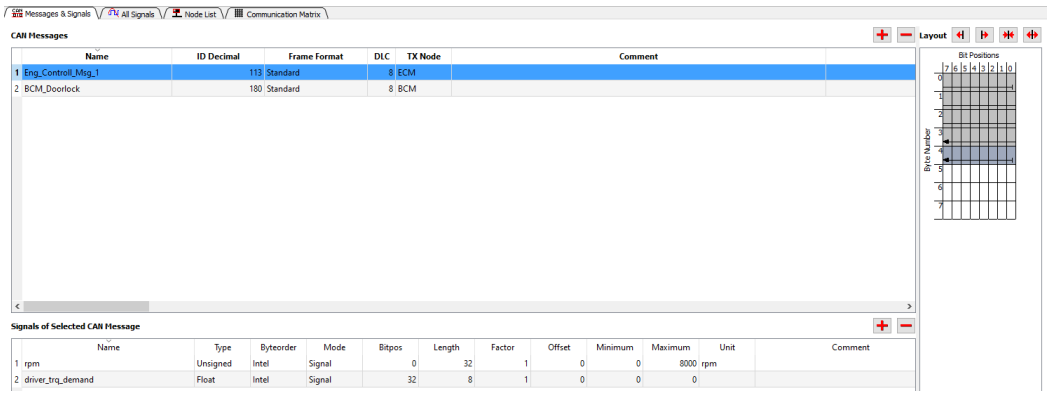


Figure 2.5: Screenshot of an example dbc file in the kvaser dbc editor

Figure 2.5 shows as an example that the CAN message with the id 113 and message name Eng_Controll_Msg_1 decodes to 2 signals: RPM and driver_trq_demand.

[2]to reproduce use *candump can0 -L*

### SecOC

Because CAN does neither provide confidentiality, integrity nor authenticity, a new mechanism was needed to introduce authenticity to CAN communication. Protection has become a strong requirement because of the new vehicles being interconnected with the World Wide Web. As attackers already have shown that it is feasible to get remote root access to an ECU it becomes important to ensure at least the authentication of messages being sent on the bus. Because of that, in 2014 AUTOSAR [3] released [AUT21] the standard for SecOC. SecOC aims to introduce authenticity and replay protection on the CAN line and to prohibit unauthorized third parties from sending valid CAN messages. Valid means that an attacker can still send a message, but he is not able to calculate the correct Message Authentication Code (MAC) for the message.
Therefore a CAN message gets reduced in size. not all of the 8 bytes contain a payload, but some of them are used to attach a freshness value and the MAC. The MAC is based on symmetric ciphers (Advanced Encryption Standard (AES) with 128-bit).

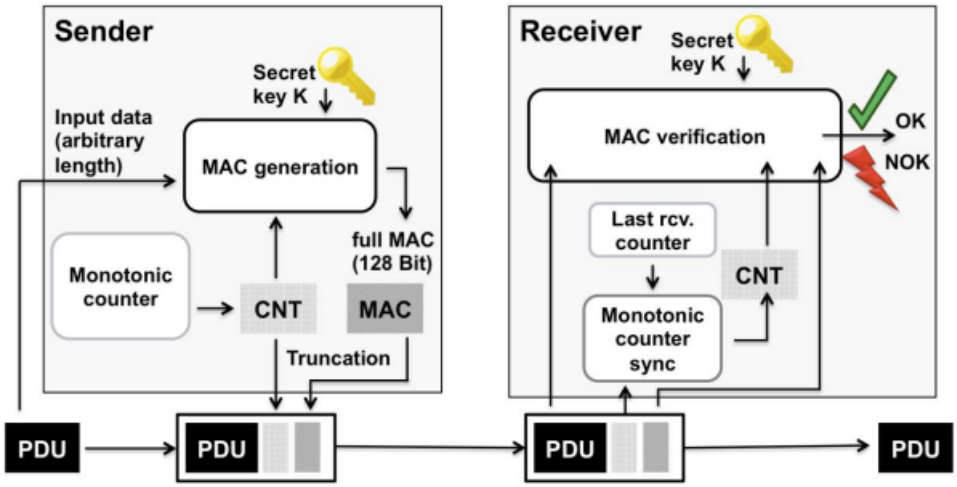


Figure 2.6: Image taken from [AUT21] Figure 3. and outlines how the data flow works in an AUTOSAR setup using SecOC

Figure 2.6 outlines how the data flow works with SecOC. A Protocol Data Unit (PDU) of arbitrary length is used as input for the SecOC module and the corresponding MAC is calculated. PDU[4] is introduced to create an abstraction layer in regards to the used protocol on the CAN line. One example is the conventional communication used in the vehicle which is based on a DBC file, but there are other protocols spoken on the CAN

[3] AUTOSAR: AUTomotive Open System ARchitecture, is an open platform of all OEMs, suppliers and more to standardize software in a vehicle
[4] PDU is just a general term of the message send. In our case a PDU would simply be a CAN message.

line like X-Calibration Protocol (XCP)[5] or UDS[6].
Because of the counter introduced in this module, also replay attacks should be mitigated. A replay attack is an attack where an attacker records some messages and executes an attack by just sending (replaying) the recorded messages again.
However, how the symmetric keys are exchanged, or how to truncate the AES-128 MAC is not defined in the SecOC standard. Also, there is an issue related to CAN and the payload size: the recommended size of the truncated MAC is 64 bits which means 8 bytes. For that reason there are two options mentioned: send the message and the payload first and attach the Mac in a following second message, or accept a higher security risk and add an even smaller MAC. This is up to each OEM on its own. Further, a strategy on how to deal with missed or lost messages is also not defined - as the receiver will be out of sync with the last received counter compared to the sender.

## 2.2 Unified Diagnostic Service

UDS is one of the most important CAN protocols within a vehicle. This protocol is used to retrieve diagnostic information from a vehicle and further to update software in a vehicle. This can be done by physically connecting to a vehicle using the OBD port (Figure 2.7), which is the diagnostic port also used by mechanics if you visit a car workshop, or by using a remote backend service that makes use of the vehicles internet connection, which will also utilize UDS. Of course, it is not sufficient to have an OBD plug available, it is also needed to have the appropriate software available that can utilize the UDS protocol for the specific vehicle and its ECUs. Very similar to SecOC the UDS protocol is standardized, but not its implementation. Because of this, the protocol is used differently at each OEM.

The protocol UDS is described in the [com15] standard and specifies how a tester (tester can be seen as a client) can make use of diagnostic functions of an ECU. The ECU is basically acting like a server in this protocol. The specification of UDS is done on the OSI layer 7, which means that the protocol itself is independent of the data link layer (so it can also be used on other bus systems than CAN). As part of the standard UDS implements an authentication requirement, which should protect critical functions like the firmware update from being used without authorization to do so.

The UDS protocol works very similar to a web service, where requests and responses are exchanged between a client and a server. A tester (can be seen like a client), is usually software running on a car workshop computer and connects to an ECU (acting as a server), see Figure 2.8, by sending a certain message with a specified message-id. Each ECU has its own CAN message ID, on which it is listening for UDS messages. Further,

[5] XCP is heavily used during development for changing the calibration of ECUs in a vehicle. XCP is disabled after the start of production.

[6] UDS is very similar to XCP but also provides the ability to overwrite the software of an ECU. It is the standard protocol used after production, also in workshops to access a vehicle and to retrieve e.g. diagnostic information

Figure 2.7: Picture of a connected OBD plug in a conventional vehicle

it is using another CAN message ID to respond. Table 2.9 shows an example message ID mapping for 4 ECUs.

The message sent by the tester will contain a so-called Service Identifier (SID) which is used to trigger a certain function on the ECU (compare able with a port and certain API call on a server). If the request is correct, the ECU will answer on another message ID with the corresponding response SID indicating a positive or negative processing. Due to this setup, it makes no difference, if there is a direct connection between the Tester and the target ECU, or if there is one or more gateways in between. It is only required that the gateway is forwarding the correct messages. If UDS is used over the CAN line, there needs to be two CAN message IDs reserved per ECU for communication. It is best practice to have an offset of 8 between the read and write addresses. However, this is not used everywhere.

As the 2.10 outlines only some service IDs as examples, there are many more. Some can be carried out without authentication and some do require authentication. Reading the

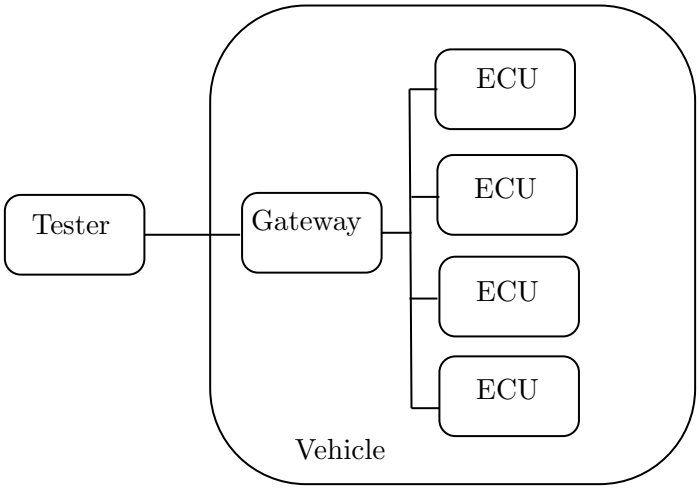


Figure 2.8: UDS setup and communication flaw

| ECU | UDS Read ID | UDS Write ID |
|---|---|---|
| ECU A | 7C0 | 7C8 |
| ECU B | 7D0 | 7D8 |
| ECU C | 9A1 | 9a9 |
| ECU D | 9CA | 9CF |

Figure 2.9: Example mapping for UDS CAN IDs within a vehicle

| Service | Request SID | Response SID | Description |
|---|---|---|---|
| Diagnostic Session Control | 0x10 | 0x50 | Common ECUs support three different types of sessions: a programming session, an extended diagnostic session, and a safety system diagnostic session |
| Security Access | 0x27 | 0x67 | Desc Sec Access |
| Tester Present | 0x3E | 0x7E | Desc Tester Present |
| Read Data By Address | 0x23 | 0x63 | Desc Read |
| Write Memory By Address | 0x3D | 0x7D | Desc Write |
| Request Download | 0x34 | 0x74 | With Download here actually the ECU (Server) is going to download the software from the Tester (Client) |
| Request Upload | 0x35 | 0x75 | Here, the ECU uploads its firmware to the tester |

Figure 2.10: Some examples of UDS commands and their service IDs

diagnostic trouble codes on service ID 0x19 is possible for everyone without protection. However, accessing more sensitive functions, like the upload of firmware to the ECU, requires authentication first. If you want to flash (upload) new software to an ECU one first needs to carry out the authentication process successfully. If request security access (0x27) is used, e.g. in case of a firmware update on a normal CAN bus, this works as follows (please refer to Figure 2.11 to have a visual flaw of the message exchange):

1. The tester first of all checks if the ECU is ready and available by using the tester present command (8A0 is the CAN id where the target ECU is listening and where the tester is sending his message on, 3E00 is the UDS service ID (3D) + payload (00))

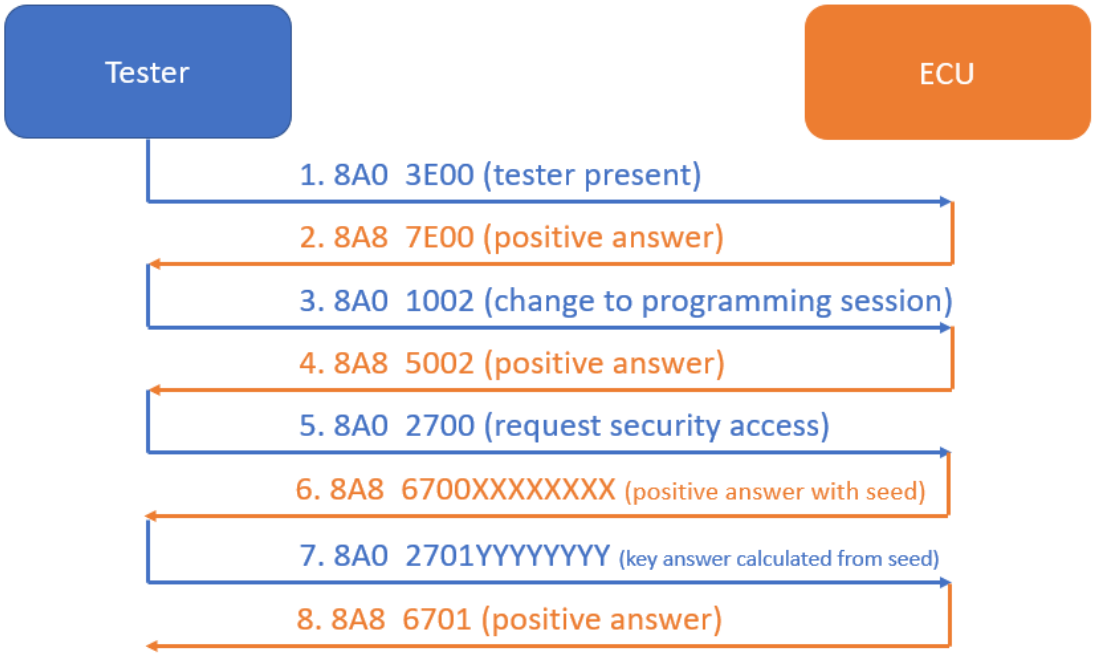


Figure 2.11: UDS Authentication flow example for successful authentication. 8A0 is the CAN ID on which the ECU is listening. On CAN ID 8A8 it is answering, here the Tester is listening on for this specific ECU.

2. If the ECU is ready it will answer with a message containing positive acknowledgment of the request on message ID 8A8. The Tester needs to listen on this CAN message ID.

3. Now a request is sent to change the programming session. 10 means change session, 02 is the identifier for the programming session.

4. The ECU again answers with a positive acknowledgment of the request. 50 is the positive answer to the request[7], and 02 confirms the programming session.

5. Now the request for security access is sent by the tester

6. ECU answers with a positive acknowledge and appends <6 bytes of data (represented with XXXX in Figure 2.11). This data is called seed and is used by the tester to calculate a session key.

7. The tester uses a secret function (containing one or more secret keys) and the received seed to calculate the key which he sends back to the ECU (time which can

[7]it is common practice that the positive answer is of hex 40 higher than the request, but as Miller and Valasek outlined in their work, not every OEM stick to this best practice.

be used for calculating the session key is usually very short. If it takes too long the ECU will not accept the key and abort the session).

