Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Csongor Ferenczi

DESIGN AND DEVELOPMENT OF A CAN SECURITY TESTING DEVICE

UNDER SUPERVISION OF

András Gazdag

Dr. Levente Buttyán

BUDAPEST, 2020

Table of contents

HALLGATÓI NYILATKOZAT

Alulírott **Csongor Ferenczi**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2020. 12. 10.

...…….

Csongor Ferenczi

Összefoglaló

Napjaink járművei tele vannak elektromos vezérlőegységekkel (ECU-kkal), melyek a különböző szenzorok adatainak feldolgozásáért, valamint az egyes alkatrészek vezérléséért felelősek. Az összehangolt működés megvalósítása érdekében ezen ECU-knak kommunikálniuk kell egymással, melyre az egyik legelterjedtebb megoldást a broadcast alapú Control Area Network (CAN) hálózat adja. A CAN busz létrehozása során nagy hangsúlyt fektettek a hálózat megbízhatóságára, azonban annak biztonságosságával már kevésbé foglalkoztak. Az autókat izolált rendszereknek tekintették, melyhez nem férhet hozzá egy potenciális támadó, napjainkra azonban ez a helyzet megváltozott. Az autók számos interfésszel rendelkeznek a külvilág felé, mint például a Bluetooth, Wi-Fi, vagy a fedélzeti diagnosztikai port (OBD), melyek mind potenciális támadási felületek lehetnek egy támadó számára.

A CAN busz egyik legnagyobb hiányossága, hogy nem támogat üzenet hitelesítést, így az ECU-k csupán az üzenetekben található ID mező alapján döntik el, hogy figyelembe veszik-e az üzenet tartalmát, vagy sem, annak eredetéről nem tudnak megbizonyosodni. Ennek következtében könnyedén lehet üzeneteket beinjektálni a buszra, vagy akár módosítani meglévő üzeneteket, és ezzel hibás működésre késztetni egy ECU-t. Az üzenet injektálásos módszert relatíve egyszerű megvalósítani, azonban detektálása sem kifejezetten kihívás sajátosságai miatt. Ezzel szemben, az üzenet módosításos módszer jóval nehezebben kivitelezhető, valamint detektálható, mivel az üzeneteket küldő ECU-k az üzenet küldése során visszahallgatják a buszt, hogy meggyőződjenek róla, hogy az általuk küldött üzenet kerül kézbesítésre. Ezen védelmet csak a küldő és fogadó ECU közötti CAN busz két részre szakításával lehet áthidalni, melyhez egy speciális eszközre van szükség, mely képes Man-in-the-Middle támadás végrehajtására.

Jelen dokumentumban bemutatjuk az általunk épített proof-of-concept hardware eszközünket, mely valós időben képes CAN üzenetek módosítására. Az eszköz két CAN interfésszel rendelkezik, melyek segítségével továbbítani, és akár módosítani is tudja az áthaladó üzeneteket, azok ID-ja alapján. A dokumentumban mérési eredmények bemutatásával is alátámasztjuk, hogy eszközünk mindezt a CAN szabványban leírt késleltetési korlátok betartása mellett képes véghez vinni.

Abstract

Modern vehicles are full of Electronic Control Units (ECU) responsible for different functionalities involving processing information from sensors and controlling actuators. To perform their functions, ECUs need to communicate with each other. Most vehicles use a Controller Area Network (CAN) for this purpose. The CAN bus is a broadcast channel where messages with a simple format can be transmitted. The original design of the CAN bus was focusing on safety and reliability properties. Security was not an issue because these networks were considered to be isolated systems. However, modern vehicles have many interfaces towards the outside world (e.g., Bluetooth, Wi-Fi, wireless TPMS, or the On-board Diagnostics (OBD) port), which could render the internal network accessible to an attacker.

The CAN bus does not support message authentication. The ECUs decide to act upon a message or not, based only on the CAN ID field of the message. As of this, it is possible to inject fake messages, or potentially, to modify messages on the CAN, and hence, forcing some ECUs to act upon these fake messages, which may influence the overall behaviour of the vehicle. In an injection attack, extra messages are added to the regular traffic. The original messages and the injected messages could appear identical, and the increased frequency of messages may influence the behaviour of the ECUs that react on these messages. It is relatively easy both to execute and to detect an injection attack. On the other hand, modification attacks are more complex both to carry out and to detect. The main difficulty is that the sender checks whether the transmitted bits correctly appear on the bus or not due to safety concerns. The only way to circumvent this protection is to physically separate the sender, and the attacked ECU on the CAN bus. This can be achieved with a physical layer Man-in-the-Middle attack, using a hardware component installed in front of the attacked ECU.

We built a proof-of-concept hardware device capable of modifying the CAN traffic in real-time to show that this attack is possible. It has two CAN interfaces and a microcontroller to read messages from the original CAN bus and either just forward or modify-and-forward traffic to the attacked CAN bus, based on the ID of the message. We showed with measurements that we can perform a message modification attack while keeping the introduced delay within what is allowed by the CAN specification.

1 Introduction

Modern cars are not just simple mechanical devices with an engine and four wheels like they used to be. They are full of Electronic Control Units (ECU) that control the car, monitors the drivers' every movement, and tries to keep them safe on the road, while providing convenience features to them. Thus, more and more components have to communicate with each other in a car, which means we need reliable communication channels between these components. One of the most common communication solutions is the Controller Area Network (CAN) bus.

When it comes to manufacturers, they put a great emphasis on the safety of the cars. There are ABS, ESP, crash avoidance systems, or even better and better crumple zones in every car. The inter-component communication channels are protected against errors caused by the high noise environment, but the cyber-physical security of their products itself is not that important for them. The internal components are not designed to be secure from a potential attacker, while it is not unthinkable that a malicious device could be installed into a car and do some harm. For instance, it only takes one mechanic to plug such a device into an On-Board Diagnostics (OBD) port of a targeted car, and we have access to the vehicle's CAN bus. Moreover, with the spread of Bluetooth OBD debugging probes, which can connect to the owner's smartphone, users themselves connect more and more devices onto the CAN bus.

Unfortunately, the CAN bus has no security, only safety measures. It was designed to be a robust communication bus that can withstand a high amount of noise while providing relatively high transfer speeds. At the end of every CAN message, a Cyclic Redundancy Check (CRC) code ensures that if the content of the message changes during transmission, the receiver will notice it. On the other hand, the CAN standard does not provide support for message authentication. Thus, just by receiving a message with a given ID does not guarantee that the source of the message was any particular device. This leads to the issue that messages can be injected onto the bus without the communication partners ever noticing that a message was not even sent by the correct device and could contain false data.

Besides injecting messages, an attacker could modify the content of the message, but achieving this is much harder. With the current techniques, it is only feasible by

compromising an ECU, and sending messages with modified values. This attack produces no extra messages, nor highly deviating values; thus, it is much harder to detect, but until now, it required a much higher knowledge[1] to perform this attack than just plugging in a device to the OBD port.

In this thesis, we present our solution to modify CAN messages in real-time, with a minuscule introduced delay without compromising any ECU.

We have created a device, which is capable of modifying ISO 11898 high speed CAN messages in real-time. It can handle up to 100% bus load with a bus speed of 500kbps or less, and about 60-100% busload at 1Mbps, which is the maximum speed specified in the standard. It introduces a delay of ~260us, which is even at 1Mbps is well within the delay of resending a message due to high traffic or a transient error on the bus. The device itself provides a wireless interface that can be used to remotely configure the attack parameters. All this, while consisting of cheap, commonly available parts.

In chapter 2, we are going to take a deeper look at the current attacks against the CAN bus and their mitigations, and how our solution differs from them. In chapter 3, we summarize the important parts of the CAN standard. In chapter 4, we present our first approach to the issue by showing the creation of a malicious CAN controller, which despite not being effective in our real-life tests, provides a potential attack interface to the CAN bus. After that, in chapter 5, we discuss the design principles of the introduced device, the hardware, and software architecture; and take a look at the measurements of the device. Finally, in chapter 6, we conclude our work.

[1] http://illmatics.com/remote%20attack%20surfaces.pdf (last accessed: 2020.11.17.)

2 Related work

In this section, first, we are going to discuss the current attack methods against the CAN bus and take a look at a few theoretical mitigation attempts. After that, we are going to briefly mention the differences between the current attacks and our solution.

2.1 Current attack methods against the CAN bus

There is a large amount of published research on the (in)security of the CAN bus. Many of them propose a solution to protect the CAN bus [1][2], but there are several papers discussing different kinds of attacks against it [3][4]. However, we can generally categorize the attacks into two groups: message injection, and message modification. Let's take a look at the first one.

2.1.1 Message injection attacks

Message injection attacks [5] are based on multiple shortcomings of the CAN standard. First of all, since the CAN bus is a broadcast channel, during operation, every ECU connected to the CAN bus receives all of the messages. This cost-saving measure saves a lot of money for the manufacturers, since they only have to wire a few twisted pairs throughout the whole car in order to connect the ECUs to each other. When receiving a message, the ECUs decide whether they are interested in the given message based on its ID and process it or discard it. On the other hand, this very principle makes it easy for an attacker to eavesdrop on messages on the bus, monitor the values of the sensors in real-time, and reverse engineer the messages and their purposes in a given vehicle type.

The second issue with the standard is that the CAN bus does not support any kind of cryptographic message authentication measures at all. Any ECU can create a message with an arbitrary ID and send it to the other ECUs via the CAN bus, and the ECUs will not be able to differentiate the messages coming from two different ECUs, if they have the same ID. This makes an attacker able to craft arbitrary messages and send them to the ECUs via the CAN bus. An attacker can have multiple goals to exploit these vulnerabilities, for instance, overwriting values or creating a Denial of Service state.

Overwriting values

In many cases, there might be safety measures in ECUs that if the ECU receives rapidly deviating values, only the most frequent value gets processed. Thus, if the attacker sends the crafted messages with a high enough periodicity, the crafted message's values are overwriting the original values in the ECU. This concept can be seen in Figure 2.1.

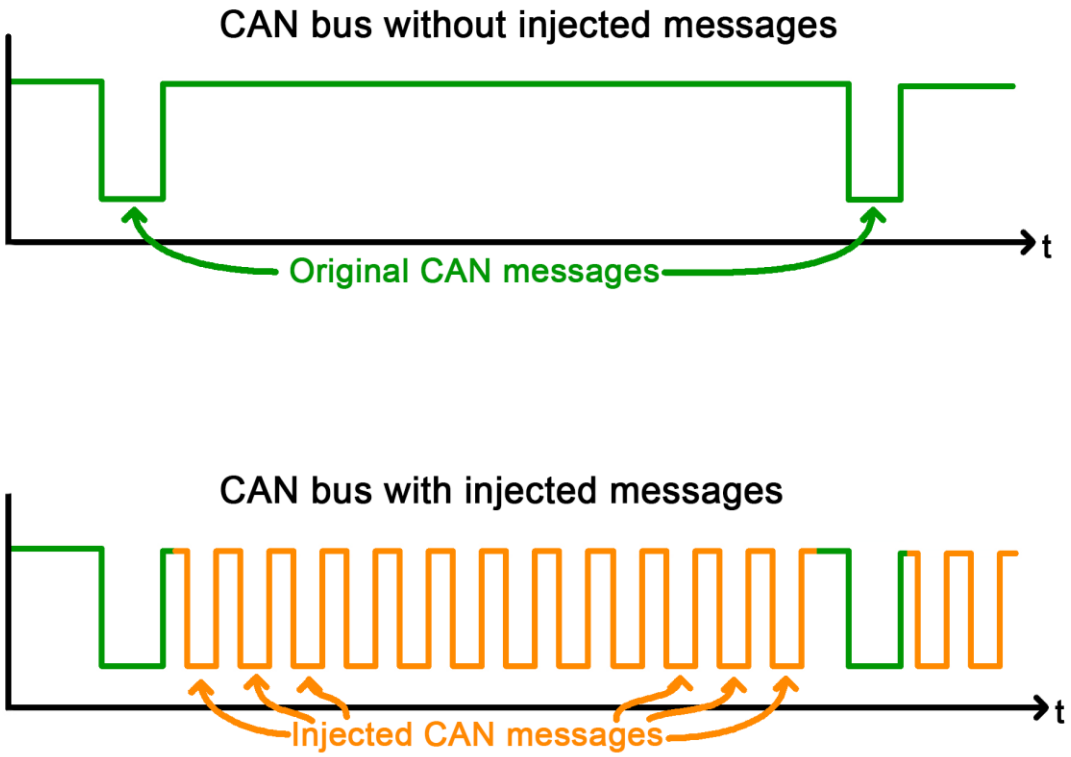


Figure 2.1 - Visualization of the message injection attack on the CAN bus

However, this attack has several drawbacks. First of all, both the rapidly changing value behaviour and the at least doubled message periodicity is easily detectable [6]. Upon detection, the targeted ECU might turn into a fallback mode, where it ignores both the original and the injected values. Secondly, the ECU might have safety margins built-in. For instance, most, if not all of the lane assist systems have a maximum steering angle it is allowed to perform in order to keep the car in the lane. If the attacker tries to induce a higher steering angle than this maximum value, the lane assist system might just deactivate.

Denial of Service (DoS)

A DoS attack is pretty straight forward. The attacker's goal is to render a given CAN bus unusable, which can be achieved in two ways: by adhering to the CAN standard and by

breaking it. The first option is to send as many messages to the CAN bus with the lowest possible ID as physically possible. As we describe in section 3.2, when the bus is idle, if two or more ECUs want to transmit at the same time, the one with the lowest ID will have priority. Thus, since the zero ID has priority over every other message ID, none of the regular messages will win the arbitration against the injected message, which will lead to the starvation of the regular ECUs. The second method is to force the CAN_H and CAN_L wires into dominant state and hold it there as long as the attacker wants to. While the second method could be easier to implement, it triggers the error detection in the ECUs; thus, the connected subsystem will detect that there is an error with the CAN bus. On the other hand, the first solution does adhere to the rules of the CAN, and the ECUs might only think that everything is okay with the CAN bus, except it is busy at the moment. Nonetheless, using both solutions, an attacker can render a given CAN bus unusable, and prevent the connected ECUs to send messages to each other.

2.1.2 Message modification attacks

Message modification attacks are based on the idea that instead of injecting messages, we could achieve somehow to modify the message that is being sent. However, as we present in section 3.4, there are several safety measures in the CAN standard that makes it harder if not impossible to modify messages transmitted by another ECU on-the-fly (e.g., CRC, bit stuffing, sender ECU monitoring the bus during transmission, etc.). Thus, until now, the easiest way to realize this attack was to compromise an ECU [7] and modify the messages before they are sent.

Furthermore, whether we are talking about message injection attacks or message modification attacks, there are several issues that affect them. First of all, not everything is controlled via the CAN bus. For instance, usually, a lot of the sensors directly connect to the drive train ECU, which makes it inaccessible for the attacker without compromising the given ECU. Secondly, there are messages that contain informative data only and have no effect on the control of the car. Attacking these messages could mess with the dashboard, for instance, but would have no real effect on the car. And finally, there could be message-chains consisting of multiple messages, transmitted after each other. While this issue is easily solvable from the attacker’s perspective, it can make reverse engineering and the implementation of the attack significantly complicated, real quick.

Now that we have seen the most common attacks against the CAN bus and their drawbacks, let us take a look at a few mitigation attempts.

2.2 Mitigation attempts

There are several proposed solutions to the issues mentioned in the previous section. However, while the following measures could increase the security of the CAN bus, they are more of a theoretical than an applied solution.

2.2.1 CANAuth

As mentioned before, one of the biggest deficiencies of the CAN bus is that it lacks message authentication. CANAuth [8] solves this issue by using a symmetric key based HMAC.

The main idea behind the CANAuth is the following. As described in section 3.3.1, every bit transmitted on the CAN bus get sampled at the 75% point of the bit time to ensure a reliable sampling. However, technology has developed a lot since the introduction of the CAN bus, and microelectronics are much faster nowadays. Thus we could use a higher sampling rate in order to hide authentication data in the propagation segment of every CAN bit, as you can see in Figure 2.2 labeled with *CAN+ transmission window*.

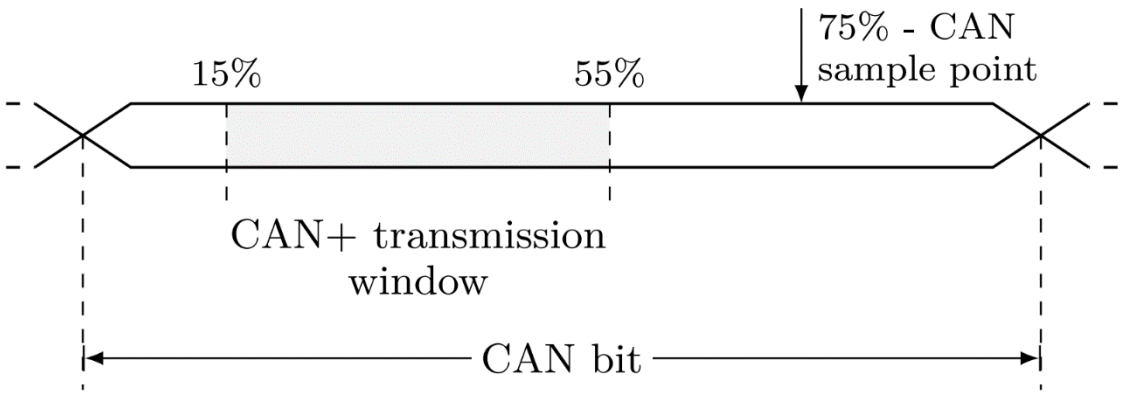


Figure 2.2 - The transmission windows of the CANAuth messages in a single bit of a regular CAN message

Before sending a message, the HMAC of the message could be calculated using a shared key and transmitted in the CAN+ transmission window. After the reception, the receiver ECU could create its own HMAC imprint with the same, shared key, and verify whether the received and the calculated HMACs are identical, thus eliminating message injection

attacks. Since older CAN transceivers and controllers simply ignore the propagation segment of every transmitted bit, CANAuth is backward compatible with older devices.

