HOCHSCHULE
MITTWEIDA
University of Applied Sciences

BACHELOR THESIS

Mr.
Sönke Adrian Eichelbaum

A practical analysis of smart lock security

Mittweida, August 2024

Faculty of **Applied Computer Sciences & Biosciences**

BACHELOR THESIS

A practical analysis of smart lock security

Author:
Sönke Adrian Eichelbaum

Course of Study:
Applied Computer Science; It-Security

Seminar Group:
IF21wI1-B

First Examiner:
Ronny Bodach, Prof. Dr. rer. pol.

Second Examiner:
Martin Schöniger, Dipl. Ing. Elektrotechnik

Submission:
Mittweida, 19.08.2024

Defense/Evaluation:
Mittweida, 2024

Faculty of **Applied Computer Sciences and Biosciences**

BACHELOR THESIS

Eine praktische Analyse der Sicherheit von elektronischen Schließanlagen

Author:
Sönke Adrian Eichelbaum

Course of Study:
Angewandte Computerwissenschaften; IT-Sicherheit

Seminar Group:
IF21wI1-B

First Examiner:
Ronny Bodach, Prof. Dr. rer. pol.

Second Examiner:
Martin Schöniger, Dipl. Ing. Elektrotechnik

Submission:
Mittweida, 19.08.2024

Defense/Evaluation:
Mittweida, 2024

Bibliographic Description

Eichelbaum, Sönke Adrian:
A practical analysis of smart lock security. – 2024. – 41 S.
Mittweida, Hochschule Mittweida – University of Applied Sciences, Faculty of Applied Computer Sciences & Biosciences, Bachelor Thesis, 2024.

Referat

Smart locks and electronic locking systems are becoming more and more relevant and prevalent in our more and more digitized society. This thesis tries to analyze different systems, focusing on their implementation, building tools to aid in such investigations. It attempts to exploit those systems, in order to allow unauthorized access to their secured areas, trying to show alternatives and give recommendations for secure implementations.

Contents

List of Figures

Listings

1 Introduction

Smart locks and electronic locking systems are becoming more and more relevant and prevalent in our ever increasing digitized society. In order to be able to replace physical locks and secure entry to buildings and private property, those systems need to be secure, hardened, be able to withstand elemental and malicious influence and still be easily useable. In order to be easily usable, those systems need to have user-friendly interfaces and be as simple as possible. These systems can have many different places of deployment, ranging from residential homes to offices, and even more restricted areas like industrial location, warehouses, et cetera. With each of these requirements comes additional cost. For them to be secure and hardened, it takes engineering effort, both in hardware, as well as in software. In order to be able to withstand natural and malicious influence, it again needs engineering skills. To be user-friendly it requires UI and UX research and design.

1.1 Problem statement

The problem with the above mentioned requirements is, that each and every one of them requires additional work, finances or trade-offs to be made. The goal of this thesis is testing different products and technologies, whether they bothered to invest this additional effort and resources, or if they are vulnerable to attacks, potentially enabling attackers to gain unauthorized access to a lock or locking-system.

1.2 Chapter structure

The thesis is structured into two different sections. Each talking about a specific implementation or product, explaining the topic, system and concepts, where appropriate. After the introduction, a possible vector of exploitation is shown, if applicable with example code and screenshots. The end of each section will discuss the viability of the attack and mention possible mitigation strategies.

2 The case MIFARE

MIFARE is a brand of NXP Semiconductors, a global player in the semiconductor space, focused on production and integration of such chips, concerned with contactless IC products, like wireless badges and tokens.[21]. MIFARE offers a range of different products, covering many different applications, ranging from cloud offerings to smart cards supporting 3DES and AES encryption hardware.[22]
Most of their products implement at least a subset of the ISO/IEC 14443 standard, an ISO standard describing different parts of a solution for personal identification using cards or other devices, defining things like the radio communication[18], the communication protocol[19] and physical characteristics[17] of the cards.

2.1 The MIFARE Classic electronic locking system

The first system tested was an electronic locking system, commonly implemented for office buildings, hotels or similar applications.

2.1.1 The system

The system consists of three parts.

Figure 2.1: An NFC tag
Source: Own image

First: The access cards. In this case they were just simple round tags, like shown in image 2.1, that can be easily attached to a keychain and is supposed to be carried by the occupants of the building, granting them access to their respective areas, unlocking doors and opening gates. In reality those tags can take any shape or form, as the actual chip is only a small chip and a really thin antenna, which fit easily into any shape or form, as shown in image 2.3. The cards used in the tested system had to be activated once per day at an online terminal, before any access to any room would be granted.

Second: The "online" key terminals. In theory these are not strictly necessary, but hugely streamline the management aspect of the whole system. In this test, and probably most deployments, they are positioned on key points, like entries to the building, the lobby and (underground) parking. They are online because they, in contrast the "offline" key readers, are connected to the management system. In addition, the terminals are able to write to the tags, not just read them. This enables applications like building managers registering a new key in the system, which then automatically gets its permissions when first tapped to such a terminal, as well as adding or removing permissions from tags.

