University of Tartu

Faculty of Science and Technology

Institute of Technology

Jürgen Laks

Reverse Engineering a Car Immobilizer

Master's thesis (30 ECTS)
Robotics and Computer Engineering

Supervisor:

PhD Arnis Paršovs

Tartu 2023

Abstract

Auto immobilaiseri pöördkonstrueerimine

Autode immobilaiserid on seadmed, mis ennetavad autode varguseid autode immobiliseerimise teel, juhul kui vastav kaart, mida auto omanik endaga kaasas kannab, ei ole auto läheduses. Käesolev töö dokumenteerib samme, mida kasutati Skybrake DD2+ immobilaiseri pöördkonstrueerimiseks, eesmärgiga avastada selles esinevaid turvanõrkuseid. Sealhulgas koguti andmeid, mis liikusid immobilaiseri komponentide vahel, ning analüüsiti kogutud andmeid. Samuti katab töö immobilaiseri püsivara välja lugemise ning analüüsimise protsessi. Käesolev töö toob välja mitmeid antud immobilaiseri haavatavusi ja võimalikke ründevektoreid. Sealhulgas demonstreeritakse viise möödumaks turvakihist, mida immobilaiser pakub.

CERCS: T125 Automatiseerimine, robootika, control engineering; T121 Signaalitöötlus; T170 Elektroonika

Märksõnad: immobilaiserid, pöördkonstrueerimine, püsivaraanalüüs

Reverse Engineering a Car Immobilizer

Car immobilizers are devices that prevent car thefts by immobilizing cars if a specific token, usually carried by the intended driver, is not in proximity to the car. This work documents the steps taken to reverse engineer the Skybrake DD2+ immobilizer in order to discover its security weaknesses. These steps include capturing the data that is transferred between the components of the immobilizer and analyzing the captured data. The firmware image of the immobilizer was dumped and analyzed. This work presents multiple possible attack vectors that this immobilizer is vulnerable to. Most notably this work demonstrates methods for bypassing the layer of security the immobilizer is supposed to provide.

CERCS: T125 Automation, robotics, control engineering; T121 Signal processing; T170 Electronics

Keywords: immobilizers, reverse engineering, firmware analysis

Contents

List of Figures

List of Tables

Acronyms

ASCII – American Standard Code for Information Interchange

CIS – Commonwealth of Independent States

CRC – Cyclic Redundancy Check

EEPROM – Electrically Erasable Programmable Read-Only Memory

EU – European Union

FPGA – Field Programmable Gate Array

GFSK – Gaussian Frequency Shift Keying

GPIO – General Purpose Input and Output

ICSP – In-Circuit Serial Programming

LED – Light Emitting Diode

MOSFET – Metal Oxide Silicon Field Effect Transistor

MSB – Most Significant Byte

PC – Personal Computer

PCB – Printed Circuit Board

RAM – Random Access Memory

RF – Radio Frequency

SDR – Software Defined Radio

SPI – Serial Peripheral Interface

1 Introduction

Car immobilizers are devices that prevent car thefts by immobilizing cars if a specific token, usually carried by the intended driver, is not in proximity to the car. Most car immobilizers work by utilizing a small radio tag that the driver can keep in their pocket. The car can be started only if the tag is in proximity to the receiver that is located inside the car (later referred to as the car module). This additional security measure is useful in cases where a thief gets physical access to the inside of the car or steals the car key but not the tag. Unless the thief has obtained the tag, the thief cannot start the car, regardless of whether they are trying to do so using the original car key or via hot-wiring the car. The immobilizer has to be well hidden inside the car or else it could be bypassed by the thief, just like hot-wiring a car bypasses the ignition key. The immobilizer immobilizes a car by disconnecting either the fuel pump or the engine control unit. In addition to being able to block the car from starting, the immobilizer can also disable the car while the car is running. This would be needed in case the car is hijacked after it has been started.

This thesis will focus on reverse engineering the Skybrake DD2+ immobilizers, which are developed, manufactured and sold by a Latvian company named Autonams. The aim of reverse engineering this immobilizer is to assess its security features or the lack thereof.

1.1 Problem description

As cars can be relatively expensive, car thefts can result in a large monetary loss for the owner. An immobilizer in a car should prevent the theft of the car. While vendors of car immobilizers claim that their products make cars more secure, there is not much research done on how secure the communication protocols used in these systems really are. These devices are essentially black boxes that users have to trust. The specific security measures and protocols for the immobilizers are not documented in the manuals. Security test reports for these devices, if they exist at all, are not publicly available. This thesis aims to verify the existence of the immobilizer's security features the vendor claims it has.

1.2 Aim of this work

The aim of this work is to reverse engineer the Skybrake DD2+ immobilizer in order to analyze the security it provides. The main questions that this work aims to answer are:

1. What data do the car module and the tag transfer over the radio channel?
2. How does the car module authenticate the tag?
3. Is it possible to clone or spoof the tag or the signals transmitted by the tag?

4. Can the signals transmitted by the tag be used to uniquely identify the tag and hence track its holder?

1.3 Methods used

In this research, we used several methods to find answers to the proposed research questions:

1. All publicly available information about these immobilizers was analyzed. This mainly consisted of reading usage and installation manuals of the immobilizer.
2. In order to verify the gathered information, three DD2+ immobilizers were obtained and then bench-tested in a lab environment.
3. In order to reverse engineer the immobilizer's circuitry, the hardware of the immobilizer was disassembled and then the components used in the circuitry were identified. The datasheets for each integrated circuit were studied in order to understand the purpose and capabilities of the components.
4. In order to be able to connect debugging tools to the circuitry, extra connectors were added to the most critical parts of the circuits.
5. Utilizing the added connectors, an attempt was made to dump the memory (i.e., the firmware image).
6. In order to do further analysis, a logic analyzer was used to capture and save the data sent between the microcontroller and the RF transceiver on the circuit board.
7. The captured data, containing the encrypted messages exchanged wirelessly between the car module and the tag, was analyzed in order to find its entropy and any repeating patterns in the data.
8. In order to replay or transmit specially crafted data packets to the immobilizer, a custom hardware setup was built.
9. In order to obtain the firmware image from the tag and the car module, a commercial firmware dumping service was utilized.
10. In order to understand the instructions present in the firmware of the immobilizer, the flash memory dump was disassembled and analyzed.
11. In order to document the meaning of the bytes in the contents of the EEPROM (Electrically Erasable Programmable Read-Only Memory), dynamic and static analysis were performed.
12. The information obtained was analyzed in order to discover vulnerabilities to which the immobilizer is susceptible to.
13. A custom device was built in order to demonstrate the trackability of the Skybrake DD2+ tags and to demonstrate methods for capturing the serial number of an immobilizer kit, which is needed to clone tags.

1.4 Structure of the thesis

This work has been structured as follows. The main body of the thesis starts with Chapter 2, which provides an overview of the Skybrake DD2+ immobilizer. The contents of this chapter are based on publicly available manuals and materials. Chapter 3 “Black-box analysis” focuses on the analysis of the data transferred between the car module and the tag. In Chapter 4 “Memory dumping and analysis” an analysis is done on the firmware image in order to figure out what the microcontroller does internally based on the instructions present in the firmware. In Chapter 5 “Overview of the attack vectors” the feasibility of multiple attacks is discussed.

2 Skybrake DD2+ immobilizer

This chapter describes some background on the Skybrake DD2+ immobilizer and gives an overview of its features.

2.1 Background

The Skybrake DD2+ immobilizers are developed, manufactured, distributed and installed by a Latvian company named Autonams. The development of the Skybrake DD2 immobilizer started in 2002. [4]

The Skybrake brand of products had a difficult start because the first batch of over 700 immobilizers turned out to be defective and had to be recalled. Due to this setback, Autonams lost over 50'000 lats and the reputation of Skybrake decreased significantly amongst users. Still, the company had the courage to continue with the Skybrake series of immobilizers, which led to the development of the Skybrake DD2+ immobilizer. [5]

In 2004, Autonams started exporting the Skybrake immobilizers to EU (European Union) and CIS (Commonwealth of Independent States) countries [4]. As the products were exported to nearby countries, the scale of manufacturing increased. More than 10'000 Skybrake immobilizers were produced in 2005, over 32'000 in 2006 and over 60'000 in 2007 [6]. In 2008, more than 43'000 Skybrake immobilizers were produced [7].

About 90% of the production was exported to Russia, Belarus, Ukraine, Lithuania and Estonia, the remaining 10% were installed on cars in Latvia [6]. In 2008, Autonams started exporting Skybrake immobilizers to France, Spain, Great Britain and Poland [7]. As a result, in 2008, Skybrake DD2+ reached the final of the "Export and Innovation Prize" organized by the Ministry of Economy of Latvia and LIAA [4]. In 2010, Autonams started exporting Skybrake DD2+ immobilizers to Croatia [8].

By 2010, Skybrake DD2+ had increased the market share of the new generation of immobilizers in Russia to 10% and had thus become the fourth most purchased 2.4 GHz immobilizer there [4,8]. By that time, a lot of clones of Skybrake immobilizers had appeared on the Russian market. The clones were visually and technologically very similar to those created by Autonams. However, the clones did not have the Double Dialogue wireless data transmission solutions that the Latvian product had. [9]

In addition to securing cars, Autonams started offering the opportunity to also equip motorcycles, tricycles and quadricycles with an updated version of the Skybrake DD2+ immobilizer [10]. In 2012, Ukrainian Toyota and Lexus dealers started installing Skybrake DD2+ immobilizers to all of their clients' cars [11].

In 2018, a newer version of the immobilizer called Skybrake DD5 was released [4]. Today the Skybrake DD2 and DD2+ immobilizers are not manufactured nor sold anymore, however, we can assume that there are still cars that have it installed, since these immobilizers have been installed on thousands of cars over a period of about 15 years.

2.2 Skybrake DD2+ package contents

The Skybrake DD2+ immobilizer product is sold packaged in a cardboard box. Its contents can be seen in Figure 2.1. The box includes a car module (on the left) and two tags (in the top-right corner). The package also includes a buzzer and a service card that lists the product's serial number, PIN-code and service code. The PIN-code of an unopened kit is covered and needs to be scraped in order to be revealed.

Figure 2.1: Contents of the original box of the Skybrake DD2+ kit [1]

2.3 Specifications and features

There are numerous materials available on the internet containing information about the Skybrake DD2+ immobilizer. This section describes the features and specifications of the Skybrake DD2+ immobilizer listed by some of these materials.

2.3.1 Basic working principle

When the ignition switch of a car is turned on, power is supplied to the car module and the car module enters the activation phase. In the activation phase, the car's engine can be started (car is not immobilized) and the car module is waiting for a signal from a tag. If a signal from a tag is received, the car's engine will be kept running and the immobilizer enters the authenticated phase. In case a signal from a tag is not received within 18 seconds since the car module was powered up, the car module enters the warning phase. [2]

In the warning phase, the car's engine will be kept running (car is not immobilized). In this phase, sound signals are emitted from the buzzer that is connected to the car module, warning the driver of the vehicle about the absence of a tag within receiving range. If a signal from a tag is received, the car's engine will be kept running and the immobilizer enters the authenticated

phase. In case a signal from a tag is not received within about 1.5 minutes, the immobilizer enters the panic phase. [2]

In the panic phase the car is immobilized, which would result in the engine stalling if it was running. In this phase, uninterrupted series of sound signals are emitted from the buzzer, notifying the driver of the vehicle that the car is immobilized due to no tags being within receiving range. If a signal from a tag is received, the car's engine can be started again (car is not immobilized) and the immobilizer enters the authenticated phase. [2]

In the authenticated phase, the car's engine is not immobilized. In this phase, the buzzer will not make any sounds. Depending on the settings of the immobilizer, the car module may still listen for signals from the tag, and if no signals are seen for some time, the immobilizer enters the warning phase.

2.3.2 Connections

Figure 2.2 shows how the car module (designated as the "control unit" in the image) should be connected to a car. The car module needs power (+12V) and ground (chassis) connected in a way that turning the ignition key of the car would power the car module. The car module also needs its "output to buzzer" pin to be connected to a passive buzzer for sound signals to work. Finally, the car module has two wires that are used for engine cut-off (denoted as the "blocking circuit" in Figure 2.2). These wires can be connected, for example, such that the power to the car's fuel pump is passed through this circuit, meaning that when the blocking circuit is open (enabled), the fuel pump will not receive power even if the car attempted to supply power to the fuel pump. [2]

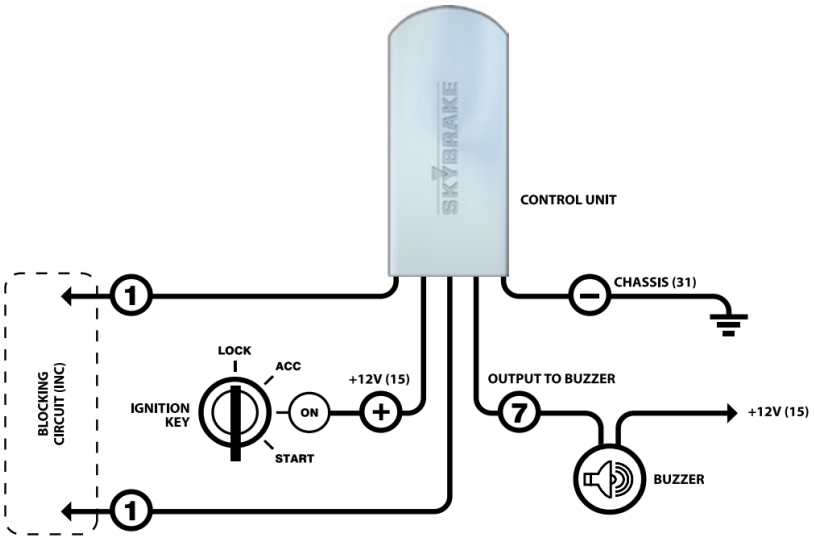

Figure 2.2: The connection diagram provided in the immobilizer's usage manual [2]

2.3.3 Tags

Each immobilizer kit comes with two tags. Figure 2.3 shows each of the three parts of a tag: the plastic case, the circuit board and a CR2430 battery. The circuit board is powered by the CR2430 battery. There is an LED (Light Emitting Diode) on the circuit board that should flash four times when a battery is inserted, indicating that the tag has been powered and is

functioning [2]. The LED should flash again when the car's ignition is turned on, indicating normal radio contact with the car module [2].

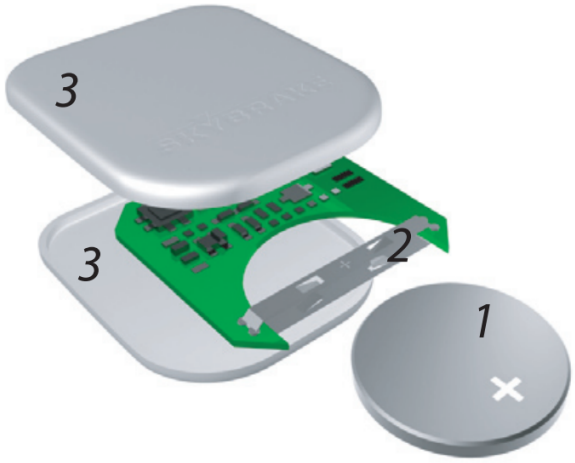

Figure 2.3: Three parts of a tag: the battery (1), the circuit board (2) and the plastic case (3) [3]

The battery in the tag should last for about 12 months on average [3]. Although this immobilizer uses a larger battery than competing products, some users have found the battery life to be less than advertised – between 6 and 12 months on average [12]. When the battery in a tag is low and the ignition key of a car is turned, the buzzer connected to the car module will emit three short beeps to indicate that the battery of the tag needs replacing [2].

In addition to the two tags that come with the immobilizer kit, it is possible to program up to three additional tags to work the car module, so 5 in total [3]. There is a sticker on the circuit board of the tag that lists the serial number of the immobilizer and a number that shows which of the five tags it is.

2.3.4 Anti-theft feature

Previously, we mentioned that if a signal from a tag is received, the blocking circuitry will be kept disabled. In case a car is started with the tag nearby, but then the tag goes out of range of the car module, the car module will enter the warning phase. This is the so-called "anti-theft" feature. In other words, the anti-theft feature continues searching for tags even after a signal from a tag has been successfully received and the car is not immobilized. The anti-theft feature can be turned on and off in the device's settings. Additionally, the exact time between the tag going out of range and the car being immobilized can be configured. [2]

This anti-theft feature however comes with a safety concern. A driver could potentially lose control of the car if the engine of the car would stall at a high speed. Due to the safety concerns, the Skybrake DD2+ anti-theft function is prohibited in the Russian Federation and the EU [2].

2.3.5 Motion detection feature

With the motion detection feature enabled, the car is immobilized only when its speed has been 0 km/h for at least 10 seconds and movement begins again [2]. This feature eliminates the safety concern posed by the anti-theft feature, as a car would never be immobilized while it is moving, but only when it starts moving. The motion detection feature is also useful, for example, during winter when warming up the car and leaving it open with the key in the ignition.

The motion detection is however an optional feature and requires an additional chip on the circuit board of the car module. In case the motion detection feature is available, it can be turned on and off in the device's settings [2]. The motion-detection sensitivity is also configurable [2].

2.3.6 Additional modules

It is possible to install up to three additional radio-controlled engine cut-off modules. These additional modules are made for car modules with specific serial numbers and cannot be attached to other car modules. The technical parameters, wire marking, installation requirements and the connection diagram for additional cut-off modules are similar to the ones for the car module. [2]

2.3.7 Emergency mode

There are cases where the car has to be started without a tag nearby, for example, in case the battery of a tag has drained completely. In order to start a car without a tag nearby, the car module needs to be put into emergency mode. [2]

In order to enter the emergency mode, the PIN-code that came with the immobilizer kit must be entered. The PIN-code can be entered by turning the ignition key in a specified sequence. In order to enter the digit `X`, the user would have to first turn the ignition on, wait for `X` beeps from the buzzer and then turn the ignition back off. This process is repeated for each of the 4 digits in the PIN-code. After the correct PIN-code is entered, the immobilizer is disabled and the car can be started without the tag being present. To start the car again, either the tag has to be present or the PIN-code has to be entered again. [2]

2.3.8 Changing settings and pairing new tags

In order to change settings, certified installers have special equipment and software to perform computer-assisted programming [2]. However, at least some settings can also be changed without special equipment and software by turning the ignition key in a specified sequence. This sequence is similar to the sequence used for entering the PIN-code. [3]

There are ignition key sequences available for changing the following settings: [3]

1. enabling and disabling the anti-theft feature
2. enabling and disabling the motion sensor
3. changing the sensitivity of the motion sensor
4. changing how quickly the engine will be immobilized in case the tag is not detected while starting the car or after the car is started, but the tag disappears (anti-theft feature)
5. pairing additional tags
6. removing paired tags

The PIN-code needs to be entered as part of the ignition key sequence for pairing and removing a tag. This PIN-code is not needed for changing other settings.