However, there are two issues with this solution. First of all, using a symmetric key requires all of the participating ECUs to possess the key, but handling and storing these keys are not an easy task. If every car has a different key, replacing OEM parts with spare or aftermarket ones will require the key to be updated. On the other hand, if a manufacturer sets the same key for a whole series of cars, if at least one person shares the key online (by reverse-engineering the key storage, for example), the whole series is compromised, and changing keys in this volume is not feasible. Secondly, in order to work properly, every important ECU should use CANAuth, which would require a lot of manufacturers to upgrade their devices.

2.2.2 CAN firewall

Another mitigation method is to use firewalls [9]. By using a firewall, we can physically split a CAN bus into multiple separate segments and interfere with the traffic going between them. For instance, we can apply whitelist or blacklist-based message filtering on each of the different segments, introduce rate limiting, etc. Thus, we can limit the possibility of message injection and DoS attacks. Adding a CAN firewall to an existing car should not require redesigning the car, since the firewall itself is just a simple device with two CAN interfaces, which can be inserted between the separated CAN segments.

While this solution could appear as the ultimate solution to the shortcomings of the CAN standard, it has several issues. For instance, we have to separate every important ECU or at least every important segment with a firewall, in order to be effective, which creates an excess cost for the manufacturer. An even bigger issue is the management of the firewall. Who gets to write the rules? How are the rules updated, if an update is required? Who is responsible if an important packet gets dropped unintentionally? What if an airbag does not open during an accident due to a malformed firewall rule? While these edge cases could seem unimportant, can be the differences between a manufacturer implementing a technology, or not using it.

2.2.3 Secure CAN transceivers

The third and final mitigation technique we are going to introduce is the secure CAN transceivers [10] proposed by NXP[2]. Their idea is to introduce a new security defense layer at the CAN transceiver level. Using this new layer, they can prevent message spoofing in both the transmitting, and the receiving side; detect malicious ECUs, and evade DoS attacks.

The transmit side message spoofing prevention is achieved using the following methodology: During production, the manufacturer can set the list of valid IDs (passlist) in the transmitter that its ECU allowed to transmit. While in operation, the CAN transceiver only transmits messages whose ID is in the preconfigured passlist. In case its ECU tries to transmit a message with a different ID (e.g., due to being compromised), the transceiver discards the message and blocks any future message from the ECU. On the receiving side, the message spoofing prevention is achieved using active invalidation. When an ECU starts to transmit a message, every secure CAN transceiver listens to it, and in case they detect that the mentioned ECU transmits a message from their passlist, they send an active error frame, which breaks the message, and prevents any other ECU from receiving and processing it. If the ECU tries to resubmit the crafted message, it will get broken again, until the sender ECU's transmit error counter hits its upper limit and switches itself into suspend-transmission behaviour.

Malicious ECU detection is based on a similar idea to what we present in section 4.1. According to the CAN standard, during transmission, if an ECU wins the arbitration, it continues to listen to the bus during the whole process, in order to detect any occurring during transmission. With a secure CAN transceiver, in case a malicious ECU breaks a benevolent ECU's message during transmission, and then sends a crafted message with the same id the benevolent ECU was sending, the benevolent ECU recognizes that the malicious ECU is trying to spoof messages, and signals the ongoing attack. This detection also works even when the benevolent ECU is in an error-passive state (see section 0); thus, an attacker will not be able to force a benevolent ECU into bus-off state.

---

[2] https://www.nxp.com/products/interfaces/can-transceivers/secure-can-transceivers:SECURE-CAN (last accessed: 2020.12.02.)

Finally, the DoS prevention is achieved using a leaky bucket mechanism. Every ECU has a “bucket”, which fills when the ECU sends a message, and empties continuously with time. In case the bucket overflows, the ECU stops transmitting messages. This method limits the number of allowed messages under a given time period. The number of allowed messages can be fine-tuned by changing the bucket size, in order to allow burst-like messages.

Based on the mitigation techniques introduced in the previous sections, we can say that while these mitigations are great theoretical results, and could more or less enhance the security of the CAN bus, we have to emphasize that they are still not widely used in the industry and thus do not provide protection against current attacks. NXP’s Secure CAN transceivers could be the next easy to implement security measure, but our solution introduced in this paper could still circumvent it after a little bit of modification.

2.3 How our solution differs from the current attack methods

Unlike the previously introduced attacks, our solution does not leave the original message on the bus, thus evading the jumping message value issue, and does not modify the periodicity of the messages, like a message injection attack. Also, it does not require the attacker to compromise an ECU or turn it into debug mode. We have created a method that can modify messages on-the-fly in such a way that it is not detectable with current detection methods.

3 A short summary of the ISO 11898 High-speed CAN standard

In this section, we summarize the important parts of the ISO 11898 High-speed CAN standard [11][12].

The CAN bus is a broadcast, serial communication protocol used mainly in vehicles. It was designed to be robust, withstand high external RF noise, while providing a high-speed communication link between the ECUs. The CAN bus is a cost-effective solution, because it enables the manufacturers to connect the ECUs by placing only one twisted pair cable between them.

While the standard itself is a broad document with many revisions, in this chapter, we are only going to discuss the main physical properties, the different layers and frame types, timing considerations, and error handling.

3.1 Physical properties

The CAN bus uses two wires called CAN high (CAN_H) and CAN low (CAN_L), in order to implement differential signalling. The cables are twisted pair cables. On each end of the cables, the wires are connected to each other using a terminating resistor in order to achieve a nominal 120 Ω impedance. The CAN bus has two states, driven, and not driven. When it is not driven, the CAN_H and CAN_L wires get pulled to about the same 2.5V nominal voltage using the passive pull resistors placed in the CAN transceivers. This state is also called a “recessive” bit, which represents a binary 1. When the CAN bus is driven, at least one CAN node pulls the CAN_H wire to 3.5V and CAN_L wire to 1.5V nominal voltage. This state is also called a “dominant” bit, which represents a binary 0. You can see a visualization of the CAN states and the represented bit values in Figure 3.1.

It is important to note that these nominal voltages can differ significantly during operation due to external noise affecting the wires of the CAN bus. However, by using differential signalling, this noise gets cancelled out during the reception.

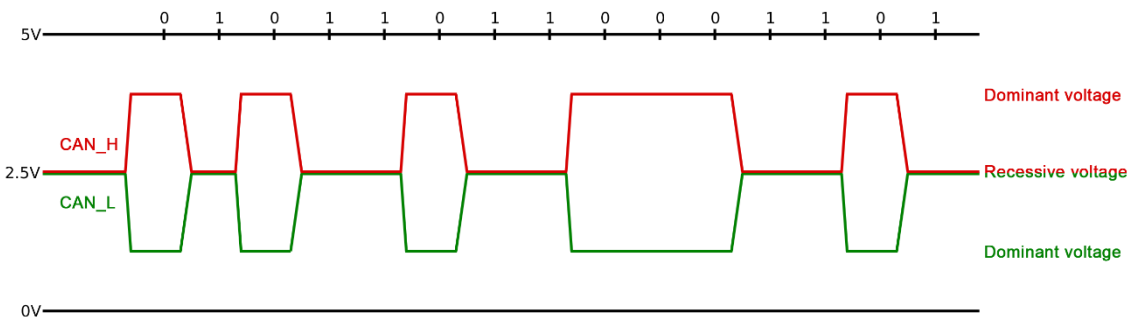


Figure 3.1 – Visualization of the CAN bus states and the represented bit values

3.2 Layers and frame types

As you can see in Figure 3.2, the CAN architecture is divided into two layers, the Data Link Layer (DLL), and the Physical Layer (PL). The DLL can be divided into two sub-layers, the Logical Link Control (LLC) and the Medium Access Control (MAC) layer. The PL can be divided into three sub-layers, the Physical Signalling (PLS), the Physical Medium Attachment (PMA), and the Medium Dependent Interface sub-layers.

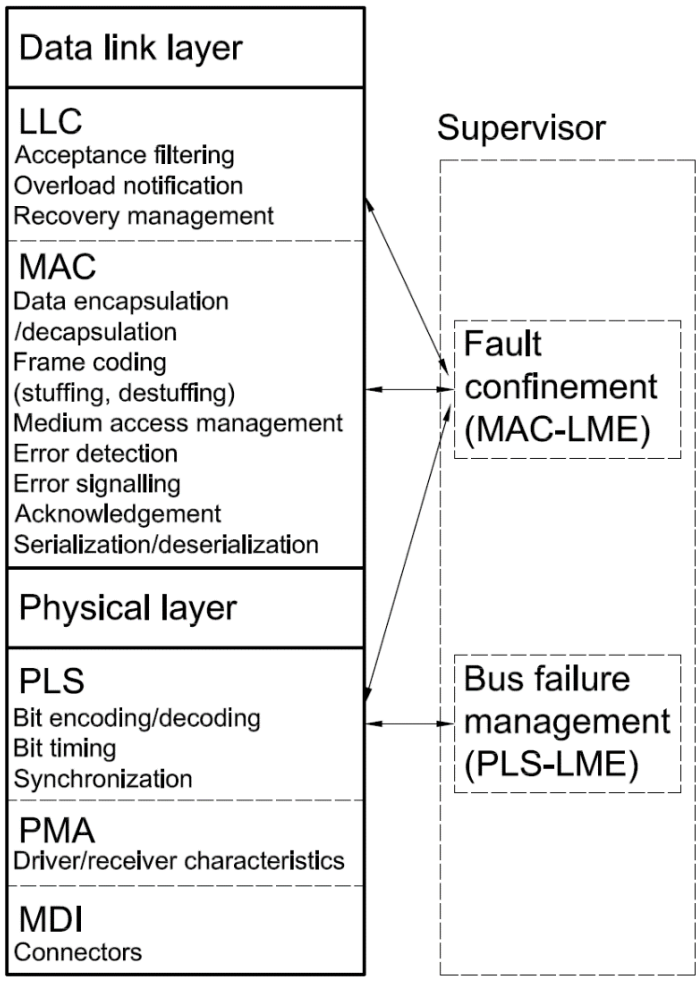


Figure 3.2 – The detailed layer structure of the CAN

However, in order to keep this chapter short, we are going to categorize these layers a bit differently. In the following 3.2.1 and 3.2.2 sections, we are going to introduce the frame structures of both the LLC and the MAC layer. After that, in section 3.3, we are going to go into details about the bit and frame timing; and finally, in section 3.4, we are going to talk about the error handling exclusively, since these details are one of the most important things in the standard from our point of view.

3.2.1 LLC layer

In general, every LLC layer frame consists of three bit-fields: the identifier (ID), the Data Length Code (DLC), and the LLC data field, as seen in Figure 3.3.

Figure 3.3 – The structure of the LLC frames

- The **ID** is a unique identifier that also represents the priority of the message, as we are going to discuss in section 3.3.2. The ID consists of three parts: An **11-bit base ID**, a **1-bit extension flag**, and an **18-bit ID extension segment**. In case the extension flag is 0, the ID extension is ignored.
- The **DLC field** consists of **4 bits**, which represents the number of data bytes in the frame. Since we can encode numbers from 0 to 15 on four bits, any value over 8 is considered as 8.
- The **LLC data** field can contain **up to 8 bytes of data** as it is set by the DLC. It is important to note that there are two kinds of frames in the LLC layer: data and remote frames. While data frames are being used to send data from one ECU to another, remote frames can be used to request data from a given ECU. However, the remote frames do not contain a data field.

3.2.2 MAC layer

Just like in case of the LLC frames, MAC frames also have **data** and **remote frames**, which is the first two frame type we are going to introduce. Both frames are wrapped and slightly modified LLC messages consisting of seven segments: Start of Frame (SOF), arbitration field, control field, data field, CRC field, ACK field, and finally, the End of Frame (EOF) field, as you can see in Figure 3.4.

Figure 3.4 – The structure of the MAC data and remote frames

- The SOF is a single dominant bit. It is used by the ECUs to synchronize themselves to the sender ECU. It only shall be sent when the bus is idle.
- The arbitration field has two formats based on whether the ID extension flag is set (extended format) or not (base format). The field contains the following bits:
  - **Base format** arbitration field**:**
    - **11-bit base ID**, passed from the LLC layer
    - **1-bit Remote Transmission Request (RTR) bit**, which differentiates the data and the remote frames. In case of a data frame, the RTR bit is dominant; otherwise, it is recessive.
  - **Extended format** arbitration field**:**
    - **11-bit base ID**, passed from the LLC layer
    - **1-bit Substitute Remote Request (SRR)** that substitutes the base format RTR bit. It is always recessive
    - **1-bit Identifier Extension Flag (IDE)**, which distinguishes the base and the extended format. Shall be recessive
    - **18-bit ID extension**, passed from the LLC layer
    - **1-bit RTR**
- The **control field** also has two formats based on whether the ID extension flag is set or not:
  - **Base format** control field:
    - **1-bit IDE**
    - **1-bit reserved**, shall be dominant until specified, but the receiver shall accept recessive too
    - **4-bit DLC**, passed from the LLC layer
  - **Extended format** control field:

- **2-bit reserved**, shall be dominant until specified, but the receiver shall accept recessive too
- **4-bit DLC**, passed from the LLC layer

- The **data field** contains **0-8 bytes of data** passed from the LLC layer.
- The **CRC field** contains a **15-bit CRC sequence** followed by a **1-bit CRC delimiter**, which is always recessive.
- The **ACK field** contains a **1-bit ACK slot**, which is being transmitted as a recessive bit, and gets pulled to dominant if any of the ECUs receives the frame successfully. This bit is followed by a **1-bit ACK delimiter**, which is always recessive.
- At the end of the frame, the **EOF field** consists of 7 recessive bits.

The visualization of an example CAN data frame can be seen in Figure 3.5.

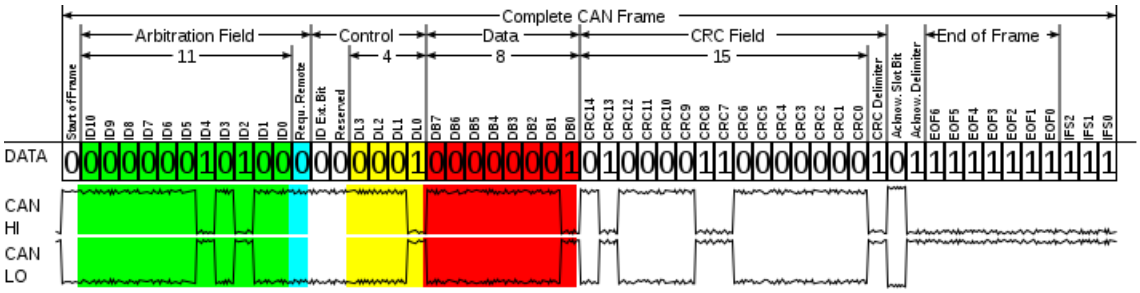


Figure 3.5 - Visualization of an example CAN data frame

There are two other frame types in the MAC layer, called the error and the overload frame. The **error frame** consists of a **6 to 12-bit long error flag** containing all dominant or all recessive bits based on the error state of the ECU, followed by an **8-bit long error delimiter,** as shown in Figure 3.6.

Figure 3.6 - The structure of the MAC error frames

The **overload frame** is similar to the error frame. It contains a **6-bit overload flag**, which is all dominant, and an **8-bit overload delimiter**, which is all recessive. An overload frame is sent in case an LLC layer component requests it due to overload, or in case a MAC layer component detects the timeout of an LLC layer component. The structure of the MAC overload frames can be seen in Figure 3.7.

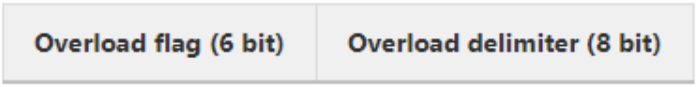


Figure 3.7 - The structure of the MAC overload frames

3.3 Message transmission

In this section, we go through the different layers of message transmission, starting with how a bit can be transmitted.

3.3.1 Bit Timing

Every CAN bus has a nominal bitrate (*BR*), which gets preconfigured in the ECUs by the car manufacturer. The maximum bitrate specified by the standard is 1Mbps, but 500kbps and 250kbps is also frequently used bitrates. We can calculate the bit time (t_B) using Equation 3.1.

$$t_B = \frac{1}{BR}$$

Equation 3.1 – Calculating the nominal bit time from the nominal bitrate

Every bit time can be divided into the following four segments:

- **synchronization segment** (Sync_Seg): This segment is used for synchronization. At the SOF, every receiving ECU synchronizes itself to the trailing edge of the first bit, which is called a “hard sync”. There is also another kind of synchronization called bit resynchronization, which is performed at the synchronization segment of each bit, by the ECU fine-tuning its inner clock based on the deviance between the expected time of a potential edge, and the actual time of its detection.
- **propagation time segment** (Prop_Seg): This segment is used to compensate for physical delay times within the network (e.g., signal propagation time, internal delay of the ECUs, etc.)
- **phase buffer segment 1** (Phase_Seg1) and **Phase buffer segment 2** (Phase_Seg2): These segments are used for edge phase error compensation. The sampling of the bus occurs after the phase buffer segment 1. The length of these segments can be fine-tuned by resynchronization, and thus, the sampling point can be moved backward or forward. A visualization of the four segments can be seen in Figure 3.8.