Figure 2.2: An NFC door reader
Source: Own image

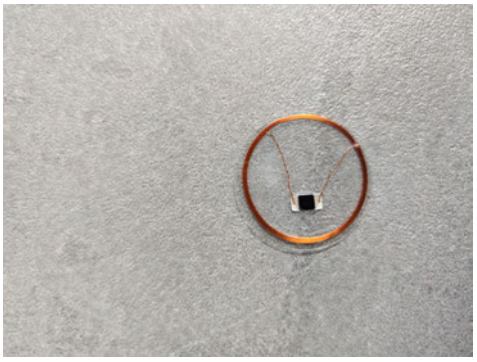

Figure 2.3: Internals of a NFC tag
Source: Own image

Third: The "offline" key readers. In order for access to be granted, the permissions of the tag have to be checked. This task is done by the key readers. In case of the tested system, they looked like shown in image 2.2. As can be seen in the image, the reader consists of an actual reader part, the black part in front, connecting to the bolt mechanism in the middle, and ending in a door knob, which can be operated without a tag. The whole lock is directly to be inserted into the door, replacing the existing lock. But like with the tags, these key readers can come in many different forms and sizes. The only relevant part is, that there is a reader somewhere, where it can talk with its associated locks. So an alternative design could also be a badge reader next to the door, that then connects to the lock in the door.
"Offline" is specified here because those readers do not connect to a central system, but are installed once, fitted with a battery, and then operate from there on out without further connections to a management system.

So in general the system would work like this: A new employee gets issued a badge. That badge was previously registered with the building manager, specifying that it should have access to the entry doors to the office of the employer, as well as specific rooms, like her office. This tag was send to her before her first day. On her first day she enters the lobby and scans the tag for the first time at an "online" terminal to enter the building. The terminal checks with the management system and writes the appropriate permissions registered in the system to the tag. The terminal grants access and now the employee is able to gain access to all "offline" readers for which her tag carries the credentials.

2.1.2 MIFARE Classic

The tags used by the system under testing are MIFARE Classic tags. The data sheet describes them as:

> "Mainstream contactless smart card IC for fast and easy solution development"[14]

Meaning MIFARE Classic is a type of tag, that adheres to a specific design, implements specific commands and follows a specific layout, all detailed in the associated data sheet.

Most importantly, MIFARE Classic supports/implements the following:

- Parts of the ISO/IEC 14443 standard
  Most notably the anti collision mechanics, allowing multiple tags to work simultaneously with the same reader
- Three pass authentication
- Two keys per sector for access control to those sectors
- An immutable UID per card
  The immutable part only applies for genuine MIFARE Classic cards. Aftermarket cards, also called "magic cards", support changing the UID, therefore disarming all security measures relying on the uniqueness of the UID.[14]

The tag is designed to hold data. Therefore it needs a memory layout, which is described in the data sheet. While this specific data sheet is concerned with the 1K version, there exists a 4K version. Their differences will be mentioned at the appropriate time.

Each card consists of 16 sectors, 39 for the 4K version, each sector consisting of 4 blocks, each 16 bytes. In the 4K version, only the first 32 sectors contain 4 blocks, the remaining hold 16 blocks each. All but the last block can be used for data storage, except in sector 0, where the first block contains the UID and manufacturer data. The last block of each sector contains the two keys, the mandatory A-key and the optional B-key, as well as the access bits for the data.[14][15] These two keys manage read/write access to the data stored in the sector. With this model, for example, the "online" terminal can be configured to know both keys, A and B, while the "offline" readers only need to know one key, lets say the A-key. The card can be configured to only allow writing using the B-key, while allowing key A to read everything. This enables the use-case as described above in the example, following the principle of least privilege. Using this method, only the terminal can read and write, while readers can only read. The permissions for each key are configured using the access bits. Read/write access to those access bits are, in turn, again managed using the A- and B-key, meaning with an unfortunate change to the access bits, the card can be bricked to never be read or written to again.
The tag has three different block types, read/write, value and sector trailer. The read/write type can store arbitrary data, while value blocks are intended for digital wallet functions, meaning they hold a single value (with redundancy), easily allowing adding and detracting "currency" from the block. The tag supports 4 different permissions/operations, read, write, increment and decrement. The read/write commands are valid for all block types, reading and writing the value, while increment and decrement are only allowed for value blocks. Each block is designated as a specific type by the access bits of the sector.[14]
The data sheet describes the access bits as being 3 bits for each block, which are stored twice, once inverted, once non-inverted. The bits for each block are stored interleaved, meaning first the first bit of each block is stored, then the second and so on. This is repeated twice, starting with the inverted version, followed by the non-inverted.[14]

2.1.3 Tag structure with example

After the previous section dryly discussed the structure of a MIFARE Classic tag, this section will attempt to visualize the description by showing all those features for one single sector, mentioning and highlighting each feature, as well as the composition of the access bits.

The image 2.4 is a screenshot of an empty sector 0, as decoded by the Mifare Classic Tool (MCT) Android app[2] Here, the structure of a sector is very well presented. Each of the 4 blocks is represented by a row. Each of the 16 bytes of a block is represented by 2 hexadecimal digits per byte. The first block holds the UID, which is redacted here, as well as the manufacturer information. The data sheet does not provide information on how the manufacturer block is structured, except for the first 4/7 bytes (depending on the type of tag) being the UID. MCT seems to interpret the last 2 bytes as a data of manufacture, decoding them as the last byte being the year and the second last being the week of the year the tag was manufactured.[3] In this case it would suggest that the tag was manufactured in week 40 (0x28) of the year 18 (0x12), probably meaning 2018. The second and third row in this example can contain arbitrary data, the 0x69 byte in the last row can also contain data, and the key B, if not in use, can also contain data. The A- and B-key in this example are configured to a well known set of default keys and highlighted in light and dark green respectively.