2.4 Security claims by the vendor

The immobilizer is advertised to be using "Double Dialogue encryption" [13–15] that "ensures protection against hacking" [16]. Double Dialogue (DD) is defined by the vendor as "a wireless data transmission technology, which enables highly secure, interference-free and electronics friendly exchange of information between several system elements in a car" [17]. Additionally, the vendor claims that "the Double Dialogue coding has low energy consumption and maximum protection against electronic cracking" [17]. According to a representative of Autonams, their algorithm "synchronously changes the frequency and therefore car thieves' code scanners cannot read it" [5]. The Director of Sales and Development of Autonams has said that even though clones have appeared on the Russian market, it is still "impossible to imitate our know-how technology – the Double Dialogue wireless data transmission system and the unique data coding algorithm" [9].

Online sources (including different materials released by the vendor) do not agree on the used frequency range and the number of frequencies used [2, 3, 15, 17]. According to one of the installation manuals released by the vendor, the immobilizer is using 125 frequencies between 2400 MHz and 2480 MHz, data is transmitted at a speed of 1 Mbps using GFSK (Gaussian Frequency Shift Keying) modulation and the radio transmission output power is limited to 1 mW [2].

3 Black-box analysis

This section contains an in-depth analysis on how the Skybrake DD2+ immobilizer functions. The term "black-box" conveys that we do not know what is happening inside the firmware of the immobilizer, but we do have access to the physical device itself – the "black box". We can see which components it is built from, what data it is transmitting and how it responds to received data.

For the analysis, we performed our experiments on three different DD2+ immobilizer kits, the serial numbers of which were 37236, 70251 and 137630. Assuming that the serial numbers started from one and are incremented by one for each device, this shows that there were at least 100'000 DD2+ immobilizers produced, which is confirmed by online sources [6].

The three DD2+ immobilizer kits we analyzed had slightly different hardware, but seemed to behave the same way. Although we did most of our experiments on just one DD2+ immobilizer kit, we verified that the findings also apply to all three DD2+ immobilizers we had gathered for experimentation. An overview of different versions of the immobilizers and how they compare to each other is given in the Appendix.

3.1 Internals of the car module and the tag

In order to get a better understanding of how the Skybrake DD2+ immobilizer works, we took the car module and the tag apart for analysis. Neither the car module nor the tag had any conformal coating nor potting added onto the circuit board in order to make it more robust, water-resistant or to make reverse engineering harder.

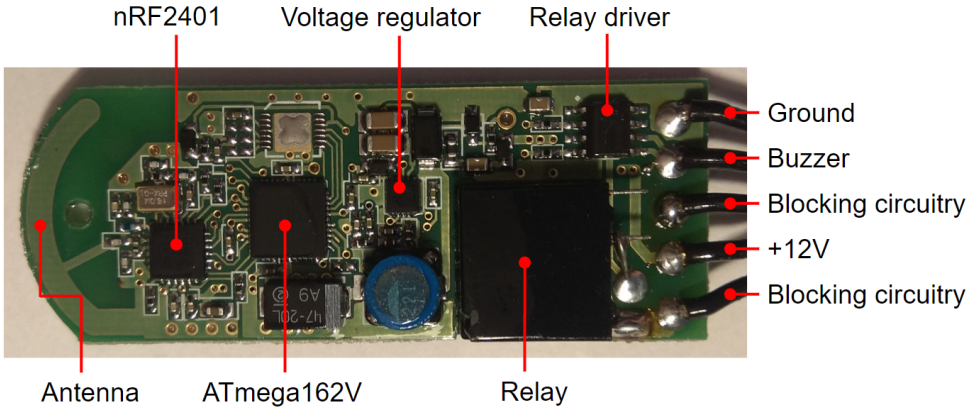

Figure 3.1: Top side of the PCB of the Skybrake DD2+ car module

Figure 3.1 shows the Printed Circuit Board (PCB) of the car module. There are five wires coming out of the PCB on the right side of the image: ground, buzzer, +12V power input and

two wires for the blocking circuit. The purposes of these connections were explained in detail in Section 2.3.2. The components on the PCB are an ATmega162V microcontroller, a relay, a relay driver, an nRF2401 radio transceiver and a switch-mode voltage regulator. The relay is the device that physically disconnects power from components of a car in order to immobilize the car. These components can be, for example, the fuel pump or the starter. We refer to the relay as the blocking circuit. The relay type is normally closed, which means that if the car module is not powered, the blocking circuit is disabled, meaning that the car is not immobilized. The nRF2401 RF (Radio Frequency) transceiver enables radio communication between the car module and the tags. This transceiver handles addressing of data packets [18] to make sure only data packets intended for the specific device get processed. Packets from other devices running near the same 2.4 GHz frequency get ignored and will not be forwarded to the microcontroller. This transceiver also handles the integrity of packets by adding checksums to data when transmitting and checking the checksums when receiving data [18]. The ATmega162V microcontroller runs firmware that handles communications with the tag. The microcontroller also makes decisions on whether to enable or disable the blocking circuit. On the PCB, there is also a footprint added for an external integrated circuit, presumably a motion sensor. This component was not populated on any of the devices we gathered for this research.

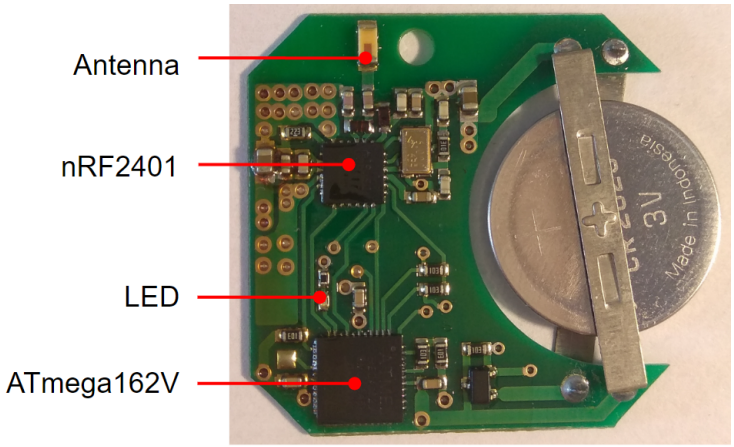

Figure 3.2: Top side of the PCB of the Skybrake DD2+ tag

The tag's circuitry is shown in Figure 3.2. It contains circuitry which is just a subset of the car module's circuitry with an added LED. It contains the same microcontroller and RF integrated circuit. Compared to the car module, the power regulation and load switching parts are not present on the tag. The power to the tag comes directly from a CR2430 button cell battery.

3.2 Breakout boards for experimentation

The most interesting parts of the circuits of the tag and the car module for reverse engineering are the microcontroller's In-Circuit Serial Programming (ICSP) pins and the communication bus used to send data between the microcontroller and the RF transceiver. The ICSP pins allow for (re)programming the microcontroller and thus reading from and writing to different types of memories inside the microcontroller. The communication bus between the microcontroller and the RF transceiver will carry all data that will be transmitted and received by the tag and the car module.

In order to make it possible to connect external analysis tools to the aforementioned pins of the microcontroller, we broke the pins out into external 2.54 mm headers. The headers and the PCB were mounted on a rigid base, which prevented the thin wires from moving and coming loose during testing. The breakout of the tag can be seen in Figure 3.3. At the top of the image, there is a cable going to an ICSP programmer header. The connector on the right breaks out inter-chip communication lines, most notably the clock line (CLK1) and the data line (DATA).

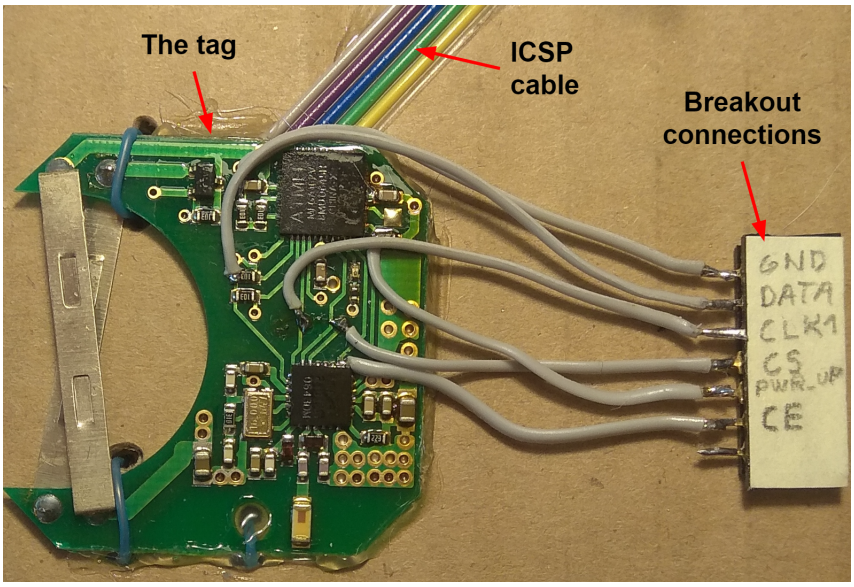

Figure 3.3: Traces on the PCB of the tag broken out into external connections

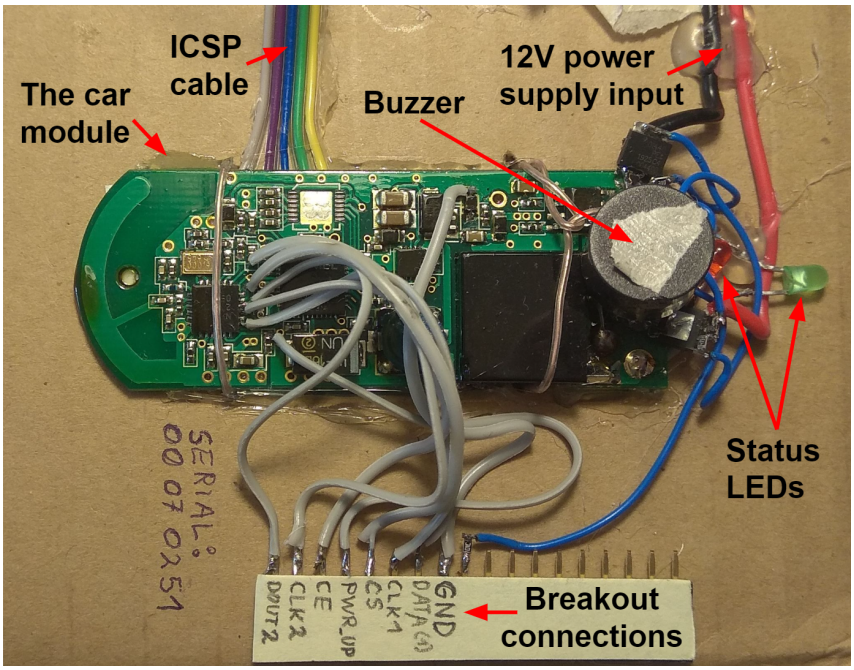

Figure 3.4: Traces on the PCB of the car module broken out into external connections. The connector at the bottom has pins available for sniffing inter-chip communications, the leftmost cable at the top goes to an ICSP programmer.

The breakout for the car module can be seen in Figure 3.4. We added a cable that goes to an ICSP programmer header at the top-left in the image. The red and black wires in the top-right corner were added for supplying power to the car module. The connector at the bottom of the image breaks out the multiple SPI-like (Serial Peripheral Interface) buses for the inter-chip communications and also some other data lines. While most of the connections can be viewed as outputs from the car module, there is one pin that is an input to the car module. It is a pin that, with the help of two additional MOSFETs (Metal Oxide Silicon Field Effect Transistor) we added, can control whether power from the red and black wires gets to the car module or not. This is useful for externally resetting the car module during testing. The breakout for the car module has two other output features. There is a buzzer, which gives audible feedback of the device's state during testing. It is the same buzzer that would be used in a car, connected like the manual suggests, just mounted closer to PCB to keep wires shorter, and covered with tape to decrease its volume. Furthermore, we added two LEDs, red and green, at the rightmost part of the image. The green LED lights up when the tag is authenticated and the immobilizer would let the car start (blocking circuit disabled). The red LED lights up when the car is immobilized, or in other words, when the immobilizer would try to block the car from starting (blocking circuit enabled).

3.3 Analyzing inter-chip communication

In order to capture data flowing from the tag to the car module, we chose to capture the data "on the wire" (data transmitted on the bus between the microcontroller and the RF transceiver) instead of listening for wirelessly transmitted packets. The first reason for this decision is that the wired communication stream carries more information than the wireless data stream. Most notably, it carries the RF configuration data, such as the frequency and the length of the data. The other reason for capturing packets on the wire is ease of filtering. When listening for wireless packets, it would be needed to filter out signals transmitted by all other devices nearby communicating on the 2.4 GHz band. However, when sniffing data from the wired communication stream, the amount of noise from nearby devices is negligible.

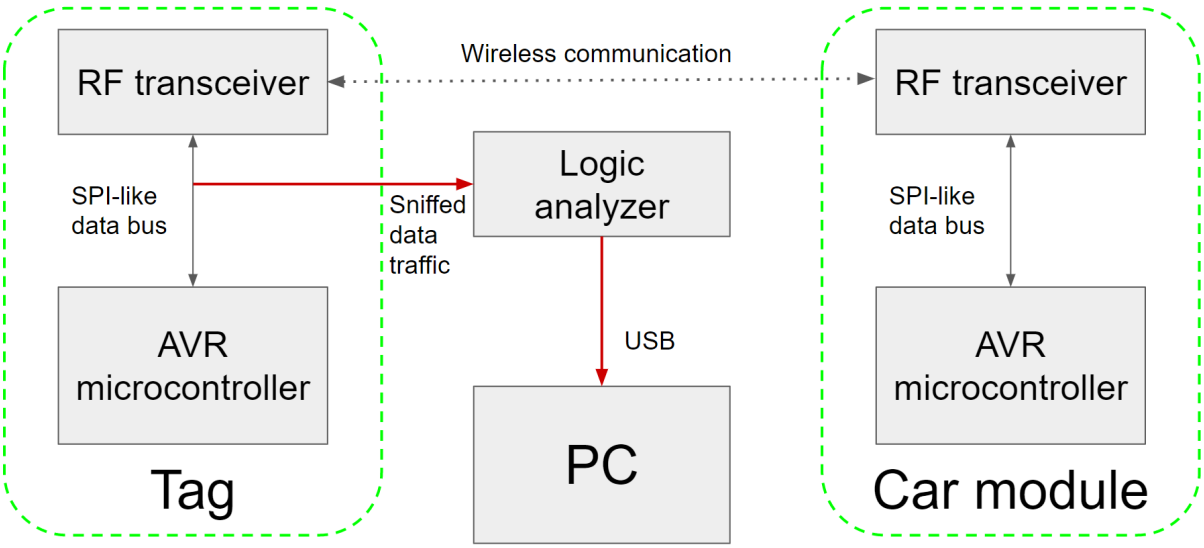

Figure 3.5: Setup for capturing communication data between the tag and the car module

A general overview of the test setup we used for capturing the data flowing between the microcontroller and the RF transceiver can be seen in Figure 3.5. This setup allows saving all data sent between the two components to a computer. This data would include packets that are transmitted from the tag to the car module and the packets that the tag received from the car module. It does not matter whether the data is saved on the tag's side or the car module's side since the same packet data passes though both of them.

3.3.1 High-level summary of the protocol

Based on our observations, experiments and the analysis of the captured data (described later in this work), we were able to provide the following high-level description of the protocol (see Figure 3.6). We are providing this description here, as it will be helpful to better understand the context behind the experiments later in this work.

As soon as power is supplied to the tag (a battery is inserted into the tag), the tag starts transmitting identical data packets *(data a)* in regular intervals – roughly every 10 seconds. We call the packets that are transmitted in regular intervals the "ping packets". The packet transmission itself takes less than a millisecond to complete. These ping packets are always transmitted on two frequencies. The tag transmits each ping packet on 2439 MHz and then the same packet is immediately retransmitted on 2447 MHz.

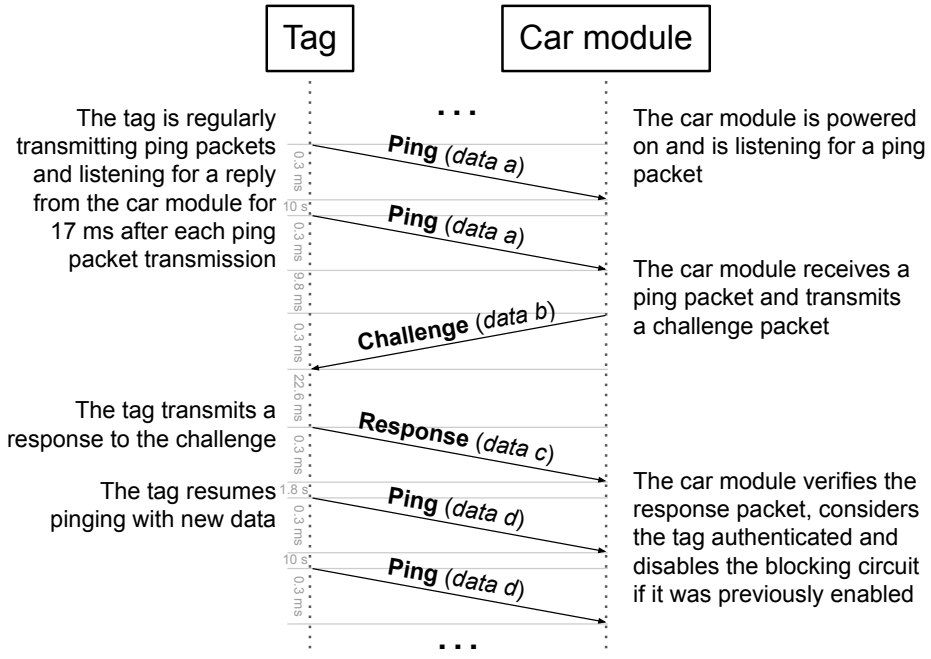

Figure 3.6: Data transmission between the tag and the car module

As soon as the car's ignition key is turned on, the car module gets powered on and it starts to listen for ping packets sent by the tag. Since it might take up to 10 seconds for a ping packet to be received by the car module, the car module decides the initial state of the blocking circuit based on its previous state. If the car module was in the immobilized state (blocking circuit enabled) before being powered off, it now enables the blocking circuit, ensuring the car would not be able to start. If, however, the blocking circuit was disabled before being powered off, the blocking circuit is kept disabled, meaning that the engine can be started immediately even

before the tag is detected. However, in case packets from a tag are not received within about a minute, the blocking circuit is enabled, immobilizing the car.