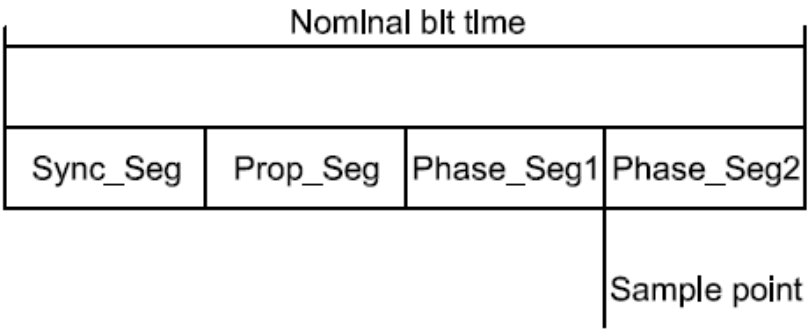


Figure 3.8 - The four segments of the bit time

3.3.2 Frame timing

Messages can be transmitted based on events (e.g., receiving a remote frame, a sensor value triggering a transmission) or by a trigger coming from an internal timer. The internal timer method is called Time-Triggered Communication (TTC). During communication, after every frame's EOF segment, there is a 3-bit long intermission period. After these 3 recessive bits, the bus is considered idle, and any following dominant is considered as the SOF of the next frame. When the ECUs detects that the bus is idle, any of them may start to transmit. If two or more ECUs happen to transmit at nearly the same time, the conflict between them is resolved using contention-based arbitration.

Contention-based arbitration

When a node starts to transmit a frame, it monitors the bus during the arbitration segment of the MAC frame in order to check whether the data on the bus is the same as it is transmitting. In case it detects that despite transmitting a recessive bit, the bus is still in dominant state, it knows that another ECU tries to transmit data, and it terminates the arbitration process and turns into a receiver. This allows the other ECU to transmit its message as nothing happened, and the other ECU that lost the arbitration, can retry sending its message at a later time. Using this method requires the bus has to adhere to three key elements:

- The ID of the message types has to be unique.
- A data frame with a given ID and a non-zero DLC value may only be sent by one ECU
- The remote frame's DLC value has to be the same as the data frame it requests.

Using this contention-based arbitration ensures that the ECU with the higher priority frame will always win the arbitration, because its ID contains more dominant bits, than the other potential transmitter ECU frame's ID.

3.4 Error handling

In this section, we are going to discuss the most important measures of the error handling in the CAN standard, starting with error detection.

Error detection

The standard defines numerous error detection techniques in order to increase the safety and reliability of the CAN bus.

- **Monitoring the medium**: During transmission, the sender ECU monitors the bus and compares the bit value available on the bus with the one it is currently transmitting. If the sender ECU sees that the two bit differs from each other, it signals an error frame starting from the next bit. This error is also called a bit error.
- **CRC**: The CAN standard defines a 15-bit CRC algorithm, which can be used to ensure that a message has not been changed during transmission due to noise, or a transient error. The CRC is calculated on the destuffed bitstream of the following segments: SOF, arbitration field, control field, data field (if present).
- **Bit stuffing**: The SOF, arbitration field, control filed, data field, and CRC are coded with a 5-bit bit stuffing. This means that if a transmitter detects five consecutive bits with the identical value in the transmission bitstream, it inserts an extra, complementary bit into the stream. The CRC delimiter, the ACK, and the EOF fields are not stuffed.
- **Frame check**: During operation, if a frame is transmitted, the receiver ECUs check whether a fixed-form bit field contains one or more illegal bits. For instance, the SRR bit should always be recessive in an extended format MAC data frame; thus, a dominant SRR bit would trigger a frame check error.
- **Acknowledge check**: All ECU acknowledges every consistent frame they receive. If a frame is not acknowledged by any other ECU, the sender knows that the frame has been corrupted during transmission, and shall be retransmitted.

Error counting

Within every ECU, there are two counters responsible for tracking the recent errors and switching error states, the Transmission Error Counter (TEC) and the Receive Error Counter (REC). If an error occurs during transmission or reception, the corresponding counter gets increased by 1 or 8 (based on multiple conditions, which is out of scope for this paper to discuss). On the other hand, with every successful transmission, the TEC gets lowered by one, and with successful receptions, the REC gets smaller by one. Based on these two counters, the error state of the ECU can be the following.

- If both the TEC and the REC is below 128, the ECU is in an **error-active state**. In this mode, the ECU participates in the bus communication per usual. If the ECU senses an error, it sends an active error frame to the CAN bus.
- If either the TEC or the REC gets higher than 128, the ECU switches into the **error-passive state**. In this mode, the ECU also participates in the bus communication, but if it senses an error, it sends a passive error frame to the CAN bus, which has no effect on the communication of the other nodes.
- If the TEC gets over 256, the ECU switches into the **bus-off state**. In this mode, the ECU does not participate in the bus communication either via sending or receiving messages. Unlike the error-passive state, where the ECU can switch back into the error-active state automatically, recovering from this state can only be initiated by the user. Once the user initiated the recovery, the ECU waits for 128 occurrences of 11 consecutive recessive bits before changing to error-active state, as it can be seen in Figure 3.9.

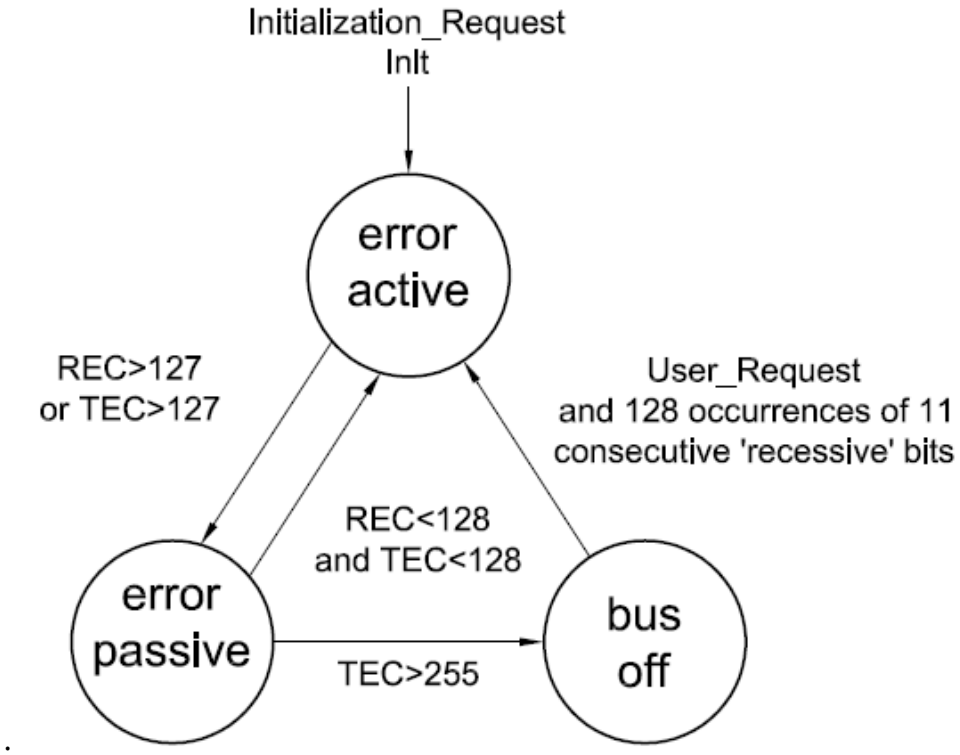


Figure 3.9 – Visualization of the transitions between the error states

Error signalling

Whenever a frame gets flagged either by the sender or a receiver ECU, due to one of the errors presented above, the sender ECU aborts the transmission and sends an error frame based on its error state. In order to give time for a potential transient error to pass, there is a recovery time between the detection and the retransmission of the frame, which is 17-31-bit time. After the recovery time passes, the ECU re-queues the message for automatic retransmission, just like in case of a lost arbitration. However, not every message is automatically retransmitted. According to the CAN standard:

„in case of TTC, the automatic retransmission shall be disabled" [9]

While this sentence might not seem to be extraordinary, in the next section, we are going to introduce our first solution to modify CAN messages in real-time, based on this one line.

4 The first approach – A malicious CAN node

Initially, we had a few different ideas on how to achieve our goals, like we could try to force a recessive state on the bus, despite one of the ECUs pulling it to dominant state, but since the CAN bus was created to be a robust, noise-tolerant bus, it has numerous safety measures built-in, as we discussed in section 3.4, which makes this impossible. However, we found a potential flaw in the CAN standard, which lead to the following ideas, introduced in the next section.

4.1 Designing the attack

As discussed in section 3.4, all messages have to be retransmitted in case it loses arbitration or if an error occurs during transmission, except the TTC messages, which shall not be resent. However, we can exploit the fact that the TTC messages are not automatically retransmitted, with the following method.

When a TTC message arrives, we can break it on purpose using a malicious CAN node. When the sender ECU detects that the message has been broken, it sends an error flag, but based on the standard, after signalling the error flag, it will not resend the original TTC message. Thus, after breaking the original message, we could wait for the end of the error frame and inject our own message with the values we would like to send. Since the original message gets interrupted, none of the other ECUs are going to detect the original message, only the crafted one that we send. The visualization of the attack can be seen in Figure 4.1.

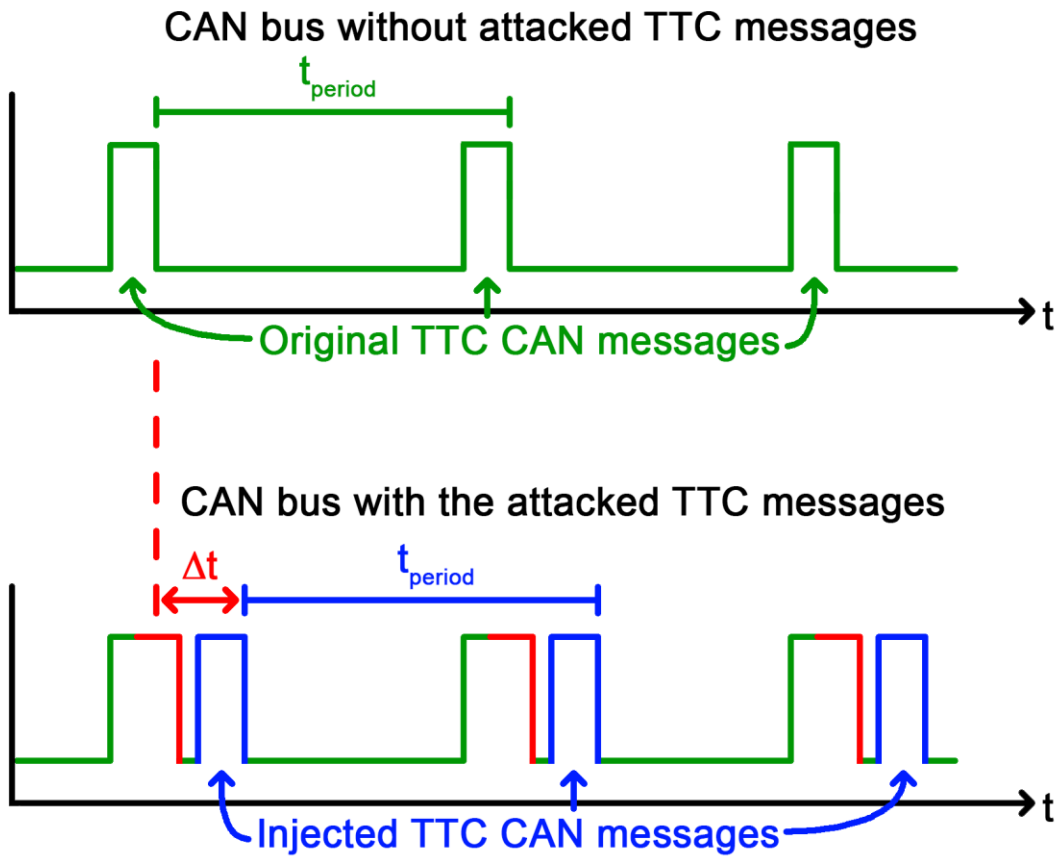


Figure 4.1 - The visualization of the malicious CAN node attack

As you can see in the figure, with this solution, when sending the first message, the arrival time of the TTC messages gets delayed by Δt. However, this only happens once, and after that, every message gets sent with the original period time. If we manage to get the Δt low enough, the ECUs would see it as a skew in the clock of the sender ECU, and thus, we could hijack any TTC message ID.

After designing the attack, now let us discuss, how we implemented the attack.

4.2 Implementation

In order to execute the attack, we need a fairly low-level access to the CAN bus. Since we are going to do something that's against the CAN standard, we have to implement our own communication device, which requires us to gain access to the CAN bus.

4.2.1 Gaining access to the CAN bus

One of the easiest ways to implement this attack is to use a Raspberry Pi 3 with a PiCAN 2 shield, shown in Figure 4.2. The Raspberry Pi running a Raspbian OS provides us a

versatile[3], Linux based environment, and it has plenty of power for signal processing. The PiCAN 2 shield is useful for two reasons. First, we can easily set up a CAN network for testing; and secondly, we can check its schematic to see how it works, because it is an open hardware device. With this, we can get a brief idea of what components we are going to need in order to implement this attack.

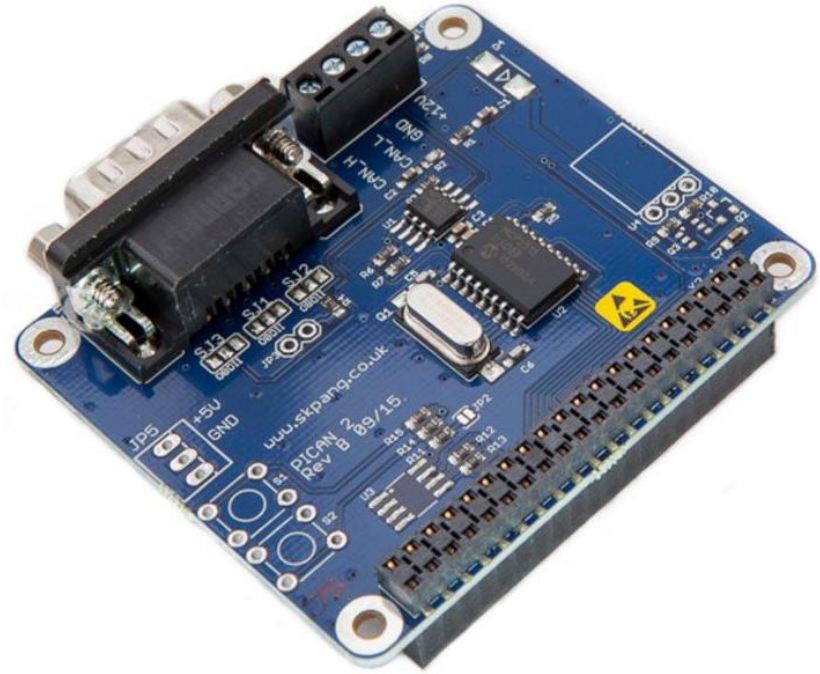
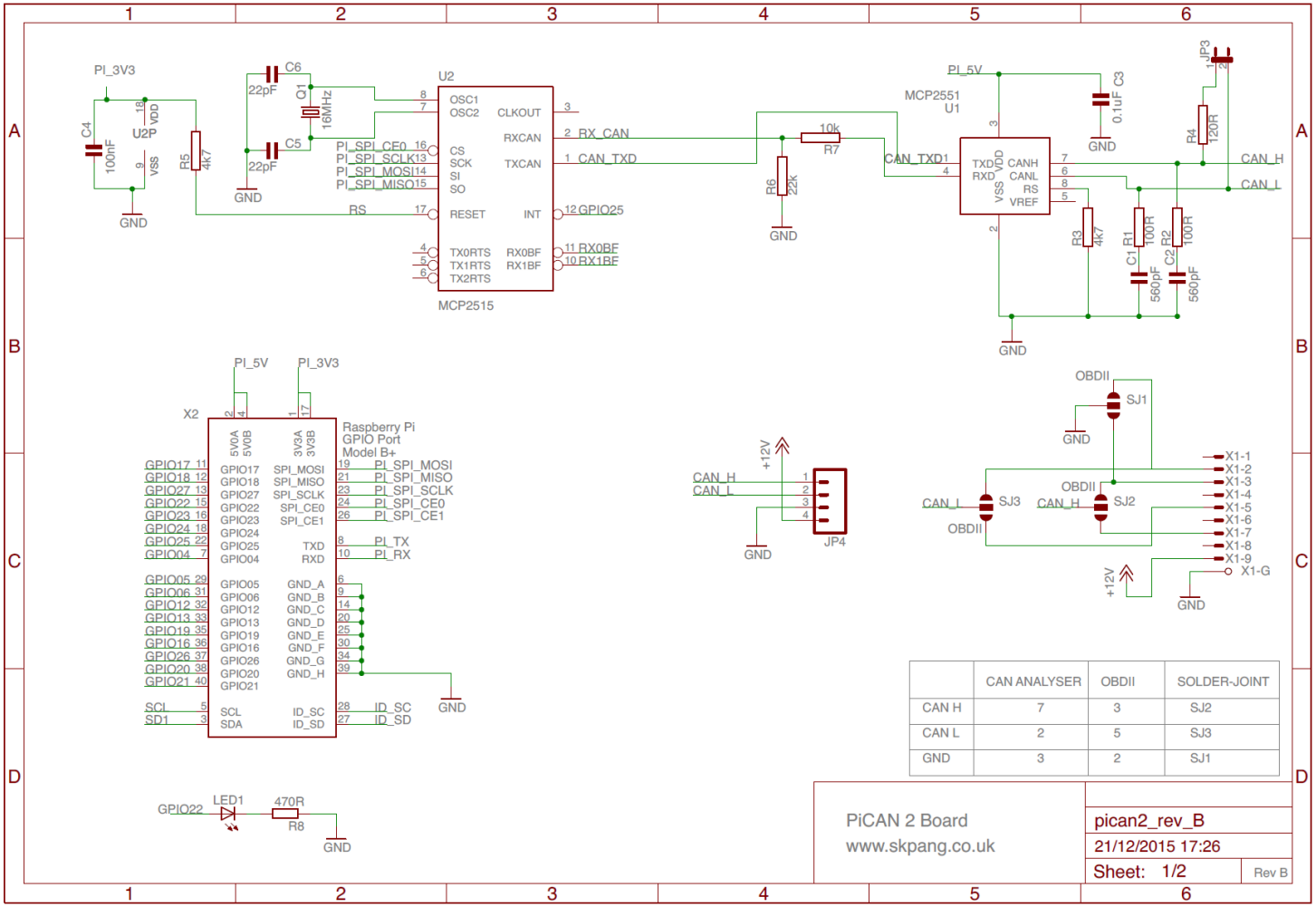


Figure 4.2 - The PiCAN 2 shield

As you can see in Figure 4.3, the PiCAN 2 shield consists of two main components: an MCP2251 High Speed CAN Transceiver [13], and an MCP2515 Stand-Alone CAN Controller with SPI Interface [14]. According to their datasheets, the CAN transceiver can be used to convert the CAN_H, CAN_L differential signalling into a 0-3.3V TTL transfer (Tx) and receive (Rx) signal; and the CAN Controller implements the CAN standard, and drives the transceiver. The Raspberry Pi can send and receive messages via the CAN Controllers SPI interface, and the CAN Controller handles the message transmission, the bit stuffing, the error handling, and so on.