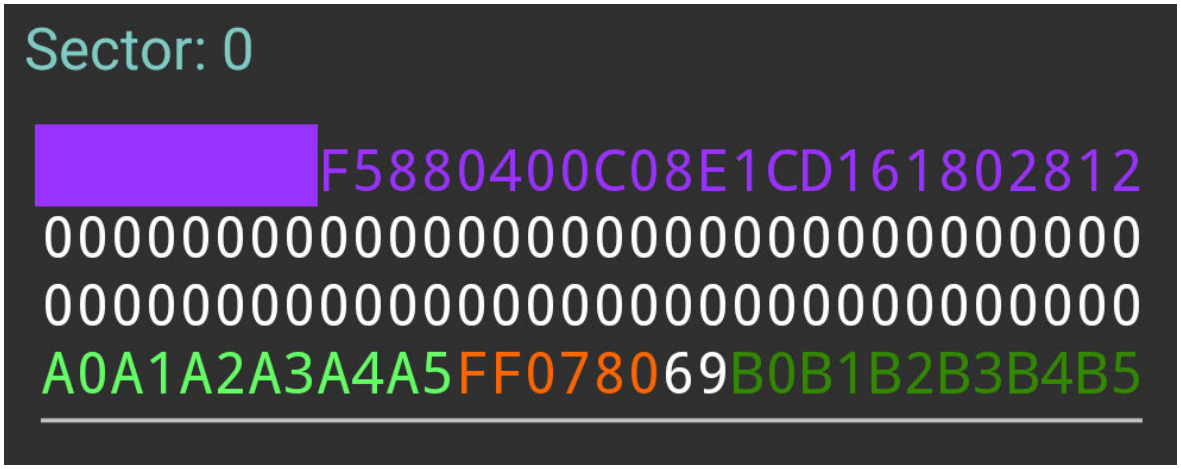

Figure 2.4: Example sector 0 of a MIFARE Classic 1K tag

Source: Own image

The last remaining part of the sector are the access bits, in this case 0xFF0780. Decoding them into the individual bits like this:

```
1111 1111
0000 0111
1000 0000
```

Figure 2.5: Decoded access conditions

Now, taking the last 12 bits, reading right to left from the top, gives following bit patter: 0b000000000001, meaning the access conditions are:

- block 0: 0b000
- block 1: 0b000

[2] More at: https://github.com/ikarus23/MifareClassicTool

[3] See here: https://github.com/ikarus23/MifareClassicTool/blob/3d8e2dbcea6b65449853dc7b16278fc47001c273/Mifare%20Classic%20Tool/app/src/main/java/de/syss/MifareClassicTool/Activities/DumpEditor.java#L754

- block 2: 0b000
- block 3: 0b001

Referencing table 8 of the data sheet, the access conditions of the data blocks can be decoded like this:

- block 0: can be read, written, incremented, decrement by key A and B
- block 1: can be read, written, incremented, decrement by key A and B
- block 2: can be read, written, incremented, decrement by key A and B

while the last block has to be decoded with table 7, as it is the sector trailer, leading to following access conditions

- block 3: key A cannot be read, but can be used to read/write the access bits, as well as key B, or if key B is not in use, the data stored there instead.

This is referred to as the transport configuration.

2.1.4 Layout of the researched tag

In order to determine how the system under testing could be attacked, it was paramount to first scan a tag registered in the system in order to gleam the used layout and data. This could also reveal information about how the systems handles authentication, as in, it only checks for the UID, or whether it uses a more sophisticated approach.
Scanning the tag with the MCT Android app reveals the layout shown in image 2.6. The screenshot does only show sector 0 through 6, the other sectors are not included in this thesis as they do not hold any relevance for the work and do not differ from sector 2.

The first thing that catches the attention is the missing B-key for sector 0 and 1 and the red sector 5, indicating it could not be read. These 3 sectors are also the relevant ones for this research as they hold the UID as well as data used by the system. This assumption can be made, as the transport configuration uses default keys, different access bits and does not contain data in the sectors. At first glance making reasonable statements about the format of the data is not possible, and will be discussed in a later chapter, where dynamic analysis is used to gleam the format.
Merely one observation can be made, and is highlighted in green in the image 2.6, the combination of the hexadecimal digits 0x0d0518. An important note for this combination is, that the date this tag was activated at, at an online terminal, and was read, was the 13th May 2024. The attentive reader might already have seen the relation. Converting those digits to integer numbers to base 10, they are 130524, which correlates to the date. Confirming this by scanning the tag the next day and reading again, this hypothesis holds true. This "field" in the data holds the last date the key was activated, which then might be used by the readers, in order to check, whether the key did not "expire" and was valid for the day.