8. After the ECU verified the received key it answers with a positive acknowledgment.

After doing this security handshake the functions which are assigned to the programming session can be used by the user/tester. The work done in this thesis will be mostly focused on synthesizing the function request security access function.

## 2.3 Security standards in the automotive domain

For testing and validation of security features in the automotive domain several standards have to be considered. Most important is the ISO21434 Road Vehicles Cybersecurity, which specifies how secure development in the automotive domain takes place. Further, the UNECE R155 brings additional requirements related to cyber security to OEMs if they want to get a successful type approval for their vehicles.

### 2.3.1 ISO 21434

The ISO 21434, or ISO/SAE 21323 Road Vehicles – Cybersecurity Engineering, is a standard focusing on the cybersecurity of road vehicles. It is very closely related to ISO 26262, a safety standard for road vehicles, which was also developed by the International Organization for Standardization (ISO) and the Society of Automotive Engineers (SAE). Iso 21434 aims to provide guidelines as well as requirements for the implementation of cybersecurity controls to protect vehicles from cybersecurity threads. The standard covers all phases, beginning with design, development, testing, production, and in-use phases. Further, it encourages engineers to design dedicated end-of-life concepts. In easier words, ISO 21434 defines a framework to help engineers build a vehicle with security in mind through the complete lifecycle. The ISO 21434 can be abstracted into 4 major chapters:

1. Risk Management: the standard wants not only to establish a security culture within the company but also to continuously identify potential cybersecurity risks and threads to vehicles and systems. These risks need to be addressed and managed and strategies need to be developed to mitigate these risks.

2. Design and Development: Many guidelines are defined to integrate cybersecurity into every stage of the vehicle's lifecycle. Software, hardware as well and different protocols used within the vehicle, e.g., for communication, are in scope.

3. Security Validation and Verification: Several methods for testing and validation are defined through the complete lifecycle of the vehicle. These methods are not only on the architecture or formal level but also on the lowest hardware level, including the requirement for penetration testing.

4. Incident Response: Finally, also incident response is required by the standard, to make sure that security is maintained through the very long lifecycle of a vehicle.

ISO 21434 aims to provide several benefits. First of all to enhance the security of vehicles. Secondly, it aims to provide a framework to standardize and address cyber security in vehicles for the industry. That enables the industry to use common language, exchange best practices, and improve the security posture for the whole industry. Regulatory requirements are met on a more standardized basis, as they are already required by UNECE R155 and will be required even more in the future. Therefore, it is an essential part for a company to be ready to meet potential future regulatory challenges. In the end, also consumer trust will be enhanced, as several reports in the news outline that vehicles are easy to hack. It will be enhanced because vehicles are engineered with security in mind and the possibility of adaptions in case of need threats are discovered.

### 2.3.2 UNECE R155

UNECE Regulation No. 155, also known as UNECE R155, is a regulation developed by the United Nations Economic Commission for Europe (UNECE). It addresses cybersecurity and software updates for vehicles, specifically focusing on ensuring the security and robustness of vehicle software systems and managing software updates throughout the vehicle's lifecycle. UNECE R155 is part of several regulations to improve vehicle safety and cybersecurity.

The UNECE R155 defines requirements that are necessary for the type approval of new vehicles. These requirements consist of organizational, as well as technical requirements. Organizational requirements are for example: the establishment of a security culture or the introduction of a certified cybersecurity management system (CSMS). Technical requirements are for example the introduction of a hardware security module for the storage of keys, encrypted connections, authenticated software updates, and so on. Without a successful type of approval, new vehicles are not allowed to be sold within certain regions.

### 2.3.3 ISO 21434 and UNECE R155

The UNECE R155 and the ISO 21434 are complementary standards, partly addressing different aspects of cybersecurity within a vehicle but still having the same goal in mind: enhancing the security of vehicles. UNECE R155 specifies requirements that must be met by the manufacturer to obtain the type approval. It is doing so by requiring necessary cybersecurity measures to enhance the security resilience of vehicles and defining a minimum-security concept for firmware updates. ISO 21434 supports these requirements by providing a comprehensive framework to engineers to help create a secure vehicle including the proper development and testing documentation – and it is doing so by bringing security into all stages of a vehicle lifecycle.

## 2.4 Syntax Guided Synthesis

As already mentioned, SyGuS is used to break the authentication mechanism during the man-in-the-middle attack. This section describes what SyGuS is and how it works.

Syntax-Guided Synthesis (SyGuS) is a technique used in computer science, particularly in the field of program synthesis, where the goal is to automatically generate a program that meets a specified set of requirements. It combines elements of formal language theory, constraint solving, and program analysis to create programs that adhere to a given syntax while satisfying certain semantic conditions.

A synthesis conjecture looks as follows (also called spec(f)):

$$\exists f.\forall x.P(f,x).$$

The property P is part of a background theory such as e.g. bit-vector or linear arithmetic. With this, it is already solvable with a Satisfiability Modulo Theories (SMT) solver. In addition, in SyGuS, as defined by [ABJ+13], the body of the function is generated by a grammar, starting with A (see Section 2.4.1 for the grammar definition):

$$A -> A + A| - A|x|y|0|1|ite(B,A,A)$$
$$B -> B \wedge B|\neg B|A = A|A \geq A| \bot$$

This is called the syntax(f). Now, a solution enumerator gives solutions for f until the solution verifier is able to find f with no counterexamples with respect to the spec(f).

In contradiction to the classical SMT problem solutions, the CVC4 synthesizer used utilizes state information from the solution verifier and returns it back to the solution enumerator to improve and fine-grain the calculations to solve the conjecture.

### 2.4.1 SyGuS input specification

The work in this thesis follows the SyGuS language standard in version 2.0 as defined in [Muk19]. Overall many of the constructs that are present in this language standard are also present in the SMT-LIB 2.6 standard. [Muk19] defines the input for a SyGuS solver as follows[8]:

SyGuS input is created out of 0 or more commands.

```
<Sygus> ::= <Cmd>*
A command can be:
<Cmd> ::=   (check-synth)
      |     (constraint <Term>)
      |     (declare-var <Symbol> <Sort>)
      |     (set-feature :<Feature> <BoolConst>)
```

[8] this definition of SyGuS is not the complete one. It is restricted to the definitions needed for this work

```
|   (synth-fun <Symbol> (<SortedVar>*) <Sort> <GrammarDef>)
|   <SmtCmd>

<SmtCmd> ::= (declare-datatype <Symbol> <DTDec>)
          |  (define-fun <Symbol> (<SortedVar>*) <Sort> <Term>)
          |  (set-logic <Symbol>)
```

SmtCmds are, as mentioned, from the SMT-LIB 2.6. The Cmd is a pure SyGuS definition. A Term is used to specify the grammar of the program as well as the constraints:

```
<Term> ::= <Identifier>
        |  <Literal>
        | (<Identifier> <Term>+)
```

Literals are a sequence of characters. These are used to express constant terms or values. A literal can be either a Numeral, Decimal, BooleanConstant, HexConst, BinConst or a StringConst. Within the input grammar for the synthesizer, a hex value starts with #x, while a binary value starts with #b. Identifiers are an extension of symbols. Symbols are a non-empty sequence of upper and lower case letters as well as digits and predefined special characters. The only exceptions are reserved words. These can be found in Annex A of [Muk19]. With this sequence of commands, a synthesis conjecture is defined and passed over to a solver.

# Chapter 3

# Related Work

In this chapter, all the related work other researchers have already done is discussed. However, it needs to be outlined, that finding good examples of automotive security research is not very easy. The biggest hurdle is, that very often no detailed information is disclosed, which makes it hard to understand how a vehicle got hacked and influenced. Therefore, we discuss the most important remote hack to a vehicle, namely the so-called Jeep Hack from 2014 in Section 3.1. Then we continue with an overview of other hacks in the automotive domain, which show what can go wrong but do not directly follow the approach we are following in this thesis. Last but not least, we discuss in Section 3.3 a paper that also utilizes Z3 to generate a secret key in a way different from ours.

## 3.1 The 2014 Jeep Hack

This section describes the related work to this thesis from Charlie Miller and Chris Valasek. The main work of them, which is "Remote Exploitation of an Unaltered Passenger Vehicle" [VM15b], is described in this section. Further, from public media this paper is also known as "The Jeep Hack".

In 2012 the authors started to work on automotive cyber security, with the goal of developing a testing platform that would help engineers to enter the automotive domain for security testing more easily. In the same year, they could show that it is already possible to influence the steering of vehicles by sending CAN messages. According to [VM15b] this is possible because new features like a parallel parking assistant are introduced to vehicles. Still, the authors faced the issue, that this is only possible due to the physical access to the vehicle. In 2014 they released their remote attack surface paper [VM14], which is also used in section 2.1 and onwards, to generate a basic understanding of the architecture of the vehicle. While working on this, they could identify the Jeep Cherokee, built in the year 2014, as a target worth testing. This was for basically three reasons: simple architecture, many interesting features and a price which was in the range for them to buy the vehicle and test with it. They had a clear goal: To show the industry that it is possible to attack an unaltered vehicle. At this point it is important to mention, what the authors of the paper are mentioning: They choose the car because it looks like it is most promising for for them to be successful. This however does not mean, that other cars can not be hacked. The network architecture of the Jeep under

attack is represented in Figure 2.3. The authors further identify that the radio system looks very promising as a target for attack, as it is directly connected to all interesting CAN-lines which could be utilized to influence the behavior of the vehicle. They continue by explaining several features of the vehicle, like the ACC, forward collision warning, and so on. We are skipping this part here, as it is referenced in section 2.1.1 of the background chapter of this work.

### 3.1.1 Remote attack surface

For the Jeep, the authors identified the following remote attack surface: The remote keyless entry system and the tire pressure monitoring system on one ECU and Bluetooth, FM/AM/XM, cellular network, and internet/apps on the infotainment system (referenced as radio in the paper). As previously mentioned, the infotainment system is connected to both CAN-lines which makes it of greatest interest. They made the following observations:

- Tire Pressure Monitoring System: It is likely that an attack can trick the car into safety-critical behavior by tricking the sensors. However, the attack surface for remote code execution is small.
- Remote Keyless Entry: Here it is the same, the technique is rather small and even thus it is sending radio signals, the attack surface related to remote code execution is again very small.
- Bluetooth: Is of great interest to the authors. First of all, because they say that the Bluetooth stack is big and already has known vulnerabilities. They determine that there are 2 scenarios for an attack: the dangerous one, via an unpaired phone, as this service can be reached by everyone, and secondly via a paired smartphone.
- WiFi (cellular network): The authors outline, that the WiFi hotspot a vehicle is creating to share the internet connection of the vehicle with the passenger is also of interest. Further, they state that hacking access points have been frequently documented.
- Telematics/Internet/Apps: The telematics unit is referenced as the "holy grail". This is due to the fact, that even if it is not directly connected to any CAN bus line, it is still connected to microphones, and speakers and is remotely transferring data/void to other locations. In the example of the Jeep, they again outline that the telematics unit is connected to all CAN lines. The telematics unit in the Jeep is from the brand Harman Uconnect.

### 3.1.2 Uconnect System

In this chapter, they explain the Uconnect system in detail as it is their main focus point for further attacks. It offers cellular connection, WiFi, navigation, and apps. Further, the authors outline that these systems are not only used within the Jeep, but also at

other OEMs like Fiat Chrysler Automotive, Dodge, Ram, and so on. The Uconnect system is operated on a QNX operating system on a 32-bit ARM processor. Further, the authors outline, that the ISO images of this operating system can be downloaded from the internet and reverse-engineered, which they ultimately also did. They continued with disassembling different binaries of the image and made some very interesting observations related to the generation of the WiFi password. They identified that the password is generated when the ECU boots for the first time. With this information, a list of possible passwords can be generated and an attacker can brute force into the WiFi. The authors assume, that it is possible to calculate all possible values of a month in 2 minutes and a year in less than half an hour. However, as they investigated how the time is set, they noticed in their example, that the ECU takes 00:00:00 o clock on 1 January 2013 as the starting date. Then the ECU tries to fetch the real date and time. However, a WiFi password is generated before the time is synchronized successfully, which restricts the possible values of a password rapidly. In the next step, they checked which ports were open in the WiFi network. The open ports are displayed in Figure 3.1.

```
# netstat -n | grep LISTEN
tcp        0      0  *.6010                 *.*                    LISTEN
tcp        0      0  *.2011                 *.*                    LISTEN
tcp        0      0  *.6020                 *.*                    LISTEN
tcp        0      0  *.2021                 *.*                    LISTEN
tcp        0      0  127.0.0.1.3128         *.*                    LISTEN
tcp        0      0  *.51500                *.*                    LISTEN
tcp        0      0  *.65200                *.*                    LISTEN
tcp        0      0  *.4400                 *.*                    LISTEN
tcp        0      0  *.6667                 *.*                    LISTEN
```

Figure 3.1: This image from [VM15b] lists all open ports found in the QNX operating system.

Very soon the authors identified port 6667 to be of interest for them. They learned, that this is a D-Bus[1] over IP service and that they can call remote procedure calls (RPC) using this port. The authors found several interesting RPC calls on this port, enabling them to for example turn up the radio volume. In the next step the authors of the paper targeted to jailbreak the Uconnect system. They did this by abusing a flaw in the verification of the USB stick while the ECU was rebooting at the installation of the update. However, the authors note that later on they discovered that it is not needed to jailbreak the system to remotely attack the Jeep. However, for better understanding and learning all of these findings, it was necessary to go down this road. With a modified firmware update they managed to change the root user password and connect to the system using SSH.

[1] D-Bus is a service providing inter-process communication on the same machine.