[3]https://www.computerweekly.com/news/252492564/Belgian-security-researcher-hacks-Tesla-with-Raspberry-Pi (last accessed: 2020.11.29.)



	CAN ANALYSER	OBDII	SOLDER-JOINT
CAN H	7	3	SJ2
CAN L	2	5	SJ3
GND	3	2	SJ1

Figure 4.3 - The schematic of the PiCAN 2 shield

Since the CAN Transceiver does not implement any CAN logic, we reuse it in our implementation; however, the CAN Controller has to be reimplemented. We have created the CAN Transceiver circuit on a breadboard, as you can see in Figure 4.4.

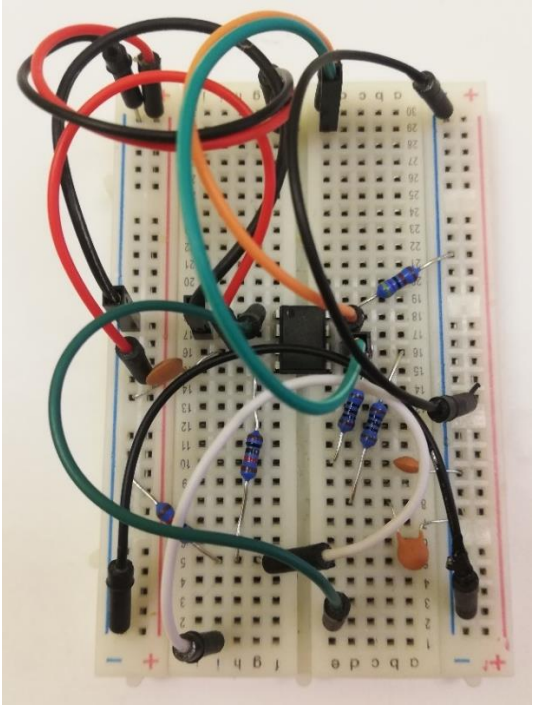


Figure 4.4 - The breadboard version of the CAN Transceiver

In order to validate the voltage levels and the functionality, we made measurements using an oscilloscope. As you can see in Figure 4.5, the CAN Transceiver circuit worked as expected.

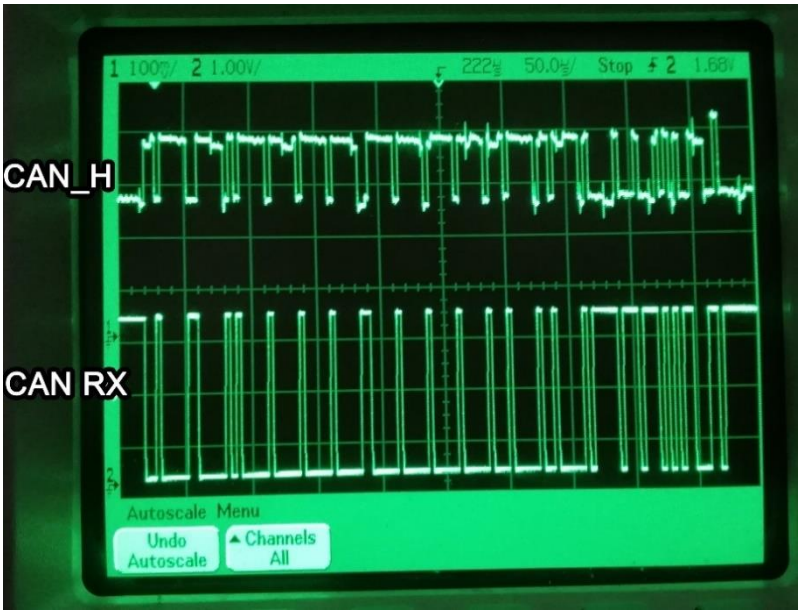


Figure 4.5 - The validation of the CAN Transceiver circuit

However, during the measurements, we noticed that there are occasional connectivity issues with the breadboard circuit; thus, we soldered the circuit on a prototype board, and connected the Tx and Rx wires to the Raspberry Pi as seen on Figure 4.6.

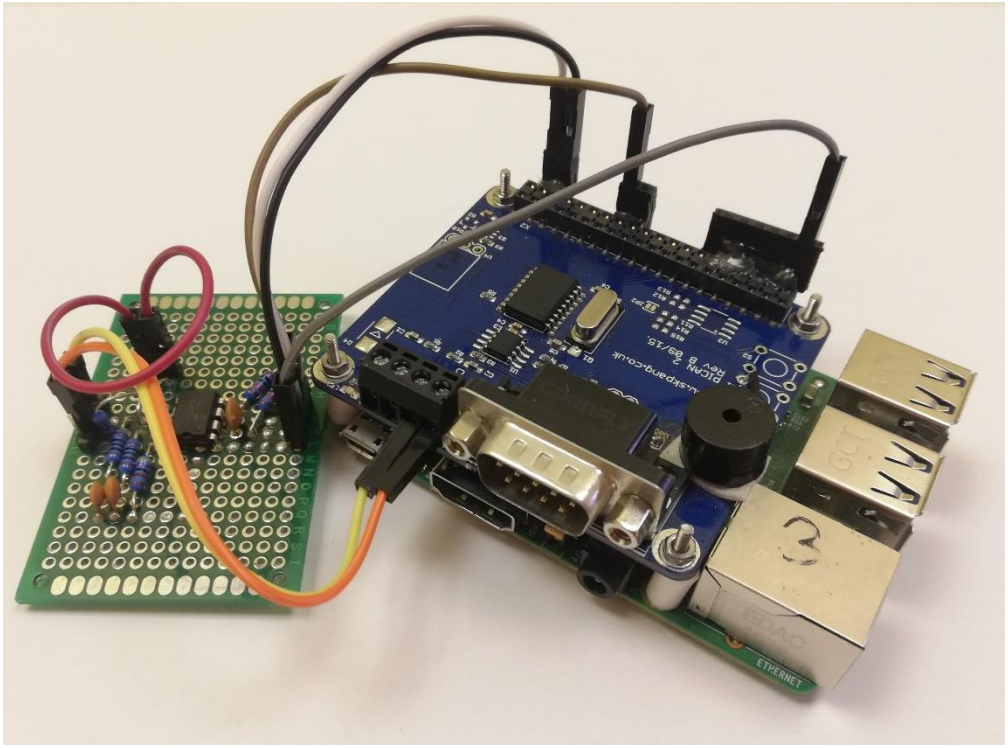


Figure 4.6 - The soldered CAN Transceiver connected to a Raspberry Pi

After connecting the wires, we can start to implement the malicious CAN controller itself.

4.2.2 Implementing the malicious CAN controller using a Raspberry pi

Our malicious CAN Controller's task can be summarized in four steps:

- First, the malicious CAN Controller has to sample the bus constantly, and wait until it detects a TTC message with the ID we want to attack. While this is a simple task, the sampling and the signal processing has to be fast enough, so it does not introduce significant delays in the system, which is a cornerstone for the attack.
- After we detect that the TTC message is in transmission, the malicious CAN Controller has to break the message, while it is still in transmission. Speed is key here also.
- After breaking the TTC message, the malicious CAN Controller has to wait until both the error frame and the grace time passes
- And finally, the malicious CAN Controller can send the crafted TTC message.

Before going into deeper levels of the implementation, the most important question was whether the Raspberry Pi could handle the requirements to be a real-time CAN Controller, which are the following

- We are going to handle bus speeds up to 1Mbps; thus, we must sample the I/O pin of the Raspberry Pi, with great speeds. In case of other serial transceivers, manufacturers usually use a bus oversampling of 4-8-16 times, so the Raspberry Pi also has to be able to sample the bus with at least 4MS/s.
- The signal processing has to be fast enough to extract the ID of the message, and if it is equal to the one we want to attack, intercept it by sending dominant bits, before the message gets acknowledged.
- During the attack, when we get to the time slot where the crafted TTC message should be sent, the Raspberry Pi has to send the signals precisely at the right timing, because the ECUs expect a standard CAN message with tight timing regulations.

In order to determine whether the Raspberry Pi 3 is up to these tasks, we measured the maximum I/O pin changing speed (IOPS) using bit-banging, and the precision of the I/O timing, with an oscilloscope. What we should see on the RPi I/O line is four precisely 1us long peaks, followed by a longer intermission, and then five bit-banging section, placed equally distant. The visualization of the expected measurement results can be seen in Figure 4.7.

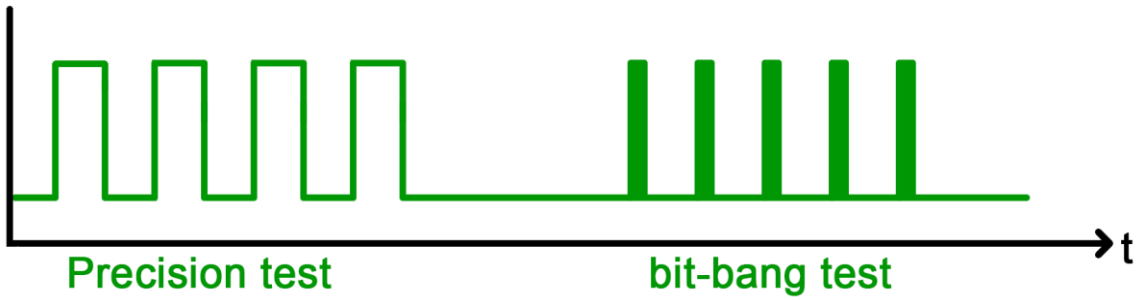


Figure 4.7 - The expected measurement results of the Raspberry Pi I/O

However, the actual measurement results differ greatly from the expected ones, as you can see in Figure 4.8.

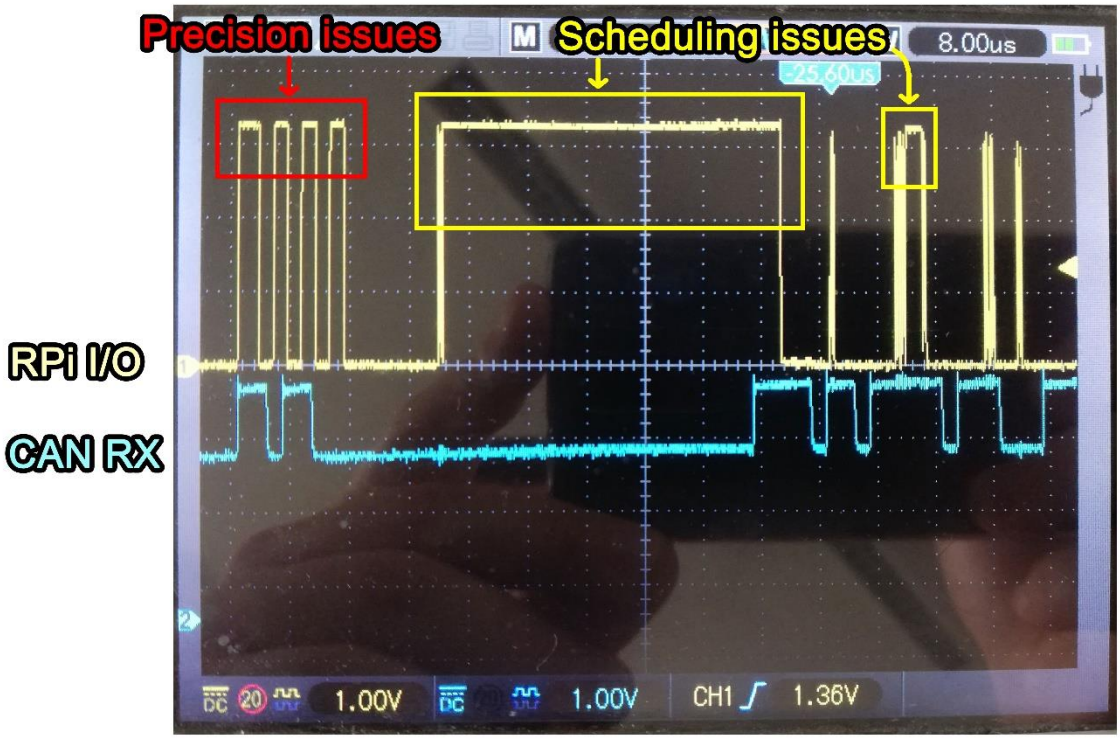


Figure 4.8 - The results of the initial testing of the Raspberry Pi

There are two major issues with the results:

- First of all, while it is not really visible on this overall picture, there were large deviations in the precision of the 1us peaks (marked red). What happened is that during the precision test, the scheduler of the Linux kernel can relatively precisely allow our code to run in time; however, the Raspberry Pi is a soft real-time system at best, which means that we are never going to get the precision we need for the CAN Controller.
- Secondly, while the max IOPS was sufficient, the bit-banging peaks are not spaced at an equal distance, and they are not really peaks, but more like a weird signal (marked yellow). This issue is also due to the scheduler. During the bit-banging test, at one point, our code triggers a CPU watchdog, because we need to sample the bus indefinitely, which the scheduler does not really like. Thus, at one point, the scheduler takes away the CPU time from our software, and the last I/O state remains the same, until next time, when the scheduler lets the software run. These states are the long horizontal lines marked with yellow.

We tried both a pre-emptive and a real-time kernel; nice values and so on. While we managed to lower the maximum time the software was forced to the background, it did not solve any of the issues mentioned before. At this point, it was clear to us that we

cannot use a regular computer for this task. We need a lower-level device, for instance, a microcontroller (uC).

4.2.3 Implementing the malicious CAN controller using microcontrollers

We tried two microcontrollers, the STM32F103 and the NodeMCU ESP32s. While the STM32F103 is a bit older, but a widely used 72MHz microcontroller, the NodeMCU ESP32s is a two-core, 240MHz top-of-the-line, powerful IoT device. However, upon initial testing, we found out that the maximum IOPS we could get even from the faster NodeMCU ESP32s is around 800k IOPS, which is not even nearly enough for our case. Thus, we concluded that even a microcontroller would not be fast enough. We need a different kind of device which can be used for high-speed signal processing, such as an FPGA.

4.2.4 Implementing the malicious CAN controller using an Arty A7 FPGA

The Arty A7 FPGA provides a flexible platform for signal processing, which is easily suitable for our needs, and much more. The state machine of the FPGA's code can be seen in Figure 4.9.