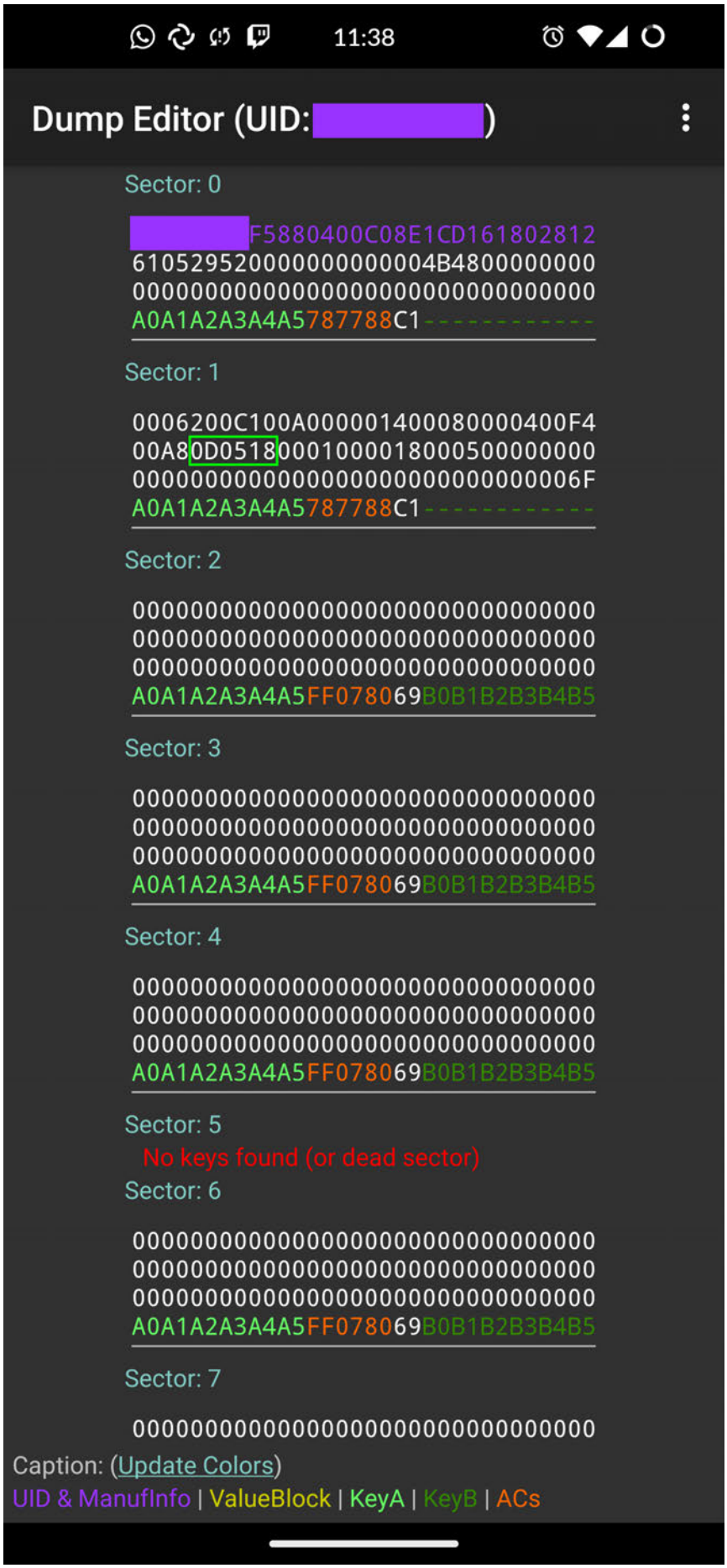

Figure 2.6: Sector 0 of a MIFARE Classic 1K tag registered in the researched system

Source: Own image

2.2 The attack

2.2.1 The planned attack: Exploiting cryptographic weaknesses and binary search

The system under testing consists of several offline key readers as well als online terminals connected to a management system. The offline key readers do not have any possibility of connecting to the management system, that is reserved for the few online terminals. The investigation was conducted without access to any internals of the system, i.e. no management system, software, firmware or not-publicly-accessible documentation.

During the investigation in the system, different possible approaches were considered. The first approach, which is applicable to any radio frequency (RF) technologies, is a relay attack, where a device acting as a scanner scans a tag and relays the signal to another device acting as a tag. Using this technique no encryption has to be broken[4] and attacks can be carried out over arbitrary distances. This attack does only provide the permissions of the relayed tag, which, depending on the scenario can or cannot be beneficial. The proposed attack in this thesis is reverse engineering the tag, giving the ability to create tags with arbitrary permissions, potentially leading to master access to all locks in the system. This attack has more prerequisites than the relay attack, as, in addition to gaining access to a tag, like with the relay attack, also requires breaking the encryption and reverse engineering of the format/system used by the electronic locking system. Each of those aspects will be focus of the next two chapters.