### 3.1.3 Exploitation of the D-Bus Service

Because the D-Bus service is available without authentication and is responsible for carrying out inter-process communication, the authors targeted this service. They thought, that because this service is located internally, it might be implemented with fewer security measures.
In this D-Bus service, they found a service called 'NavTrailService' which they could abuse in a way, such that they are easily able to execute arbitrary commands. To the author's surprise, it was even more easily possible to execute arbitrary commands, as there was also a method called "execute", allowing the execution of commands.

```
#!python
import dbus
bus obj=dbus.bus.BusConnection("tcp:host=192.168.5.1,port=6667")
proxy object=bus obj.get object('com.harman.service.NavTrailService','/com/ha
rman/service/NavTrailService')
playerengine iface=dbus.Interface(proxy object,dbus interface='com.harman.Ser
viceIpc')
print playerengine iface.Invoke('execute','{"cmd":"netcat -l -p 6666 |
/bin/sh | netcat 192.168.5.109 6666"}')
```

Figure 3.2: This image from [VM15b] shows the short exploit written in Python to open a remote root shell using the D-Bus service

Because the Uconnect system has a preinstalled version of Netcat, the exploit for the D-Bus service was easily written by the authors. The exploit can be seen in Figure 3.2. With this, it was clear, that jailbreaking the head unit was not necessary. With this remote shell, the authors can explore the complete infotainment system and utilize all the features present. Even writing exploits is possible, due to the root rights. The authors continued to abuse present LUA scripts to influence the behavior of the radio of the vehicle. Some of the findings on this interface are:

- GPS: the authors managed to retrieve the GPS coordinates of the vehicle via port 6667. With this, they can track the vehicle.
- Heating System: With another call, they could easily increase the fan speed of the heating system within the vehicle.
- Radio Volume: was also possible to be manipulated via this interface.
- Radio Station: selection of a different radio station is possible.
- Display: only under the two conditions to place an image on the file system and to format it in the correct resolution they have also been able to easily display an image of their choice on the infotainment systems screen.
- Killing service: further they found a way to kill functions in such a way, that for example, the volume button does not work anymore. With this, they can just turn up the volume and kill the buttons functionality and the driver will not be able to turn the value down.

### 3.1.4 Cellular Exploitation

Until now the authors of the paper have only been able to execute these commands, while jailbreaking the firmware or by having a WiFi connection established. Both versions require physical interaction with the vehicle. Therefore the next step was to explore how the cellular connection works to see if it is possible to completely remotely compromise the vehicle. The authors continued to look in the Uconnect system for the IP address it is using to communicate with the internet. They observed, that the internet-facing IP address is a Sprint network internal IP address. They identified two class-A address blocks to be presumably reserved for vehicles: 21.0.0.0/8 and 25.0.0.0/8. Last but not least they identified that the vulnerable D-Bus service on port 6667 is bound to this sprint internal IP address and that it can be called from the sprint network. In the next step they acquired a burner phone[2] Sprint device. With this, they could verify that they are still able to remotely interact with the Jeep under test. Even further they noticed, that they can interact with each Sprint device all over the country. They used their Sprint connection to scan through the network and identified 16 different vehicle types as potentially vulnerable. Potential, because they did not try to exploit any of them, but they identified that these vehicles are listening for the D-Bus service. With this, they estimated to have found between 292000 and 471000 vulnerable vehicles.

### 3.1.5 Getting access to the CAN bus

After being able to remotely compromise any vehicle using the Sprint network and the D-Bus service, the authors started to investigate what is necessary to access the CAN bus. The chip (the chip is called Renesas V850/Fx3) in the Uconnect system used to read CAN messages was only able to read messages but could not send messages. Because for the V850 there is no cryptographic signature used to validate a firmware, they were able to reverse engineer the firmware and reflash it with a version that is able to write on the CAN bus.
While being able to send messages to the CAN line, Miller and Valasek now have been in need of a better understanding of the meaning of these messages. To do so, they purchased the mechanic's tools (previously referenced as Tester in the UDS protocol). By reverse engineering the firmware of this tool, they acquired the necessary information for actions like the activation of windshield wipers. However, their goal was to Figure out the SecurityAccess algorithm used to unlock the ECU. They found several encrypted files, which are used for the SecurityAccess handshake, containing the function to calculate the key from the seed. However, it had to be reverse-engineered, which of the files is the correct one for the ECU they want to attack.

However, as flashing the ECU was not in the author's scope, they continued to investigate how the CAN messages work to manipulate the behavior of the vehicle. With the help of the mechanics tool they were able to identify CAN messages and send the following messages:

[2] burner phone is an inexpensive phone often used for anonymous usage

00A6.esu 00A6.su

```
// javascript
var name = "S5_ADCM_MSRT_Sup0026_Var40";
var level = 5;
var version = "$Revision: #1 $";

var KEY_CONSTANT_1 = strToBigInt("70e90220", 16);
var KEY_CONSTANT_2 = strToBigInt("29e3e35b", 16);
var AND_CONSTANT = strToBigInt("ffffffff", 16);

function calculateKey(seed) {
    if (seed.length < 4) {
        throw "seed length must be 4";
    }

    var seedBigInt = arrayToBigInt(seed, 0, 4);

    var tempSeed = leftShift(andInt(rightShift(seedBigInt, 16), 0xFF), 24);
    tempSeed = add(tempSeed, leftShift(andInt(rightShift(seedBigInt, 24), 0xFF), 16));
    tempSeed = add(tempSeed, andInt(rightShift(seedBigInt, 8), 0xFF));
    tempSeed = add(tempSeed, leftShift(andInt(seedBigInt, 0xFF), 8));

    var shiftSeed = and(add(leftShift(tempSeed, 11), rightShift(tempSeed, 22)), AND_CONSTANT);

    var key = and(xor(shiftSeed,xor(KEY_CONSTANT_1, and(KEY_CONSTANT_2, seedBigInt))), AND_CONSTANT);

    return toArray(key, 4);
}
```

Figure 3.3: This image from [VM15b] shows the decrypted file containing the SecurityAccess function used to calculate a key from the seed provided by the ECU

- Turn signal: With the CAN message with ID 04F0 they can activate the indicators of the vehicle.
- Locks: with message 05C§ they can lock or unlock the vehicle.
- RPMS: with 01FC and a checksum algorithm which they reverse-engineered, they could even manipulate the tachometer.
- Kill engine: within a diagnostic session (UDS protocol session type) they could turn off the engine at low speed.
- No breaks: If a diagnostic session can be established all breaks can be disabled, leading to a massive safety impact.
- Steering: By turning one ECU off, they managed to turn the steering wheel on at low speed on the author's demand.

Overall the authors managed, that after making this issue aware to Fiat Chrysler Automotive, on 24.07.2015 Sprint cellular network blocked all traffic for port 6667 (the vulnerable D-Bus service) and Chrysler voluntarily recalled 1.4 million vehicles. After this is fixed, these attacks can only be carried out in close range to the vehicle, by having WiFi access or by establishing a femtocell[3] connection.

[3]here, a Sprint network cellular tower is set to simulate a real Sprint network cellular tower, so that the vehicle connects to the femtocell

### 3.1.6 Authors conclusion

Miller and Valasek conclude, that after 3 years of research in the automotive domain, they were able to perform a remote attack on Fiat-Chrysler vehicles all over the United States. As they managed to influence the physical behavior of the vehicle, they published the work in the hope, that more secure vehicles could be built in the future.

## 3.2 Vehicle hacks from public news

However, since Jeep got hacked in 2014 the field of vehicle hacking has become increasingly popular. This Section lists several hacks in the automotive domain that have been published recently.

#### DefCon 2015: How to Hack a Tesla Model S

In 2015, two security engineers presented at the DefCon conference in Las Vegas [RM15] how they hacked into a Tesla Model S. To do so, Marc Rogers and Kevin Mahaffey took a Tesla Model S and physically analyzed its components. Over several months they learned how the system works and how they can hack into this vehicle. In the end, they managed to achieve big results: they could open the trunk of the vehicle, lock and unlock it and last but not least remotely kill the engine from an iPhone, while the Tesla was driving. However, the authors of the hack outline, that this was not a remote hack, but required physical access to the vehicle first. This is similar to how Miller and Valasek started their work on the Jeep, but in the end, they managed to create a remote exploit for the vehicle. Rogers and Mahaffey outline in addition, that they did not reverse engineer the CAN and how to write there, but they used an API within the infotainment system to influence the behavior of the CAN. Exploitation of the CAN was out of scope for them. Even thus the authors of *How to Hack a Tesla Model S* outline that the overall architecture of the Tesla is very secure, Tesla issued a patch just the day before they had their talk at the DefCon conference for the complete fleet.

#### Opening a BMW using a ConnectedDrive vulnerability

Also in 2015, the German automotive driving club ADAC reported [Hol15], that it was easily possible to open a locked BMW because of a vulnerability in the ConnectedDrive system of BMW. The setup used for this was barely easy: you needed a notebook and an IMSI[4] catcher. Overall, the attackers utilized an unencrypted connection between the vehicles TCU and the backend service of BMW. In summary, several flaws in the usage of crypto primitives have been uncovered in this hack, as for example symmetric keys have been shared over all vehicles, which makes it only necessary to steal the symmetric key ones and abuse it over the complete fleet. Last but not least, the major outcome if this analysis was, that BMW implemented everything and had the TLS certificates in

[4] A IMSI catcher is a device used to fake a mobile tower to smartphones. With that someone can perform a men-in-the-middle attack on the connected smartphones

place but did not activate TLS for this on the TCU. This was done after these findings had been notified to BMW to fix the issue.

**Web Hackers vs. The Auto Industry**

At the beginning of 2023, security researcher Sam Curry published a new post in his blog [Cur23], summarizing the findings he and some other researchers made when testing the automotive sector for vulnerabilities. KIA, Honda, Mercedes-Benz, BMW, Rolls Royce, and Ferrari are just a few of the big names for which they found vulnerabilities. This article is very interesting for 2 reasons. First of all, because they targeted several OEMs, without having physical vehicles in mind, but only from a web-pentesting point of view. Second, they did not only target the vehicles but also found vulnerabilities enabling attacks in development (e.g. access to a git repository), targeting the supply chain of a vehicle. In summary, they managed to:

- Remotely lock and unlock vehicles, start the engine, or stop it and honk vehicles. All of this for different OEMs. Mostly by only knowing the vehicle identification number (VIN) or the email addresses of the owner, which both are not considered secret information.
- Some accounts in the systems used to remotely manage a vehicle have been taken over completely, including the potential disclosure of private information about the vehicle owner. Further, they managed to lock the user out of the portal and change ownership of the vehicle.
- Access cameras of the vehicle to get up to 360° live views from vehicles.
- Vulnerabilities in SSO configurations enabled them to access Github instances, internal toolchains, and internal vehicle-related APIs,...

According to the author, the OEMs reacted very fast to the reported vulnerabilities and all of them have been fixed. However, these vulnerabilities show in a very frightening way, how attachable connected cars are.

## 3.3 Key generation using the Z3 SMT Solver

In 2015 Dennis Yurichev participated in a capture the flag hacking challenge where he was challenged with generating a password to unlock a file. He finally approached this topic by utilizing the Z3 constraint solver and wrote a whitepaper [Yur15].
As he analyzed the algorithm applied to verify the password, the challenge became clear: it is not only about finding a 32-bit value but about finding the correct number, represented as a string, to which certain properties for verification hold. The verification function applies several operations on the password string, which all need to be true for the file to be decrypted successfully. Figure 3.4 outlines, how the author utilized Z3 to generate the password to decrypt the file. Z3 found the shortest possible password, consisting out of 12 digits.

```
# Initialize the solver
s = Solver()

# Add the constraints
# Sum of first 5 digits should equal 21
s.add(d0 + d1 + d2 + d3 + d4 == 21)

# Product of 4 digits after the first should equal 480
s.add(d1 * d2 * d3 * d4 == 480)

# All digits must be within 0-9 inclusive
s.add(d0 >= 0, d0 <= 9)
s.add(d1 >= 0, d1 <= 9)
s.add(d2 >= 0, d2 <= 9)
s.add(d3 >= 0, d3 <= 9)
s.add(d4 >= 0, d4 <= 9)
for i in range(11, num_digits):
    s.add(  globals().get('d{}'.format(i)) >= 0,
            globals().get('d{}'.format(i)) <= 9
         )

# Digits from 5 to 10 should be 914323
s.add(d5 == 9, d6 == 1, d7 == 4, d8 == 3, d9 == 2, d10 == 3)

# Constraint involving Math.abs
s.add(Or(d1 + d2 - d3 == 1, d1 + d2 - d3 == -1))
s.add(Or(d1 + d2 - d4 == 1, d1 + d2 - d4 == -1))

# Last digit must be 9
s.add(globals().get('d{}'.format(num_digits)) == 9)
```

Figure 3.4: This image from [Yur15] shows how the author transformed the operations applied to the password into a constraint set for the Z3 solver in order to find the correct password.

The author concludes, that with the Z3 solver and formulation, the constraint satisfaction problem for this task leads to a much quicker solution than brute force. Further, he recommends that SMT solvers are a must-have tool for bug finding and fuzz testing.

# Chapter 4

# Vehicle Security Testing

This chapter describes the practical part of our work. First, it will describe the phase of analyzing the attack paths (section: 4.1), which finally leads to the decision to attack the UDS path of vehicle communication. It is very important to have at least read the chapter 2.2 to understand the UDS protocol and how the security access works. This security access function will be further referenced as an authentication function and be the primary subject of interest. The section In depth Security Testing UDS 4.2 describes the work done to achieve the goals of this thesis. The setup used to interact with the ECU using the CAN line is explained in 4.2.1 as well as the libraries used for SyGuS, the implementations to reach the goals of this thesis are described in 4.2.2 and the benchmarks for different attack methods to the algorithm is presented in 4.3.

## 4.1 Attack Paths

This section provides the reasoning, why UDS is of such importance and therefore our in-depth testing target. It creates a basic understanding of the impact within a conventional vehicle if this protocol has significant weaknesses. The major outcome is, that UDS is working on most likely every ECU within the vehicle and grants administrative access. It can be abused either by using an external interface like the OBD interface or by applying a direct physical men-in-the-middle attack.

To outline the meaning and importance of the UDS protocol, we created a very simple thread model of modern vehicle architecture. This simple model is visible in 4.1.

The model shows first of all two boundaries: the physical vehicle boundary, with which it can be determined if the attack vector can be executed from remote or with physical access to the vehicle. The second boundary shows the trusted environment, where all the operational and safety-critical ECUs are located.

- PTCM - Powertrain Control Module This module controls the engine, acceleration, and many more functionalities. It is internally connected via the Powertrain CAN to interact with other ECUs and with the PTCM vehicle CAN to the outer side of the vehicle, to be reachable for diagnostics using the OBD connection, as well as to receive remote firmware updates using the TCU. Last but not least it can route CAN communication from the PTCM vehicle CAN to the Powertrain CAN (e.g. diagnostic relay for internal ECUs like the BMS to the OBD port).