Once the car module receives a ping packet from a tag, the car module responds with a challenge packet *(data b)*. Immobilizers with lower serial numbers[1] transmit the challenge packet on two different frequencies. Immobilizers with higher serial numbers send the challenge packet on only one frequency, but the same packet is transmitted 11 times in a row. Regardless of the serial number, the frequencies used for sending the challenge packet are seemingly random. However, this frequency is known by the tag as it has to listen on that frequency for the challenge packet.

Once the tag receives this challenge, it calculates a response to the given challenge *(data c)* and transmits it on two different frequencies. The frequencies used by the tag to transmit the response packet are seemingly random. After the tag has transmitted its response packet, the tag resumes transmitting identical ping packets, but these ping packets now contain new data in them *(data d)*. The new ping packets are still transmitted on the same frequencies as before – 2439 MHz and 2447 MHz. The car module verifies the challenge response and if the response is correct, the car module considers the tag authenticated and disables the blocking circuit, if it was not disabled already, allowing the car to be started.

As part of the anti-theft feature, the car module continues listening for ping packets in order to verify the presence of the tag after receiving a valid response, but the car module does not respond to the ping packets with challenge packets anymore. Based on the analysis of the data gathered during experimentation, the car module transmits the challenge packet only after it receives the first packet since being powered on, and also when it receives an outdated ping packet from the tag. There is a specific ping packet corresponding to each possible challenge transmitted by the car module. Any ping packet that does not correspond to the latest challenge packet is considered outdated. If a tag is rebooted (its battery removed and reinserted), the tag does not continue transmitting the same ping packet as before rebooting, but a new one instead. As this is not the ping packet the car module is expecting to receive, this packet is considered outdated and the car module issues a challenge packet even if a tag was previously successfully authenticated. During normal operation, where outdated packets are not present, this means that the tag is forced to solve the challenge every time the car is started. As a test, we left a car module and a tag running for over 7 hours, and we observed that no additional challenge packets were transmitted by the car module during that time.

The car module is powered from the car's battery that is regularly charged, but the tag is only powered from a small button cell battery. Because of this, the power usage of the tag needs to be a lot lower than the power usage of the car module. This constraint also affects the times of transmitting and listening. In general, data transmission takes a lot more power than listening for data. However, since most packets from the tag are transmitted with intervals of multiple seconds and the packets can be transmitted in less than a millisecond, over time the power consumption needed for transmitting data averages out to a lower value than continuously listening for packets. Because of this, with an interval of about 10 seconds, the tag transmits a ping packet and afterwards it only listens for a response for a few milliseconds. If the car module receives the ping packet, the car module will respond immediately (in under 10 ms), thus it is not needed for the tag to listen for longer. This increases the battery life by many orders of magnitude when compared to continuously listening, since the tag does not need to transmit nor listen continuously. The car module, however, listens continuously, but power consumption on that side is not a problem because the battery of the car is significantly larger than the battery of the tag. Furthermore, the battery of the car gets charged while the car is

[1] Serial numbers 70251 and lower based on the devices we were able to test with.

running. To summarize, the tags are noisy, periodically transmitting packets, while the car module is mostly quiet, infrequently transmitting packets while being powered on.

3.3.2 Hardware layer for communications

Both the tags and the car module use nRF2401 RF transceivers for communication. These integrated circuits form the hardware layer (physical layer + data link layer) of communications.

In order to differentiate between wired and wireless data transfers, we use the word "send" to refer to data being sent over a wire and the word "transmit" to refer to data that is transmitted wirelessly.

Physical layer

The nRF2401 RF transceiver has two sets of registers – the configuration registers and the data registers. The configuration registers can be written to in order to change the transceiver's configuration parameters. These parameters include the frequency the transceiver is working on, the address of the packets it should be listening for, the data transmission rate and more. The configuration data is not transmitted, while the data written into the data registers is wirelessly transmitted. [18]

The data lines used by the microcontroller for communicating with the nRF2401 are similar to an SPI bus, but in this case there are multiple chip select lines, called CS and CE. In case CS is high and CE is low, data is clocked into the configuration registers. In case CS is low and CE is high, data is clocked into the data registers. When both of the lines are pulled low, data is wirelessly transmitted from the data registers and then the transceiver starts listening for incoming packets. Data is clocked out on the DATA line on the rising edge of CLK. [18]

Data Link Layer

The nRF2401 transceivers handle the addressing of packets. The transceiver has to be configured with a reception address of 8 to 40 bits (1 to 5 bytes). If the address in a received packet matches the configured reception address, the data in the received packet will be forwarded to the microcontroller. The transceiver will remove the address from the data prior to forwarding the data to the microcontroller. For transmitting a packet, the microcontroller needs to prepend the recipient's address to the payload data when sending the data to the transceiver. [18]

In addition to handling the addressing of packets, the nRF2401 transceivers also handle the integrity of packets. A two-byte CRC (Cyclic Redundancy Check) checksum is appended to the end of a transmitted packet. Once a packet of the length defined in the configuration registers is received, the transceiver verifies the received payload against the CRC. The received data is forwarded to the microcontroller only if the CRC matches. The CRC is removed from the payload before forwarding the payload to the microcontroller. [18]

All the nRF2401 transceivers for an immobilizer kit (both the tags and car module coming from the same box) are configured with almost identical configuration. The reception address for all transceivers is a 5-byte value corresponding to the serial number of the immobilizer kit prefixed with two bytes `0x0707`. For example, the reception address of our immobilizer that had the serial number 70251 was set to `0x070701126B`. The packet size is set to 25 bytes for all packets. This size does not include the address of the recipient nor the CRC, which means that, in reality, 32 bytes will be transmitted for each packet (5 byte address + 25 byte payload + 2 byte CRC).

The only parts of the configuration that change are the transmission frequency and the working mode – whether the transceiver should be receiving or transmitting packets. The nRF2401 transceiver can transmit on 128 different frequencies that are evenly spaced between 2400 MHz and 2527 MHz, each 1 MHz apart from the next [18]. For reception, the transceiver has two channels. The frequency of the first channel can be set between 2400 MHz and 2524 MHz in 1 MHz steps [18]. The frequency of the second channel, if it is used, is exactly 8 MHz higher than the frequency set for the first channel [18].

3.3.3 Communication protocol

A logic analyzer capture of a packet transmission captured from the tag's side is shown in Figure 3.7. The green areas are writes to the configuration registers and red areas are writes to the data registers. The following subsections describe the contents of these areas.

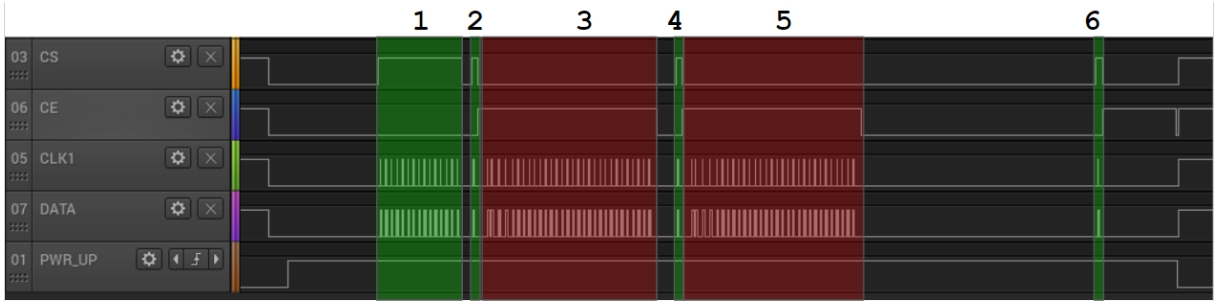

Figure 3.7: Logic analyzer capture of the transmission of a packet, captured on the tag's side. Green areas are writes to configuration registers, red areas are writes to data registers.

Area #1

This area configures the RF transceiver. This configuration is the same every time the tag is powered up or wakes up from sleep (which happens roughly every 10 seconds). The data sent in this area and its meaning can be seen in Table 3.1. Bytes on the wire are sent in the order from top to bottom in Table 3.1, but based on the manufacturer's documentation, they are numbered in reverse (the lowest byte index is sent last) [18].

As Table 3.1 shows, the transceiver is configured so that in listen mode, it would listen for 25-byte packets with the address `0x070701126B` using only a single channel (listening on a single frequency at a time). In this configuration, the transceiver is set up for transmitting a packet on 2439 MHz at a baud rate of 1 Mbps.

Areas #2, #4 and #6

Areas #2 and #4 set the transceiver up for transmitting a ping packet on two different frequencies and area #6 sets the transceiver back into listen mode. These areas only send one configuration byte. In case only one configuration byte is sent to the transceiver, only the first byte in the configuration register is updated, the other bytes are left unchanged. This first configuration byte sets the mode and frequency. The contents of the three areas can be seen in Table 3.2. Each row in the table corresponds to a single configuration byte from a different area.

As it can be seen from Table 3.2, the first two configurations set up the transceiver for transmission and the last configuration turns the RF transceiver back into listen mode. Between each of these configurations (in areas #3 and #5), the tag transmits the ping packet.

Table 3.1: Configuration bytes in Area #1 that prepare the transceiver for transmitting a ping packet

Byte index	Value	Meaning
#15	0xC8	Channel #2 data width 200 bits (25 bytes)
#14	0xC8	Channel #1 data width 200 bits (25 bytes)
#13	0x07	Channel #2 receive address 0x070701126B (1st byte)
#12	0x07	Channel #2 receive address 0x070701126B (2nd byte)
#11	0x01	Channel #2 receive address 0x070701126B (3rd byte)
#10	0x12	Channel #2 receive address 0x070701126B (4th byte)
#09	0x6B	Channel #2 receive address 0x070701126B (5th byte)
#08	0x07	Channel #1 receive address 0x070701126B (1st byte)
#07	0x07	Channel #1 receive address 0x070701126B (2nd byte)
#06	0x01	Channel #1 receive address 0x070701126B (3rd byte)
#05	0x12	Channel #1 receive address 0x070701126B (4th byte)
#04	0x6B	Channel #1 receive address 0x070701126B (5th byte)
#03	0xA3	CRC enabled (16-bit), address width 40 bits
#02	0x6F	RF power 0b11 (max), clock frequency 16 MHz (0b011), baud rate 1 Mbps, ShockBurst, single channel
#01	0x4E	Transmit mode on 2439 MHz

Table 3.2: Configuration bytes in Areas #2, #4 and #6 explained

Area index	Value	Meaning
#02	0x4E	Transmit mode on 2439 MHz
#04	0x5E	Transmit mode on 2447 MHz
#06	0x47	Receive mode on 2435 MHz

As described previously, the nRF2401 transceiver is capable of listening on two different frequencies at the same time, provided these frequencies are exactly 8 MHz apart. In Table 3.2 we see that the two frequencies used for transmitting the ping packet are indeed exactly 8 MHz apart from each other as required by the RF transceiver in listening mode.

Areas #3 and #5

These areas contain ping packet data that is to be transmitted over the air. Both areas send packets that contain the same data. The first five bytes (`0x07, 0x07, 0x01, 0x12, 0x6B`) contain the address of the recipient of the packet. The next 25 bytes (in this case `0x4A, 0xA1, 0xD3, 0x93, 0x8F, 0xFD, 0x01, 0x25, 0x60, 0x17, 0xD1, 0xB2, 0x9C, 0x6D, 0x52, 0xED, 0x78, 0x80, 0xD3, 0xCB, 0x91, 0x6B, 0x02, 0x5E, 0x7A`) contain the payload data. However, so far the meaning of the payload data is not known to us.

Challenge packet

Figure 3.8 shows another logic analyzer capture. This time the car module was turned on and issued a challenge packet. Data areas #1 to #6 are the same as before, but now there was an additional challenge packet received from the car module in area #7.

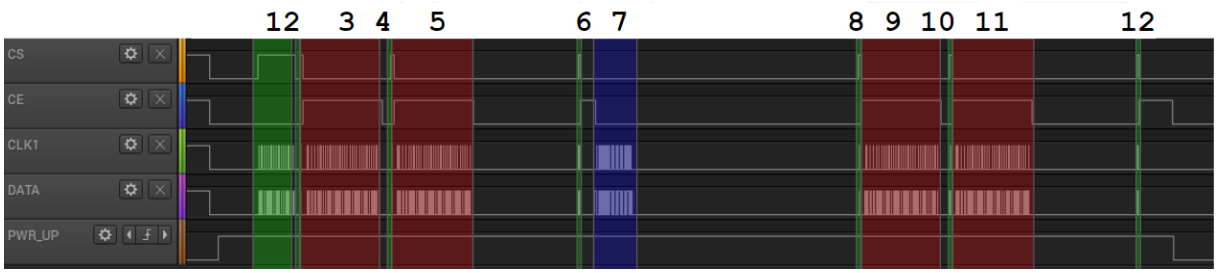

Figure 3.8: Logic analyzer capture of transmitting a packet, receiving a challenge and transmitting a response, captured on the tag's side

In areas #9 and #11, the tag responds with a response to the challenge packet. Both of the response packets contain identical data, but are transmitted on different frequencies. Areas #8 and #10 are once again there for setting up the transceiver for transmission and #12 for setting the transceiver back to listen mode. The exact data for areas #7, #9 and #11 is omitted, since the data does not carry much significance due to our limited understanding of its contents. The communication still follows the same structure: all received and transmitted packets are 25-bytes long, exactly as was seen in the configuration data before.

One thing to note is the frequencies that are used. The tag transmitted the initial ping packet on frequencies 2439 MHz (area #3) and 2447 MHz (area #5) and waited for the car module's challenge on **2435 MHz** (area #7). After responding to the challenge packet, the frequencies used for the ping and challenge packets were 2439 MHz, 2447 MHz and **2471 MHz** respectively. This shows that the car module does not change the frequency it is listening on for ping packets, as the frequencies 2439 MHz and 2447 MHz, which are used by the tags to transmit data, did not change (are fixed). This makes sense as the car module has to be able to receive packets from multiple tags. In other words, if two tags would transmit on two separate frequencies (that are not exactly 8 MHz apart), then the car module would not be capable of listening to packets from both tags. However, the car module transmits each challenge packet

on a different frequency, in this case 2435 MHz and 2471 MHz. Additionally, we noticed that when the tag responds to the challenge, the frequency was also not fixed. This seems to indicate that some kind of frequency hopping scheme is being used, likely for an additional security or reliability measure. Later we verified these findings on more than 3'000 challenge and response packets – the ping packets are always transmitted on the same two frequencies and the frequencies used for challenge and response packets change for each transmission.

3.3.4 Transmitting custom packets

Instead of just looking at the packets exchanged between the car module and the tags, we could gain more insight if we were able to transmit our own packets to the immobilizer and see how it responds. In order to transmit our own packets, we built a device that consisted of an ATtiny817 microcontroller and an nRF24L01+ RF transceiver. This device can be seen in Figure 3.9.

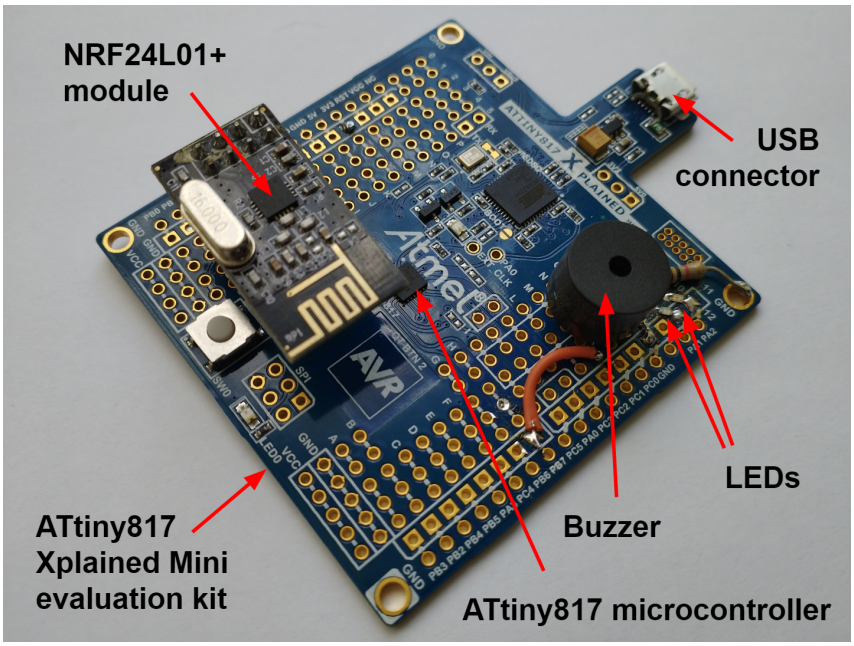

Figure 3.9: The setup used for transmitting custom packets

The nRF24L01+ RF transceiver was chosen for transmitting the packets because it is compatible with the older nRF2401 transceivers, had more configuration options and was easier to obtain in 2022 compared to the nRF2401 transceiver used by the tag and the car module. The functionality of the device we built can be controlled over a USB connection. The device also includes a buzzer and two LEDs. The cost of the components combined to build this device was around 10 euros.

Device firmware capabilities

The firmware we wrote for this device has two modes. The first mode presents a command-line interface over the USB serial interface for controlling the device for experimentation purposes. The second mode is a standalone mode, where the device listens for nearby tags on 2447 MHz and if a packet from a tag is received, the tag's serial number is saved to the EEPROM (Electrically Erasable Programmable Read-Only Memory) of the device. The serial number of the tag is extracted from the address of the received packet.

In the command-line mode, we can load data that is to be transmitted wirelessly. As the transmitted data has a checksum inside it, the device also features a command for recalculating the checksum of the loaded data in case we alter the data during experimentation. The device also features commands for selecting the transmission and reception frequencies. Finally, the device has two commands for transmitting the loaded data. One command transmits the packet immediately, the other will first listen for any incoming packets and will "respond" to it with the loaded data.

The command mode can be used to print out all serial numbers that were collected in the standalone mode. The device's EEPROM memory can be erased in the command mode too.

3.3.5 Testing for replay attacks

With our device, we are able to receive packets from the tags and retransmit them. From previous experiments we know that all sequential ping packets from a tag contain the same data, at least until the tag is forced to respond to a challenge packet or is rebooted. Since the contents of the ping packets do not change, it should be possible to replay them in order to prevent the car module from immobilizing the car. In order to test this simple replay attack, we turned on a car module and a tag. The tag then transmitted a ping packet, responded to the challenge from the car module and continued transmitting ping packets, resulting in the blocking circuit getting disabled. With our device, we captured one of the ping packets, loaded its data for transmission and started transmitting the same packets ourselves. We then removed the battery from the original tag. Based on the facts that the buzzer did not start beeping and the blocking circuit was kept disabled, the car module still believed that a valid tag was nearby because of the packets that we were transmitting. This shows that, once the car is un-immobilized (blocking circuit disabled), a replay attack can prevent the car module from going to the immobilized state (enabling the blocking circuit).