2.2.2 Cracking the keys

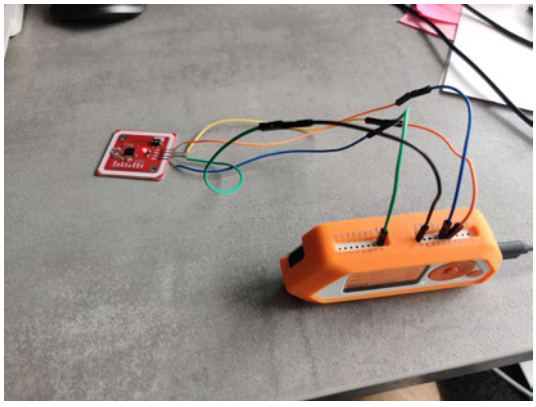

Figure 2.7: Setup for dumping the tag data
Source: Own image

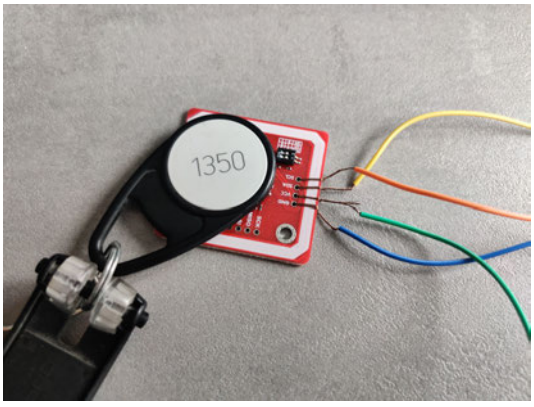

Figure 2.8: Setup for dumping the tag data
Source: Own image

In order to read the data of the tags, the non-default keys have to be found. This is possible, because the used cryptography library of the MIFARE Classic tags, `CRYPTO1`, MIFARE's own, proprietary, cryptography implementation, has a security flaw, where the used number generation has a rather small keyspace of 2^{16}. This, together with other vulnerabilities, and techniques, like measuring time distances between two authentication attempts, in order to calculate the internal

[4] At least in this scenario, there are techniques to mitigate replay and relay attacks

encryption register, can be abused, in combination with an already known key, to perform a "nested attack", as described in "Wirelessly Pickpocketing a Mifare Classic Card".[10] This allows to crack the keys for all sectors in a few minutes.

At the time of writing no method was found to execute this attack using a smartphone, so dedicated hardware, i.e. a badge reader was needed, in order to read and attack the tag from a computer. The "mfoc" tool[5] from the "nfc-tools" project was used. As the attack requires a pre-known key, the tool provides a few hard-coded default keys, but also supports specifying keys from the command line or from a key file. Just executing the tool with the default options yields following, successful, result, returning all missing keys:

```
$ mfoc -O key-dump.mfd
Found Mifare Classic 1k tag
ISO/IEC 14443A (106 kbps) target:
    ATQA (SENS_RES): 00  04
* UID size: single
* bit frame anticollision supported
       UID (NFCID1): xx  xx  xx  xx
      SAK (SEL_RES): 08
* Not compliant with ISO/IEC 14443-4
* Not compliant with ISO/IEC 18092

Fingerprinting based on MIFARE type Identification Procedure:
* MIFARE Classic 1K
* MIFARE Plus (4 Byte UID or 4 Byte RID) 2K, Security level 1
* SmartMX with MIFARE 1K emulation
Other possible matches based on ATQA & SAK values:

Try to authenticate to all sectors with default keys...
Symbols: '.' no key found, '/' A key found, '\' B key found, 'x' both keys found
[Key: ffffffffffff] -> [................]
[Key: a0a1a2a3a4a5] -> [//xxx.xxxxxxxxx]
[Key: d3f7d3f7d3f7] -> [//xxx.xxxxxxxxx]
[Key: 000000000000] -> [//xxx.xxxxxxxxx]
[Key: b0b1b2b3b4b5] -> [/xxxx.xxxxxxxxx]
[Key: 4d3a99c351dd] -> [/xxxx.xxxxxxxxx]
[Key: 1a982c7e459a] -> [/xxxx.xxxxxxxxx]
[Key: aabbccddeeff] -> [/xxxx.xxxxxxxxx]
[Key: 714c5c886e97] -> [/xxxx.xxxxxxxxx]
[Key: 587ee5f9350f] -> [/xxxx.xxxxxxxxx]
[Key: a0478cc39091] -> [/xxxx.xxxxxxxxx]
[Key: 533cb6c723f6] -> [/xxxx.xxxxxxxxx]
[Key: 8fd0a4f256e9] -> [/xxxx.xxxxxxxxx]

Sector 00 - Found   Key A: a0a1a2a3a4a5 Unknown Key B
Sector 01 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 02 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 03 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 04 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 05 - Unknown Key A               Unknown Key B
Sector 06 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 07 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 08 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 09 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 10 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 11 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 12 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 13 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 14 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5
Sector 15 - Found   Key A: a0a1a2a3a4a5 Found   Key B: b0b1b2b3b4b5


Using sector 00 as an exploit sector
Sector: 5, type A, probe 0, distance 12243 .....
Sector: 5, type A, probe 1, distance 12249 .....
...
Sector: 5, type A, probe 37, distance 12753 .....
Sector: 5, type A, probe 38, distance 12245 .....
  Found Key: A [7312c2efdd66]
  Data read with Key A revealed Key B: [000000000000] - checking Auth: Failed!
Sector: 0, type B, probe 0, distance 12237 .....
```

[5] Mifare Classic Offline Cracker; More at: https://github.com/nfc-tools/mfoc/tree/master

```
...
Sector: 0, type B, probe 5, distance 12251 .....
  Found Key: B [f7d124e09356]
Sector: 5, type B, probe 0, distance 12249 .....
...
Sector: 5, type B, probe 11, distance 12245 .....
  Found Key: B [52dfa63b1177]
Auth with all sectors succeeded, dumping keys to a file!
...
```

Listing 2.1: mfoc command to recover keys from a MIFARE classic tag

The middle block lists all the sectors and whether or not a hard-coded or provided key is valid for that sector. In accordance with the MCT screenshots 2.6 and 2.4, the B-key for sector 0, as well as both keys for sector 5 are missing, as they are not default keys. The last block then logs the attempts of recovering the keys and showing the recovered keys. These keys then can be taken and stored in the MCT app, allowing it to also read those sectors, allowing read/write access to all sectors on the chip, enabling the reverse engineering process discussed in the next section.