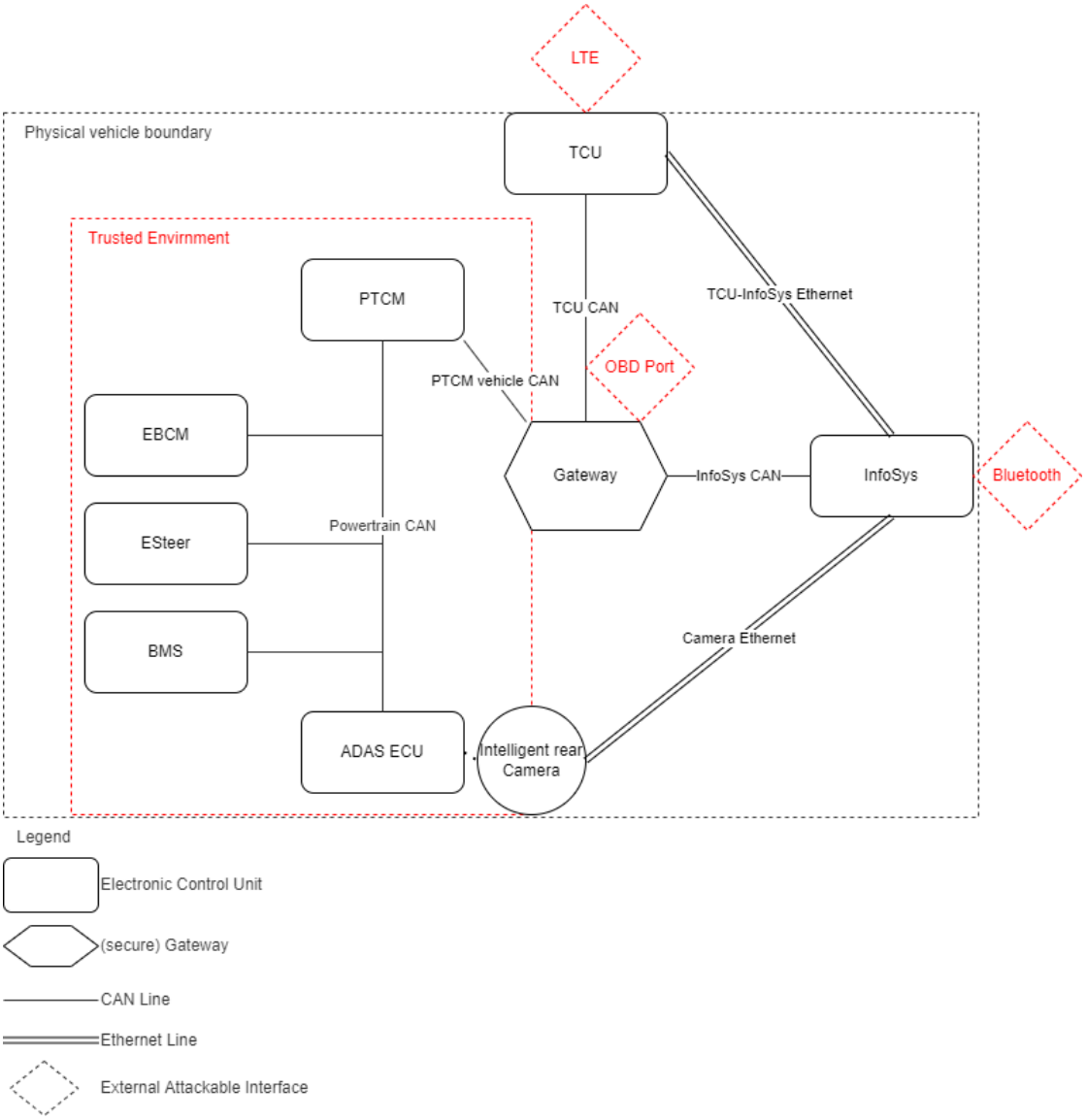


Figure 4.1: This Figure shows a simplified version of the current state-of-the-art vehicle infrastructure.

- EBCM - Electronic Brake Control Module Utilizes several sensors and actuators related to braking. Needs to communicate with the PTCM and other ECUs using the Powertrain CAN to execute its purpose safely.
- ESteer - Electronical Steering Controls several sensors and actuators related to steering. Communicates with the other ECUs using the Powertrain CAN.

- BMS - Battery Management System Controls power consumption of the battery. Communicates with the other ECUs using the Powertrain CAN.
- ADAS ECU Steers different sensors and camera information. Is highly communicating with other ECUs to command them through autonomous driving actions e.g. parking assistant. Communication is mainly done via the Powertrain CAN.
- Gateway The gateway is often also referenced as a secure gateway. It is specially hardened and is responsible for routing different messages (either on CAN or Ethernet...) between different connections. Routing is usually implemented as a whitelist, meaning that only predefined message IDs are routed from one interface to another. Very likely, the OBD port is connected to the gateway.
- TCU - Telematics Control Unit Is usually the breakout to the internet. It connects the vehicle with different backend services of the OEM. In this model it is connected via the CAN bus to the gateway, to serve functionality like firmware updates to all ECUs within the vehicle and via Ethernet to the InfoSys.
- InfySys - Infotainment System Is the visible heart of the vehicle to the end user. It provides not only navigation and radio functionality, but also it can set options within the vehicle (like the tire pressure reset) which propagates deep into the vehicle. Further, it has external facing interfaces like USB, Bluetooth, and so on. It also has an Ethernet connection to the camera system, as it will be using its display to reflect the camera's image to the driver.

When having a look at the simplified model in Figure 4.1, the three, red external attack-able interfaces jump into view. The Bluetooth and LTE connection is perfectly suited for a remote attack without a physical connection to the vehicle. However, for the master thesis, these two have the following drawbacks:
While Bluetooth would be more easily attackable, it might need a huge amount of time to reverse engineer the memory layout of the application to get an exploit properly working. For LTE, one would first need to intercept the connection between the LTE module and the receiver, which can bring one into legal issues and we would have to deal with all security measures implemented in the LTE stack first.

When having a look at the OBD port, one CAN protocol becomes very important: the UDS protocol. As this protocol is used, for example in car workshops to read out error codes and to perform firmware updates, it must be able to reach every relevant part of the vehicle. This means, via this channel, we can utilize every connection marked red in Figure 4.2. The gateway would also not block the messages, as it has to implement some rules to forward the messages, as well as the PTCM which serves as a local gateway. With this, we can reach almost every ECU from the OBD port onwards. Further, the same operation will likely be possible from the TCU CAN, in case remote firmware updates are implemented, or from the InfoSys. As by the nature of CAN, we could also open up the CAN line at any point within the vehicle and manipulate the firmware of ECUs, if

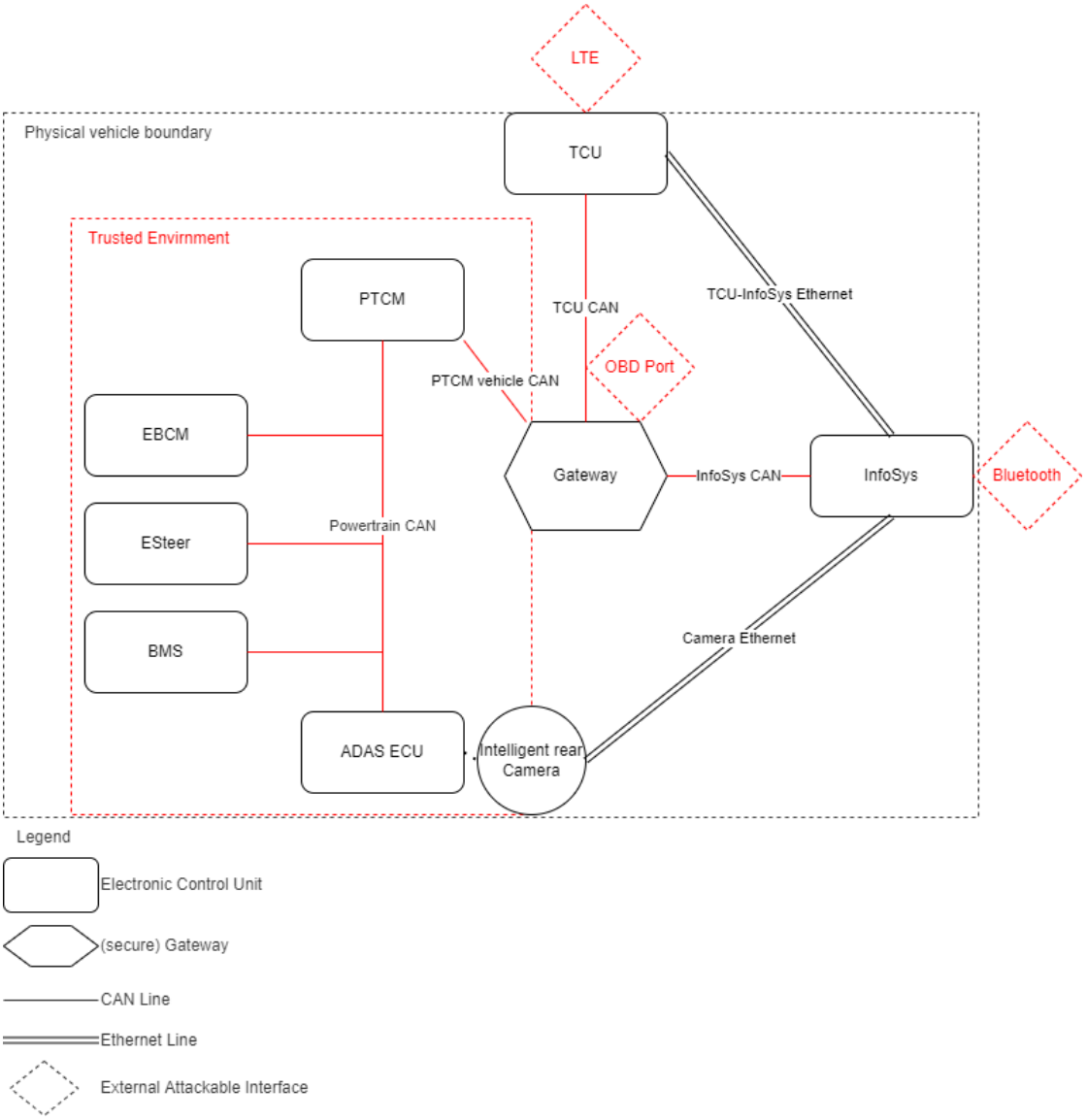


Figure 4.2: This Figure shows a simplified version of the current state-of-the-art vehicle infrastructure, highlighting in red the CAN paths that can be infected if we manage to break the UDS authentication mechanism.

we manage to break the UDS authentication protocol, because CAN assigns messages to a CAN ID, but it can't verify who send the message. So, if the message is correct, the recipient will not know if it was sent by the expected sender or by a man-in-the-middle. Last but not least, if one manages it to take over, for example, the InfoSys, the UDS protocol would be the ideal choice to further escalate privileges into the vehicle.

For that reason, we choose UDS to be one of the most important in-depth protocols that we like to test and attack, as UDS is the core protocol to manage a vehicle that is on the public road.

## 4.2 In depth Security Testing UDS

This section explains the hands-on and exploitation part of the work. First, in section 4.2.1 the technical setup used to prepare the men-in-the-middle attack is described. In 4.2.2 the SyGuS work is described, outlining the authentication function 4.2.2 we are synthesizing. Further, it explains the journey from our first try with the synthesizer 4.2.2 and the issues we faced, followed by how the synthesizer finds two big issues within the design of the function in section 4.2.2. As we fail in section 4.2.2 to synthesize the function because of RAM limitations, we try to achieve the goal of synthesizing the key in section 4.2.2 and synthesizing the function given the key 4.2.2 in a 16-bit environment instead of 32 bits. As this is also not working as expected the try to interpolate the synthesizing time from 16 bits to 32 bits fails in section 4.2.2. To overcome these issues we adjust the setup in section 4.2.2 by moving to a Kubernetes cluster with 100GB of RAM. The main result on how we managed to synthesize the function is described in section 4.2.2. Finally, in section 4.3 we benchmark the different approaches to break the authentication algorithm with different knowledge. The meaning of all the observations done in this chapter is outlined in the next chapter 5.

### 4.2.1 Technical Setup

Here we explain the setup used to connect to the ECU via a CAN line, and that we choose CVC4 as the synthesizer to perform SyGuS.

After we have aligned on attacking UDS, we create a suitable test setup to implement an attack. Here, AVL provided an ECU in an exemplary setup, which is behaving in the same way as a real ECU would do. However, the UDS specification, as well as a flashing software written in Python was available, which seems to be a promising testing setup.

As seen in Figure 4.3, a power supply was bought, and cabling was created to connect the CAN pins of the ECU on a d-sub adapter with a Peakcan USB dongle to a notebook. This was serving as our reference setup to start the attack on UDS.

With this, the setup is close to a real-world scenario, where our D-Sub adapter would just end in the OBD port of a vehicle. In Figure 4.4 our laboratory reference setup is visualized in blue. This is how our reference setup would look like in a real vehicle.