This attack is however very limited because once the car's ignition is turned off and on again, the car module reboots and requires the tag to solve a challenge. As a result, the recorded ping packet would become outdated and would not keep the blocking circuit disabled anymore. The challenge packets we observed previously were always unique, so it would not be possible to record all possible challenge packets and the responses to them for replaying later.

Challenge packets from a car module are only transmitted in response to ping packets from a tag. Could it be that a given specific ping packet is always followed by the same challenge packet? In other words, could it be that the challenge packet is calculated deterministically based on only the data provided by the tag in the ping packet? If this were the case, it would be possible to force the car module into issuing a specific challenge, to which we could record an answer beforehand and replay later, after receiving the challenge.

In order to test this theory, we powered up the tag and the car module. Using the logic analyzer connected to the car module, we recorded the ping packet from the tag, the challenge issued by the car module and the response from the tag. The tag was then powered off and the car module was rebooted. We then used our device to retransmit the same exact ping packet that the original tag transmitted previously. The car module responded with a challenge packet that was not the same as before. This shows that the challenge packets transmitted by the car module are always unique and we cannot force it to issue repeating challenge packets by replaying ping packets. This proves that simple replay attacks do not work for responding to challenges. Since the challenge packets are issued after a reboot of a car module, this also proves that simple replay attacks do not work against this immobilizer if the car module is rebooted.

3.3.6 Fuzzing packets

Using the device we built, it is also possible to get a better understanding of the structure of the data in ping packets. We captured a ping packet that contained the data "6B62A3B537BBA6AD7B64ABB52BA6373B71ACAAE755A1343B78" and transmitted it to the car module. Since the car module had been rebooted after we captured this packet, the car module considered this ping packet outdated and responded with a challenge packet. Even though the packet was considered outdated, it was still valid enough that the car module recognized the packet we transmitted as a ping packet.

For the following experiments, we used the device we built for transmitting packets (see Section 3.3.4). However, since we did not know which frequency to listen on to receive the challenge packets, the challenge packets were sniffed on the wire with the logic analyzer. This ensured that we could capture all challenge packets, no matter which frequencies they were transmitted on.

We then started substituting bytes in the packet with zero bytes one-by-one and transmitting the modified packet to the car module. For example, we transmitted the packet "0062A3B537BBA6AD7B64ABB52BA6373B71ACAAE755A1343B78", to which the car module did not respond, then "6B00A3B537BBA6AD7B64ABB52BA6373B71ACAAE755A1 343B78", to which the car module did not respond, then "6B6200B537BBA6AD7B64ABB52B A6373B71ACAAE755A1343B78", to which the car module responded. This indicates that the value in the third byte is not related to the structure of a ping packet, rather it is some data that is not protected by the integrity measures of the Skybrake DD2+ protocol. Continuing this process, we arrived at a minimal working ping packet: "6B62000000000000000000000000000000000E75500000000". This shows that only four bytes determine the format of a ping packet, the rest is data that can have any value.

We repeated the steps in order to generate more minimal packets. The collection of minimal packets we generated (shown in Figure 3.10) confirmed that the format of a ping packet is determined by just four bytes. Two of the four bytes were always at the beginning of the packet, however the locations of the other two bytes differed between packets.

01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
6B	62	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	E7	55	00	00	00	00
F5	A9	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	D3	00	00	00	00	00	10	00
2F	D4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	2C	1C	00	00	00	00
82	52	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	B7	00	00	1D	00	00	00
2B	1B	00	00	00	00	00	00	00	00	00	00	00	00	00	55	00	00	00	00	00	05	00	00	00
DD	D0	00	00	00	00	00	00	00	00	00	00	00	9D	00	00	00	00	00	00	00	00	00	7A	00
AC	1C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	EB	00	00	00	00	00	7D	00	00
F5	14	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	E1	00	00	00	00	00	EC	00
51	EF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	DD	00	00	00	49	00	00	00
FF	8E	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	6F	93	00	00	00	00

Figure 3.10: A list of ten minimal 25-byte packets with non-zero values highlighted.

We noticed that when we transmit the same minimal packet to the car module multiple times, the car module would always respond with a challenge packet on the same frequency. This shows that the frequency used to transmit the challenge packet is determined by the data contained in the ping packet. Furthermore, the frequency is only determined from the four

bytes of the minimal packet. In other words, the other 21 bytes could be set to any value and the challenge packet would still be sent on the same frequency.

A side effect we noticed while transmitting these minimal packets was that the anti-theft feature seemed to crash. The anti-theft feature is supposed to be checking the presence of valid ping packets from a tag even after a tag is authenticated. In case the car module has authenticated a tag at least once since being powered on, after the car module receives a minimal packet, the car module seems to get stuck in whatever state it is currently in. If the blocking circuit was disabled, it will stay disabled, even if there are no tags nearby anymore. If the buzzer was beeping to indicate that the car will be immobilized soon, it will keep on beeping but will not immobilize the car. This happens even if the minimal packet is based on an outdated ping packet. The anti-theft feature will start to work again only after either the tag or the car module is rebooted.

3.3.7 Collecting protocol messages

In order to get meaningful results from the analysis of the communication protocol based on just data traffic, we need an amount of data that is orders of magnitude larger than what we have collected so far. Since gathering large amounts of data takes time, it was necessary to automate this process.

Methods for creating an automated packet capture setup

The goal of the packet capture setup is simple: capture protocol messages exchanged between a tag and a car module. In order to do this, we want to capture raw data from the SPI-like data bus and decode the data into a readable format. In order to capture the challenge packets too, the car module additionally needs to be forced to transmit those packets. This could be achieved by either transmitting an outdated packet to the car module or by momentarily removing power from the tag or the car module.

Our initial plan was to use the built-in SPI peripheral of an ESP32 microcontroller to read and decode the data on the SPI-like data bus. However, since the SPI-like data bus of the nRF2401 has multiple chip-select pins for different modes and the standard SPI only has one, the SPI peripheral would not receive all packets on the bus.

Our second plan was to connect the GPIO (General Purpose Input and Output) pins of a microcontroller directly to the data bus and use hardware interrupts for detecting changes of the data bus and saving the data. This however failed because the time it took for a hardware interrupt to execute was too slow for capturing the data bus. We tested this with ATtiny817, ESP32 and TM4C123G development boards. For the SPI clock speed of 1 MHz, an interrupt latency of less than 500 ns would be needed. Although the clock speeds for the microcontrollers were well above the clock speed of the bus (for example, 80 MHz was used for the TM4C123G development board), all interrupt latencies achieved were still well above 1 µs. We also tried a variation of this method that used polling instead of interrupts. This method worked semi-successfully as the latency between changes appearing on the data bus and the microcontroller reading the data was reduced. With this method we were able to receive some packets, but other packets would get corrupted. Overall, this method proved to be too unreliable.

Our third plan was to use an FPGA (Field Programmable Gate Array). Using an FPGA, the sampling of an input signal would happen almost instantly after a trigger because there would not be any software context switching needed to handle the interrupt. While this method worked reliably for some packets, the FPGA configuration quickly grew unreasonably complicated

to manage for decoding different types of packets sent to and received from the nRF2401 transceiver.

Our fourth and final plan was to use a logic analyzer for capturing the data on the data bus. A logic analyzer can however only be used to receive data. For manipulating the car module (forcing it to transmit challenge packets), an additional microcontroller was used. This microcontroller disconnected and reconnected power to the car module, making the car module reboot.

We initially tried to avoid using the logic analyzer because it was difficult to interface with. At the time of creating the automated packet capture setup, no reasonable methods existed to programmatically control the logic analyzer we had. The Logic 2 Automation API [19] that we eventually ended up using was released only in September 2022 [20], which was after we had created all the previous packet capture setups.

The functional automated packet capture setup

Our goal is to capture data in a challenge packet transmitted by the car module and the response to the challenge transmitted by the tag in large scale and in an automated manner. We also want to capture the content of the ping packets transmitted by the tag that follow the challenge and response packets. We did not analyze the ping packet preceding the challenge packet because it was shown in Section 3.3.5 that the contents of a challenge packet are not determined solely from the ping packet preceding it.

From previous experiments, it is known that the tags transmit a sequence of identical ping packets until a challenge packet is received from the car module. This means that, before the car module issues a new challenge packet, there is not much variation in the data that could be analyzed. In order to capture these sequences of "challenge $\rightarrow$ response $\rightarrow$ ping" packets, we need to force the car module into issuing a challenge packet. To achieve this, we transmit an outdated ping packet to the car module, that is, a ping packet that differs from the packet that the legitimate tag would transmit. As a result, the car module transmits a challenge packet. Note that the legitimate tag will not receive this challenge packet, as most of the time the tag is asleep and only wakes up for a few milliseconds roughly every 10 seconds, missing the challenge packet, and will thus not respond to the challenge. Our setup, the issuer of the outdated ping packet, also does not respond to the challenge packet.

The tag will finally wake up and transmit a ping packet. However, this ping packet will be considered outdated by the car module because the car module had transmitted a new challenge packet earlier. As a result, the car module will issue a new challenge packet that the tag then responds to. After this, the tag starts transmitting new ping packets. This is the moment when we can gather the sequences of "challenge $\rightarrow$ response $\rightarrow$ ping packets".

For collecting bulk data, we improved the setup described in Figure 3.5 by connecting the logic analyzer to the SPI-like bus on the car module instead of the tag. The data bus for the car module carries data for packets heard on two channels, as opposed to the tag's one channel. In order to transmit outdated packets to the car module, which would force the car module to transmit challenge packets, we connected the same hardware that we used previously for transmitting custom packets (an ATtiny817 microcontroller and an nRF24L01+ RF transceiver) to the computer. Figure 3.11 shows a general overview of the automated capture setup that was used for gathering data. A photo of the actual setup can be seen in Figure 3.12.

The software side of this setup heavily relied on the Logic 2 Automation API [19], which allowed controlling the Saleae Logic 2 software through a Python interface, which in turn communicated with the logic analyzer. The used Python script did the following:

- Configured the Logic 2 software for use with the logic analyzer

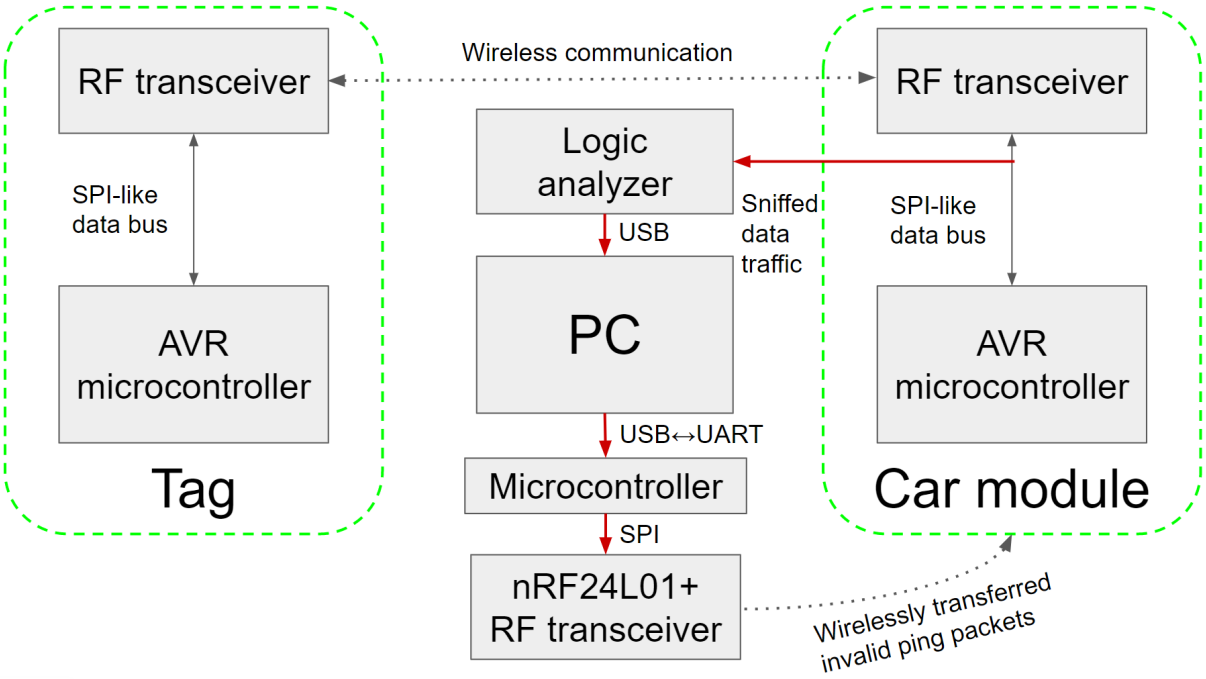

Figure 3.11: Overview of the automated packet capture setup

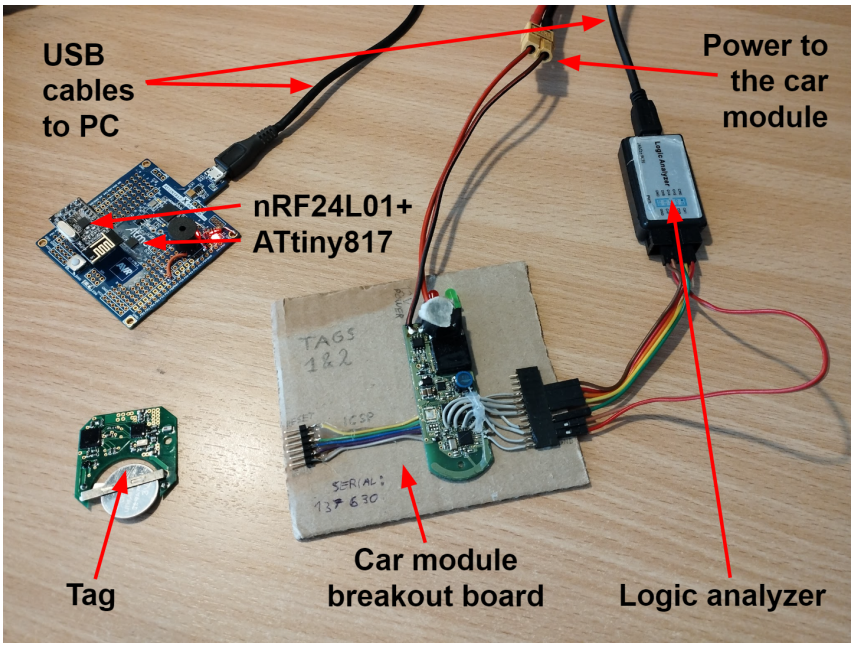

Figure 3.12: Annotated photo of the automated packet capture setup we used

- Configured names for all input channels
- Configured the sample rate to 8 MHz
- Configured 3 different SPI protocol decoders (for RX, TX and configuration data)
- Configured the capture start and stop conditions (trigger channel and 15-second capture length)

- Started the logic analyzer capture
- Initiated the transmission of an outdated ping packet
- Waited for the capture to trigger and finish capturing
- Exported the captured and decoded SPI data into a .csv file
- Reformatted the .csv file into a text file, where every line is a separate packet

This setup was left running overnight. In about 30 hours, it captured roughly 36'000 sequences of "challenge → response → ping" packets. We only captured the sequences that were initiated by the tag, or in other words, the challenge packets, that immediately followed the outdated packets we transmitted, were discarded.

Because some packets were transmitted multiple times in a row, such as the ping packets with identical packet contents, there were a lot of repeated lines in the file generated in the previous step. Because of this, further data cleaning was performed by removing all repeated lines and the repeating packet destination addresses. Only the 25-byte long packet contents were kept for further analysis. Finally, the captured data was split into three different files: one for challenge packets, one for response packets and one for ping packets.

3.3.8 Entropy analysis

Now that there is a sufficient amount of data collected, analysis can be done on each generated file. The product advertises that it uses "double dialogue encryption" [13–15] that was specially developed for this product [16]. Although this is likely just an advertising term, if it really is a good encryption scheme, packets encrypted with it should look random [21]. We measured the probability of different byte values appearing in the data. For the following analysis, some random data was downloaded from www.random.org and this data was compared to the data in the files generated in the previous steps. Each of the files for comparison contained about 90 KB of data.

Using a Linux tool called `ent`, statistics were calculated for each of the files. The results can be seen in Table 3.3.

Table 3.3: Entropy analysis results

Parameter	random.org	challenge packets	response packets	ping packets
Entropy (bits per byte)	7.997862	7.977662	7.997412	7.996693
Chi-Square distribution	272.64	2864.65	322.88	422.58

The theoretically maximum entropy value is 8 bits per byte. The entropy values for data in all files was close to 8, but entropy for the data in the challenge packets was noticeably lower than for response and ping packets (see Table 3.3).

Another method of determining whether some data is encrypted is to find the Chi-Square distribution value of the data. Truly random data should have a value of around 224 and anything higher than 490 may be questioned for randomness [21]. The results shown in Table 3.3 confirm that the data in the challenge packets is likely not encrypted well.

We also analyzed the entropy for each byte in the transmitted challenge packets. Each point on an entropy graph for truly random data should have a value close to the maximum possible

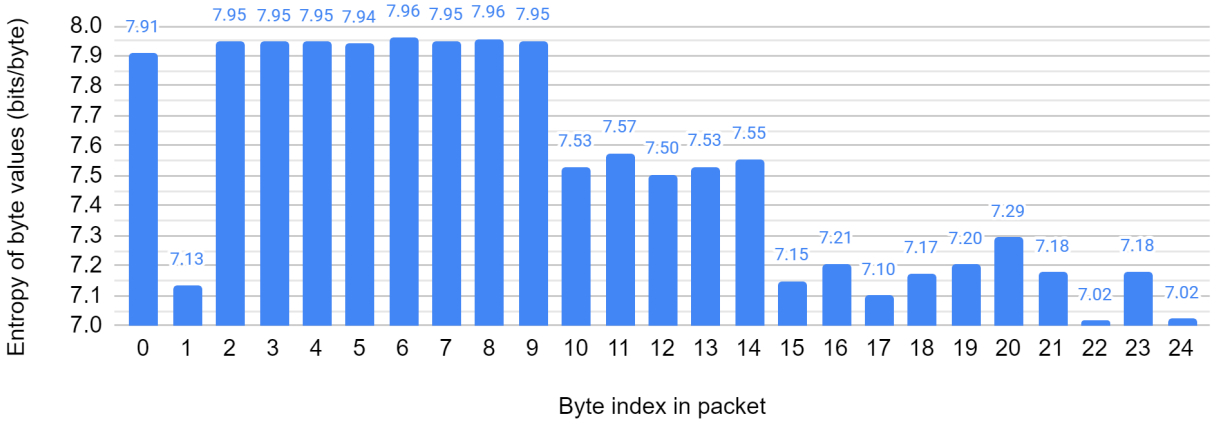

Figure 3.13: The entropy of values for each byte position in challenge packets

value (8 bits/byte), which was observed for almost all captured files. The only entropy graph where some points were significantly lower than 8 was of the data in challenge packets, which is shown in Figure 3.13. This graph shows that the second byte (with index 1) in the packet has a significantly lower entropy than the first and following bytes. It is also visible that the 8 bytes following the second byte have a high entropy, after which the entropy falls again.