2.2.3 Reverse engineering and developing a master key

With the keys acquired in the previous section, the extracted data from the tag now looks like this:

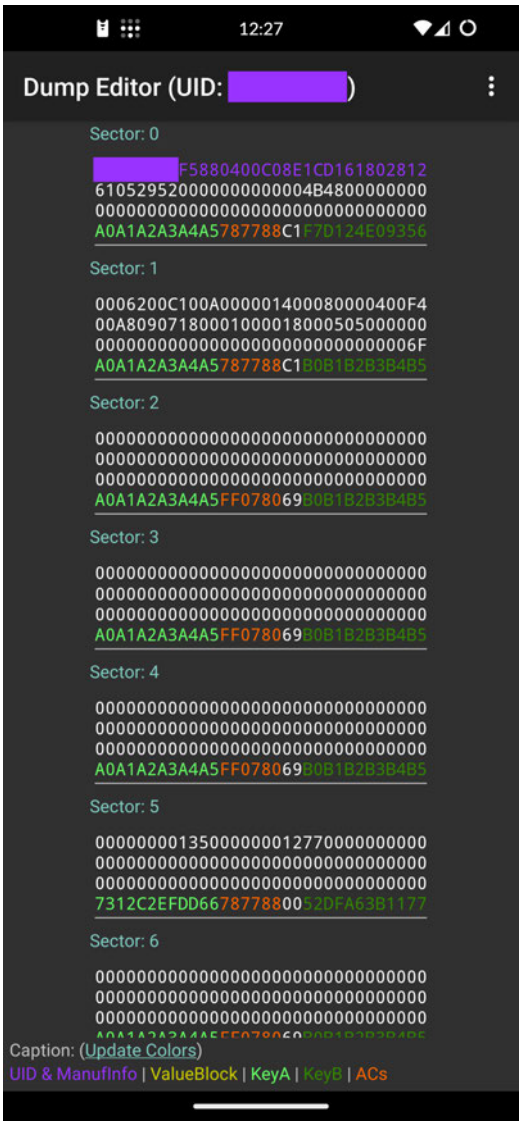

Figure 2.9: NFC tag read with all keys cracked
Source: Own image

Sector 0 and 1 did not change, except the B-keys are now also displayed, giving write capabilities to those sectors. The most notable thing: sector 5 is now on full display, with both keys being available. Now that every sector can be read and every key has been exposed, therefore every sector can be written to, building a master key is just an exercise of figuring out the format of the data on the tag and the data expected by the system.

At first it was considered, how the data could be stored on the tag. Single room numbers written plain into the memory of the key would be easy for humans to decipher, but would introduce needless complexity on the reader part, by requiring an extra parser and would take quite a lot of space, as, such systems could be deployed in environments with several hundred doors and access permissions. Those problems, together with the data read of the tag, seem to suggest, that this is probably not the approach used here.
Another approach would be to use the tags just for their UID, storing them on the readers for the doors the tag should have access to. But this also poses quite a challenge, as then, every time

a tag should be able to access a new door, someone would have to physically flash every reader with the new UID. This could, of course, be circumvented by attaching the readers to a network. But the tested system does not have network capabilities for the readers, and it would introduce other challenges.
The approach with the fewest space requirements is storing permissions as bitfields, where every reader has a specific bit location on the tag it checks for. If that bit is set, it grants access. This removes the need for a special parser, readers only have to be configured once and it does not take huge amount of space.

Now the problem becomes figuring out the bit pattern required for master access. Further complicating this is the fact, as can be seen in image 2.9, 3 different sectors hold data. Where does the bit pattern need to go in order get picked up by the reader and grant access?

There exist different approaches to solve this problem.
First, large amount of data can be collected, by scanning many different tags and taking note of every permission this tag has. This is really only an option, if one has (legal) access to this amount of data, which a black box engagement basically never has.
Second, a fuzzing setup could be built. Fuzzing is a technique, "which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion."[9] In this case, the fuzzing approach would focus mostly on the "data injection in an automated fashion"-part of the definition, as it is not really concerned with finding implementation bugs. This setup could consist of a computer with hardware capable of emulating a MIFARE tag, repeatedly sending different bit patterns to an attached reader, waiting for positive responses, trying to gather information on the used format. This would require time and a reader enrolled in the system, which, again, a black box engagement would not necessarily have.