On the laptop, we choose to utilize a virtual Kali-Linux image as the operating system. The operating system however has a minor impact, as most of the development to interact with the ECU is done in Python. After mounting the CAN line in the Kali-Linux, first CAN communication could be observed by just using the candump[1] tool which is part of the can-utils library. The outcome of candump can be seen in Figure 4.5. Having this

[1] https://manpages.ubuntu.com/manpages/focal/en/man1/candump.1.html

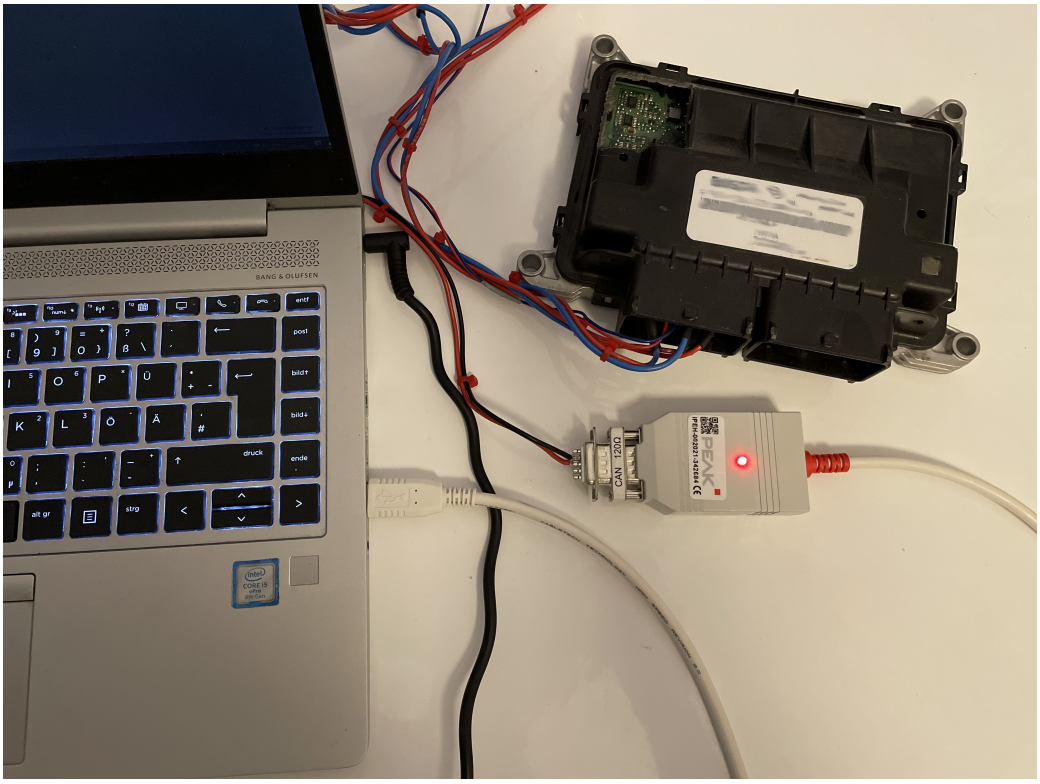


Figure 4.3: The reference setup with the ECU under attack, the self-made cabling to power it, and to bring the CAN line to the PEAK CAN USB adapter.

first milestone reached, we immediately start with the most important part: utilizing the UDS protocol by starting the flashing tool. Even though it is working nicely, we encounter the problem, that the flashing takes over 20 minutes for one successful run. To produce a good amount of data of UDS seed/key pairs on the CAN line, would take too much time. So we decided to implement a shortcut with the specification of the UDS protocol to just trigger the UDS protocol parts that are necessary (the request security access part of UDS). In this way, we can generate real seed/key pairs in less time.

Last but not least we choose CVC4[2] as implementation for the synthesizer for SyGuS because it is very promising and winning several awards. This makes it very promising in terms of reliability. As operational logic for SyGuS bit-vectors are chosen, which in this case is the obvious choice. Bit-vectors have been set to the size of 32 bits, which is already the real size of the integers used within the ECU. At this point, it was not clear if we would be able to synthesize the function with 32 bits, as this might be too big. However, we decided to try it first with 32 bits, as we can reduce the size to 16 bits or 8 bits at any point in time. 32 bits was to be preferred, as there will be no inaccuracy resulting from modifying the authentication function

[2]https://cvc4.github.io/

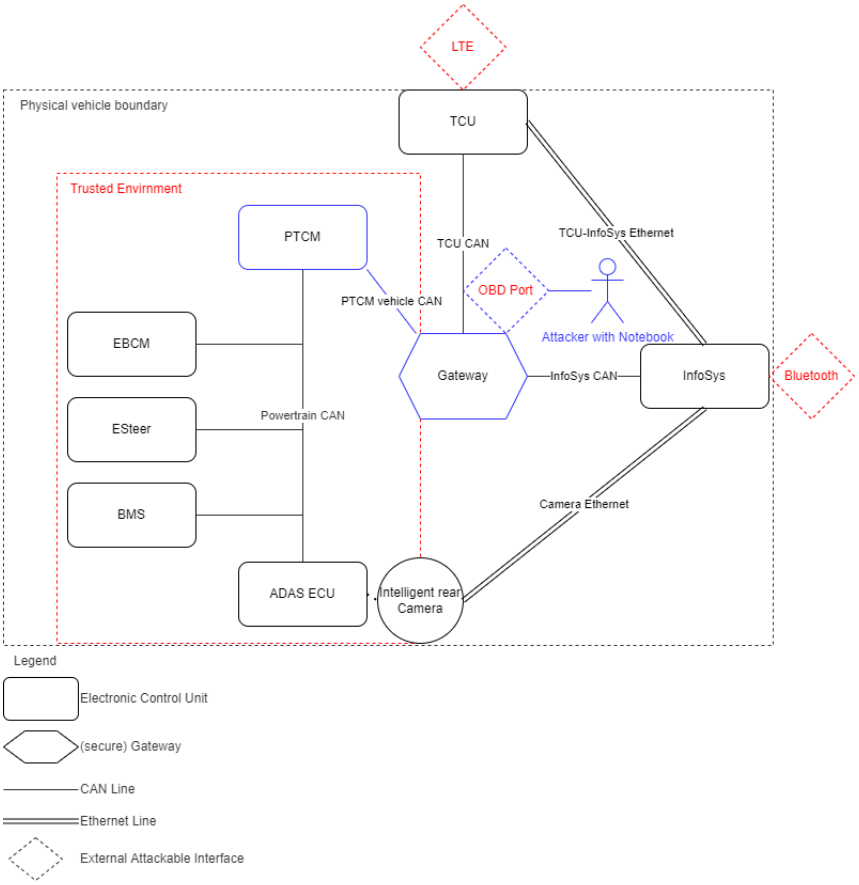


Figure 4.4: This image visualizes in the threat model our reference setup in blue and how it can be mapped into a vehicle.

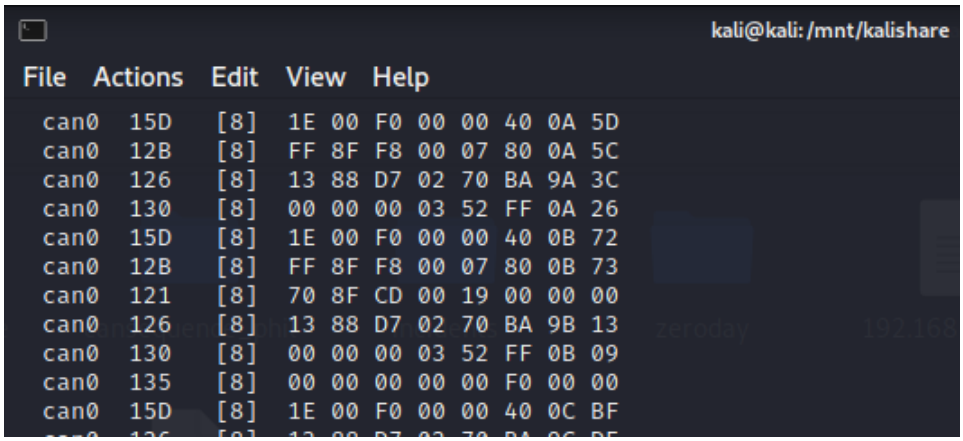


Figure 4.5: The first communication recorded by the ECU

With all the code and preparation work finished, we can finally move on to the actual phase of trying to exploit UDS using SyGuS.

### 4.2.2 SyGuS into an ECU

Here we described the journey on how to synthesize the authentication function of the ECU used within the request security access (0x27) function of UDS. In section 4.2.2 we see the first weakness in the function, which allows us to read out the secret key of the function. The solution needed to synthesize the function is referenced in section 4.2.2.

#### The target function

Through the whole work we have obfuscated the real key which was shipped with the exemplary ECU. A C-implementation of the function is shown in Figure 4.6. As the

```
uint32_t calculateKey32(uint32_t seed, uint32_t key)
{
    for (int i = 0; i < 32; i++) {
        if (seed & 0x80000000) {
            seed = seed << 1;
            seed = seed ^ key;
        }
        else {
            seed = seed << 1;
        }
    }
    return seed;
}
```

Figure 4.6: A C implementation of the function used inside the ECU/tester to calculate the seed/key value within the UDS authentication process.

trained eye can already see issues in this function, we did not have a close look at it and directly went for the SyGuS implementation.
In the real application, it is a simple function, taking the seed as input, calculating some permutations of the seed using a secret key and 32 iterations of a loop. Some additional countermeasures which are implemented in the real ECU are left out here to not leak the complete function.

### First try and learnings

For the first try, we took the ECU and values as is. The key was defined here as 0xFFFFFFFF. In addition to the random seed/key values, we also choose 1 and 0 as seeds, as well as their corresponding resulting key as input, with the target to give the synthesizer some good values to synthesize the function. In this case, input means, 0x0, 0x01, and a real seed/key pair are defined as constrained for the synthesizer.
Only three values have been chosen as constraints, to make it more easily for the synthesizer to find any solution. With this, we accept that we might not be able to synthesize the real function at the beginning. But, after having a first solution, which may be correct for only 3 inputs or 10% of all possible inputs, we aim to increase the amount of constraints to tune the synthesized function more, after having any solution.

Figure 4.7 shows this first implementation of a program to the synthesizer. The declaration of the non-terminals and the grammar that we allowed to the synthesizer were already aligned on how the real function looks like.

The synthesizer is executed on a notebook with 16 GB of RAM which unfortunately leads to an issue after 3 hours of execution. As seen in Figure 4.8, the synthesizer allocates a huge amount of memory which ultimately makes Windows start swapping memory from the RAM to the hard disk. Even thus the notebook uses an SSD as a hard disk, the notebook is only busy with swapping and not with calculation.
When analyzing this issue 3 possible solutions come to our minds: increase the memory, reduce the size of the bit vectors, calculate the difference (e.g. to 16-bit or 8-bit vectors), or to change the grammar of the synthesizer to reduce the search space and make the synthesizer more efficient. Again, to stick as close as possible to the correct algorithm option 3, rewriting the grammar is chosen to be the first try.

### Second approach: refining the grammar and unexpected result

As seen in Figure 4.9, for the second approach the grammar was rewritten. The if-then-else clause was removed, because the synthesizer is still able to build if-then-else clauses by utilizing XOR operations. To our surprise, this change of the grammar was of great impact: the synthesizer called with the program from Figure 4.9 was terminating after some seconds, bringing up a great result.

The output of the synthesizer looked as follows:

```
(define-fun f ((x (_ BitVec 32))) (_ BitVec 32)
  (bvxor (bvlshr x #b00000000000000000000000000000001)
    (bvshl #b11111111111111111111111111111111
      (bvshl #b10000000000000000000000000000000
        (bvand #b00000000000000000000000000000001 x))
    )
  )
)
```

```
;; The background theory is linear integer arithmetic
(set-logic BV)
;; (set-logic LIA)

;; Name and signature of the function to be synthesized
(synth-fun encrypt ((x (_ BitVec 32)) ) (_ BitVec 32)

    ;; Declare the non-terminals that would be used in the grammar
    ((I (_ BitVec 32)) (B Bool))

    ;; Define the grammar for allowed implementations of encrypt
    ((I (_ BitVec 32)(x
                #b00000000000000000000000000000000
                #b00000000000000000000000000000001
                (ite B I I)
                (bvand I I)
                (bvnot I)
                (bvxor I I)
                (bvshl I I)
                #xffffffff
                #x80000000
                ))
     (B Bool ((and B B) (or B B) (not B)
                 (= I I) )))
)

(declare-var x (_ BitVec 32))

(constraint (= (encrypt #x00000000) #x00000000 ))
(constraint (= (encrypt #x00000001) #xffffffff ))
(constraint (= (encrypt #x4ad88327) #xda93be6c ))


(check-synth)
```

Figure 4.7: This image shows the very first implementation try for the synthesizer to synthesize the target authentication function.

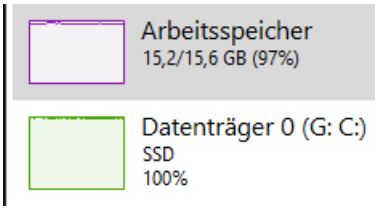


Figure 4.8: As the calculation continued the RAM of the notebook was full - resulting in the notebook putting more work into swapping between the disk and the RAM

```
1  ( set-logic BV )
2  ( synth-fun f (( x ( _ BitVec 32))) ( _ BitVec 32)
3  (( BV32 ( _ BitVec 32)) )
4  (( BV32 ( _ BitVec 32) (#x00000000 #x00000001 #xffffffff #x80000000
5      x
6      ( bvand BV32 BV32 )
7      ( bvor BV32 BV32 )
8      ( bvnot BV32 )
9      (bvxor BV32 BV32)
10     (bvshl BV32 BV32)
11     (bvlshr BV32 BV32)
12     ))
13 ))
14
15 (constraint (= (f #x00000000 ) #x00000000 ))
16 (constraint (= (f #x00000001) #xffffffff ))
17 (constraint (= (f #x4ad88327) #xda93be6c ))
18 (constraint (= (f #x4ad88326) #x256c4193 ))
19
20 ( check-synth )
```

Figure 4.9: The second draft of the input alphabet. Here the if-then-else clause was removed

Rewriting the synthesizer output into a C function is more readable:

```
uint32_t f(uint32_t seed){
   uint32_t tmp = seed & 1 ? 0xFFFFFFFF : 0;
   return ((seed >> 1)^tmp);
}
```

The synthesizer found a solution for our constraints and reduced a program having 32 iterations in a for loop down to a program having 3 lines of code. This is a real surprise, as we did not look as close to the function from the beginning. However, here we made two observations:

- The if-clause, especially the 0x80000000, makes the if very weak for certain inputs. If we put 0x01 as input to the function, it will return the key of the function used. Looking at Figure 4.6, even shows that we did overlook this at the very beginning. In line 16, the constraint is already given as

  ```
  (constraint (= (f #x00000001) #xffffffff ))
  ```

  which already shows the seed / key pair having this issue. However, this is only possible because in the else of the function, the seed value gets shifted to the left. So, if the seed is 1, then it gets shifted to the left for 31 times. Only in the last iteration of the execution of the function the if is evaluated to true. At this point,

the value of the seed is 0x80000000. The function will shift the value of seed again one time to the left, which results in the seed having the value: 0x0. The next instruction will XOR the seed with the key and return this value already. 0x0 XOR the key of course results in the key.