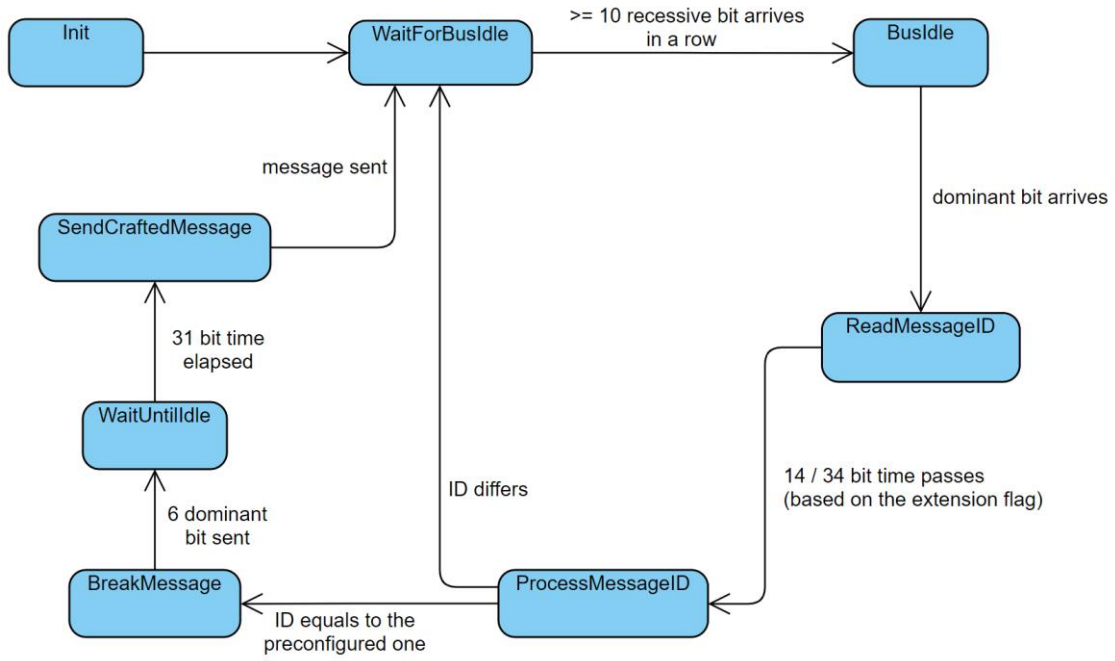


Figure 4.9 - The state machine of the FPGA

After the FPGA starts up, we go into **WaitForBusIdle** state. In this state, we do not know, whether there are messages being transmitted at the moment or not, but in case we detect at least 10 recessive bits, we know that the bus is idle; thus, we go into **BusIdle** state. In this state, if we detect a dominant bit, we know that it must be the SOF bit of the following message; thus, we change to **ReadMessageID** state, where we sample and store the content of the messages. After we sample 14 or 34 bits (based on the extension flag), we can change to **ProcessMessageID** state, where the ID of the message gets calculated. In case the ID differs from the one which was preconfigured as the one to be attacked, we go back to WaitForBusIdle state. However, in case the ID matches, we can go into **BreakMessage** state, where 6 dominant bit gets sent in order to break the sender ECU's message. This triggers the sender ECU without exception to send an error frame; thus, we have to wait until the next frame can be sent, in the **WaitUntilIdle** state. After the maximum time interval of the error frame elapses, we can send the crafted message in the **SendCraftedMessage** state, and finally, return to WaitForBusIdle state.

The code itself was written in Verilog, and it consists of five main modules:

- **CANTimer**: This module converts the 100MHz clock of the Arty A7 FPGA into a 4MHz clock, which can be used by other modules to sample the CAN bus.
- **CANShiftRegister**: This module is a modified shift register. It both samples the bus, removes the stuffed bits, and stores it in a shift register. The output registry contains only the original message's values.
- **CANMessageGenerator**: This module would generate the arbitrary CAN messages, and send them to the CAN bus, if it were implemented. More on this later.
- **AttackerLogic**: This top module is responsible for:
    - containing the previous three modules, and wiring them together
    - managing the state machine of the device, and enabling and disabling the modules based on the current state
    - handling the bus idle logic
    - processing the message ID
    - breaking the message, if the ID is correct.

Using an FPGA comes with several positive factors. For instance, we can create a virtual testbed for each of the modules, and simulate their behaviour during operation with a far

more accurate simulation, than any higher-level device would provide. In Figure 4.10, we show the results of a simulation, created after the message breaking logic was implemented.

As you can see in the figure, there are four messages traveling on the simulated CAN bus (CAN_RX line), three with a normal length ID, and one with an extended ID. From these messages, only one matches the preconfigured target ID of the device; thus, only the third message would get interrupted by the device (CAN_TX line).

At this point of the development, the next step would have been to implement the message injection functionality, but first, we wanted to test whether our theory works on a real CAN network, and the TTC messages do not get retransmitted automatically. In the next section, we present the evaluation of the real-life testing of the malicious CAN controller.

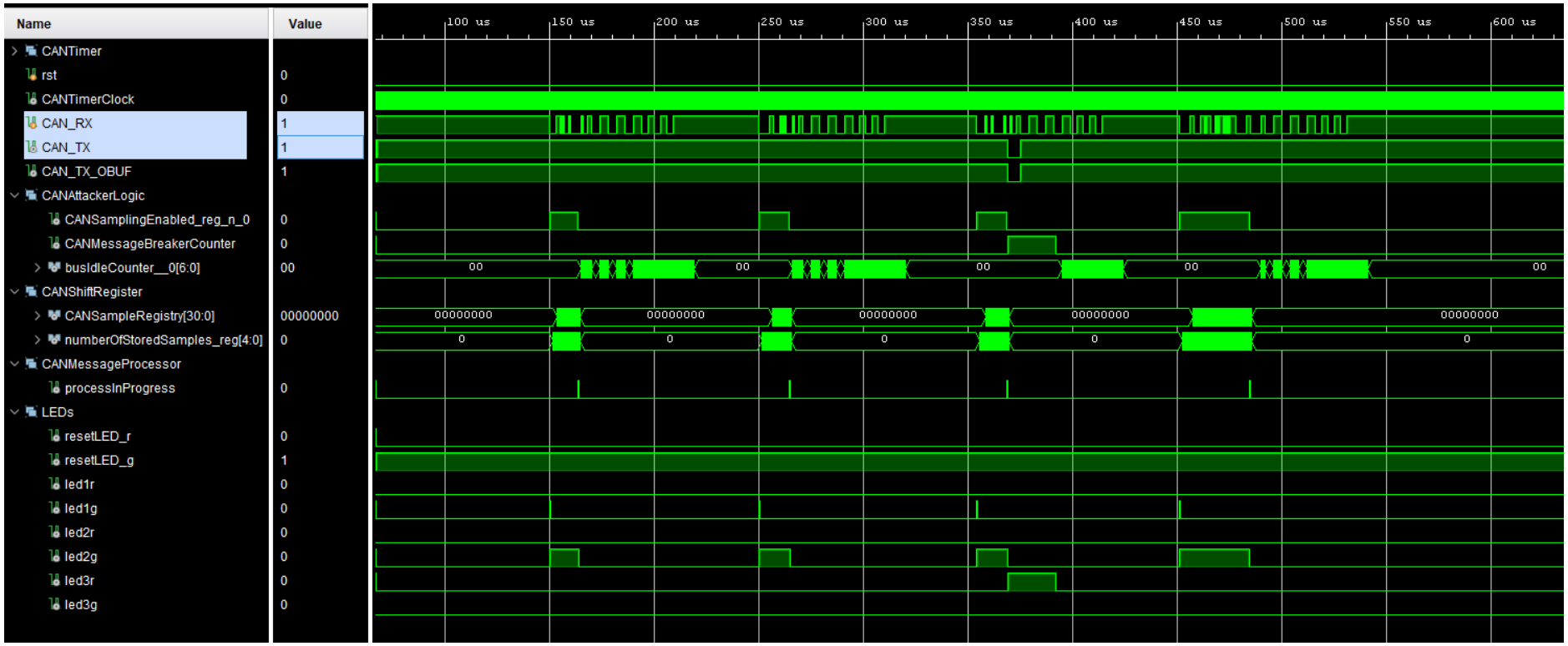


Figure 4.10 - The simulation of the FPGA's behaviour

4.3 Evaluation

In order to test the device, first, we created a testbed using a Raspberry Pi with a PiCAN shield, as seen in Figure 4.11.

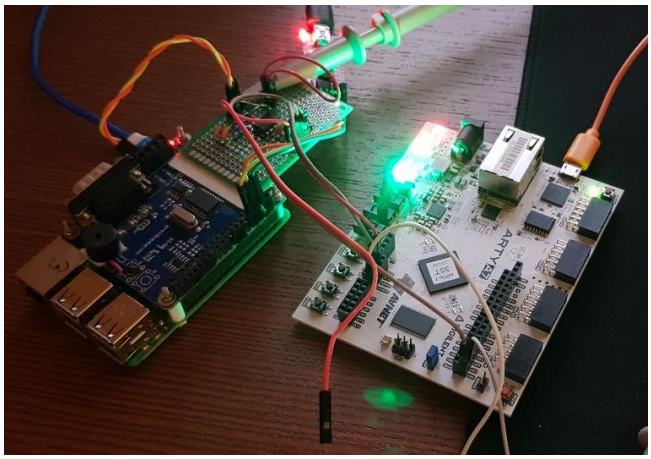


Figure 4.11 - The first testbed, using a Raspberry Pi and the PiCAN shield

After configuring the FPGA, we have verified that the simulations were accurate, and the device can break messages with an arbitrary ID, and it only breaks the desired messages. As you can see in Figure 4.12, the FPGA reads as many bits as required to verify the ID, and sends the message breaker signal after an almost instantaneous message ID processing.

Figure 4.12 - The measurement of the message breaking using the Raspberry Pi

You can notice that despite sending the message breaker signal, the CAN_RX still shows the original message arriving. During this test, the CAN_TX wire of the FPGA was only connected to the oscilloscope, and not to the PiCAN shield, because the testbed used on-demand message transmission instead of TTC; thus, the messages would have been resubmitted anyways.

After verifying the device with our local testbed, we made a more realistic measurement using the Citroen C5 test bench of the Faculty of Transportation Engineering and Vehicle Engineering. This test bench contains the electronics of a real-life Citroen C5, built onto an interface to allow students to conduct measurements. The test bench can be seen in Figure 4.13.

Figure 4.13 - The Citroen C5 test bench

First, we monitored the ongoing traffic and determined which messages could be sent by TTC. Next, we connected the device, and configured it to break one of the potential TTC messages. The results of the measurement can be seen in Figure 4.14. Sadly, as you can see in the figure, the results did not match our expectations. While the FPGA succeeded at breaking the targeted message, when the grace time passed after the error frame, the sender ECU resent the original message, which got also broken, and this cycle repeated until the sender ECU switched into first error-passive and then recovery mode due to the increase of the TEC. At this point, after a few seconds, the ECU got turned back into error-active mode, and the whole cycle started over again.

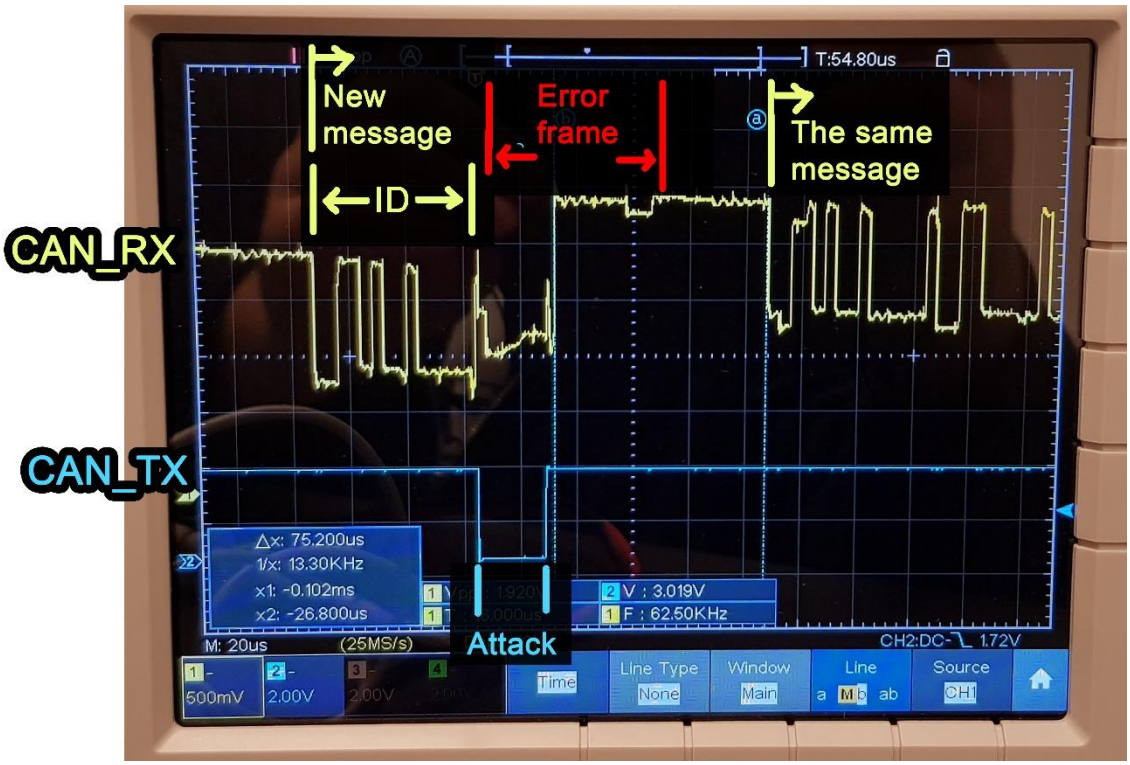


Figure 4.14 - The FPGA breaking a message on the Citroen C5 test bench

As you can see in Figure 4.15, the message resubmission gets repeated 32 times, and after that, the sender ECU turns into recovery mode.

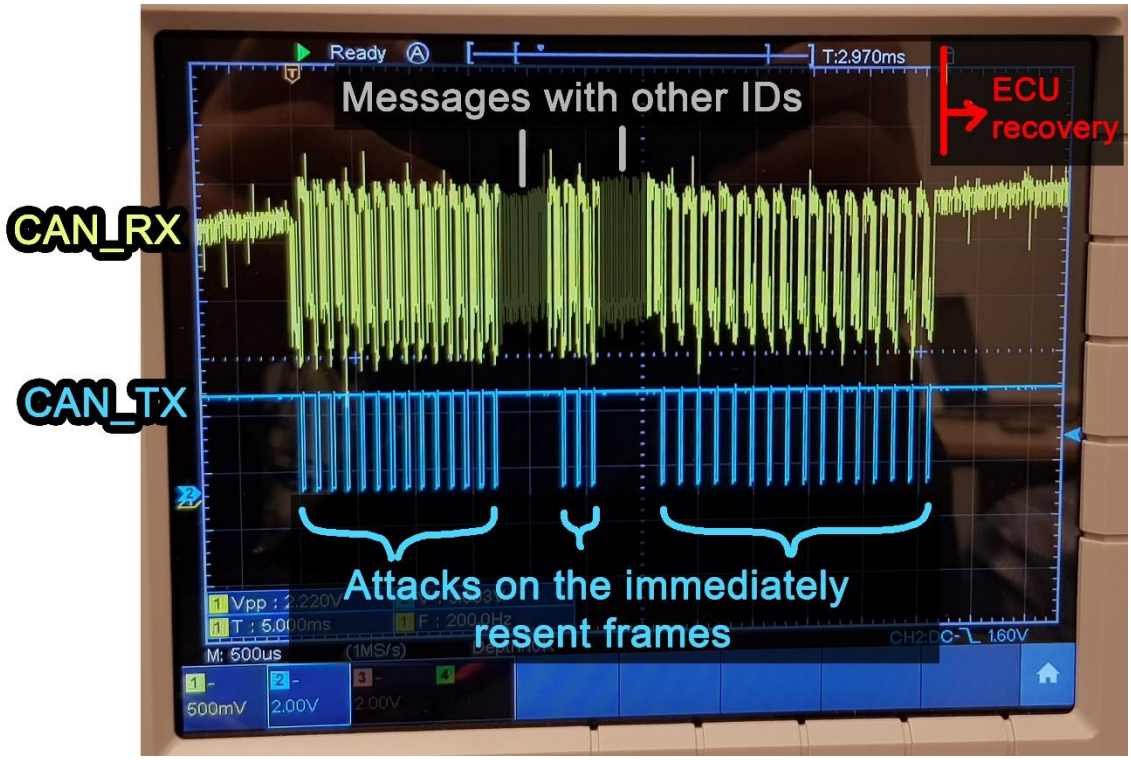


Figure 4.15 - The message breaking leads to the ECU turning into recovery mode on the test bench

Based on what we experienced, we tried other potential TTC message IDs, but the results were the same. While this method was promising, it looks like the manufacturers do not follow the standard word-by-word. However, it is important to emphasize that we only verified our theory on one real-life car test bench using a few different ECUs; thus, we cannot say for sure that this attack would not work on any other ECUs.

In the next section, we are going to show our second approach to this problem, which is based on a completely different principle than this attack was.

5 The second approach – Creating a CAN gateway

After the malicious CAN controller idea not working out exactly as expected, we took a different approach. Rather than trying to modify the messages on the bus, what if we performed a Man-In-The-Middle (MITM) attack instead? This would allow us to modify any message we would like to, without any noticeable side effects. But first, let us describe what a MITM attack is, in the next section.

5.1 MITM attacks

The concept of the MITM attack is quite simple. Let us suppose there are two persons communicating with each other, called Alice and Bob. If a third, malicious person (called Mallory) finds a way to eavesdrop on their messages, and intercept the communication between them, in case Alice and Bob do not use any kind of message authentication measure, Mallory may get to change the messages between the communication partners without them ever noticing it.

In case of the CAN network, the communication partners are the ECUs. Since the CAN standard does not support message authentication, if we manage to wedge a special device between the two ECUs' communication line, we can add, modify, or even delete messages with an arbitrary ID. The visualization of the attack can be seen in Figure 5.1.