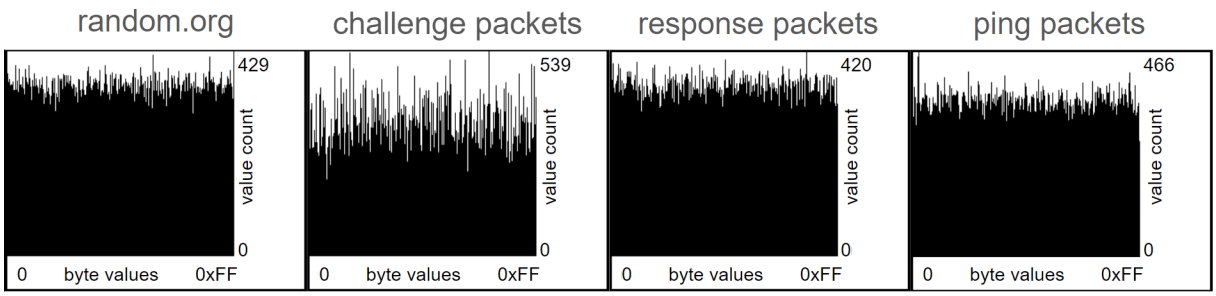

Figure 3.14: Histograms of true randomness and captured data. Byte value on the X-axis, byte count on the Y-axis.

Figure 3.14 shows four histograms, one for data from random.org and one for each file of the captured packet data. What we observe is that while the data in response and ping packets seem similar to the data from random.org, the challenge packet data has a lot more variation in how frequently each byte occurs. Although this change is a bit harder to see, bytes with values 5 and 6 in ping packets occur much more frequently than other values. There is a clear tendency towards these values and this also does not look truly random.

3.3.9 Dotplot analysis

The dotplot can be used to detect repeating patterns in data. A visualization of how it works can be seen in Figure 3.15. In this example, the input sequence is given in the upper row. Every new row has the same sequence, but shifted by one cell. For each cell, if the value in the cell is equal to the topmost value above the cell, the cell is painted white. Otherwise, it is colored black. In this case, it reveals that the sequence “BCDEF” is repeated.

Looking at the plots of true randomness and the captured data (see Figure 3.16), it is visible that there are some repeating patterns in the challenge and ping packets. There did not seem to

Figure 3.15: Example of how dotplots visualize repeating patterns

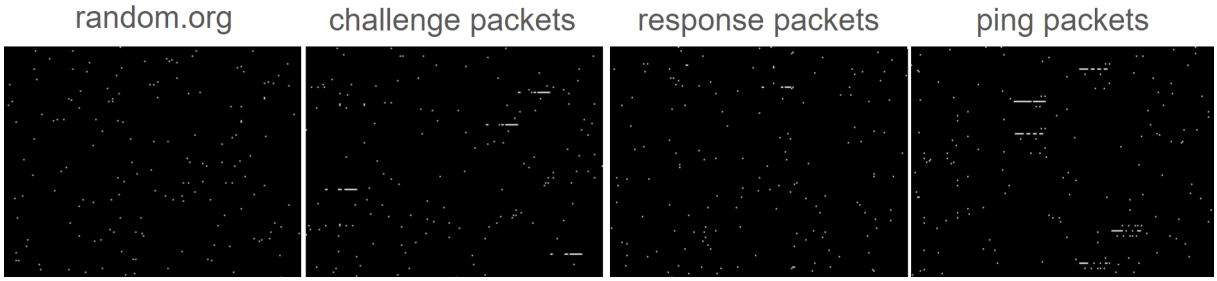

Figure 3.16: Dotplots depicting repeating patterns for true randomness and captured data

be many repeating patterns in response packets. The dotplots here once again indicate that true encryption is likely not used in the communications.

3.3.10 Channel hopping pattern

Previously, we saw that the frequencies used for transmitting ping packets are always the same, but the frequencies used for transmitting challenge and response packets changed with each transmission. In order to investigate the frequency hopping patterns, we analyzed 3'709 captures of challenge, response and ping packets.

Figure 3.17 shows the distribution of frequencies used for transmitting challenge packets. The frequency usage is not uniform. For example, in this dataset there were 59 occurrences of a challenge packet being transmitted on 2446 MHz, while there were no occurrences of a challenge packet being transmitted on 2447 MHz. Figure 3.18 shows the distribution of frequencies used for transmitting the response packets, which is also not uniform. There did not seem to be any repeating sequences of frequencies used for channel hopping in the captured

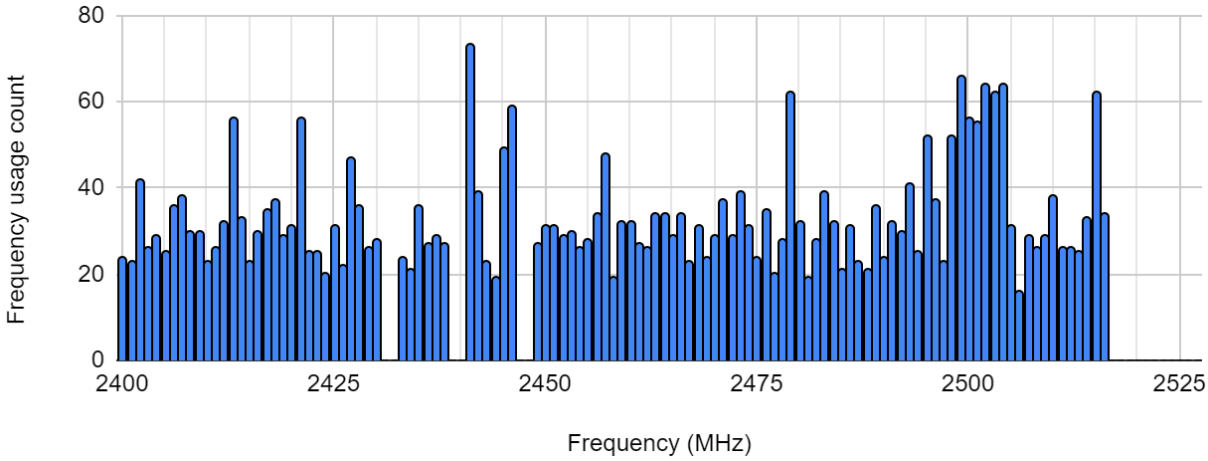

Figure 3.17: Frequency distribution of 3709 challenge packets

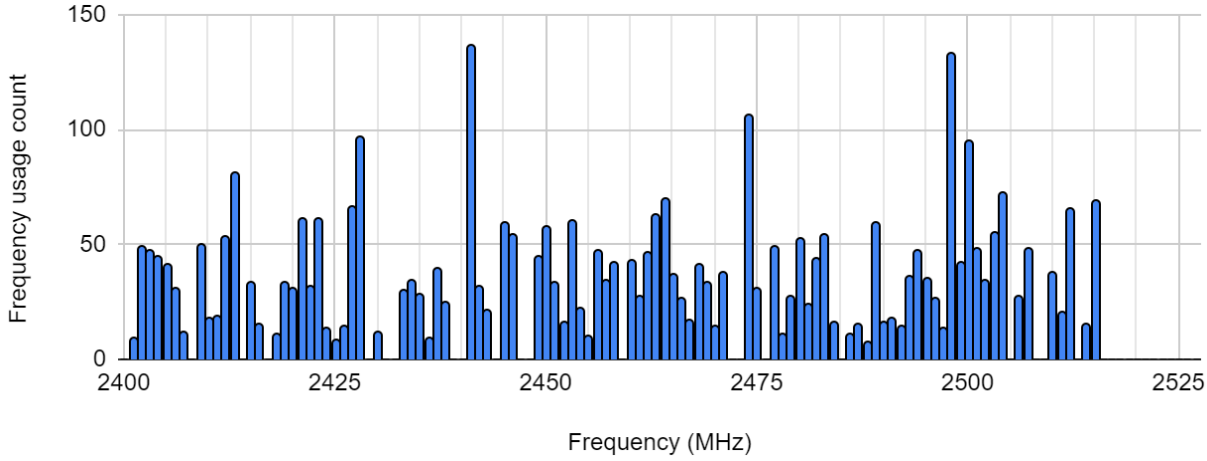

Figure 3.18: Frequency distribution of 3709 response packets

dataset. The used frequency range for all packets was from 2400 MHz to 2516 MHz, a part of which is outside the frequency range (from 2400 MHz to 2480 MHz) declared in the installation manual [2]. In total, 112 different frequencies were used for transmitting packets, which is less than the 125 frequencies stated in the manual [2]. In the dataset we see that in order to listen for ping packets, channel #1 of the car module's RF transceiver was always configured to the frequency 2439 MHz. We also see that challenge and response packets were never transmitted on the frequencies where ping packets were transmitted (2439 MHz and 2447 MHz).

3.3.11 Conclusion of the black-box analysis

As there are many patterns visible in different parts of the data packets, it is highly unlikely that there is some secure encryption scheme used. With true encryption, cases where most of the data in a packet repeats should occur rarely. However, in the collected dataset, repetitions were common. The amount of obfuscation in this protocol however is high. This means that reverse engineering the meaning of data contained in the packets from the captured data alone would be difficult.

4 Memory dumping and analysis

So far, the immobilizer was analyzed as a black box, where only its inputs could be controlled and outputs could be observed. While this method can provide clues about the behavior of the microcontroller, it does not reveal what exactly the microcontroller is doing.

Microcontrollers operate by running firmware that is located in some memory. For the ATmega162V microcontroller, the firmware is stored in the flash memory that is built into the microcontroller. Having a copy of the firmware would let us view which exact instructions the microcontroller is executing. This could give a valuable insight into what the microcontroller is doing with received packets and how it generates the contents of the transmitted packets. This chapter focuses on extracting the firmware from the microcontroller and analyzing it.

4.1 Microcontroller fuse bits

AVR microcontrollers, such as the ATmega162V that is located in the tag and the car module, have non-volatile memory called fuses that define the hardware configuration of the microcontroller. These fuses define hardware settings such as the clock type, startup times, watchdog timer settings and memory access permissions. [22]

The ATmega162V microcontroller uses SPI Serial Programming, also known as ICSP (In-Circuit Serial Programming), for reading and writing the EEPROM, flash and fuses in the microcontroller [23]. In order to read the contents of these memories, we broke out the ICSP pins into a header (see Section 3.2), after which the microcontroller could be connected to a computer through an ICSP programmer. For this purpose we used the USBtinyISP [24] ICSP programmer.

The values of these fuses for the car module and the tag were read out using the `avrdude` software. The exact command used for reading the fuses was

```
avrdude -c usbtiny -p m162 -U signature:r:-:i -v.
```

4.1.1 Fuse values of the tag

Figure 4.1 shows the console output of the avrdude command that was used for reading the fuse values. The fuse values that were read out were `0x42` for the Low Fuse Byte (lfuse), `0xCF` for the Fuse High Byte (hfuse) and `0xFF` for the Extended Fuse Byte (efuse). These fuse bytes are described in detail in the datasheet for the microcontroller [23]. Additionally, for easier comprehension of the fuse values, an online AVR fuse calculator [25] was used.

When looking at the differences between the microcontroller's default fuse values and the fuse values used in the tag, we see the following modifications done by the developers of the immobilizer:

- Startup time of the microcontroller has been made shorter by 65 milliseconds

- Watchdog timer has been enabled
- JTAG interface has been disabled
- Boot flash section size has been changed from 1024 to 128 words
- Boot start address has been changed from `0x1C00` to `0x1F80`
- Memory lock bits have been set.

```
avrdude: Using SCK period of 10 usec
avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.05s

avrdude: Device signature = 0x1e9404
avrdude: safemode: lfuse reads as 42
avrdude: safemode: hfuse reads as CF
avrdude: safemode: efuse reads as FF
avrdude: current erase-rewrite cycle count is -117834497 (if being tracked)
avrdude: reading signature memory:

Reading | ################################################## | 100% 0.05s

avrdude: writing output file "<stdout>"
:030000001E940447
:00000001FF

avrdude: safemode: lfuse reads as 42
avrdude: safemode: hfuse reads as CF
avrdude: safemode: efuse reads as FF
avrdude: safemode: Fuses OK

avrdude done.  Thank you.
```

Figure 4.1: Avrdude output showing the fuse values of a tag

The datasheet of the microcontroller explains that, with the lock bits being set, further programming and verification of the flash and EEPROM is disabled in Parallel and SPI/JTAG Serial Programming mode [23]. Verification here also includes the ability to read out the contents of EEPROM and flash. It is possible to reset the lock bits though the ICSP interface, making the memories readable again, but this process would also erase the contents of the memories. Having the ability to read the memory contents would not allow us to read the firmware as the firmware would be erased from the memory prior to us reading it.

4.1.2 Fuse values of the car module

The fuse values for the car module were read out using the same methods that were used for reading the values from the tag. The values that were read out were `0xE2` for the Low Fuse Byte (lfuse), `0xCF` for the Fuse High Byte (hfuse) and `0xFB` for the Extended Fuse Byte (efuse).

Looking at the differences between the microcontroller's default fuse values and the fuse values used in the car module, most of the differences were the same ones as the differences listed for the tag. The only additional fuse bit modifications we observed were the following:

- Brown-out detection level has been changed to 2.7 volts (disabled by default)
- Clock signal is divided by 1 (divided by 8 by default)

The brown-out detection feature immediately powers down the microcontroller completely, if the microcontroller's supply voltage goes below a certain threshold [23]. Essentially, it puts the microcontroller's core into a safe state if the voltage should fluctuate. While this feature was likely enabled to improve the reliability of the immobilizer, this change also makes power glitching attacks harder to perform, although not impossible [26] [27].

In contrast to the tag, the microcontroller of the car module sets a faster clock speed which provides increased computational performance. The clock rate is not increased for the tag, likely because a faster clock speed increases the power consumption which would negatively affect the battery life of the tag.

4.1.3 Conclusion

The firmware in flash memory and data in EEPROM that are stored in the microcontrollers of the tag and the car module are protected sufficiently well against reading them. All fuse bits that protect EEPROM and flash memory readout are set. The brown-out detection is also enabled for the car module, which would make glitching attacks harder to perform.

4.2 Methods for dumping firmware

There are multiple ways of dumping firmware from a device. Vasile et al. [28] have categorized firmware dumping techniques into three categories.

The first one is leveraging debug interfaces, such as the ICSP interface, in order to read memory contents. However, this method cannot be used in this case because memory readout protection in the microcontrollers is enabled, as was confirmed by the lock bits in the fuses being set.

The second category of methods is by using software methods to gain access to firmware, such as firmware updates, network services, etc. Unfortunately, this method does not apply for the immobilizer because it supports neither automatic over-the-air firmware updates nor even manual firmware updates by the user. The immobilizer also does not have a command-line interface for controlling it.

The third category of methods is performing a flash chip hardware memory dump. This is relatively easy to accomplish if the memory device is separate from the CPU. However, the ATmega162V combines different memories, CPU and numerous peripherals all in one package. This makes it unfeasible to connect wires between the memory and the CPU.

Another method of extracting memory proposed in [29] is using UV attacks for disabling the memory protection fuses. In order to achieve this, the microcontroller would first need to be decapsulated. Then the location of the fuse bit in the silicon die has to be determined. Once the location of the bit is known, a UV laser can be used to reset the given fuse bit to its default state. Once the memory protection fuse bits have been reset, the memory contents can be read out using debug interfaces, such as the ICSP interface.

4.2.1 Firmware dumping as a service

Although the theory of performing UV attacks is relatively simple, actually performing these attacks on microcontrollers is not. This process is time-consuming, needs special equipment, needs researching the microcontroller model in detail and is eventually prone to failure, thus requiring multiple samples of the microcontroller for the attacks to succeed.

There exist some companies that have specialized in reverse engineering microcontrollers. They have research facilities dedicated to researching different microcontroller models in order to verify whether different attacks work against a specific microcontroller model.

In order to get a firmware dump for the car module and the tag, we approached one such company in Shenzhen, China – Circuit Engineering Co., Ltd [30]. The intended process of obtaining a firmware dump of a microcontroller begins by the customer mailing the microcontroller to the company. Once the microcontroller is delivered, the customer has to pay half of the service fee. The firmware is then dumped and flashed onto two brand-new microcontrollers, essentially creating two clones of the original microcontroller. The two clones are then mailed back to the customer, who can now verify whether the firmware dump was valid by using the cloned microcontroller in place of the original. Once the customer has verified the firmware correctness and pays the other half of the service fee, the customer receives a firmware dump of the microcontroller through email.

According to our conversation with the representatives of the company, the main customers of firmware dumping services are electronics repair shops that need replacement microcontrollers for devices which have failed. Allegedly, there are a few companies in Russia that provide a similar firmware dumping service, but apart from that, there are not many known providers for such services.

4.2.2 Obtaining a firmware dump

We decided to get the firmware for the car module with the serial number 137630. For us, the process of obtaining a firmware dump did not go as smoothly as it was intended. We sent them the microcontroller and paid half of the service fee, which was around 220 USD for this microcontroller. Then however, as of the year 2022, the company was not able to source new ATmega162V microcontrollers for creating the two copies they were supposed to. We also tried searching for suppliers of this microcontroller over the span of two months, without success. Eventually, we convinced the company that we can verify the validity of the firmware image by looking at the binary files (flash and EEPROM contents). They then sent us the contents of the memories over email.

The email contained three files: (1) the content of EEPROM; (2) the content of flash; and (3) the fuse values. The EEPROM file was 512 bytes in size, which is the size of the EEPROM memory in the microcontroller. The size of the file containing the flash memory contents was 16 KB, also the same as the amount of memory in the microcontroller. The fuse bits were sent as a screenshot of some software that displayed the fuse values (see Figure 4.2).