- Secondly, the used key itself is an issue, allowing the function to be cut down as we see in the result of the synthesizer. This is due to the fact, that the functions build something like a circle. If there is a leading 1 in the seed, it will shift the seed to the left and invert it (xor with 0xFFFFFFFF). If there is a leading 0 it will only shift the seed to the left. This is done 32 times. If it were done 33 times, we would receive the seed which was the initial input to the function. Comparing this now with the C representation of the synthesized result, we can see that it is just doing the reverse operation of the for loop, namely: it shifts the seed to the right (instead of left) and depending on the rightmost bit is a 0 it does nothing, or if it is a 1, then it is inverting the integer. This is exactly the opposite of what would have been done in the normal function in the last iteration.

These are already great two findings. The next step was to examine why the key 0xFFFFFFFF was used and we learned that this is just a debug key that was given out to us without really thinking about it. So we received another software with a real key for the next steps within the work. This key however will not be published, but only be referenced as "key" within this work.

#### Continue synthesizing the real key

Even with having good observations in 4.2.2 leading to the construction of a powerful full real-world attack in 5.1, this is not the initial goal, so we went back to how to synthesize the function with the other provided key.

We calculated the seed/key pairs to keep the synthesizing scrip comparable. Then we replaced the development key with the real one and the assumption with the newly generated values and started the synthesizing again.
Unfortunately, we ran into the RAM limitation again, just as observed previously in section 4.2.2. As a real key gives more entropy to this compared to 0xFFFFFFFF, we need to make further improvements to the synthesizer parameters.
This time we choose to reduce the size to 16-bit vectors and to try to model the function in two different ways: first one, where the function is known and only the key needs to be synthesized (section 4.2.2), second approach where the operations are known, and the synthesizer needs to construct the whole function and find the key (section 4.2.2).

#### 16 bit and synthesize the key

To achieve this task, first of all, the function used to calculate the seed/key pairs has to be rewritten recursively for the synthesizer to accept it. The result can be seen in Figure 4.11.

```
1  ( set-logic BV )
2  ( synth-fun f (( x ( _ BitVec 32))) ( _ BitVec 32)
3  (( BV32 ( _ BitVec 32)) )
4  (( BV32 ( _ BitVec 32) (#x00000000 #x00000001 #x[redacted] #x80000000
5      x
6      ( bvand BV32 BV32 )
7      ( bvor BV32 BV32 )
8      ( bvnot BV32 )
9      (bvxor BV32 BV32)
10     (bvshl BV32 BV32)
11     (bvlshr BV32 BV32)
12     ))
13 ))
14
15 (constraint (= ( f #x00000000 ) #x00000000 ))
16 (constraint (= ( f #x00000001 ) #x[redacted] ))
17 (constraint (= ( f #x4ad88327 ) #x[redacted] ))
18 (constraint (= ( f #x90729a8a ) #x[redacted] ))
19
20 ( check-synth )
21
```

Figure 4.10: Here we replaced the old development key 0xFFFFFFFF with a real key. The real key is here however not displayed directly. The keys calculated out of the seeds are also updated accordingly.

A function encrypt_inner is defined, which takes a 16-bit vector x and a 16-bit vector KEY as input. x is the seed while KEY is the real key to be synthesized. The function then models if-else from the original authentication function. Everything is given to the synthesizer as terminals, except for the KEY which needs to be synthesized. However, here the synthesizer knows, that he must choose the KEY and then use it for the XOR operation. Secondly, a function encrypt is defined. This function abstracts the for loop, by recursively calling encrypt_inner 16 times. With this, the initial authentication function is modeled with 2 functions inside the synthesizer world. Last but not least, a function f is defined, which the synthesizer needs to synthesize. In this function, the synthesizer is advised to choose a constant 16-bit value for the variable KEY and to use the encrypt function, which will result in 16 calls to encrypt_inner, which should simulate the real function with a 16 iteration for loop.
The constraints have been the same, however, the keys have been reduced accordingly.

After approximately 10 minutes the synthesizer finishes its job and gives us the following output:

```
(define-fun f ((x (_ BitVec 16))) (_ BitVec 16)
     (encrypt x #b01**************1))
```

He immediately finds the correct key and prints it in binary representation. Here we have again covered it, as it would be half of the real key used.

```
( set-logic BV )

(define-fun encrypt_inner (( x ( _ BitVec 16)) (KEY ( _ BitVec 16))) ( _ BitVec 16)
    (
        ite (= ( bvand x #x8000 ) #x8000 ) (
            bvxor (bvshl x #x0001) KEY
        )
        (
            bvshl x #x0001
        )
    )
)

(define-fun encrypt (( x ( _ BitVec 16)) (KEY ( _ BitVec 16))) ( _ BitVec 16)
    (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner
)

( synth-fun f (( x ( _ BitVec 16))) ( _ BitVec 16)
(( BV16 ( _ BitVec 16)) (KEY (_ BitVec 16)))
(( BV16 ( _ BitVec 16) (
    x
    (encrypt x KEY)
    ))
(KEY (_ BitVec 16) ((Constant (_ BitVec 16))))

    )
)

(constraint (= (f #x0000) #x0000 ))
(constraint (= (f #x0001) #x[redacted] ))
(constraint (= (f #x8327) #x[redacted] ))

(check-synth)
```

Figure 4.11: This Figure shows all adaptions done for the synthesizer to synthesize only the 16-bit key used within the function.

### 16 bit and synthesize function and key

Taking motivation from the successful try in the previous section, the implementation to synthesize the function and the key, while only knowing the results of the operation in Figure 4.12.

Writing the input for the synthesizer here is much more simple. Except for the 3 standard constraints used almost everywhere in this thesis, there is now only one function f defined which the synthesizer needs to synthesize. The operations he needs to achieve his goal are defined again, including the non-terminals needed. However, the iteration depth is not modeled.

The execution of this input only took 17 seconds for the synthesizer to come up with a solution: As we can see in Figure 4.13, the result is as short as the synthesizing time. This immediately brings up the question, of how accurate this synthesized function is. Figure 4.13 shows the result already in post-processed form, in which the bit values have been replaced with hex-encoded values. We can see that the key is not contained in this solution, and deduct that this is an over-fitted solution, only valid for the constraints given.

```
1  ( set-logic BV )
2  ( synth-fun f (( x ( _ BitVec 16))) ( _ BitVec 16)
3  (( BV16 ( _ BitVec 16)) (B Bool) (KEY (_ BitVec 16)))
4  (
5   ( BV16 ( _ BitVec 16) (
6      x
7      (bvxor BV16 KEY)
8      (bvshl BV16 #x0001)
9      (ite B BV16 BV16)
10     ))
11     (B Bool ( (= BV16 #x8000)))
12     (KEY (_ BitVec 16) ((Constant (_ BitVec 16))))
13     )
14 )
15
16
17 (constraint (= (f #x0000) #x0000 ))
18 (constraint (= (f #x0001) #x[redacted] ))
19 (constraint (= (f #x8327) #x[redacted] ))
20 ( check-synth )
21
```

Figure 4.12: This Figure shows all adaptions done for the synthesizer to synthesize not only the key but also the function.

```
1 (define-fun f ((x (_ BitVec 16))) (_ BitVec 16)
2     (ite (= (bvshl (bvxor x #x4000) #x0001) #x80000)
3         x (ite (= (bvxor x #x0327) #x80000)
4             (bvxor x #x4A0F) (bvxor x #x424C)
5           )
6      )
7 )
```

Figure 4.13: This Figure shows the 6 outputs of the synthesizer which have been generated after 17 seconds.

To validate this, a validation file in the format .smt2 is written to validate the output of the synthesizer. In this validation file, there is the synthesized function f inserted as a defined function and asserts are defined, against which the constraint solver of the synthesizer is checking whether the function is valid or not. Figure 4.14 shows this script. When calling the synthesizer, he returned UNSAT. This is happening because in line 10 there is a new seed/key pair (0x9A8A/0x7F62) added as assert.
If we remove line 10, then only the assets from the synthesizing phase are present and the constraint solver is returning SAT.

Another observation is, that we told the synthesizer to apply XOR to a 16-bit vector and the constant KEY, as visible in Figure 4.12 in line 7. However, as we can see in Figure 4.13, all the xor operations are executed between the input variable x and different constants. This indicates, that the input grammar is not defined properly in the way we have been expecting it, as the synthesizer is generating one KEY per constraint and not one KEY for all constraints.

```
2   ( set-logic BV )
3   (define-fun f ((x (_ BitVec 16))) (_ BitVec 16) (ite (= (bvshl (bvxor x #b0100000000000000)
4     #b0000000000000001) #b1000000000000000) x (ite (= (bvxor x #b0000001100100111)
5     #b1000000000000000) (bvxor x #b0100101000001111) (bvxor x #b0100001001001100))))
6
7   (assert (= (f #x0000) #x0000 ))
8   (assert (= (f #x0001) #x[redacted] ))
9   (assert (= (f #x8327) #x[redacted] ))
10  (assert (= (f #x9a8a) #x[redacted] ))
11
12
13
14  (check-sat)
```

Figure 4.14: This shows the output of the synthesizer which has been generated after 17 seconds.

### Interpolating the run time of finding the key

As the execution time of using SyGuS to synthesize the 16-bit key was quite ok in section 4.2.2, we wanted to continue playing around with the size of the bit vectors, to finally calculate how long 32-bit brute-forcing using the synthesizer would take.

Therefore we refactor the script from Figure 4.12 in such a way, that we reduce the bit vector size from 16 bits to 15 bits, and change the function and all its values from hex representation to bit representation, to more easily model the size of 15 bits.

When executing the synthesizer on the resulting input in Figure 4.15, however, the testing system (here the notebook) starts to slow down and it shows unstable behavior.
This makes it impossible to continue this approach to use interpolation for the calculation of the synthesizer time for higher bit values.

### Adding more resources

To overcome the RAM limitations and the strange behavior on the notebook caused by the huge RAM utilization, we decided to move into a Kubernetes cluster with a limit of 100GB RAM for the beginning.
As first try, we again take our initial try as seen in Figure 4.10, where we ask the synthesizer to synthesize a function, only knowing the operations allowed (AND, OR, XOR, NOT, SHL, SHR) and the constants used within the function (including the key). In this environment, the execution is working longer, but in the end, we run again in the RAM limitation and fail with 101GB RAM utilization. So we still have to think about other ways of optimizing the input for the synthesizer to make this final breakthrough.

### The breakthrough

At this point, we continued with creating the benchmarks (section 4.3). However, for the ease of reading this master thesis, we continue here with the findings that have been done later on. These findings have only been possible because we gained an additional

```
(set-logic BV)

(define-fun encrypt_inner ((x (_ BitVec 15)) (KEY (_ BitVec 15))) (_ BitVec 15)
    (
        ite (= (bvand x #b100000000000000) #b100000000000000) (
            ;if
            bvxor (bvshl x #b100000000000001) KEY
        )
        (
            ;else
            bvshl x #b000000000000001
        )
    )
)

(define-fun encrypt ((x (_ BitVec 15)) (KEY (_ BitVec 15))) (_ BitVec 15)
    (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner
)



(synth-fun f ((x (_ BitVec 15))) (_ BitVec 15)
((BV15 (_ BitVec 15)) (KEY (_ BitVec 15)))
((BV15 (_ BitVec 15)) (
    x
    (encrypt x KEY)
))
(KEY (_ BitVec 15)) ((Constant (_ BitVec 15))))

    )
)
;100001001001101

(constraint (= (f #b000000000000000) #b000000000000000 ))
(constraint (= (f #b000000000000001) #b[redacted] ))
(constraint (= (f #b100001100100111) #b[redacted] ))

(check-synth)
```

Figure 4.15: This shows the reduced 15-bit version of the initial 16-bit file.

understanding of the synthesizer and how it works while doing the benchmarks.

When thinking about other optimization possibilities, to help the synthesizer to utilize less RAM, the brute force attempts came to our mind: of course, the task for the synthesizer gets less hard, if we can reduce the complexity by removing the for-loop. The previous result which we successfully received in section 4.2.2 was possible very fast, as the development key 0xFFFFFFFF removed the 32 for loop.
With this in mind, new input is written for the synthesizer, as visible in Figure 4.17.

The function f, which needs to be synthesized is still the same. However, the constraints change and the function encrypt is introduced. The constraints do not directly reference the function f anymore, but instead, the function encrypt is used. Encrypt is then 32 times calling the function to be synthesized. With this, the scope of the synthesizer should be more clear and the for loop easier understandable, to reduce processing time and the amount of memory needed for calculations.


Figure 4.16: Reducing the space to 15 bits resulted in several blue screens at executing - as well as strange system behavior right before the blue screen.

```
1   ( set-logic BV )
2
3   ( synth-fun f (( x ( _ BitVec 32))) ( _ BitVec 32)
4   (( BV32 ( _ BitVec 32)) )
5   (( BV32 ( _ BitVec 32) (#x00000000 #x00000001 #x[redacted] #xffffffff #x80000000
6       x
7       ( bvand BV32 BV32 )
8       ( bvor BV32 BV32 )
9       ( bvnot BV32 )
10      (bvxor BV32 BV32)
11      (bvshl BV32 BV32)
12      (bvlshr BV32 BV32)
13      ))
14  ))
15
16  ( define-fun encrypt (( x ( _ BitVec 32))) ( _ BitVec 32)
17   ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f
18      ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f
19          ( f ( f ( f ( f x ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
20      ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
21  )
22
23  (constraint (= ( encrypt #x00000000 ) #x00000000 ))
24  (constraint (= ( encrypt #x00000001 ) #x[redacted] ))
25  (constraint (= ( encrypt #x4ad88327 ) #x[redacted] ))
26
27
28  ( check-synth )
29
```

Figure 4.17: This shows the working solution in which the 32 iterations for-loop have been excluded from synthesizing. The synthesizer only needs to synthesize the operations.

Surprisingly the synthesizer returns a solution after 37 minutes of calculation and with a memory utilization of only 12GB of RAM. The returned solution can be seen in Figure 4.18.

```
(define-fun f ((x (_ BitVec 32))) (_ BitVec 32)
    (
            bvxor KEY (
                    bvxor (bvshl x #x01)
                            (bvshl KEY (bvand #x80000000 x))
                    )
    )
)
```

Figure 4.18: This is the synthesis function returned from the synthesizer. The real value for the key is replaced with KEY here. The variable x is the input seed

The next step is to validate the synthesized function against the real function, which works with a constraint solver, but not with the result being used as a c-function. The function from the synthesizer output was directly transformed into a c-function f and called recursively 32 times.

```
uint32_t f(uint32_t x, uint32_t KEY)
{
   return (KEY ^ ((x << 0x01) ^ (KEY << (0x80000000 & x))))
}
```

The main reason for this is the undefined behavior of shifting a variable 0x80000000 times to the left[3]. In this case, instead of shifting KEY until it is 0, the compiler seems to take only the lowest byte of the value (which is sufficient to shift 256 times) and therefore always just shifts for 0, which makes of course a difference to the outcome.