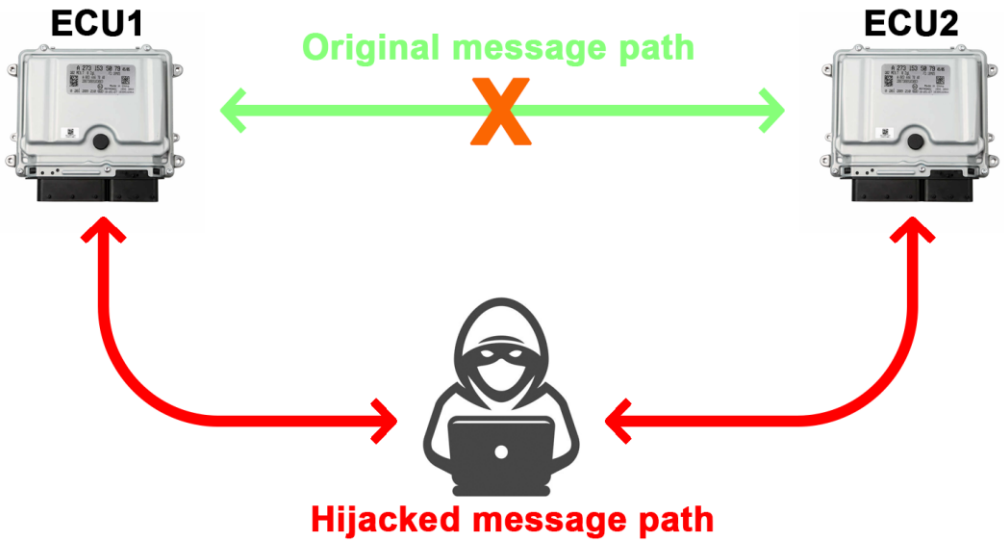


Figure 5.1 - The concept of a MITM attack in the automotive industry

Now that we have introduced the attack, let us go into details about designing the device itself.

5.2 Designing the CAN gateway

In order to create a CAN gateway, we are going to use NodeMCU ESP32s microcontrollers. As described briefly in section 4.2.3, the ESP32s is a two-core, 240MHz versatile microcontroller. It has a built-in CAN controller, Wi-Fi module, and it comes with the Espressif IoT Development Framework (ESP-IDF), which is based on the popular Free Real-Time OS (FreeRTOS) platform. Using the ESP-IDF allows us to create a hard-real-time device using its built-in scheduler, while providing convenient features, for instance, a built-in web server, which will be useful for the usage of the device. Another not insignificant reason to use the ESP32s is that during the previous attempts mentioned before, we got familiar with the microcontroller, and using this device allowed us to rapidly develop our ideas, without a fuss.

There were several requirements against the proof-of-concept CAN gateway:

- It shall be able to handle ISO 11898 High-Speed CAN messages, with a bus speed of up to 1Mbps
- It shall have as low introduced delay as possible, preferably low enough that it is not significantly greater than a lost arbitration or a transient error
- Despite being a proof-of-concept device:
    - It shall be robust enough to be used later by other researchers in the laboratory
    - It shall be easy to configure during testing, and configuration shall not require reprogramming of the device.
    - It shall be as serviceable as possible, without requiring specific hardware knowledge. The components shall be replaceable in case it is required.

The first step of the design process was to create a high-level architecture of the CAN gateway, which can be seen in Figure 5.2.

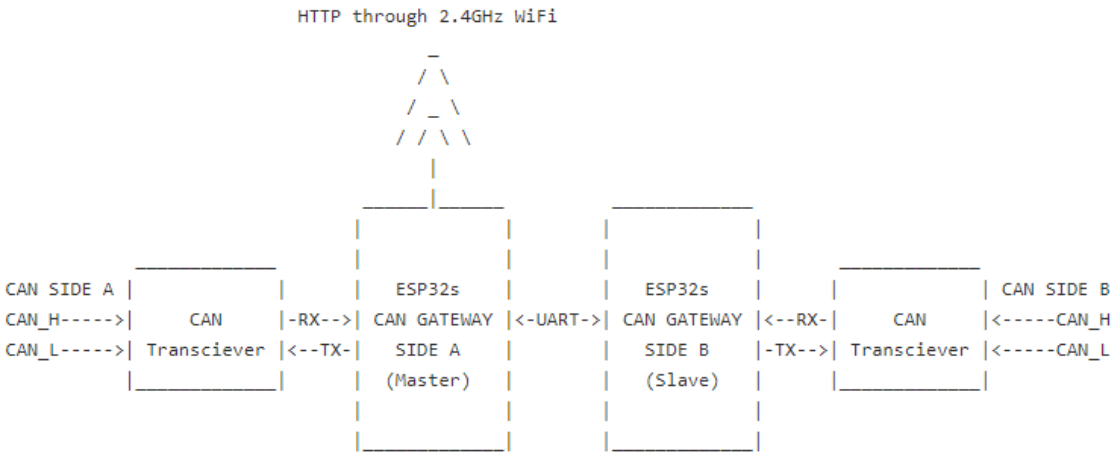


Figure 5.2 - High-level architecture of the CAN gateway

As you can see in the figure, the CAN gateway consists of two CAN transceivers, and two ESP32s microcontrollers. One of the microcontrollers is the master, and the other one is the slave. Each microcontroller handles one side of the CAN bus via its built-in CAN controller, and the corresponding CAN transceiver. The communication between the microcontrollers is realized via a UART line. The master provides a web interface for configuration through its 2.4GHz Wi-Fi module, working as an access point.

After designing the high-level architecture, we decided that since this is a very specific device that has to be reliable, the best thing would be to create a dedicated hardware for it. The schematic of the device can be seen in Figure 5.3 and Figure 5.4.

Apart from the CAN transceivers' related circuitry, there are only a few extra components:

- Each microcontroller has three feedback LED: a master, a slave, and an error LED. The master and slave LEDs are used by the microcontroller to signal what role it has chosen during the start-up (more on this later), and the error LED is being used to notify the user in case an error occurs.
- A connector for a 5V input voltage, provided by a DC-DC buck-boost converter.
- The schematic includes a combined reset circuit, which lets the user reset both microcontrollers at the same, while also enabling the master microcontroller to reset the slave during start-up. With this solution, we can run the same code on both microcontrollers, and the microcontrollers themselves decide their roles during the start-up sequence, which will be described in more detail in section 5.3.2.
- And finally, there are several testing points, which are necessary to test the device's behaviour during operation. The testing points will be discussed in section 5.3.1.

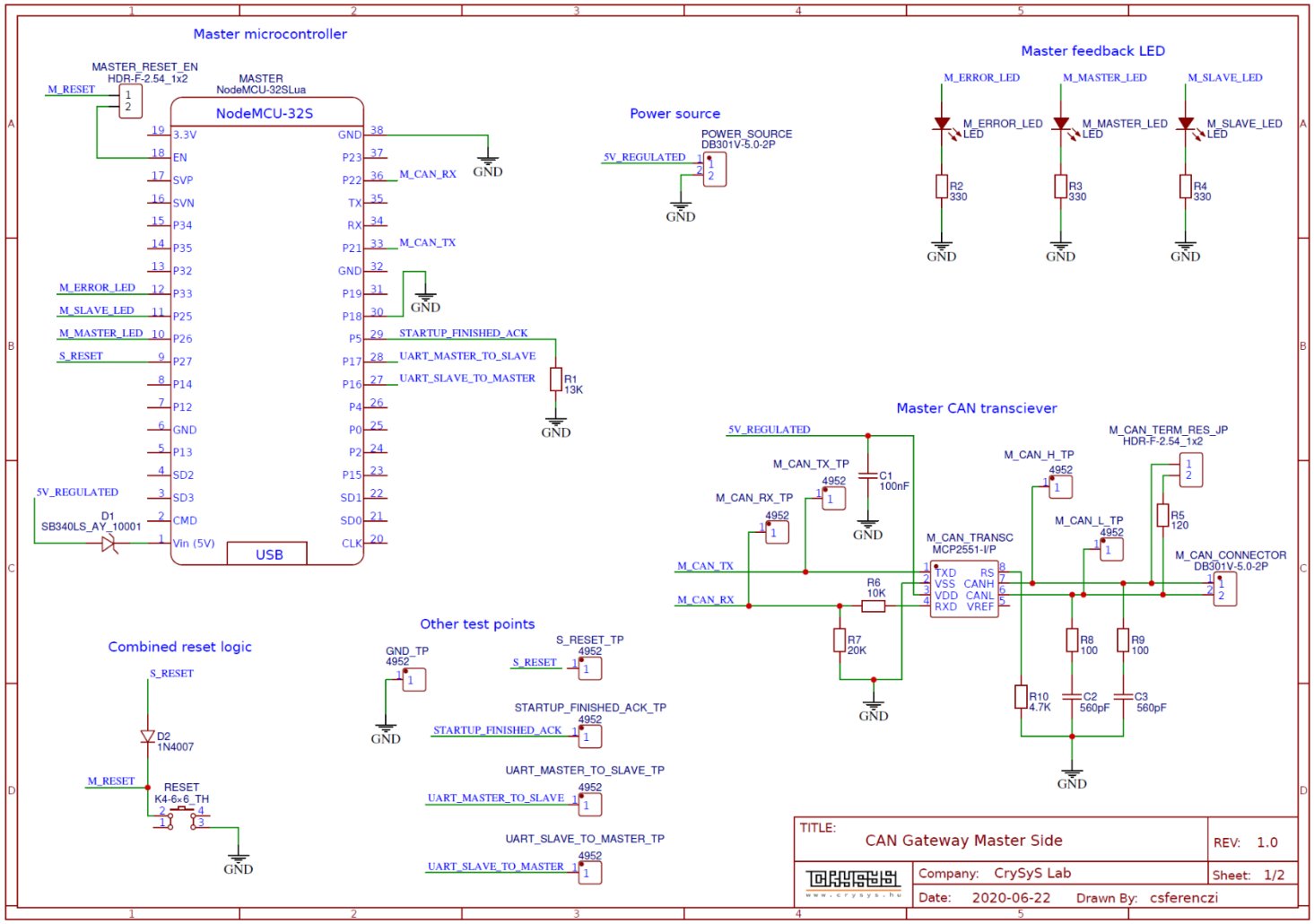


Figure 5.3 - The schematic of the CAN gateway - Master side

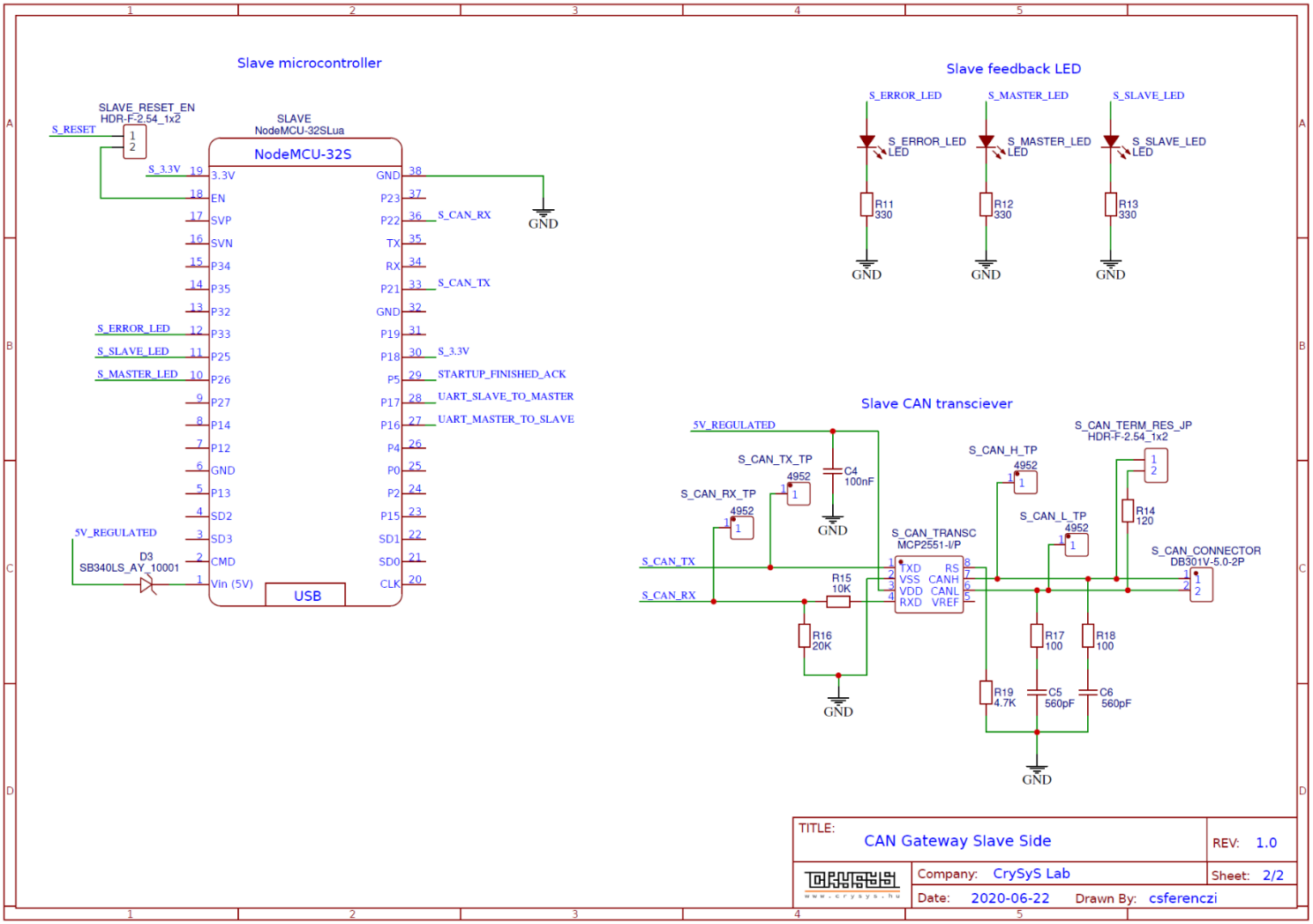


Figure 5.4 - The schematic of the CAN gateway - Slave side

5.3 Implementation

5.3.1 Hardware

After designing the schematics and verifying it on a breadboard, we have built a soldered version from the circuit. We used a protoboard as the base of the device, which allowed us to apply minor changes to the hardware design without having to rebuild the complete circuit again. The components have been soldered onto the protoboard using sockets in order to make a potential component replacement easier to achieve.

The picture of the finished proof-of-concept CAN gateway board can be seen in Figure 5.5 and Figure 5.6.

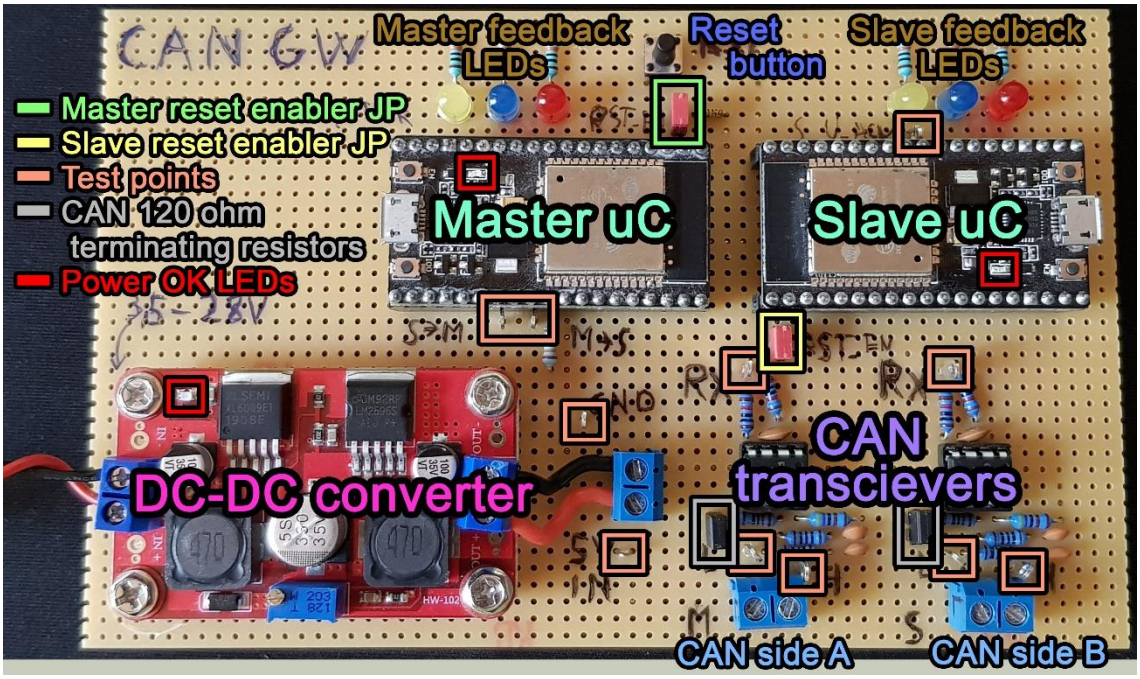


Figure 5.5 - The top view of the proof-of-concept CAN gateway board

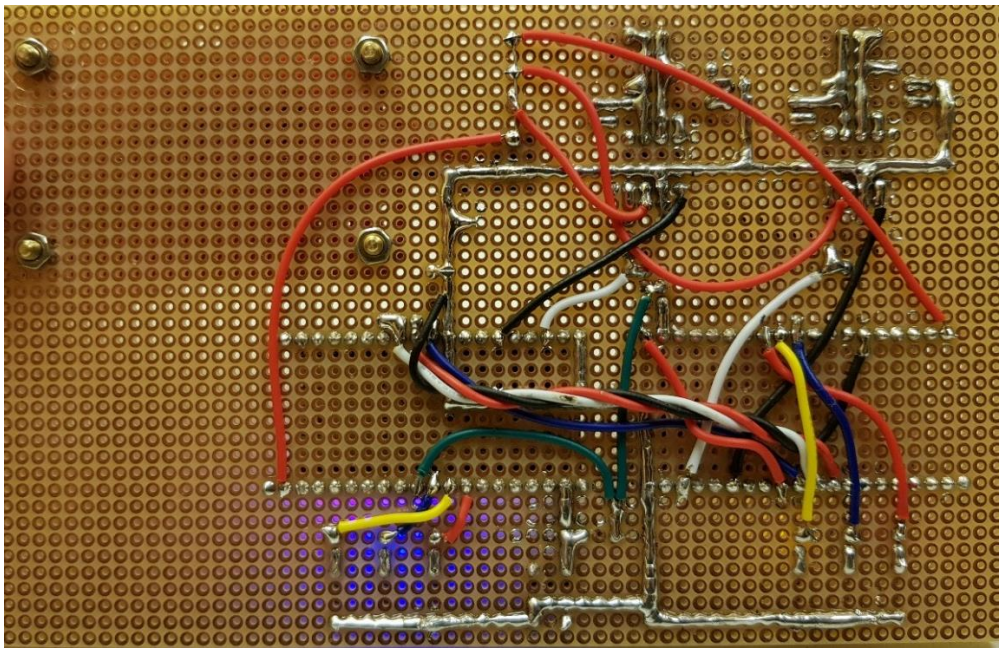


Figure 5.6 - The bottom view of the proof-of-concept CAN gateway board

Before going onward to the software implementation, one of the most important things to mention is the test points of the device. Describing them in this section will make measurement results easier to understand in later sections.