The third approach, which was used for this attack, was simple brute force using a binary search over the sections. Discarding the first sector, for now, as a management sector, not relevant for permissions, the process was pretty straight forward. Setting half a sector, while being as greedy as possible in setting bits, checking how the reader reacts, repeat that till the key works. While this is not a sophisticated attack, it yielded results without a lot of complexity in a short amount of time.
Turns out sector 0 is indeed not used for access permissions, only sector 1 and 5 were relevant, and even those not in their entirety. Configuring the tag like shown in image 2.10 gives access to every office in a section, all sections, underground parking and the basement. In the spirit of transparency, not all doors could be unlocked, at least two doors are known, which elude access. It is possible that the payload is only missing a bit, as all other door could be unlocked with the same technique, and those doors do not exhibit signs, like being high-security or guarding of special areas, that could justify special requirements. Still, all other access gained still proves the viability and severeness of the attack.

The end result looks like this:

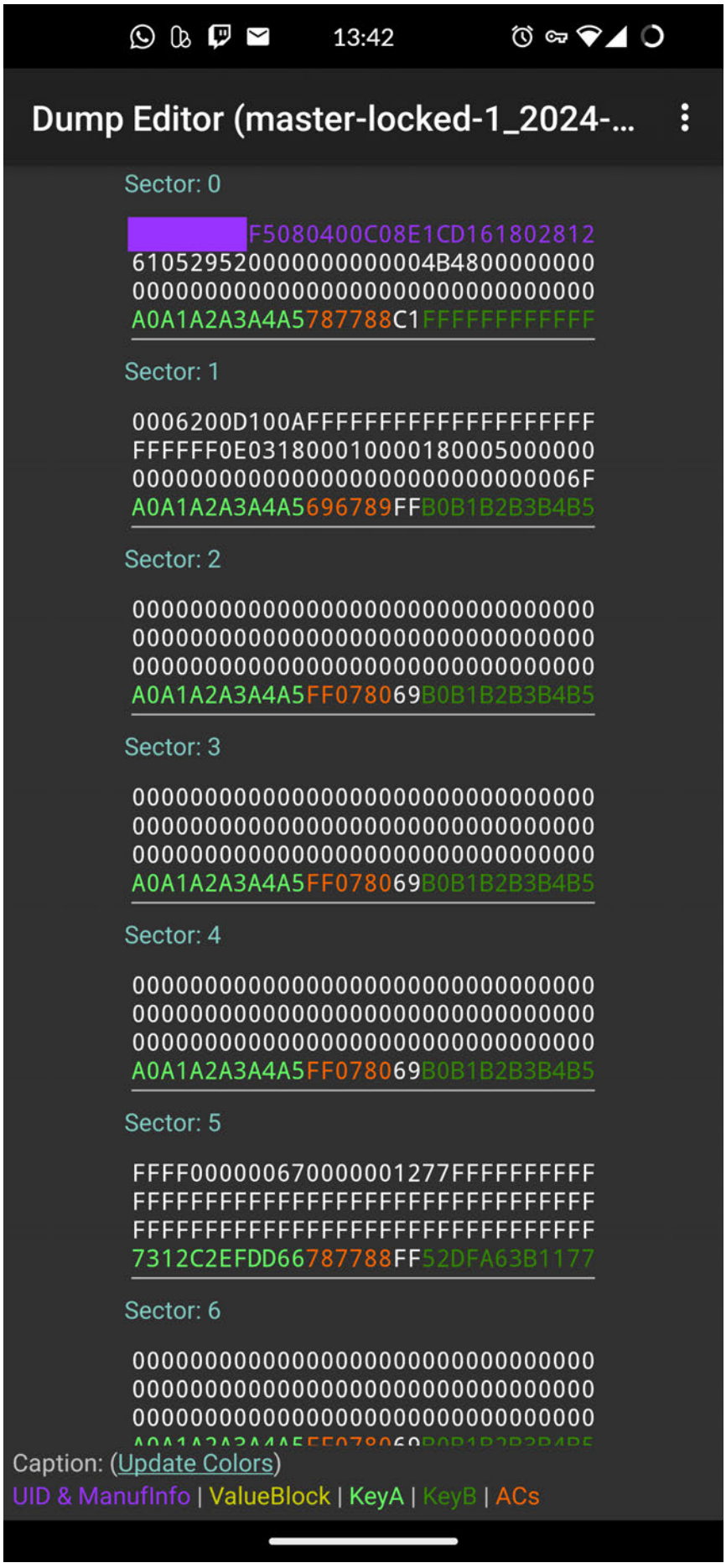

Figure 2.10: NFC tag with master permissions

Source: Own image

2.2.4 Viability and conclusion

While this attack provided, master access, with exception to the doors mentioned before, to the building, is it a viable attack that can be executed in the field?
First, the attack requires access to a tag. This can be accomplished either through stealing (even a few minutes suffice to crack the keys and make a clone), or having access to a tag organically, like working in the environment or being resident in a hotel. Another possibility would be to place a card skimmer over a legitimate reader, which is easy to do with conventional hardware like a microcontroller, which is not very noticeable and allows a bunch of other attack at the same time, like cloning, man-in-the-middle attacks, and the likes.
Furthermore, at least one key needs to be known for the attack outlined above. Most cards probably will not use every sector, so integrators potentially will not implement the system with due diligence, like changing their default keys. And even if they opt to change it, there are attacks described in "Wirelessly Pickpocketing a MIFARE Classic Card"[10], that allow cracking a key in 15 minutes[6], even if no other keys are known.
Another obstacle is the implementation of the system. The system as described in 2.1.1 is only one possible implementation. There can, and are, many different possible solutions involving MIFARE Classic cards. But given the relative insecurity of the cards themself, each and every one of them is at least vulnerable to cloning, if not more elaborate attacks, potentially granting master access like shown in this thesis.