The correctness of the fuse values were the easiest to verify because we had previously read the values out through the ICSP interface. Verifying the EEPROM contents was a bit more difficult. We looked at the hexdump of the file and found the PIN-code and the serial number of the immobilizer (more details in Section 4.3), after which we concluded that the contents of EEPROM were correct. The flash memory contents were the hardest to verify. We disassembled the firmware in the file and then examined which pins of the microcontroller the firmware uses. Once we saw that this information matches the layout of the PCB, we considered dumped contents of the flash correct. Finally, we paid the other half of the service fee (another 220 USD) to the company.

We repeated the same process in order to get the firmware for the tag too. Altogether, the service of dumping the memories of the two microcontrollers cost 882 USD including taxes, plus around 20 EUR of shipping costs.

For further experimentation, we want to be able to read the memory contents in the microcontroller ourselves. This would become useful for further experiments in case the

Figure 4.2: The screenshot we received that shows the fuse values of the microcontroller taken from the car module

firmware itself modifies the contents of the memories. In order to achieve this, we first reset the memory readout protection fuse bit. This process erased the memories in the microcontroller, but since we now have the memory dumps, we can upload the flash and EEPROM contents back onto the microcontroller. As a result, we have the car module and the tag running the original firmware while its memory readout protection is disabled. We are now able to read out the contents of the memories using the ICSP interface. The following sections will cover the analysis of the car module firmware and the data stored in EEPROM.

4.3 EEPROM analysis

The ATmega162V microcontroller, which is used in the tags and the car module, contains 512 bytes of non-volatile memory called EEPROM (Electrically Erasable Programmable Read-Only Memory). This section focuses on analyzing the contents of the EEPROM memory in the car module.

The EEPROM memory in the ATmega162V microcontroller is rated for 100'000 write/erase cycles while the flash memory is only rated for 10'000 write/erase cycles. Furthermore, EEPROM can be written a byte at a time as opposed to flash memory, which can only be written a page (128 bytes) at a time. This makes EEPROM more suitable for storing frequently changing information where only small parts need to be updated at a time, such as the running state of a device or its configuration.

Figure 4.3 shows a hex dump of the bytes in the EEPROM of the car module. In a case where an EEPROM memory is erased, the values of all bytes are reset to `0xFF`. From the amount of bytes in the hex dump having values `0xFF`, it is visible that most of the bytes in the memory are likely not used. We also notice that there are no ASCII (American Standard Code for Information Interchange) strings stored in EEPROM.

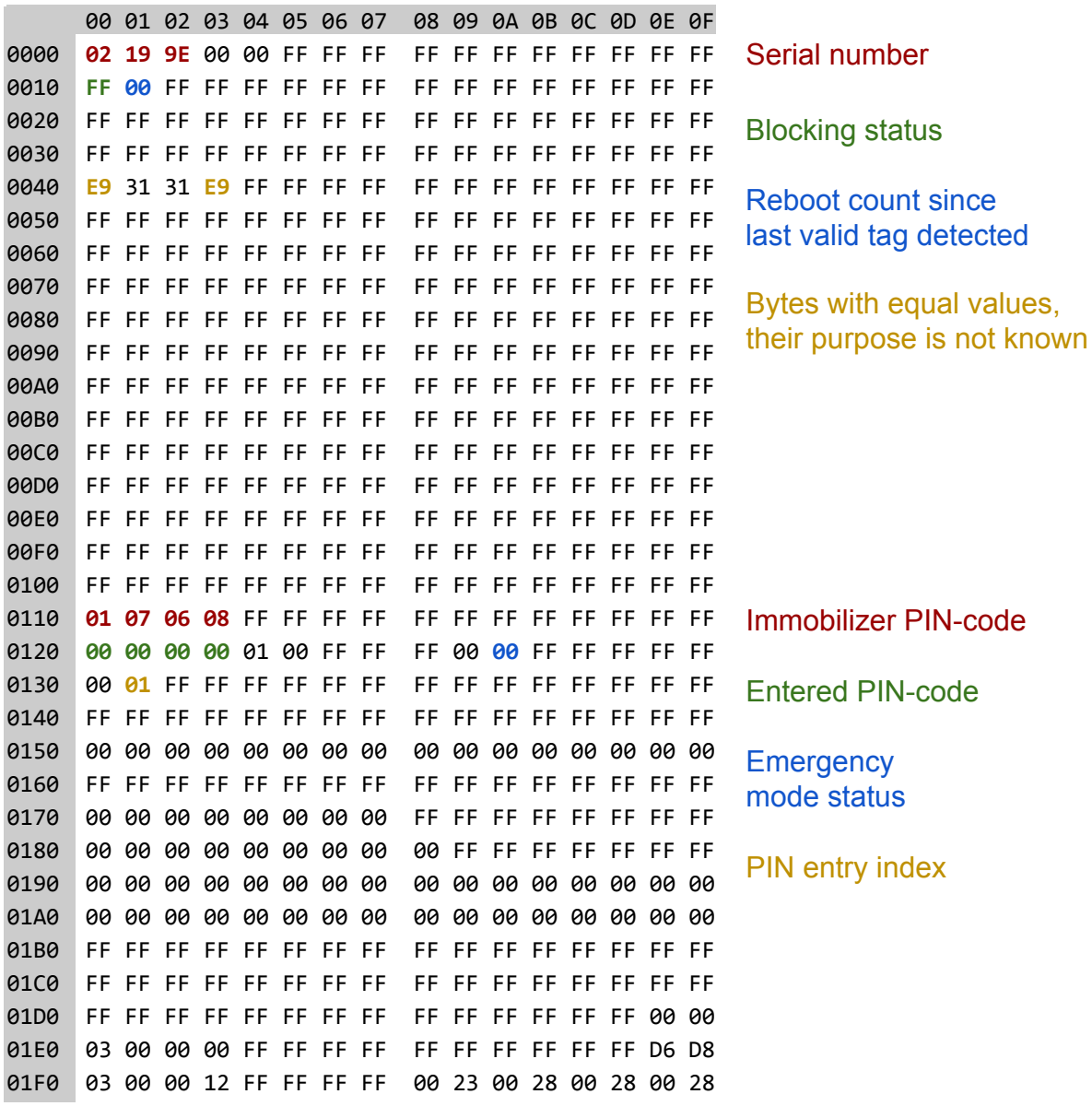

Figure 4.3: Hex dump of the EEPROM of the car module

4.3.1 Service card details

The immobilizer kit came with a service card that contained a serial number, a PIN-code and a service code. The serial number for this immobilizer was `137630`. Converting this number to hexadecimal format results in `0x02199E`, which matches with the bytes stored in the EEPROM's address range `0x0000:0x0002`.

The service card also provided a PIN-code for entering emergency mode, which was `1768`. This value can be seen in the EEPROM at the address range `0x0110:0x0113` in binary-coded decimal format.

The final number that the service card provides is a service code, in this case `F7C8D70F`. This value does not appear to be stored in the EEPROM. It might be possible that this number is only stored in some database that the vendor owns in order to keep track of when and where these immobilizers are manufactured, sold and serviced.

4.3.2 System state

As the immobilizer executes code, the values of variables are stored in volatile RAM (Random Access Memory). The values of these variables define what state the immobilizer is currently in. Some of these variables are also stored in non-volatile memory. This is apparent, for example, when the blocking circuit is disabled and the car's ignition is turned off, which in turn removes power to the immobilizer. When power is removed, the contents of RAM, where the running state of the variables are stored, are erased. After turning the ignition back on, the immobilizer powers up and knows immediately to disable the blocking circuit even before receiving the first packet from the tag, which indicates that this information had to be stored in non-volatile memory.

In order to understand the purpose of other bytes in the EEPROM, we took snapshots of the EEPROM's contents at different times, correlating the changes with events that happened just before taking the snapshot. The snapshots were taken using the ICSP interface, using the `avrdude` command-line utility.

One observation made was related to the example given above. The byte at address `0x0010` had a value of `0x00` when the blocking circuit was enabled and a value of `0xFF` when the blocking circuit was disabled.

The installation manual of the immobilizer states that if the engine is switched on and off for more than 8 times in a row without a tag being detected, the engine becomes immobilized [2]. To achieve this functionality, the immobilizer keeps track of how many reboots have occurred since detecting a valid tag. This value seems to be stored at address `0x0011`. It is reset to 0 when a valid packet is received and increases with each reboot. If the immobilizer is rebooted more than 9 times, the value will not be increased further and will stay at 9 until a valid packet is received.

4.3.3 Emergency mode

The immobilizer can be switched to emergency mode by entering a PIN-code. We observed in the EEPROM dumps that a counter stored at address `0x0131` is used to track which digit of the PIN-code is currently being entered. Digits of the entered PIN-code are stored at addresses `0x0120:0x0123`. The format of the entered PIN-code is the same as for the stored PIN-code at addresses `0x0110:0x0113` – each digit is stored in a separate byte.

The value at `0x012A` seems to show the state of the emergency mode. The value is `0x00` when emergency mode is active and `0xFF` when not in emergency mode.

We did not observe any incorrect PIN-code retry counters in the EEPROM. While this would make it possible to brute-force all possible 4-digit combinations, brute-forcing all combinations would require significantly more time than the time needed to find and remove the immobilizer from a car.

4.3.4 Other bytes

During analysis, there were other changing bytes, the purpose of which could not be determined. For example, the byte at address `0x0040` always contained the same value as the byte at address `0x0043`. These values seemed to change to a new seemingly random value a few seconds after every reboot. This value was updated only when packets were received from a tag.

4.4 Firmware analysis

This section focuses on analyzing the firmware file dumped from flash memory of the microcontroller. This firmware file is a binary executable for the AVR architecture, which contains machine code instructions for the microcontroller to execute. Machine code is not meant to be read by humans, but it can be disassembled into assembly language. Assembly language is much closer to human-readable source code than machine code. There are many disassemblers available, but for this task two were used. The first one was Ghidra [31] – a software reverse engineering toolkit developed by the National Security Agency. The second one was The Interactive Disassembler (IDA for short) [32].

4.4.1 Ghidra vs IDA

Because Ghidra is open source and free to use for anyone, most of the reverse engineering was done using that. Ghidra is also capable of generating C-like pseudocode from the disassembled assembly instructions, which the free version of IDA does not support. For nested loops and nested conditional statements, having the pseudocode made it a lot easier to comprehend the amount of code in larger functions. However, in some cases, the pseudocode generated by Ghidra was in many cases longer and harder to read than the pure AVR assembly instructions.

There were however some drawbacks to using Ghidra. For example, Ghidra did not recognize any interrupt vectors as entry points to functions. This meant that a big proportion of the codebase was never analyzed by Ghidra and no references to some functions were created. IDA, on the other hand, recognized the interrupt vector table automatically and also automatically commented each of the interrupt service routines respectively. This information could then be transferred to Ghidra by marking the jump instructions in the interrupt vector table as function entry points and running a full automated analysis of the project again.

There were still many functions that were allegedly never entered – neither Ghidra nor IDA could find any call instructions referencing those functions. There were also dynamic accesses to memory, which neither Ghidra nor IDA could link to specific memory addresses. This made finding connections between different parts of the code a very troublesome task for us. It involved manually calculating target addresses for hundreds of assembly instructions and adding them as comments. Luck had a big role in finding connections as we had to analyze hundreds of comments of seemingly random-looking addresses scattered around the codebase for some usage patterns. As a result of a lot of work, the logic behind EEPROM reads and writes was reverse engineered. SPI writes (microcontroller's communication with the radio module) were also successfully reverse engineered. However, although the code paths for reading SPI data were found, neither Ghidra nor IDA could identify places where these were called from. This meant that we were not able to uncover the full logic behind SPI communications.

4.4.2 Wireless packet addressing

The knowledge of how SPI writes work led to a part of code that sends packet data to the RF transceiver. Before sending the packet data, five additional bytes are sent that determine the wireless address to which the packet should be addressed to. The logic for determining this is shown in Figure 4.4. It shows that the first two bytes of the address are constants of value `0x07`. The next three bytes are taken from EEPROM from addresses 0 through 2. The values in these EEPROM locations, as explained in Section 4.3.1, correspond to the serial number of the device. This means that each immobilizer kit (car module + tags) is using its serial number as a unique address for transmitting and receiving. This also confirms what we observed in

Section 3.3 that the car modules and tags are transmitting their serial number in each radio packet transmitted over the air.

```
code:1b40 07 e0          ldi        R16,0x7
code:1b41 24 d0          rcall      spi_send_byte
code:1b42 07 e0          ldi        R16,0x7
code:1b43 22 d0          rcall      spi_send_byte
code:1b44 00 e0          ldi        R16,0x0
                    Sends the three bytes of serial number over SPI
code:1b45 1d d0          rcall      transmit_EEPROM[R16]_to_spi
code:1b46 01 e0          ldi        R16,0x1
code:1b47 1b d0          rcall      transmit_EEPROM[R16]_to_spi
code:1b48 02 e0          ldi        R16,0x2
code:1b49 19 d0          rcall      transmit_EEPROM[R16]_to_spi
```

Figure 4.4: Ghidra disassembly showing how the destination address is generated for an outgoing packet

4.4.3 Encryption

Looking through the disassembled firmware, there were no functions visible in the call graph that could perform any sort of encryption. This was another case of neither Ghidra nor IDA being able to determine entry points for a function and thus never disassembling that part of the codebase. We found the encryption function by forcing Ghidra to interpret all the bytes in the firmware file as executable code. This resulted in a lot of invalid instructions, but also uncovered some function-like code constructs. The whole binary file was then manually looked through instruction-by-instruction to find any new functions.

```
                    undefined crypto_xor_encrypt()
      undefined       Wlo:1          <RETURN>
                    crypto_xor_encrypt

code:0b9e 11 81          ldd        R17,Z+0x1
code:0b9f 10 27          eor        R17,R16
code:0ba0 11 83          std        Z+0x1,R17
code:0ba1 09 81          ldd        R16,Y+0x1
code:0ba2 10 27          eor        R17,R16
code:0ba3 11 83          std        Z+0x1,R17
code:0ba4 03 e7          ldi        R16,0x73
code:0ba5 12 81          ldd        R17,Z+0x2
code:0ba6 10 27          eor        R17,R16
code:0ba7 12 83          std        Z+0x2,R17
code:0ba8 0a 81          ldd        R16,Y+0x2
```

Figure 4.5: Disassembly in Ghidra showing a small snippet of the encryption sequence

One of the functions found was a long repeating sequence of XOR instructions followed by other instructions. A small snippet of this sequence is shown in Figure 4.5. It combines values pointed to by the Z register, values pointed to by the Y register and constant values using the XOR operator (`eor` instruction in AVR assembly – "exclusive or"). In this function, the Z register points to the input data, which is also where the processed data is stored after

the encryption routines finish. The Y register points to a location in memory, where the serial number of the device is stored. The random-looking constant values `0x21`, `0x15`, `0x73`, `0x01`, `0x03`, `0x39`, `0xF1`, `0x13`, `0x77`, `0xA9`, `0x41`, `0x32`, `0xE6`, `0x95`, `0x2F`, `0xB6`, `0xC8`, `0x62`, `0x3D`, `0x15`, `0x76`, `0x8E`, `0xA6` form a 23-byte long encryption/decryption key. When analyzed further, this function was just a part of a longer encryption sequence, split into multiple functions.

There appeared to be two distinct functions that, in addition to XORing together the input data, the encryption key and the serial number, also manipulated the data in different ways. These two functions were in turn called by another function multiple times in different sequences. Overall, the encryption/decryption routines were relatively complex and we could not afford to spend more time on reversing them entirely.

One thing we noticed earlier when transmitting minimal packets (see Section 3.3.4) was that the minimal packets were recognized by all car modules, even though they had different serial numbers. If the serial number is used as part of the encryption key, the ciphertext should be different for immobilizers with different serial numbers. The fact that all car modules recognized the minimal packets indicates that only 21 bytes of the packet are encrypted and the other 4, that define the minimal packet, are not.

4.4.4 Verifying encryption logic by cloning a tag

Assuming that the encryption algorithm really is just an XOR operation between the input data, a hardcoded key and the serial number, it should be possible to make tags from one immobilizer communicate with a car module of another immobilizer just by making the tag believe that it has another serial number. In order to verify this hypothesis, we used the hardware shown in Figure 4.6 to perform an experiment by executing the following steps:

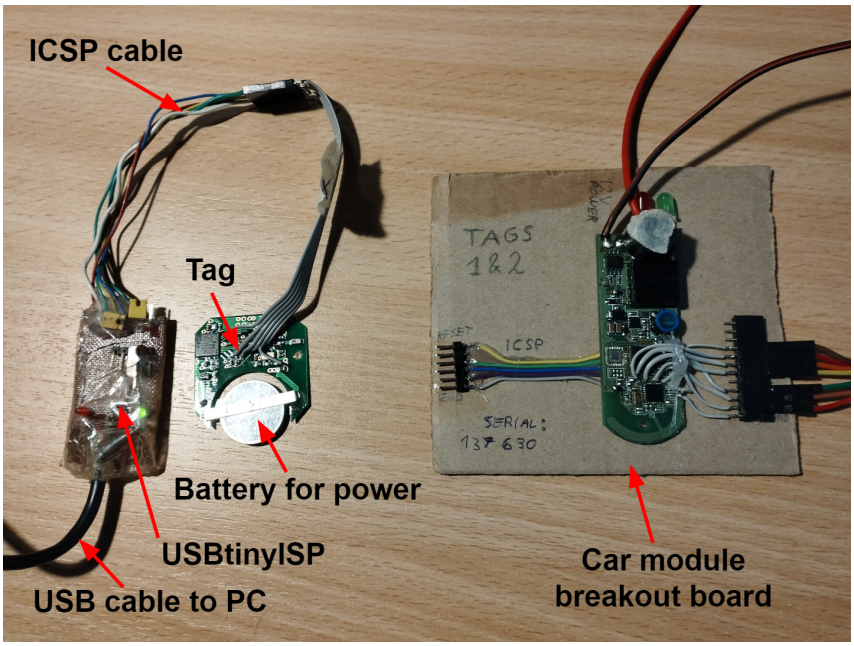

Figure 4.6: The hardware used for cloning a tag

- A car module with the serial number 137630 and a tag with the serial number 37236 were used for the experiment.

- The serial number 137630 in decimal was converted to hexadecimal: `0x02199E`.
- The EEPROM dump of the tag was modified with a hex editor: the three first bytes of the EEPROM were set to the values `02`, `19` and `9E`, effectively overwriting the serial number 37236 of the tag with the serial number 137630 used by the car module.
- The tag was then connected to a computer through an ICSP programming cable (USBtinyISP) and powered with a battery.
- The modified EEPROM dump was uploaded to the tag using the `avrdude` command-line tool.
- The tag was disconnected from the computer and its battery was removed.
- The tag and car module were powered on.