The list of test points based on their orientation in Figure 5.5 is the following:

- Above the Slave microcontroller:
    - **S_U_ACK – Start-up ack**: this wire is used to signal the end of the configuration phase, between the two microcontrollers.
- Below the Master microcontroller:
    - **S->M**: Slave to Master UART line (TX for Slave, RX for Master)
    - **M->S**: Master to Slave UART line (TX for Master, RX for Slave)
- Next to the DC-DC converter:
    - **GND**: Negative polarity connection point for measurements
    - **5V_IN**: the stabilized 5V output voltage test point for the DC-DC buck-boost converter.
- Around the CAN transceivers:
    - **L** and **H**: CAN_L and CAN_H wire testing points, in case we would like to directly sample the CAN bus.
    - **RX:** The CAN bus differential signal converted to a 0-3.3V signal.

With the usage of these test points, the behaviour of the board, and thus, our theory can be easily validated during operation. We are going to discuss the validation process in more details in section 5.4

Now that we have described the hardware to execute the attack on, we are going to show you how the software side of the device has been implemented.

5.3.2 Software

The code of the proof-of-concept CAN gateway was written in C/C++ based on the ESP-IDF framework. While the ESP-IDF environment itself is not object-oriented (OO), our code follows the OO principles as much as possible, except where the framework's C/C++ translation requires us not to.

The software consists of several components:

- **LED**: This class is responsible for driving an external LED
- **RoleSelector**: The sockets of the microcontrollers have been designed in such a way that a designated role pin is set to different voltage levels for the master and the slave. Thus, during start-up, the microcontrollers can decide whether they are the master or the slave, which allows us to run the exact same code on both microcontrollers. The RoleSelector's purpose is to read the voltage level on the role pin and decide whether the given microcontroller's role should be master or slave. The result is stored in a global **Role** object
- **WiFiHandler** and **WebServerHandler**: During the start-up procedure, the master creates a Wi-Fi access point. The user can connect to this Wi-Fi network, and access a web server through which the user can configure the attack parameters of the device, such as the bus speed, attack type, and so on, detailed in section 5.3.3. The WiFiHandler is responsible for creating the Wi-Fi network, and destroying it after the configuration; and the WebServerHandler is responsible for providing a configuration web interface, validating the request, and parsing the new configuration into an **AttackConfig** object.
- **UARTHandler**: This class is responsible for the internal communication between the two microcontrollers. During the start-up procedure, an internal connection test verifies that the UART communication is in working order in both directions. After the test passes, the UARTHandler can transfer CAN messages between the microcontrollers.

- **CANHandler**: This class is responsible for initializing and managing the CAN interface of the microcontrollers. During start-up, it verifies whether the CAN controller is in working order. After that, during the operation, it receives messages from the CAN bus, and relays them to the UARTHandler; and transmits the messages coming from the UARTHandler to the CAN bus. In case of the master microcontroller, this class is also responsible for checking the ID of the message, and passing it to the **MessageAttacker**, in case the ID matches.
- **MessageAttacker**: This component is responsible for carrying out the different attacks on the received CAN messages.
- **Main**: The top-level component, which handles the component initialization, schedules the configuration, sets the new AttackConfig in the MessageAttacker, and starts the message processing.
- **PowerManager**: There are two kinds of errors that can happen during operation. The first one is the recoverable, which can be handled within the component. For instance, in case the user sends a new configuration via the web interface, but during the parsing process, we find out that there is an issue with the configuration, the WebServerHandler can send an error message to the user, and we wait for the next configuration. However, in case a component's initialization fails, or there are issues with the UART network or with the CAN controller, the device will not be able to operate properly. In such cases, the components can use the PowerManager component to terminate any tasks, signal the error, and turn the device into a low power standby mode.

Another important aspect of the software is how the processing works. As mentioned before, the ESP-IDF is based on the FreeRTOS platform, which uses tasks as the core element of code execution. These tasks can then be set to run at the occurrence of an event, or periodically using a timer, or even indefinitely. In the CAN gateway, there are two tasks running simultaneously on the two cores, one for receiving and one for transmitting the messages. Since there are only two tasks, and both of these are bound to a different core, we can guarantee a low response time, because the scheduler does not have to schedule any other tasks during the operation, and thus, none of the tasks gets interrupted.

Now that we described how the software looks like, let us take a brief look at how to operate the device.

5.3.3 Operation

In order to utilize the device, the first step is to wire it up correctly and perform the start-up procedure, which is described in detail in the user guide. After starting up the device, it provides multiple interfaces for configuration, as we mentioned before. The easy configurability is a key element in making the device usable for laboratory purposes. As a testing device, we have to make it as easy to use as possible, because we cannot rely on the users to understand the code, change the hard-coded configuration every time, and reprogram the device. Instead, we created a web interface for the device which allows the user to configure it using a simple web browser, or via direct POST request (e.g., using curl). The graphical web configuration interface of the device can be seen in Figure 5.7.

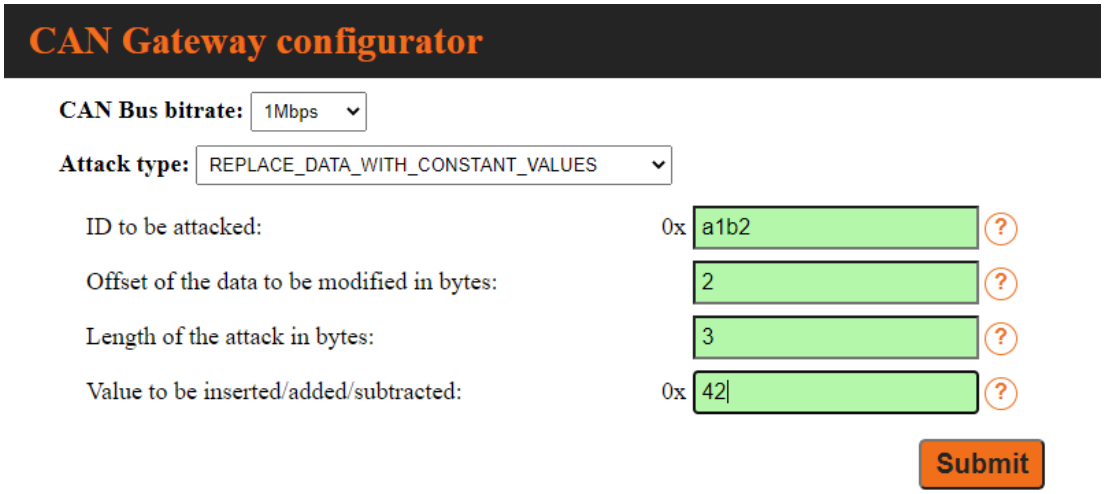


Figure 5.7 - The graphical web configuration interface of the CAN gateway

The configurable parameters are the following:

- **Bitrate**: This parameter can set the bitrate of the CAN bus.
- **Id**: The id of the CAN message to be attacked
- **Offset** and **AttackLength**: The offset controls the position of the first byte to be attacked, and the attackLength determines how many bytes will be attacked. Together they select the bytes which will be attacked by the device.
- **ByteValue**: Some of the attack types require an additional parameter, which will replace the original or will be added or subtracted from the selected byte values.
- **AttackType**: There are several different attacks the device can perform:
    - **PASSTHROUGH**: In this mode, the device relays the traffic without modifying any of the messages.

- **REPLACE_DATA_WITH_CONSTANT_VALUES**: In this mode, the device replaces the selected bytes in the message with the given ByteValue parameter.
- **REPLACE_DATA_WITH_RANDOM_VALUES**: In this mode, the device generates random bytes for each of the selected bytes in the message, and replaces them.
- **ADD_DELTA_VALUE_TO_THE_DATA**: In this mode, we add the given ByteValue parameter to each of the selected bytes in the message. In case the resulting values would overflow, it gets capped at the maximum 255 value.
- **SUBTRACT_DELTA_VALUE_FROM_THE_DATA**: Similar to the previous attack type, but the byte ByteValue is subtracted instead of added. In case the resulting value would underflow, it gets bounded at the minimum 0 value.
- **INCREASE_DATA_UNTIL_MAX_VALUE**: In this mode, we take the lowest value from the selected bytes, increase it by one and replace all of the selected bytes if the increased byte is higher than the original. This is repeated until the max value (0xff) is reached. For instance, let us suppose we attack two positions. In the original message, the first byte is 0x03, and the second is 0x11. The attack would result in the following messages:

  0x03,0x11
  0x04,0x11
  …
  0x10,0x11
  0x11,0x11
  0x12,0x12
  …
  0xfe,0xfe
  0xff,0xff
  0xff,0xff
  …
- **DECREASE_DATA_UNTIL_MIN_VALUE**: This attack type is similar to the previous one, but at the start, we take the highest value from the selected bytes and decrease it every message, until we reach 0x00.

- **REPLACE_DATA_WITH_INCREASING_COUNTER**: We start a counter from 0 and increase it by one at every occurrence of the message. The selected bytes get replaced with the counter. The counter can overflow.
- **REPLACE_DATA_WITH_DECREASING_COUNTER**: Similar to the previous attack type, but the selected bytes get replaced by a decreasing counter starting from 255, which can underflow.

After the user sends the configuration via the graphical web interface or via a direct POST request, the device validates the configuration on the server-side, and if it is correct, it starts the attack phase. During the attack phase, the device passes through messages, or attacks the selected IDs per configured.

Now that we described how to operate the device, it is time for us to show the results of the measurements we created using the CAN gateway.

5.4 Evaluation

During the evaluation, we performed two tests, similar to section 4.3. First, we created a testbed using a Raspberry Pi with a PiCAN shield; and a NodeMCU ESP32s with an additional CAN Transceiver as our CAN nodes. The nodes were configured to send messages to each other via the CAN bus; however, they could only send these messages through the CAN gateway. The testbed can be seen in Figure 5.8.

During the measurements, we tested all attack types, with different IDs, offsets, and attack lengths, while logging both the UART lines, as well as the messages on both sides of the gateway. As we found, the CAN gateway managed to modify the messages with an introduced delay of ~260us on the 1Mbps CAN bus, which is only just a bit longer, than a lost arbitration.

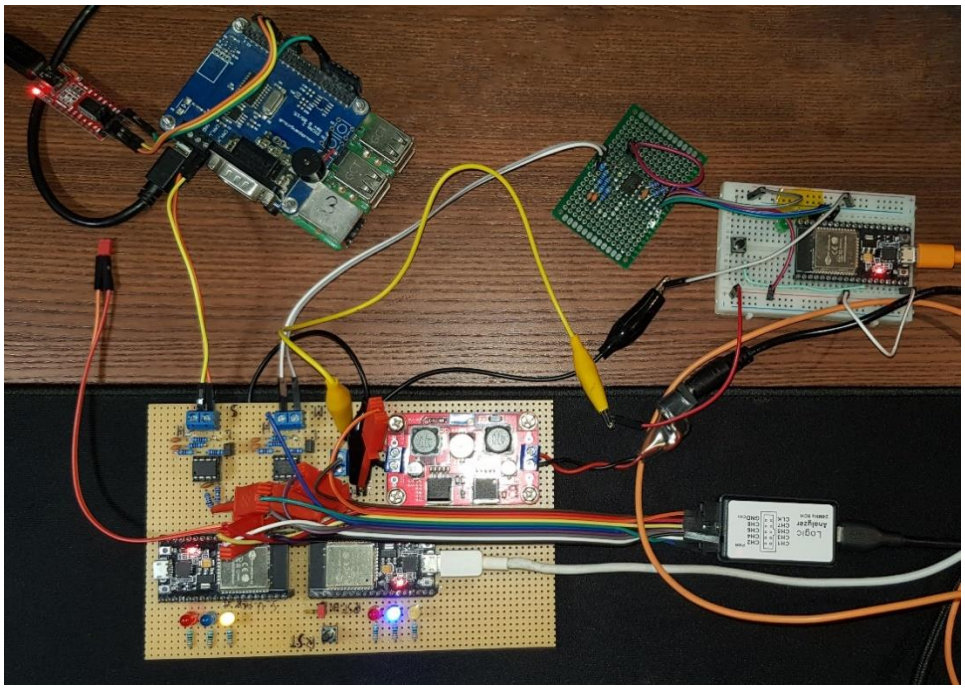


Figure 5.8 - The CAN gateway testbed using a Raspberry Pi and a NodeMCU ESP32s

Since presenting all of the different attack types are out of scope for this paper, we would like to show the results through an example. The attack parameters were the following:

- Bitrate: **1Mbps**
- ID: **0x090**
- Offset: **2**
- AttackLength: **3**
- ByteValue: **0x08**
- AttackType: **REPLACE_DATA_WITH_CONSTANT_VALUES**

A screenshot of a measurement can be seen in Figure 5.9. As you can see, there are two messages traveling on the bus at the same time:

```
Side A to B: ID: 0x090, Data: 0x00 0x80 0x80 0x80 0x41 0x41 0x00
Side B to A: ID: 0x045, Data: 0x01 0xf2 0x03 0xf4 0x05 0xf6 0x07 0xf8
```

However, after both messages go through the CAN gateway, the targeted message with the 0x090 ID arrives with changed values. On the arriving side, the following messages are present

```
Side A to B: ID: 0x090, Data: 0x00 0x80 0x08 0x08 0x08 0x41 0x00
Side B to A: ID: 0x045, Data: 0x01 0xf2 0x03 0xf4 0x05 0xf6 0x07 0xf8
```

Thus, we can say that the CAN gateway has successfully modified the preconfigured part of the message.

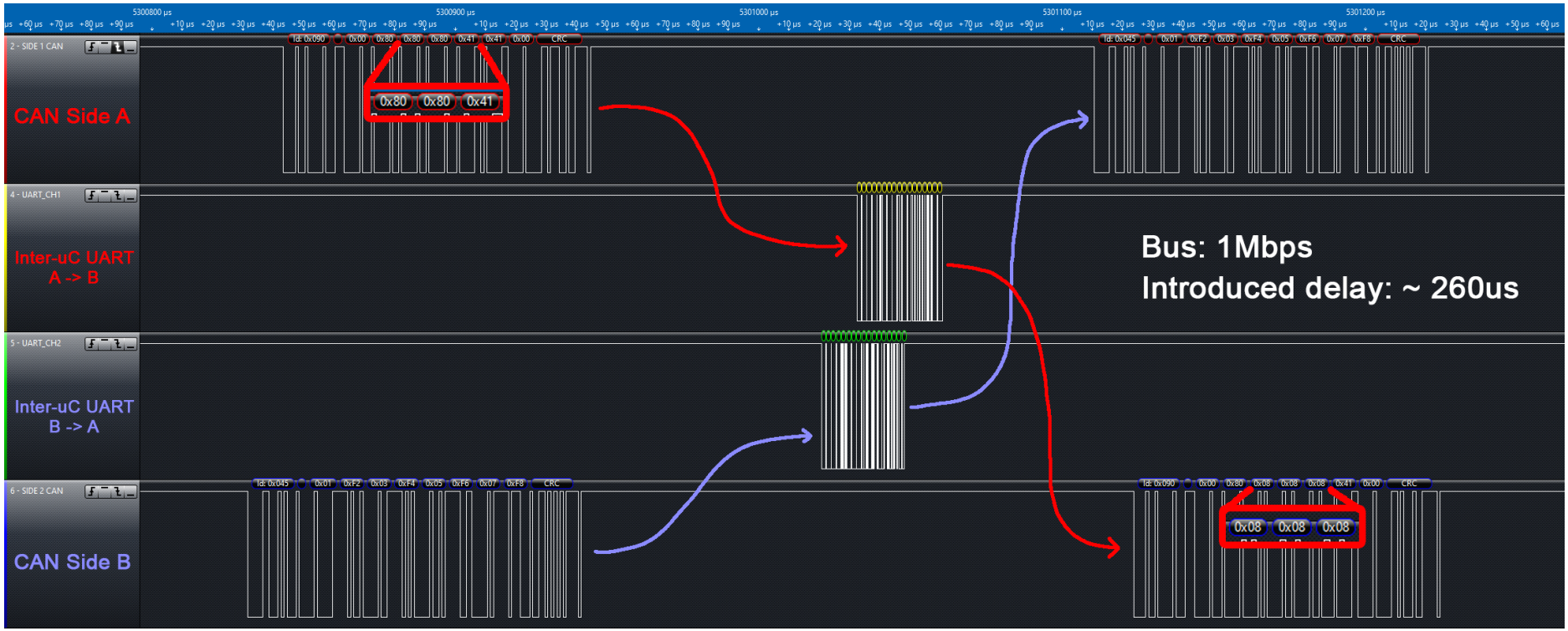


Figure 5.9 - Simultaneous message transmission in both direction

After verifying that our attack worked in our local testbed, we wanted to test it on the Citroen C5 test bench. After inspecting the internals of the test bench, we found that the components are modular, everything is connected via screw terminals, thus performing the attack could be feasible. The internals of the test bench can be seen in Figure 5.10.