Therefore, the function has to be carried over into a c representation more carefully. We decided to do this in a way that we can compare the synthesized function with the real function:

1. first replace the operations to C-style
   ```
   (define-fun f ((x (_ BitVec 32))) (_ BitVec 32)
        return KEY ^ ((x << 0x01) ^ (KEY << (0x80000000 & x)))
   )
   ```

2. rewrite the shift operation to create the if later on
   KEY << (0x80000000 & x) == 0x80000000 & x ? 0 : KEY
   this is equivalent, because if x is 0x80000000, then KEY will be set to 0, because it will be shifted 0x80000000 to the left. If x has any other value, then KEY will be shifted to 0 times and remain KEY.

[3]For further reading see Section 5.8 paragraph 1 of the ISO Working Draft, Standard for Programming Language C++: https://open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4594.pdf

```
(KEY ^ ((x << 0x01) ^ (0x80000000 & x ? 0 : KEY)))
```

REMARK: at this point, it is even more important to remove the big shift operation, because a real C implementation will run into undefined behavior. A C compiler will most likely just take the lowest byte, as with this he can shift for up to 256 times, which is larger than the provided data type. Therefore, in our example, it will just shift 0 times and not set the KEY to 0. It took quite some time to understand this and get the C–implementation running.

3. create the if–clause out of (0x80000000 & x ? 0 : KEY)

```
if(0x80000000 & x)
{
    KEY ^ ((x << 0x01) ^ 0)
}
else
{
    KEY ^ (seed << 0x01) ^ KEY
}
```

4. In the if, the xor zero operation can be ignored, as it will not change anything, while in the else, the xor KEY on the left and on the right is canceling itself out due to the associativity of xor, which leads to the following function:

```
if(0x80000000 & x)
{
    KEY ^ (x << 0x01)
}
else
{
    seed << 0x01
}
```

As we can see, the synthesizer comes up with the same function, with a still very small set of constraints. What is not visible here, is the rest of the input to the synthesizer: the 32-calls to the synthesized function f are not represented in this transformation. But adding them as a for loop would result in the complete same function.

This is a great result, as we achieve the goal of synthesizing the function, with only knowing some constants (including the key) but not the operations. With this, we can deduce interesting next questions, to further improve these synthesizer-based attacks.

## 4.3 Benchmarks

Here we describe our approach to benchmark classical brute-force attacks, carried out between a c-function, a constraint solver z3, and the synthesizer used. Interestingly, the z3 solver is the fastest one, if we abuse the weakness of the authentication function and provide a seed/key pair where the seed is 0x01. Besides that, brute force is faster and the synthesizer did most of the time not even find the correct key after several days.

Another goal of this work is to benchmark the synthesizer in finding the key against other techniques. For this purpose, we choose brute force and the Z3 constraint solver to compare against the synthesizer. Therefore the function is rewritten in C and then translated into the grammar of the synthesizer. This is necessary because the synthesizer in the used version does not handle for-loops.

With this rewritten function, a new grammar was created for the synthesizer to enable a brute force attempt of SyGuS to the algorithm. We compared the following scenarios:

- Bruteforce 0: Here the function referenced in Figure 4.6 is called, starting with a key set to 0 and increasing with each call until the expected key is returned.
- Bruteforce random: The same function is used as in Bruteforce 0, but it is called now with random 32-bit values.
- Z3 constraint solver: The input to the constraint solver for brute forcing the key is displayed in Figure 4.19. The defined function encrypt_inner defines the operations carried out with the handed-over values x (the seed) and KEY (the secret key of the function). It is recursively called by the function encrypt. In line 19 KEY is declared so that the synthesizer will search for the value for the variable KEY to resolve all the constraints provided.
- CVC4 SyGuS synthesizer: Figure 4.20 is the input to the synthesizer. Compared to the Z3 input it is defined differently from line 19 onwards, as SyGuS needs a different way to tell the synthesizer to only bruteforce the key.

Having a close look at the assertions/constraints it is obvious that they are not the same between the z3 input and the synthesizer input. This is because the z3 solver immediately returns the key if we give an assertion set that includes a seed of 0x01. This is happening because of the finding already observed in section 4.2.2, that the key is returned due to a function weakness if 0x01 is provided as input to the function. The z3 solver behaves smartly and seems to try the assertions as possible values for the key to calculate and therefore finished very fast. Due to that, we change the assertions with different values. To circumvent these issues we take 3 seeds as input: 0x4AD88327, 0x90729A8A, 0xD63A38DB, and their resulting keys as assertion. The same seeds and keys are also used with brute force. This is necessary because, with only one seed/key pair, the brute force attempt will terminate with a possible wrong key. Further, with this, the constraints to brute force are more comparable to the z3 solver constraints. The brute

```
1   ( set-logic BV )
2
3   (define-fun encrypt_inner (( x ( _ BitVec 32)) (KEY ( _ BitVec 32))) ( _ BitVec 32)
4       (
5           ite (= ( bvand x #x80000000 ) #x80000000 ) (
6               bvxor (bvshl x #x00000001) KEY
7           )
8           (
9               bvshl x #x00000001
10          )
11      )
12  )
13
14  (define-fun encrypt (( x ( _ BitVec 32)) (KEY ( _ BitVec 32))) ( _ BitVec 32)
15      (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner (encry
16  )
17
18
19  (declare-fun KEY () (_ BitVec 32))
20
21  (assert (= (encrypt #x4ad88327 KEY) #x[redacted] ))
22  (assert (= (encrypt #x90729a8a KEY) #x[redacted] ))
23  (assert (= (encrypt #xd63a38db KEY) #x[redacted] ))
24
25  (check-sat)
26  (get-model)
27
```

Figure 4.19: This is the input to the z3 constraint solver. The two defined functions are the same as in Figure 4.20. Line 19, the definition of the KEY to be determined is different due to the syntax of the constraint solver.

force and z3 solver tests are executed on a notebook, Sygus is executed on the notebook but brings only results for 16-bit secret sizes in a reasonable time. Due to the long execution time, the Sygus attempt is executed within a Kubernetes environment. The mean over 20 runs is calculated and represented in the table for brute force. Further, we use a total of 7 different keys. The results are outlined in table 4.1 with some interesting observations.

| ID | Key | Bruteforce 0 | Bruteforce random | Wrong keys found | Z3 | SyGuS |
|---|---|---|---|---|---|---|
| 1 | real key | 00:00:37 | 00:12:23 | 0,95 | 00:00:54 | 32:20:00* |
| 2 | 0x01010101 | 00:00:01 | 00:15:42 | 57,55 | 00:02:52 | |
| 3 | 0x10101010 | 00:00:18 | 00:13:46 | 2,85 | 00:45:11 | |
| 4 | 0xDeadbeef | 00:04:16 | 00:07:57 | 2,8 | 05:27:34 | |
| 5 | 0xFac239af | 00:04:47 | 00:15:18 | 23,05 | 04:57:36 | |
| 6 | 0xff99ff99 | 00:04:59 | 00:14:30 | 24,35 | 01:22:26 | |
| 7 | 0xFfffffff | 00:04:53 | 00:11:12 | 8,2 | 19:36:45 | |
| **Mean over all keys** | | *00:02:50* | *00:12:58* | *17,11* | *04:36:11* | |

Table 4.1: benchmarks of brute force attempts

1. If we include 0x01 as input (seed) to the function, which reflects a weakness of the algorithm (recall: it will return the key directly as secret) the Z3 constraint solver will immediately return the correct key. Therefore the seed 0x01 was removed for all test sets except for SyGuS.

```
( set-logic BV )

(define-fun encrypt_inner (( x ( _ BitVec 32)) (KEY ( _ BitVec 32))) ( _ BitVec 32)
    (
        ite (= ( bvand x #x80000000 ) #x80000000 ) (
            bvxor (bvshl x #x00000001) KEY
        )
        (
            bvshl x #x00000001
        )
    )
)

(define-fun encrypt (( x ( _ BitVec 32)) (KEY ( _ BitVec 32))) ( _ BitVec 32)
    (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner (encrypt_inner
)


( synth-fun f (( x ( _ BitVec 32))) ( _ BitVec 32)
(( BV32 ( _ BitVec 32)) (KEY (_ BitVec 32)))
(( BV32 ( _ BitVec 32) (
    (encrypt x KEY)
    ))
    (KEY (_ BitVec 32) ((Constant (_ BitVec 32))))

    )
)

(constraint (= (f #x00000000) #x[redacted] ))
(constraint (= (f #x00000001) #x[redacted] ))
(constraint (= (f #x4ad88327) #x[redacted] ))

(check-synth)
```

Figure 4.20: This is the input to the SyGuS synthesizer. The function to be synthesized f only contains a call to the encrypt function and a variable KEY where the synthesizer has to choose a constant value to solve all constraints.

2. SyGuS did only return one successful brute force attack on the real key, taking one and a half days to finish. The next attempt on key number 5 did not work, it was stopped after 3 weeks. The same holds for the previous successful execution, this could also not be reproduced anymore.

3. For some reason key 6 continuously took longer for brute force compared to key 7. The other side around was expected.

4. The random bruteforce approach shows interesting behavior for key 4, as it here takes only half the time it usually takes. Further, it also has only a failure rate of 2,8 when searching for the key.

5. Z3 solver showed some strange behavior as well for keys 4 and 5 as well as for keys 6 and 7.

6. Having a look at Key 1 and 2 we see that the Z3 solver is even outperforming the brute force approach.

As expected, brute force is still the fastest way to find a 32-bit value. However, the huge performance difference between the Z3 solver and SyGuS is interesting and would be worth checking.

Even though the comparison of brute force attempts ended in something that we previously already expected, it leads to the most important idea of finally enabling us to synthesize the function by only knowing the key! We abstracted the loops in the algorithm and defined recursively the 32 iterations to enable the synthesizer to more easily understand the function to synthesize. The idea is, by providing the recursion as an abstraction for the for-loop, to reduce the search space for Sygus and to reduce the large amount of memory that was needed until now. Again the remark, in the real timeline of the work, the benchmarks work was done after doing the work in chapter 4.2.2 and before chapter 4.2.2.

# Chapter 5

# Evaluation

This chapter evaluates the findings done in chapter 4. Based on these findings we can construct a real-world man-in-the-middle attack in Section 5.1, which would enable us to get root access to each vehicle if it would have gone into series production. Section 5.2 summarizes the outcome of successfully synthesizing the UDS SecurityAccess function and Section 5.3 discusses our outcome from the benchmarking, including that we failed here in utilizing SyGuS for brute forcing a 32 bit secret.

## 5.1 Constructing a real attack scenario

In section 4.2.2 we made the important observation, that it is possible to read out the secret key of the ECU if we pass in the seed of 0x01 within the UDS handshake. With this observation, we can build a real men-in-the-middle attack on a vehicle using this implementation of UDS. This attack will even follow Kerckhoff's assumption: "(...) an adversary (...) knows all details of the encryption function except the secret key.. " as written in [MOV01].

To apply the men-in-the-middle attack, we need to prepare a small device that we can put on the OBD port, which will break the OBD connection of the car workshop for us. The idea is when the car workshop starts the communication with the ECU and requests the seed, that we intercept the seed which gets returned by the ECU and replaces it with 0x01. Then, the car workshop PC will pick up this value and calculate the secret. As already explained in section 4.2.2, the function and car workshop PC will then return the real secret of the function on the CAN bus.
At this point, the UDS connection will fail and the technician will just try to establish a connection again. If so, all we need to do is to let the communication go through. Or, we just use the received key to calculate the correct key as expected by the ECU and continue to route the traffic through our man-in-the-middle attack. By this, we can just log in and learn more information. Ideally, we can even sniff a firmware update and reverse engineer it later on. After this, we can at any point in time use the recorded secret key and establish any UDS connection with the ECU we want.

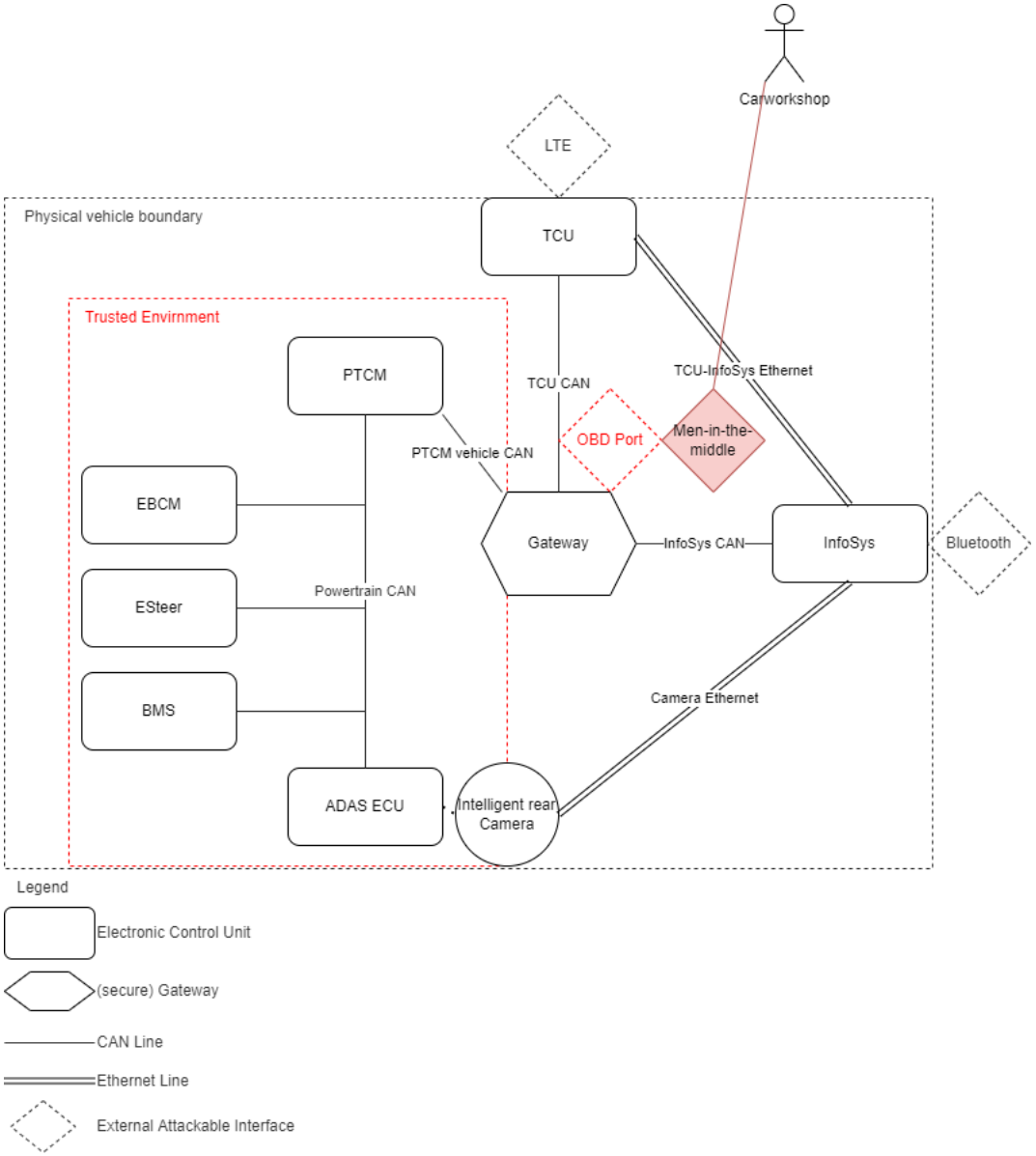


Figure 5.1: To apply the men-in-the-middle attack a modification is done at the OBD port. Here, the UDS requests will be intercepted, and a modified payload will be sent to the computer of the car workshop technician.