As a result, the car module immediately recognized the tag as a known tag and disabled the blocking circuit, allowing the car to be turned on. With this we have successfully cloned a tag. Further tests showed that the cloned tag did not interfere with the operation of the original tag, meaning that both tags were recognized by the car module.

We repeated the same experiment the opposite way. Instead of modifying the EEPROM of a tag, we cloned a car module by modifying the EEPROM of the car module. As a result, after powering on the tag and the car module, the car module once again immediately recognized the tag as a known tag and disabled the blocking circuit, allowing the car to be started.

The ability to clone tags demonstrates that the immobilizer is susceptible to the worst attack an immobilizer can have. Even though we were not able to fully reverse engineer the algorithm used by the immobilizer, we were able to demonstrate that tags can be trivially cloned just by knowing the serial number of the immobilizer kit. The serial number can be easily obtained as it is transmitted in each packet by the tag and the car module. Here we had to use the original hardware and firmware with slight modifications in order to make a cloned tag, but by completely reverse engineering and thus knowing the algorithm, this attack could be reimplemented in software using a software defined radio (SDR).

5 Overview of the attack vectors

Based on what we have learned about the functionality of the immobilizer in the previous chapters, we can analyze the main possible attack vectors of this security product.

5.1 Simple replay attack

By a simple replay attack, we mean recording a packet transmitted from a tag and retransmitting it later. We have verified that this attack would not be effective against the immobilizer.

As the black-box analysis showed, the ping packets transmitted by a tag contain the same data until a challenge packet is received from the car module. This means that, if the immobilizer is not immobilized (blocking circuit disabled) and we then recorded a packet from the tag and started replaying it, we could prevent the immobilizer from immobilizing the car even if the original tag disappears. However, the recorded packet would become outdated once the car is turned off and on again, after which the car module would reject the packet we transmit and consequently the attack would fail.

5.2 Bypassing the anti-theft functionality

As was shown in Section 3.3.6, the anti-theft functionality will crash after the car module receives a minimal packet. The implications of this attack are the same as for the simple replay attack. If the car is not immobilized then this attack would prevent the immobilizer from immobilizing the car. This attack would stop working after the car module is rebooted, just like with the simple replay attack. The only difference between this and the replay attack is that the minimal packet only needs to be transmitted once for this attack to work. For the replay attack, it was needed to keep on transmitting the packets periodically in order to keep the car from getting immobilized.

5.3 Communication relay attack

The idea behind this attack is to use a radio range extender in order to make the tag communicate with the car module even if they are out of range. There are two aspects that make the implementation of this attack difficult.

The first aspect is that each challenge and response packet is sent on a different frequency, therefore it is not known on which frequency the packets should be listened for. However, with a software defined radio, it would be possible to capture a range of frequencies at a time.

The second aspect that makes executing this attack difficult comes from the timing requirements. As was discussed previously, in order to save power, the tags only listen for responses for a brief amount of time. The captures of the communication exchanged

between the microcontroller and the RF transceiver in the tag show that the tag listens for challenge packets for 17.0 milliseconds after transmitting a ping packet. The captures of the communication exchanged between the microcontroller and the RF transceiver in the car module show it takes the car module 9.8 milliseconds to respond with a challenge packet. We assume that at a baud rate of 1 Mbps, transmitting 1 preamble byte, 5 address bytes, 25 data bytes and 2 CRC bytes would take $\frac{(1+5+25+2)*8}{1000000} \cdot 1000 \approx 0.3$ ms. This means that the time available for executing this attack would be only $17.0 - 9.8 - 0.3 = 6.9$ ms. As two packets need to be transferred within these timing constraints (the ping packet to the car module and the challenge packet to the tag), the time available per packet transfer is $\frac{6.9}{2} = 3.45$ ms. Using traditional mobile networks to relay this data between the tag and the car module would not work since latency for mobile networks exceed this timing requirement. Using a wired network for relaying data would be impractical and a custom radio solution for relaying the data would be costly. An alternative would be to just amplify the radio signals once received. This way the relaying distance would be decreased, but it would be easier to meet the timing constraints.

Although we have not implemented this attack in practice, we consider this attack possible.

5.4 Jamming the signal

This is one of the simplest attack vectors to exploit. Although the immobilizer does frequency hopping, all ping packets are still sent on the same two frequencies. Jamming these specific frequencies would produce a denial of service. This means that the car would not be able to start at all. Although we have not implemented this attack in practice, we consider this attack possible.

5.5 Tracking users

Based on the data gathered during experimentation, we know that the tags transmit fixed-length data packets on a certain frequency with a known addressing scheme. It is thus possible to identify people carrying the Skybrake DD2+ tags by searching for data packets with similar features. Furthermore, in addition to being able to identify packets coming from different tags, we can also capture the serial number of the immobilizer kit the tags belong to, which can be used for making a clone of the tag.

In order to implement a device that is able to capture packets from tags, we repurposed the same hardware (ATtiny817 and nRF24L01+) that was used in Section 3.3.4 to transmit outdated packets. We wrote firmware for detecting Skybrake DD2+ tags, extracting their serial numbers and saving them to non-volatile memory. Later, the serial numbers can be extracted and used to make clones of the tags.

The heuristic used for detecting Skybrake DD2+ tags relies on the packet's transmission frequency, addressing scheme and CRC algorithm. From the firmware disassembly, we know that the addresses are composed of two hard-coded bytes `0x0707` and three bytes of the serial number. The nRF24L01+ can be configured to listen for packets of different addresses with different address lengths [33]. Although the datasheet for the nRF24L01+ documents only three possible address lengths – 3, 4 and 5 bytes – there is an undocumented feature, marked "illegal" in the datasheet, which enables using addresses with a length of only two bytes [34]. This makes it possible to configure the nRF24L01+ radio for listening and filtering packets starting with the address of just `0x0707`. By setting the address length to just two bytes, the nRF24L01+ transceiver assumes everything following it is data, including the three bytes of the

serial number. Filtering packets based on the value of just two bytes, however, produces quite a lot of false positives because of the noisy radio environment. In order to filter out the false positives, we also check the CRC of the packet. The two-byte address, along with the two-byte CRC, make a reliable filter that only packets from Skybrake DD2+ tags can come through.

According to our experiments, this device is able to detect tags in 10-meter radius in open space. A similar range for detecting tags in open space was observed for the car module. The device was able to detect all Skybrake DD2 and DD2+ tags we had gathered for research, which shows that throughout different versions of the Skybrake DD2 immobilizers, the addressing scheme has been the same. This device could not detect Skybrake DD5 tags because a different addressing scheme is used in that version.

5.6 Cloning the tag

As we were able to demonstrate, it is possible to clone tags just by knowing the serial number of the immobilizer kit. The serial number can be obtained from any packet transmitted by either a car module or a tag.

5.7 PIN-code recovery without damaging the coating on the card

The immobilizers come from the factory with a card that contains the serial number, PIN-code and service code. The PIN-code is covered with a coating that needs to be scraped off by the user in order to reveal it. Assuming that an attacker has access to the immobilizers before being sold to the customers, it would be possible for the attacker to see the PIN-code of the immobilizer kit by shining a bright light through the card.

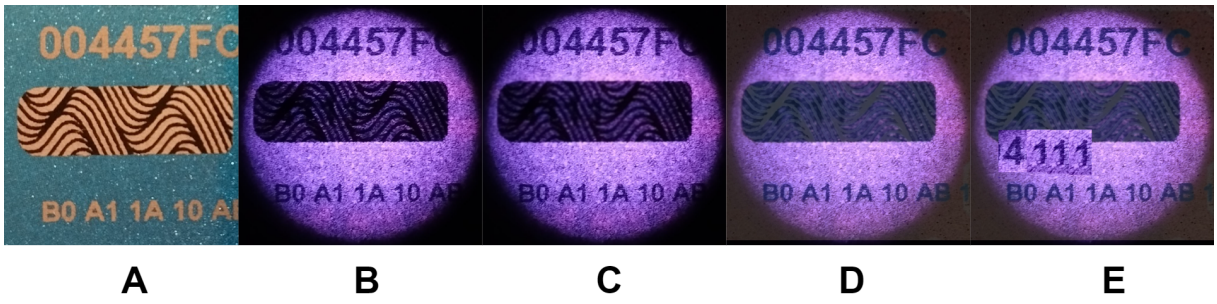

Figure 5.1: Extracting the PIN code by shining a light through the card. From the left to the right, each image has been manipulated to bring out the PIN-code better.

We tested this in practice and the result can be seen in Figure 5.1. Part A shows the original PIN-code that is covered. Part B shows how the PIN-code looks when a bright light is shining through it. In step C, we eliminated some high-frequency noise in the scrambling pattern by low-pass filtering the PIN-code portion of the image. In step D, we partly subtracted a thresholded inverse of the scramble pattern in order to make it less noticeable. Part E has the correct PIN-code with the original font and size placed below the covered PIN-code for comparison. We consider that this attack is possible.

5.8 Detecting cars equipped with the immobilizer

For attackers, knowing whether a car is equipped with this immobilizer could be valuable knowledge as then they could either come prepared with a cloned tag or just skip trying to steal the given car. As the car module only responds to messages from tags, it is not possible to determine the existence of a car module by passively listening to radio transmissions. It is however possible to transmit packets with every possible serial number and see if we get any responses. The requirement for this attack to work is that the ignition has to be turned on, i.e. the car module must be powered up. This requirement makes this attack impractical for many use-cases.

In order to transmit packets with every possible serial number, we would have to brute-force three bytes. We implemented this attack using our ATtiny817 and nRF24L01+ hardware. We transmitted ping packets to all possible addresses identified by two bytes (65536 in total), which took 1 minute and 48 seconds. As the serial numbers are three bytes in length, going through all values would then take 256 times longer or a bit less than 8 hours. Although, at first, 8 hours seems like a large amount of time for scanning a single car, in practice the scan result could be obtained in about 5 minutes. This is due to the fact that the serial number is incremented with each immobilizer kit produced and the most significant byte (MSB) of the address likely has a low value. During testing different immobilizer kits, we only encountered three different values for the MSB of the address: 0x00, 0x01 and 0x02. In the 5-minute period, about 182'000 serial numbers could be tested, which covers more than the range of all serial numbers of the immobilizers we were able to gather for experiments.

Theoretically, the time required could be decreased, since in our case the radio was transmitting only 20% of the time, staying idle for the rest of the time. Packets could also be transmitted on two different frequencies simultaneously to further speed up the process of brute-forcing the address of the immobilizer kit. Because this was supposed to be a quick proof-of-concept, we kept the attack as simple as possible to implement and did not optimize it for speed.

For this attack, it would be necessary to know what frequency the car module would respond on. Experimentally, we determined that when we transmit a packet with the address of the car module and contents "6B6200000000000000000000000000000000E75500000000C0C1" (hex-encoded 25 data bytes and 2 checksum bytes) on 2447 MHz, the car module would always respond with a challenge packet on 2433 MHz.

This attack can be taken one step further. Once we capture a challenge packet from the car module, we can extract the serial number of the immobilizer kit from the address of the challenge packet. Having obtained the serial number, we can create a cloned tag that will disable the blocking circuit of the immobilizer, allowing the car to be started.

6 Discussion and future work

This chapter discusses the answers to the questions proposed in the beginning of the thesis and also lists some potential topics for future work.

6.1 Discussion

In the beginning of the thesis we proposed four research questions. As a result of our research, we are able to give an answer to each of the proposed questions.

The first question that we proposed was what data do the car module and the tag transfer over the radio channel? The manuals for the immobilizer contained a part of the answer. Based on the manuals, the battery level of the tag is transmitted from the tags to the car module. As it is possible to use only up to five tags with a single car module, an identification number of the tag is likely also transmitted from the tag to the car module so that the car module would know which of the five tags it received a packet from. The rest of the answer has been found by reverse engineering the immobilizer. At a high level, there are three types of packets transferred between the tag and the car module: ping, challenge and response packets. On a lower level, we know that these packets contain the address of the packet, part of which is the serial number of the immobilizer, and the frequency on which responses will be listened for.

Another question we proposed was how does the car module authenticate the tag? The car module can differentiate tags based on the ping packets transmitted by the tags. We observed in the logic analyzer captures that if the car module decides to authenticate a tag, the car module responds to the ping packet from the tag with a challenge packet. The tag must then calculate a response to this challenge issued by the car module. After the tag has transmitted its response and the car module has received it, the car module will consider the tag authenticated.

The third question we proposed was whether it is possible to clone or spoof the tag or the signals transmitted by the tag? We showed that, in some cases, just replaying packets is enough to trick the car module into believing that a legitimate tag is nearby. We also showed that in other cases, where replaying is not enough, it is possible to make a full clone of a legitimate tag.

In order to make a full clone of a tag, the immobilizer's serial number is needed. We demonstrated multiple methods for acquiring the serial number without having physical access to the immobilizer. As tags periodically transmit packets with the serial number contained in the packet's address, one method for obtaining the serial number is to capture a single packet from the tag. In a real world attack scenario, this means that in order to bypass the immobilizer, a prepared attacker would have to walk by the person carrying the tag prior to entering the car.

We also demonstrated a method for retrieving the serial number of the immobilizer from the car module. Since the car module does not transmit packets regularly, obtaining the serial number involved brute-forcing it. Although this method takes a few minutes longer to succeed than listening for a packet from a tag, from the perspective of a prepared attacker trying to bypass the immobilizer, this method removes the need for walking by the user carrying the

tag and capturing a packet from it. This clearly demonstrates the seriousness of the security weakness. An attacker with special hardware could bypass the immobilizer in a few minutes without having to plug anything into the car.

The final research question we proposed was whether it is possible to uniquely identify the tag and hence track its holder? Using the logic analyzer, we were able to observe how the RF transceivers are configured. This configuration showed us the addressing scheme that was used by the immobilizer. Based on this information, we were able to build a device that can track the tags by recognizing the ping packets that the tags transmit.

In conclusion, we were able to answer all of the research questions that we proposed. Additionally, we were able to find multiple security flaws in the immobilizers and were able to demonstrate attacks against the security flaws in practice.

6.2 Future work

While our work proved that cloning a tag is possible, the cloning process required special hardware. One area of further research could be fully reverse-engineering the firmware and implementing tag scanning and spoofing using a software defined radio.

We also demonstrated a brute-force attack for obtaining the serial number of the immobilizer from the car module. The demonstration worked as a proof-of-concept, but was not optimized to be as fast as possible. An area of further research could be optimizing the method we used to brute-force packet addresses.

As the DD2+ version of the immobilizer is discontinued, another area of research could be analyzing the protocol used by the DD5 immobilizer, the successor to the DD2+ immobilizer. As the electronics in the DD5 version are largely similar to the electronics used in the DD2+ version, this thesis could be used as a basis for further experiments, outlining the major differences between the versions.

Summary

The aim of this thesis was to reverse engineer the Skybrake DD2+ series of car immobilizers in order to uncover its security weaknesses. We started the reverse engineering process by looking through online materials. Once we had a good overview of the alleged capabilities of the immobilizer, we continued by taking the immobilizer apart and connecting measurement devices to it. This way we were able to capture the data that was sent between the car module and the tag.

The collected data was then analyzed in order to evaluate the Double Dialogue protocol. In the analysis, we saw that the allegedly encrypted data that is exchanged between the car module and the tags contained a lot of repeating data, and the entropy of the exchanged data was low. From this we concluded that the data is not encrypted well.

We also dumped the firmware of the car module and the tag. By modifying the part of the memory contents that contained the serial number of the immobilizer, we were able to create clones of tags. We also analyzed the immobilizer's firmware and discovered that serial numbers of immobilizer kits were used as wireless packet addresses. Putting these two pieces of information together, we were able to demonstrate practical attacks.

One of the attacks demonstrated was creating a clone of a legitimate tag after capturing a single packet from it. In another attack, we demonstrated a method of brute-forcing packet addresses in order to determine the serial number of a car module. As a result, the immobilizer could be bypassed wirelessly without ever having to have access to any of the tags that the car module is supposed to be used with.

Overall, the findings of this work show that the design of this immobilizer was not secure. The entire security of this product relied on the vendor's assumption that the attacker would not be capable of accessing the firmware. Reverse engineering the immobilizer uncovered weaknesses in the communication protocol, making it possible to attack the immobilizer, completely defeating the purpose of this security product.

Acknowledgements

I would like to thank my supervisor Arnis for providing me with such an interesting research topic and helping me tackle it every step of the way. Even when I thought this work had come to a dead end, you were determined to continue and motivated me to pursue. The numerous hours of work you have dedicated into this work are far beyond what I would have expected from a supervisor.

I would also like to thank Eva and Meelis for proofreading my work and keeping me motivated throughout the writing process.

Bibliography

[1] AvtoTachkiAdmin. "Skybrake immobilizer: principle of operation, features, installation and dismantling". https://en.avtotachki.com/immobilayzer-skybrake-princip-raboty-osobennosti-ustanovka-i-demontazh/. Accessed: 2023-04-25.

[2] www.skybrake.com. Skybrake DD2+ installation manual (English). https://alfapolish.hu/letolt/skybrake_dd2.pdf, 2008. Accessed: 2023-01-20.

[3] Autonams. Skybrake DD2+ installation manual (Russian). https://www.autostudio.ru/files/291.pdf, 2006. Accessed: 2023-01-21.

[4] Autonams. Autonams website: About us. https://www.autonams.lv/en/about-us/. Accessed: 2023-01-18.

[5] Sanita Jemberga. Kirson's secretary went to get a new suit (in Latvian). https://ir.lv/2010/11/24/kirsona-sekretare-aizgaja-pec-jauna-kostima-2/, https://ir.lv/rubrika/people/page/162/, 2010. Accessed: 2023-02-05.

[6] DIENA.lv. Autonams last year's turnover increases by 62% (in Latvian). https://www.diena.lv/raksts/pasaule/krievija/autonams-perna-gada-apgrozijums-pieaug-par-62-35010, 2008. Accessed: 2023-05-02.

[7] iAuto. "Autonams" has started the production of SKYBRAKE fleet management systems (in Latvian). https://iauto.lv/zinas/10513-autonams-uzsacis-skybrake-autoparku-vadibas-sistemu-razosanu, 2008. Accessed: 2023-02-05.

[8] TVNET. "Autonams" immobilizer market share in Russia reaches 10% (in Latvian). https://www.tvnet.lv/5071110/autonama-imobilaizeru-tirgus-dala-krievija-sasniedz-10, 2010. Accessed: 2023-02-05.