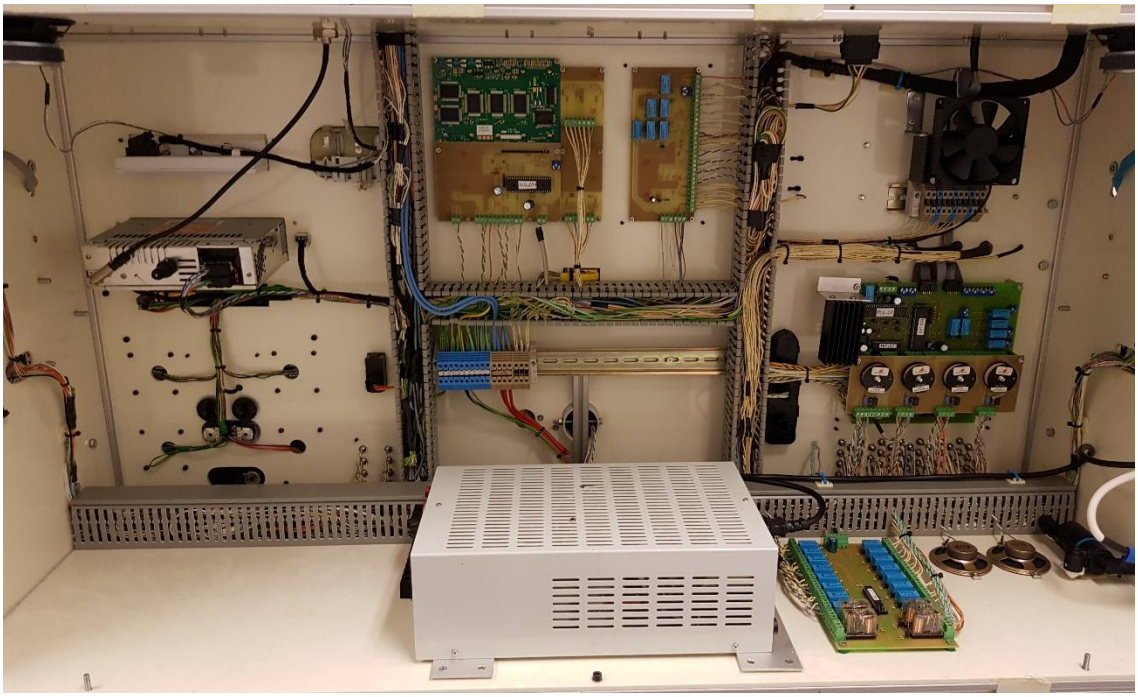


Figure 5.10 - The internal components of the Citroen C5 test bench

Using the documentation of the test bench and making measurements, we have located an easy to access point of the CAN bus, located below the main feedback LCD of the test bench. Connecting the CAN gateway between the main feedback LCD and the rest of the CAN bus lets us verify that the device is working as expected. The connected gateway can be seen in Figure 5.11. The attack parameters were the following:

- Bitrate: **250kbps**
- ID: **0x348**
- Offset: **1**
- AttackLength: **2**
- ByteValue: **0x80**
- AttackType: **REPLACE_DATA_WITH_CONSTANT_VALUES**

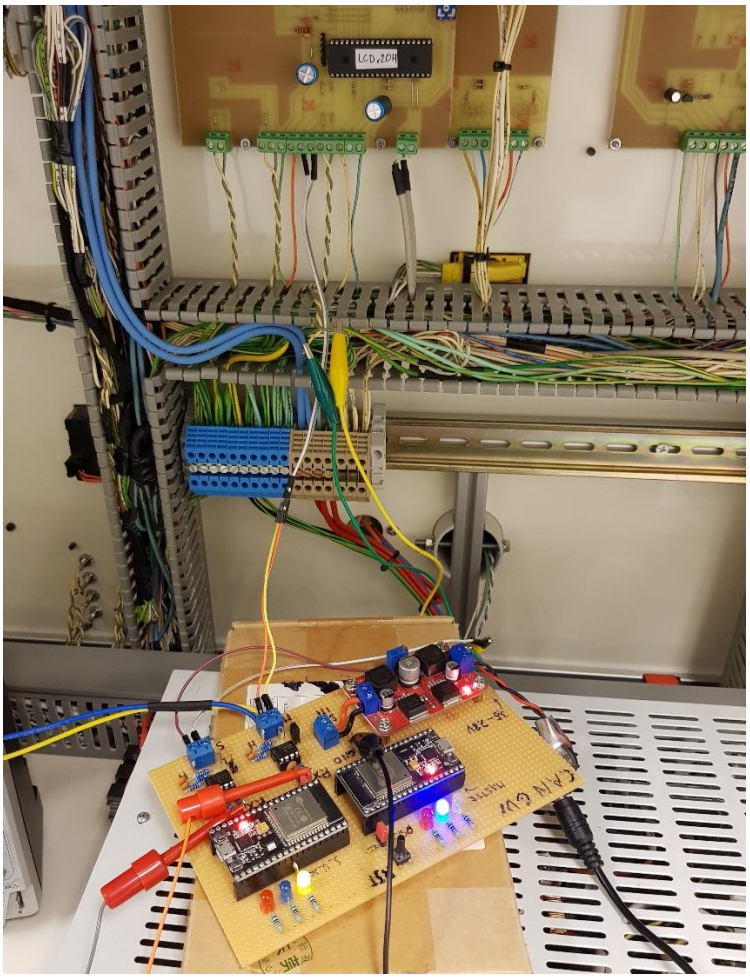


Figure 5.11 - The CAN gateway connected between the ECU and the main feedback LCD

As you can see in Figure 5.14, as the result of the attack, the original 0x26 values got overwritten to 0x80. While this test verifies that our solution works, we wanted to perform a more spectacular test, using the dashboard. After finding a CAN bus connection point between the dashboard and the ECU, we connected the CAN gateway to this point, as it can be seen in Figure 5.12.

The attack we performed was to overwrite the tachometer value with a constant, and thus force the dash to show our engine rpm instead of the real one. The attack parameters were the following:

- Bitrate: **250kbps**
- ID: **0x208**
- Offset: **0**
- AttackLength: **1**
- ByteValue: **0x30**
- AttackType: **REPLACE_DATA_WITH_CONSTANT_VALUES**

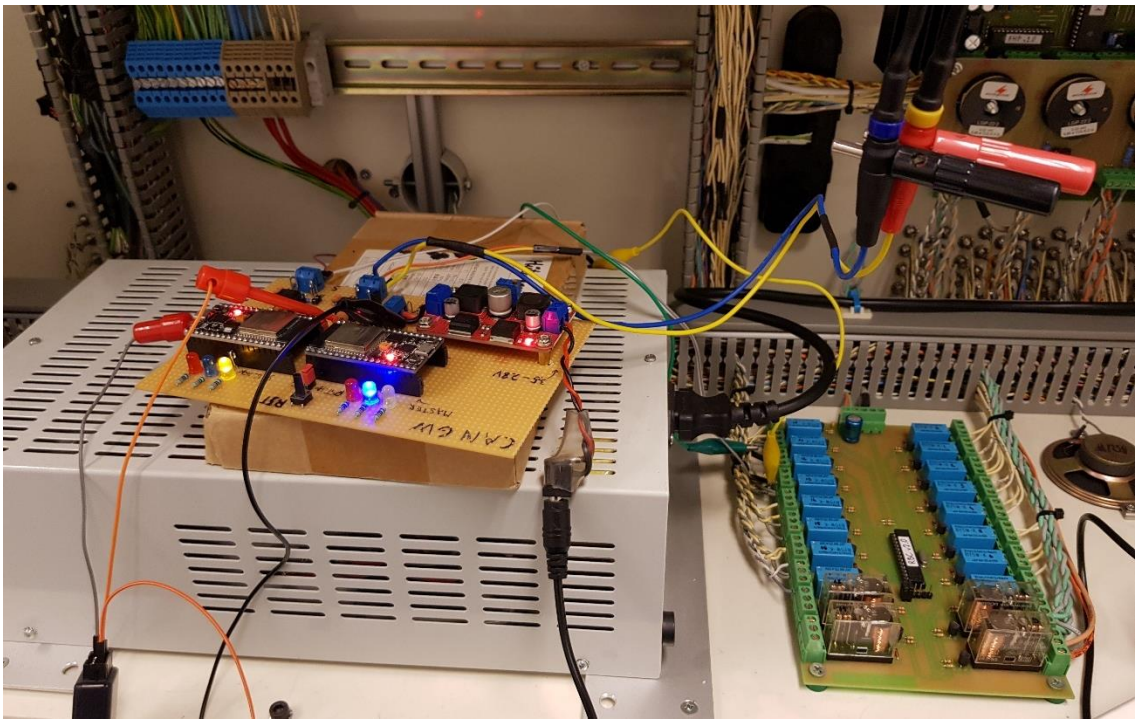


Figure 5.12 - The CAN gateway connected between the ECU and the dashboard

As you can see in Figure 5.13, despite the engine idling at 810rpm, the dashboard shows the modified values of around 1500rpm.

Figure 5.13 - The attack of the tachometer displays different rpm than the real one

The message modification at the CAN gateway can be seen in Figure 5.15.

Figure 5.14 – The attack of the gas pedal angle sensor, setting a constant value

Figure 5.15 - The attack of the tachometer setting a constant 1500 rpm

After performing a constant value replacement attack on the tachometer, we tried something even more spectacular, replacing the tachometer value with an increasing counter. The attack parameters were the following:

- Bitrate: **250kbps**
- ID: **0x208**
- Offset: **1**
- AttackLength: **2**
- AttackType: **REPLACE_DATA_WITH_INCREASING_COUNTER**

While you can sort of see in Figure 5.16 that the tachometer gauge is blurred from the motion, we created a few videos during both of the attacks, available on youtube[4][5][6], which show the tachometer's behaviour in motion.

Figure 5.16 - The tachometer value replacement with an increasing counter attack

Now, at last, with a successfully working CAN gateway at hand, let us conclude our work in the following section.

---

[4] Constant RPM attack: https://youtu.be/2lDvY7b4cak (last accessed: 2020.12.10.)

[5] Increasing RPM attack: https://youtu.be/Z179lHE-gBI (last accessed: 2020.12.05.)

[6] Increasing RPM attack: https://youtu.be/ItWnO_L-4rQ (last accessed: 2020.12.10.)

6 Conclusion

In summary, the CAN bus, which is one of the most common communication solutions for ECUs, has several security flaws, since security was not in focus during its creation. There are no message authentication measures; the ECUs decide to act upon a message or not based only on the CAN ID field of the message. As of this, it is possible to inject fake messages, or modify existing messages on the CAN, and hence, to force some ECUs to act upon these fake messages, which may influence the vehicle's overall behaviour.

While message injection attacks are easy to implement, they provide several side effects, making them trivial to detect. On the other hand, message modification attacks are hard to realize and require a more profound knowledge of the field, but they are much less detectable. In this work, our goal was to design and implement a device, capable of modifying ISO 11898 high-speed CAN messages in real-time. We presented two solutions: a malicious CAN node, and a CAN gateway.

The malicious CAN node was developed based on the premise that according to the CAN standard, the TTC messages shall not be retransmitted automatically in case an error occurs during transmission. Our goal was to create a device that would break the targeted messages on purpose during transmission, and retransmit the message with our values instead. While we managed to create such a device using an Arty A7 FPGA, during the testing, we found that the manufacturers of the tested ECUs were not following the standard word-by-word, and the ECUs retransmit the TTC messages too. We cannot say for sure whether this attack could work with other ECUs, but it did not yield the expected outcome in our case.

The second solution, the CAN gateway, was designed to perform a Man-in-the-Middle attack by separating the targeted ECU and the rest of the CAN bus; and wedging the CAN gateway between them. By using this solution, we can modify the content of any message passing through the CAN gateway without any excess message, or increase in the busload. The created device is capable of modifying ISO 11898 high-speed CAN messages in real-time with a minuscule introduced delay of ~260us, without being detectable by current measures. It can handle up to 100% bus load with a bus speed of 500kbps or less, and about 60-100% busload at 1Mbps. The device is built from cheap, commonly available

parts, and it provides a wireless interface that can be used to remotely configure the attack parameters, which makes it easy to use during laboratory works in the future.

Future work

In case of future development, we have several ideas on how to improve the device.

- First of all, the second microcontroller could be eliminated from the system, as it is mostly a by-product of the changing architecture during the rapid development.
- Also, a PCB could be designed and produced to increase the reliability of the device, and a 3D printed housing could be created for it.
- After that, the passthrough mode could be moved to a hardware solution, using two relays and a secondary, direct CAN path. This path could take the traffic until the device is configured; thus, the device could be started-up and restarted independently from the car's internal CAN network.
- Finally, the software could be improved. At the moment, the two sides of the gateway's CAN bus is independent of each other. We could ACK a message on one side, and then not be able to send it on the other side due to high traffic or transient noise. This could result in an overall system malfunction, due to the "lost" messages.

Acknowledgement

I would like to express my sincere gratitude towards András Gazdag and Dr. Levente Buttyán for being enthusiastic and encouraging about the multi-semester project; and being available and helpful whenever I needed them. I also would like to thank the CrySyS lab for the tremendous knowledge I managed to learn during my studies, and the opportunity they gave me. Finally, I would like to thank my friends and my family for helping me and encouraging me during my studies.

The project has been partially supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00002); and by the European Commission via the H2020-ECSEL-2017 project SECREDAS (Grant Agreement no. 783119).

References

[1] A. TAYLOR, N. JAPKOWICZ AND S. LEBLANC," *Frequency-based anomaly detection for the automotive CAN bus," 2015* World Congress on Industrial Control Systems Security (WCICSS), London, 2015, pp. 45-49.

[2] H. M. SONG, H. R. KIM AND H. K. KIM," *Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network"* 2016 International Conference on Information Networking (ICOIN), Kota Kinabalu, 2016, pp. 63-68.

[3] C. MILLER AND C. VALASEK," *Adventures in Automotive Networks and Control Units"*, IOActive Labs Research, Tech. Rep., Aug. 2013. [Online]

[4] E. EVENCHICK," *Hopping On the CAN Bus"*, Black Hat Asia, 2015

[5] K. KOSCHER, A. CZESKIS, F. ROESNER, S. PATEL, T. KOHNO, S. CHECKOWAY, D. MCCOY, B. KANTOR, D. ANDERSON, H. SNACHM, AND S. SAVAGE, *"Experimental security analysis of a modern automobile"*, 2010, pp. 447462.

[6] A. GAZDAG,D. NEUBRANDT,L. BUTTYÁN,Z. SZALAY:*Detection of Injection Attacks in Compressed CAN Traffic Logs*, in: Security and Safety Interplay of Intelligent Software Systems. CSITS 2018, ISSA 2018. Ed. By SECURITY,I. 2. SAFETY INTERPLAY OF INTELLIGENT SOFTWARE SYSTEMS. CSITS 2018, 2019.

[7] C. MILLER AND C. VALASEK, *"Remote exploitation of an unaltered passenger vehicle."* Black Hat USA, 2015 [Online]

[8] VAN HERREWEGE, ANTHONY, DAVE SINGELEE, AND INGRID VERBAUWHEDE. *"CANAuth-a simple, backward compatible broadcast authentication protocol for CAN bus."* ECRYPT 2011.

[9] ARILOU: *"Feasible car cyber defense"*, ESCAR 2010

[10] BERND ELEND, THIERRY WALRANT, GEORG OLMA: "*Securing CAN Communication Efficiently With Minimal System Impact"* NXP, 2020, [Online] available: https://www.nxp.com/docs/en/white-paper/SECURECARTRANA4FS.pdf (last accessed: 2020.12.02.)

[11] ISO 11898-1:2015: *"Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling"*, ISO 2015

[12] ISO 11898-2:2016: *"Road vehicles — Controller area network (CAN) — Part 2: High-speed medium access unit"*, ISO 2016

[13] Microchip MCP2551 High-Speed CAN Transceiver datasheet, available: http://ww1.microchip.com/downloads/en/devicedoc/21667e.pdf (last accessed: 2020.12.02.)

[14] Microchip MCP2515 Stand-Alone CAN Controller with SPI Interface datasheet, available: https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf (last accessed: 2020.12.02.)