In summary, with this algorithm in use, one can build a small adapter, rent a car, go to the car workshop, and only need one time to intercept the connection of the vehicle to be able to take over full control over the vehicle.

## 5.2 Synthesizer lessons learned

Besides from reaching the target of the thesis, to synthesize the key of the authentication function and to synthesize the function, if we only have limited knowledge, successfully, we can show a much more interesting use case of utilizing SyGuS. SyGuS showed us, that there exists a very weak key for this function (see 4.2.2), which results in a very simple function with only 2 lines of code, completely throwing away the 32 iterations for-loop. This was only possible, because the bad key 0xFFFFFFFF at this point, did not make the function more random, but built something like a circle operation on the integer out of the function. The two-line function only did the reverse operation and went one step to the left in the circle, where the input seed is treated at the starting point of the circle. Further, this finding made it more obvious, that a seed with the value 0x01 will return the secret key of the function used. This evaluates the fact, that SyGuS can be used to verify a key within its used environment. Even though this was only a development key, this could also happen to other values in other functions. And SyGuS can be used to check if there exists a simpler version of a function, doing the same and making the function under test useless.
Further, when we synthesized the function with limited knowledge, it not only synthesized a function that gives us 30% accuracy, it synthesized the real function, where we could even exchange the key and still get correct results!

We managed to achieve the goals for this function, but more complex functions are not yet tested. More complex functions could cause more troubles for the synthesizer to solve the result and might result in more solutions, where only the constraints will be valid, but any other input to the synthesized function might fail, as encountered in section 4.2.2. Or we would finally come down to a situation where we are only able to synthesize a function that is of 30% accuracy.

## 5.3 Benchmark interpretation

It is still very interesting, that SyGuS manages it very easily to synthesize the function, but not to brute force the key. We have not been able to identify, what makes the synthesizer fail in finding a solution, even with turning the verbosity on a high level at execution.
Z3 however showed that it has a very smart approach to finding a key. Because the Z3 constraint solver only took less than a second to find the key, if we keep the assertion with the seed == 0x01 in the constraint set. This makes the constraint solver also an interesting tool for the validation of a key in development. Still, we observed unclear behavior in the execution time of Z3 for different keys. As seen in table 4.1, Z3 finds key 5 (0xFAC239AF) 30 minutes faster than key 4 (0xDEADBEEF). Even more strange is the huge difference between key 6 (0xFF99FF99), key 7 (0xFFFFFFFF), as well as key 4 and key 5. Finding key 6 is hours faster than key 4 or 5. And last but not least finding key 7 is exponentially slower than all the others. However, there is also something positive to be mentioned, as we can see, that the Z3 solver is faster compared to the

random brute force approach for keys 1 and 2.
For the random brute force approach, there is an interesting observation possible: key 4 (0xDEADBEEF) is brute force in half the time that the others take. This might be related to the used C++ rand_s function and an indicator that there is an accumulation for this value in the generation of pseudo-random numbers. However, rand_s was chosen as according to Microsoft this should be a secure function[1].
In summary, this makes the classical brute force attack still much faster, however, it seems to be worth to also use Z3, as it is very fast if the key is either very low or if there is a flaw in the algorithm used.

[1] https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/rand-s?view=msvc-170

# Chapter 6

# Conclusion

Our main goal was to show how weak some implementations of UDS are in vehicles, because of the way how they are implemented. Further, we wanted to to utilize SyGuS for this purpose, by showing that it is capable of calculating the function with very little knowledge (only knowing the key). As we achieved this successfully, we also had some interesting findings due to the usage of SyGuS which brought some design flaws in the function to our attention. With this design flaw, we could even construct a real-world men-in-the-middle attack in section 5.1 to vehicles, which would enable us very easily to get root access to any ECU. This could ultimately result in an impact on the physical safety of a driver, if this access was used to manipulate the functionality of the vehicle, causing an accident harming the driver.
The next steps could be, to check how to utilize SyGuS in development, as suggested in 5.2, to improve the overall quality of functions and reduce weaknesses introduced in critical functions which are not uncovered by the engineers. Also, as synthesizing the function worked out in section 4.2.2, a next target can be to try to remove the constraint of providing the key as input to the alphabet of the synthesizer. As Charlie Miller and Chris Valasek did not only reverse engineer the Jeep as discussed in the related work section, they also inspected other vehicles and they mentioned in [VM15b], that they see reoccurring patterns, also in the flashing routines of ECUs. With these patterns, we could aim to create a more general input to the synthesizer, which would enable it to challenge more UDS implementations. Last but not least, together with Section 3, we outlined further interesting attacks that vehicles face currently in the real world. Further, we can show there, that other people, such as described in [Yur15], are similarly using the Z3 solver as we used SyGuS to break the UDS authentication.

# Acronyms

| | | |
|---|---|---|
| ACC | Adaptive Cruise Control | 10, 11, 12, 13, 28 |
| ADAS | Advanced Driver Assistant Systems | 12, 14 |
| AES | Advanced Encryption Standard | 18, 19 |
| AVL | Anstalt für Verbrennungskraftmaschinen List | 2 |
| BCM | Body Control Module | 9, 13 |
| CAN | Controller Area Network | viii, xi, 1, 4, 7, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 27, 28, 31, 32, 33, 40 |
| CRC | Cyclic Redundancy Code | 14, 15 |
| DBC | Database CAN | 17, 18 |
| ECM | Engine Control Module | 9, 13, 17 |

| | | |
|---|---|---|
| ECU | Electronic Control Unit | iv, viii, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 18, 19, 20, 21, 22, 23, 28, 29, 31, 32, 36, 40, 43, 60, 64 |
| ESP | Electronic Stability Program | 8, 9, 17 |
| INSYS | Infotainment System | 9, 10, 11, 13, 14 |
| LIN | Local Interconnect Network | 12, 13 |
| MAC | Message Authentication Code | 18, 19 |
| OBD | On-Board Diagnostics | viii, 19, 20, 36 |
| OEM | Automotive Original Equipment Manufacturers | iv, 1, 2, 8, 9, 10, 11, 17, 18, 19, 22, 23, 29, 34, 38 |
| PA | Park Assist | 13 |
| PDU | Protocol Data Unit | 18 |
| RKE | Remote Keyless Entry | 9, 12, 13 |
| SecOC | Secure Onboard Communication | viii, 18, 19 |
| SID | Service Identifier | 20 |
| SMT | Satisfiability Modulo Theories | 25 |
| SOTA | Software Update Over the Air | 11 |

| | | |
|---|---|---|
| SyGuS | Syntax Guided Synthesis | iv, v, viii, ix, 1, 3, 4, 5, 6, 7, 8, 25, 26, 36, 40, 41, 43, 51, 56, 57, 58, 59, 60, 62, 64 |
| TCU | Telematics Unit | 10, 11, 12, 13, 33, 34 |
| TPMS | Tire Pressure Monitoring System | 9, 12, 13 |
| UDS | Unified Diagnostic Service | iv, v, viii, 1, 3, 4, 6, 7, 19, 20, 21, 31, 32, 36, 41, 43, 60, 64 |
| XCP | X-Calibration Protocol | 19 |

# Bibliography


[ABJ+13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-Guided Synthesis*. 2013.

[AUT21] AUTOSAR. *SECOC*. 2021. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf (visited on 07/05/2021).

[CDKT15] Mohamed Chawki, Ashraf Darwish, Mohammad Ayoub Khan, and Sapna Tyagi. "Cybercrime: Introduction, Motivation and Methods". In: *Cybercrime, Digital Forensics and Jurisdiction*. Cham: Springer International Publishing, 2015, pp. 3–23. ISBN: 978-3-319-15150-2. DOI: 10.1007/978-3-319-15150-2_1. URL: https://doi.org/10.1007/978-3-319-15150-2_1.

[cE03] European commission and the European Parliament. *nformation and Communications Technologies for Safe and IntelligentVehicles*. Communication document. 2003. URL: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX%3A52003DC0542.

[Com15] ISO/TC 22/SC 31 Comittee. *ISO 11898-1:2015 Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling*. IACR Cryptology ePrint Archive, Report 2015/097. 2015. URL: https://www.iso.org/obp/ui/#iso:std:iso:11898:-1:ed-2:v1:en.

[com15] ISO/TC 22/SC 31 Data communication. *ISO 14229-1 Road vehicles — Unified diagnostic services (UDS)*. 2015.

[Cur23] Sam Curry. *Web Hackers vs. The Auto Industry: Critical Vulnerabilities in Ferrari, BMW, Rolls Royce, Porsche, and More*. 2023. URL: https://samcurry.net/web-hackers-vs-the-auto-industry/ (visited on 12/17/2023).

[Edw20] Joseph White Edward Taylor Norihiko Shirouzu. *How Tesla defined a new era for the global auto industry*. 2020. URL: https://www.reuters.com/article/us-autos-tesla-newera-insight-idUSKCN24N0GB (visited on 02/12/2023).

[FDC11] Aurélien Francillon, Boris Danev, and Srdjan Capkun. "Relay Attacks on Passive Keyless Entry and Start Systems in Modern Cars". en. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 18th Annual Network & Distributed System Security Symposium (NDSS Symposium 2011); Conference Location: San Diego, CA, USA; Conference Date:

February 6-9, 2011. Zürich: Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2011. DOI: 10.3929/ethz-a-006708714.

[Hol15] Martin Holland. *ConnectedDrive: Der BMW-Hack im Detail*. 2015. URL: https://www.heise.de/news/ConnectedDrive-Der-BMW-Hack-im-Detail-2540786.html (visited on 12/17/2023).

[Mik18] Mike Metzger - Flexible Creations. *Letting the Air Out of Tire Pressure Monitoring Systems*. 2018. URL: https://www.defcon.org/images/defcon-18/dc-18-presentations/Metzger/DEFCON-18-Metzger-Letting-Air-Out.pdf (visited on 04/05/2021).

[MOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001. URL: http://www.cacr.math.uwaterloo.ca/hac/.

[Muk19] Abhishek Udupa Mukund Raghothaman Andrew Reynolds. *The SyGuS Language Standard Version 2.0*. 2019.

[Por21] Porsche Austria GmbH & Co OG. *We Connect Service*. 2021. URL: https://www.volkswagen.at/konnektivitaet-mobilitaetsdienste/konnektivitaet/we-connect (visited on 03/14/2021).

[RM15] Marc Rogers and Kevin Mahaffey. *DEF CON 23 - Marc Rogers and Kevin Mahaffey - How to Hack a Tesla Model S*. 2015. URL: https://www.youtube.com/watch?v=KX_0c9R4Fng (visited on 12/17/2023).

[Rob21] Robert Bosch GmbH. *CAN*. 2021. URL: https://www.bosch.com/de/stories/das-controller-area-network/ (visited on 05/24/2021).

[SSS+21] Christoph Schmittner, Abdelkader Shaaban, Svatopluk Stolfa, Jakub Stolfa, Jan Plucar, Marek Spanyik, Alen Salamun, Richard Messnarz, Damjan Ekert, Georg Macher, and Alexander Much. "Automotive Cybersecurity - Training the Future". In: *Systems, Software and Services Process Improvement*. Ed. by Murat Yilmaz, Paul Clarke, Richard Messnarz, and Michael Reiner. Cham: Springer International Publishing, 2021, pp. 211–219. ISBN: 978-3-030-85521-5. DOI: 10.1007/978-3-030-85521-5_14.

[VM14] Chris Valasek and Charlie Miller. *A Survey of Remote Automotive Attack Surfaces*. Technical Whitepaper, IOActive. 2014. URL: https://ioactive.com/wp-content/uploads/2018/05/IOActive_Remote_Attack_Surfaces.pdf.

[VM15a] Chris Valasek and Charlie Miller. *Adventures in Automotive Networks and Control Units*. Technical Whitepaper. 2015. URL: https://illmatics.com/car_hacking.pdf.

[VM15b] Chris Valasek and Charlie Miller. *Remote Exploitation of an Unaltered Passenger Vehicle*. Technical Whitepaper. 2015. URL: https://illmatics.com/Remote%20Car%20Hacking.pdf.

[Wen05] Victor Wen. *Security on Tire Pressure Monitor System*. Midterm Report, EE241, Computer Science Division, U. C. Berkeley. 2005. URL: http://bwrcs.eecs.berkeley.edu/Classes/icdesign/ee241_s05/Projects/Midterm/VictorWen.pdf.

[Yur15] Dennis Yurichev. *Keygenning using the Z3 SMT Solver*. Technical Whitepaper. 2015. URL: https://yurichev.com/mirrors/SAT_SMT_crypto/Keygenning%20using%20the%20Z3%20SMT%20Solver.pdf.