[9] iAuto. Aftermaths of Autonam's unique products have appeared (in Latvian). https://iauto.lv/zinas/10602-autonama-unikalajiem-produktiem-paradijusies-pakaldarinajumi, 2008. Accessed: 2023-02-05.

[10] iAuto. Installation of Skybrake DD2+ for motorcycles begins (in Latvian). https://iauto.lv/zinas/13343-uzsak-skybrake-dd2-uzstadisanu-motocikliem, 2011. Accessed: 2023-02-05.

[11] Georgs Rubenis. Autonams company overview_2011_v7 (in Latvian). https://www.slideshare.net/georgsrubenis/autonams-uznemuma-parskats2011v7-9235827, 2011. Accessed: 2023-02-05.

[12] oborudow.ru. Skybrake dd2 problems. SkyBrake immobilizer is a reliable security system for a car. Possible problems and solutions. https://oborudow.ru/en/tips/skybrake-dd2-problemy-immobilaizer-firmy-skybrake-nadezhnaya-ohrannaya. Accessed: 2023-02-05.

[13] Autonams. Skybrake DD2 immobilizers (in Latvian). https://web.archive.org/web/20120504012145/http://www.autonams.lv/auto-drosiba/auto-imobilaizeri/skybrake-dd2-serija/. Accessed: 2023-04-08.

[14] A-Autoalarm. "SKYBRAKE DD2+" product listing. https://www.a-autoalarm.ee/immobilaiserid/skybrake-immobilaiser. Accessed: 2023-04-08.

[15] Ugona.net. "Immobilizer SkyBrake DD2+" (in russian). https://www.ugona.net/archive/immo/skybrake/dd2-603.html. Accesed: 2023-04-08.

[16] Autonams. Vehicle immobilizers. https://web.archive.org/web/20170928122242/http://www.autonams.lv/en/vehicle-security. Accessed: 2023-04-08.

[17] SkyBrake. DOUBLE DIALOGUE TECHNOLOGY. https://web.archive.org/web/20080331061458/http://www.skybrake.com/?section=14. Accessed: 2023-04-08.

[18] Nordic Semiconductor. nRF2401 Single Chip 2.4 GHz Radio Transceiver PRODUCT SPECIFICATION. https://www.sparkfun.com/datasheets/RF/nRF2401rev1_1.pdf, 2004. Accessed: 2023-02-05.

[19] Saleae. Documentation for the Saleae Logic 2 Automation API. https://saleae.github.io/logic2-automation/automation.html. Accessed: 2023-04-03.

[20] Mark Garrison. Logic 2 Automation API release announcement. https://discuss.saleae.com/t/saleae-logic-2-automation-api/1685. Accessed: 2023-04-03.

[21] Pavel Tšikul. Encrypted Data Identification by Information Entropy Fingerprinting. MSc thesis, Tallinn University of Technology, 2019. https://digikogu.taltech.ee/en/Download/3b18c60d-8d50-4dce-be32-56783e30fe75.

[22] Microchip. Microchip Developer Help page "AVR® Fuses". https://microchipdeveloper.com/8avr:avrfuses, 2021. Accessed: 2023-05-27.

[23] Atmel. Atmel AVR ATmega162 datasheet. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2513-8-bit-AVR-Microntroller-ATmega162_Datasheet.pdf, 2013. Accessed: 2023-02-05.

[24] Adafruit. USBtinyISP – AVR programmer & SPI interface. https://learn.adafruit.com/usbtinyisp. Accessed: 2023-05-08.

[25] The Engbedded Blog. Engbedded's AVR® Fuse Calculator. https://www.engbedded.com/fusecalc/. Accessed: 2021-12-30.

[26] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):199–224, Feb. 2019.

[27] Chad Spensky, Aravind Machiry, Nathan Burow, Hamed Okhravi, Rick Housley, Zhongshu Gu, Hani Jamjoom, Christopher Kruegel, and Giovanni Vigna. Glitching demystified: Analyzing control-flow-based glitching attacks and defenses. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 400–412, 2021.

[28] Tom Chothia Sebastian Vasile, David Oswald. Breaking all the Things – A Systematic Survey of Firmware Extraction Techniques for IoT Devices. 2018.

[29] Sergei P. Skorobogatov. Semi-invasive attacks – A new approach to hardware security analysis. Technical Report UCAM-CL-TR-630, University of Cambridge, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, April 2005.

[30] Circuit Engineering Company Limited. Circuit Engineering Company Limited website. https://www.ic-cracker.com/. Accessed: 2023-19-05.

[31] NSA's Research Directorate. Ghidra software reverse engineering suite of tools. https://ghidra-sre.org/. Accessed: 2023-04-04.

[32] Hex-Rays. IDA Pro website. https://hex-rays.com/ida-pro/. Accessed: 2023-04-04.

[33] Nordic Semiconductor. nRF24L01+ Single Chip 2.4GHz Transceiver Preliminary Product Specification v1.0. https://www.sparkfun.com/datasheets/Components/SMD/nRF24L01Pluss_Preliminary_Product_Specification_v1_0.pdf, 2008. Accessed: 2023-04-20.

[34] Travis Goodspeed. Promiscuity is the nRF24L01+'s Duty. https://travisgoodspeed.blogspot.com/2011/02/promiscuity-is-nrf24l01s-duty.html, 2011. Accessed: 2023-04-24.

[35] www.skybrake.com. Skybrake DD5 installation guide. https://alfapolish.hu/letolt/skybrake_dd5.pdf. Accessed: 2023-05-10.

Appendix: Comparison of different versions of the Skybrake immobilizers

While this thesis focused on the Skybrake DD2+ version of the immobilizer, we also gathered other versions of the Skybrake immobilizers for research. This section provides an overview of the differences between various Skybrake immobilizers.

DD2

When the Skybrake DD2 product was first released, the car module had a relatively large circuit board. This version is shown in Figure 6.1. On the PCB, there were two ATmega162V microcontrollers, two relays, an nRF2401 radio transceiver, a battery, large transistors and a large voltage regulator. There were also 16 wires coming out of the circuit board.

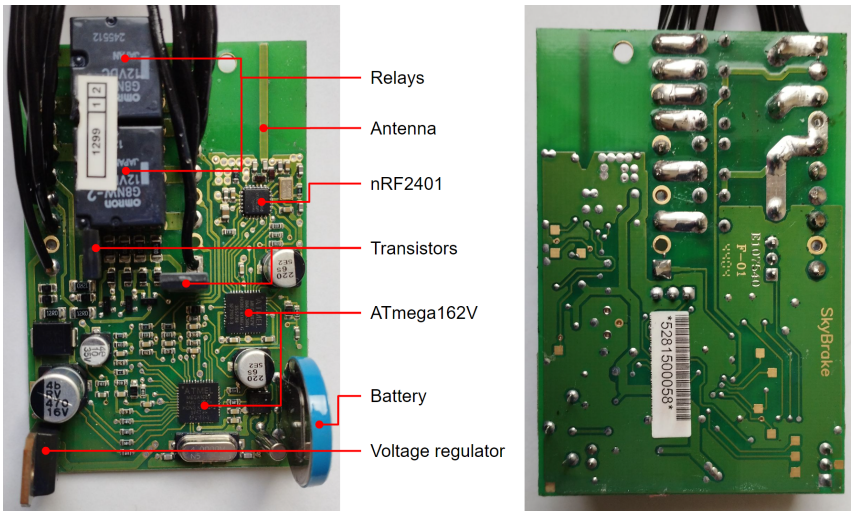

Figure 6.1: Top and bottom sides of the PCB of the car module of the Skybrake DD2 immobilizer

The PCB (see Figure 6.1) also contains a sticker with the serial number of the product – 1299. Assuming that the serial numbers started from zero, we can assume that there were at least a few thousand of these revisions manufactured. This also indicates that this is likely one of the first designs of this product.

We verified that the communication protocol used by the DD2 version is compatible with the communication protocol used by the DD2+ version. It is likely that exactly the same communication protocol is used by both versions of the immobilizers.

DD2+

The car module on the revised version, Skybrake DD2+, is a lot more compact. This version is shown in Figure 6.2. While it mainly consists of the same components – an ATmega162V microcontroller, an nRF2401 radio transceiver and a mechanical relay – it cuts down on size in many ways. Instead of having two relays and two microcontrollers, it only has one of each. The large through-hole transistors were replaced by smaller surface-mount transistors. Similarly, the large through-hole linear voltage regulator was replaced with a surface mount one. The battery, presumably used for powering memory for non-volatile storage, has been removed and data is stored in EEPROM instead. The number of wires coming out of the device is reduced to just five. The PCB (see Figure 6.2) also contains a sticker with the serial number of the product – 37236 – which suggests that there could be about 30'000 such boards produced.

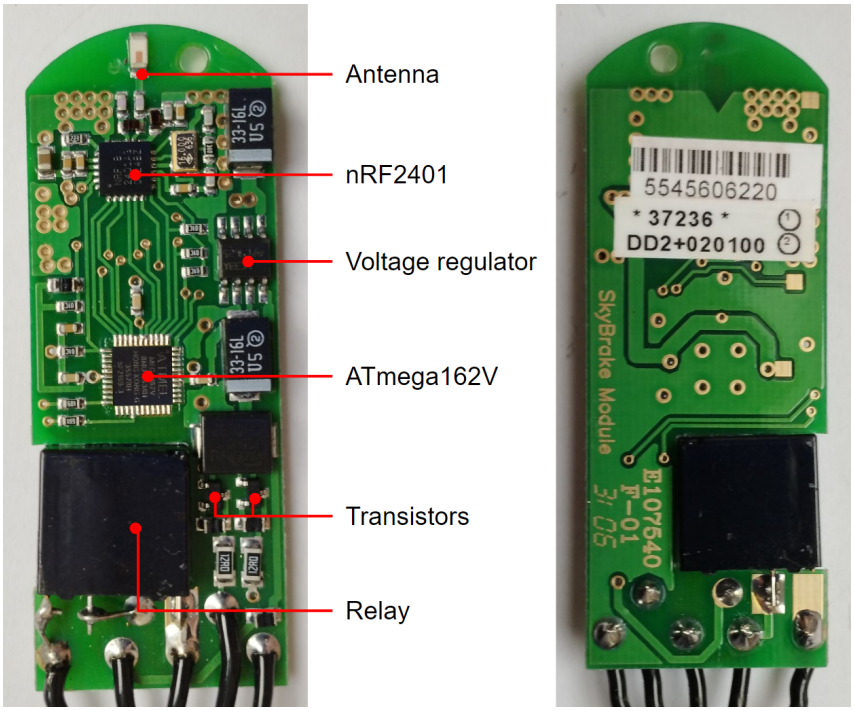

Figure 6.2: Top and bottom sides of the PCB of the car module of the Skybrake DD2+ immobilizer

Then there was a newer revision of the immobilizer that was released under the same Skybrake DD2+ name. This version is shown in Figure 6.3 on the left. The car module of this version had exactly the same case dimensions as the previous version, but internally had a lot more components. The PCB thickness was changed from 1.0 mm to 1.6 mm, providing more rigidity. The linear regulator was replaced with a switching regulator, making the immobilizer more power-efficient. The transistor used for switching the relay was switched out for a larger relay driver, presumably making the immobilizer more reliable. A footprint was also added for an external integrated circuit, presumably a motion sensor. This component was not populated on any of the devices we gathered for this research. The connections between the main components stayed the same. This is based on the fact that soldering the microcontroller from the previous version of the immobilizer to this version of the immobilizer resulted in a fully functional immobilizer. Changes in microcontroller firmware, if any, were minor. This is based on that fact that a new tag could be successfully paired with an older car module and

vice-versa. This proved that the communication protocol and encryption algorithms had to be the same throughout different revisions of the product.

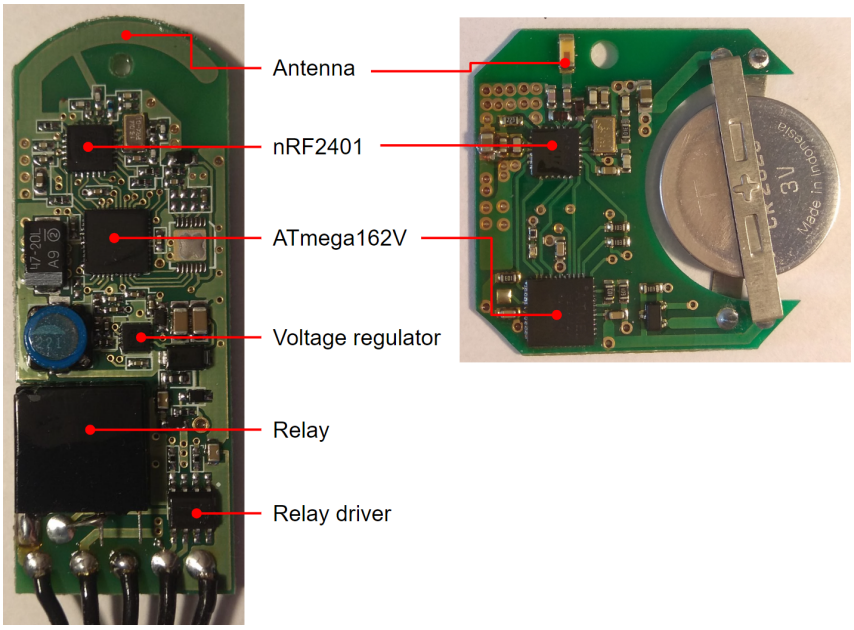

Figure 6.3: Top sides of the PCBs of the Skybrake DD2+ car module (left) and tag (right)

The exact devices with this PCB layout gathered for research had serial numbers 70251 and 137630. This shows that at least about 70000 DD2+ immobilizers produced were very similar in design.

The tags for the DD2 and DD2+ revisions of the immobilizer were electrically equivalent. One of these tags is shown in Figure 6.3 on the right. There were some changes in the PCB layout, but connection-wise everything stayed the same. The firmware for the tags has been mostly unchanged. Testing showed that the firmware used in the tags of the older DD2 version is compatible with firmware used in the newer DD2+ version car modules.

DD5

In 2018, a newer version of the immobilizer was released. This new version, named Skybrake DD5, included many changes while keeping the same physical dimensions. This version is shown in Figure 6.4. Most upgrades were changes to components. The components on the DD2 versions are hard to source due to some of them being discontinued. Most notably, the radio chip was replaced with a newer version – nRF24L01+. The microcontroller was also replaced with a PIC24F32KA microcontroller. Additionally, a fuse and a reverse polarity protection diode were added. The tags for the DD5 version (see Figure 6.5) followed similar changes – the microcontroller and radio chip were changed to the same ones as used now in the car module.

The car module and tags of the DD5 version are not compatible with the car module and tags of the DD2+ version. The DD5 immobilizer uses 4-byte addresses for packets and the size of each packet is 16 bytes. The ping packets of the DD5 tags do not repeat and each packet is sent on a different frequency. According to the DD5 installation guide, the immobilizer uses AES-128 encryption [35].

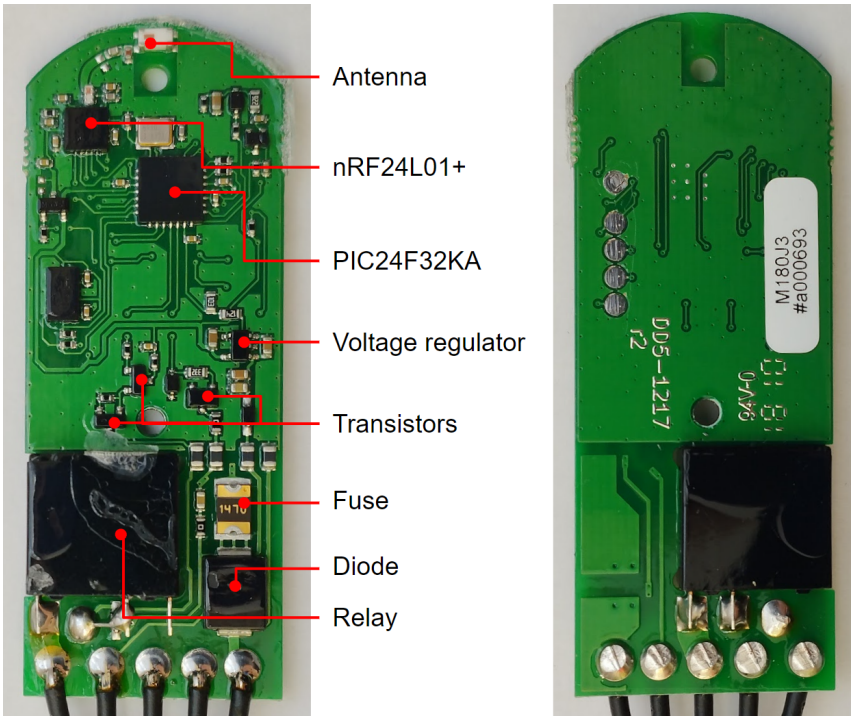

Figure 6.4: Top and bottom sides of the PCB of the car module of the Skybrake DD5 immobilizer

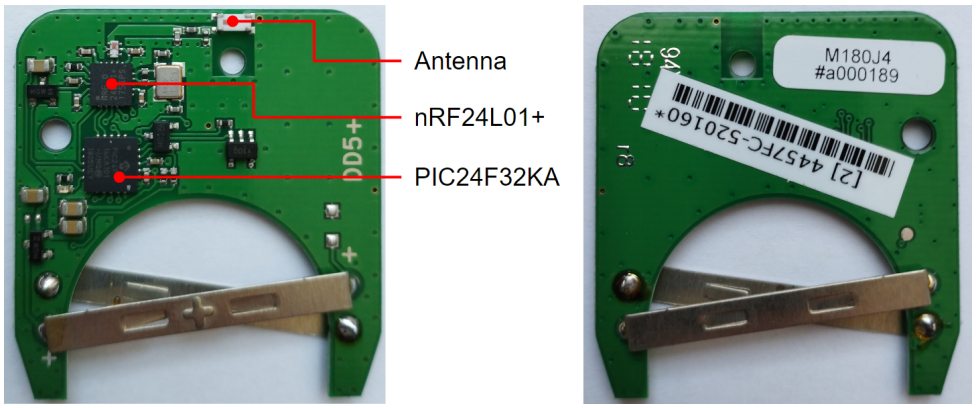

Figure 6.5: Top and bottom sides of the PCB of the tag of the Skybrake DD5 immobilizer

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Jürgen Laks,

1. grant the University of Tartu a free permit (non-exclusive licence) to:
   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

   "Reverse Engineering a Car Immobilizer"

   supervised by Arnis Paršovs

2. I grant the University of Tartu the permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work from 01/07/2024 until the expiry of the term of copyright.

3. I am aware that the author retains the rights specified in points 1 and 2.

4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Jürgen Laks **20.05.2023**