This thesis shows, that it is possible to take a tag, and, without prior knowledge of and access to the system, build a replica of a key granting master access to the building. In whole, the process took two working days, from start to finish, demonstrating the viability of the attack.

2.3 The alternative: MIFARE DESfire

Knowing about the mentioned security issues with their Classic lineup, NXP created several replacements, mainly the DESfire and Plus family, as mentioned on the MIFARE Classic webite.[7]
The main focus here is on DESfire, as this is what was used in the successor of the tested system.[8]
DESfire, according to the website, offers a "Broad choice of open crypto algorithms based on DES, 2K3DES, 3K3DES, or AES".[23]. With DES and 3DES being long considered insecure or just currently having been deprecated and disallowed by NIST[20][6], it leaves only the AES standard, which is still the recommended encryption standard by NIST, not yet withdrawn or deprecated at the time of writing, as the only safe encryption standard for the tags, possibly opening systems up for compromise, if not considered carefully and implemented correctly.
While there are some attack, as outlined in "An investigation of possible attacks on the MIFARE DESFire EV1 smartcard used in public transportation"[8], they only operate on a smaller scale, as in a person trying to circumvent paying a fee for public transport, or charging up more than they actually paid, and do not offer a way for larger scale attacks, as outlined in this thesis. Furthermore, there are some side-channel attacks, as outlined in "Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World"[16] able to recover the whole 112-bits of the 3DES

[6] As stated in the paper
[7] Found here: https://www.nxp.com/products/rfid-nfc/mifare-hf/mifare-classic:MC_41863
[8] No further investigation went into that system, only cursory, revealing the tag hardware, prompting research into other implementations

secret key, which, if executed correctly, could allow attacks as shown here, but still not as easily as the attacks presented in this thesis. The attacks for recovering the DES key still required hardware worth 3000$, as well as several hours[16], which will not hinder motivated malicious actors, but, for now, probably enough to thwart drive-by attacks. Furthermore they focus on the DES part, which is insecure. Should the implementation rely on AES encryption, the tag is still considered secure.[9]

In conclusion, while there are attacks for the DESfire lineup, they rely on the usage of DES by the system integrator, require specialised hardware and time, or focus only on attacks for personal cards by the owner. If system integrators opt to use AES and a proper integration into the locking system, the cards can still be considered secure, but proper usage is paramount.

2.4 Results

The above chapter analyses an electronic locking system relying of MIFARE Classic tags in order to provide authentication. For this it uses 3 different types of interfaces, the tag, the door readers and an online terminal integrated into a management system. The system, relying on MIFARE Classic, also uses MIFARE's own proprietary cryptography implementation `CRYPTO1`, which has several severe security flaws, as mentioned above, enabling complete compromise of the card. While the attack relied on the fact that at least one key is known in advance, other measures could have been taken to recover the keys, even if no other key was known. Using the `mfoc` tool, all keys for all sectors could be recovered, enabling full read/write access to the cards, making creating a master key feasible. Instead of reverse engineering the exact format used by the system, a brute force approach was selected. Using binary search over the sectors, toggling bits lead to a satisfying result in a short time, in the end granting master access to the locking system. This also revealed the storage format used by the system, which is bitfields encoding the room numbers. Each door lock has a specific bit it checks for, if the bit is set, it grants access, if not it denies access. Knowing this also provides an explanation for doors not affected by the master key: They are missing a bit somewhere in the payload.
The system cannot be implemented in a secure way, as long as it depends on data on the card, as that data can be arbitrarily changed or cloned, because of a fundamentally flawed cryptography implementation.

The suggested alternative, MIFARE DESfire, as used in the alternative system, relies on open and tested cryptography implementations, DES, 3DES as well as AES. Only one of those implementations, AES, is not yet deprecated and forbidden by NIST. DESfire is still considered secure, with not many vulnerabilities published for the technology. While only secure as long as sensible defaults are implemented, as in no default keys or weak keys, DESfire can rely on the confidentially of its payload, for building electronic locking systems on top of it.

[9] As long as it is implemented securely, i.e. no standard keys

3 The case Welock

The second locking system, or rather lock, was the "Welock Smart Lock Fingerprint Door Lock Cylinder SECBN51". As the name suggests, it is an electronic door lock, including cylinder.

3.1 The Welock electronic lock

3.1.1 The hardware

Figure 3.1: Welock front
Source: Own image

Figure 3.2: Welock side front
Source: Own image

Figure 3.3: Welock side
Source: Own image

The lock shows a similar structure to the door readers from the previous chapter. It features a main body, handling the authentication, on the "outside", a cylinder and a simple handle that can be operated without authentication.
For authentication it features a fingerprint reader, as well as a NFC reader. Further it supports unlocking via Bluetooth, but that will be highlighted in the next chapter.

3.1.2 The Software and Bluetooth communication

3.1.2.1 The app

In addition to the hardware authentication methods, like fingerprint and NFC tags, Welock also offers an app integration for their smart locks, supporting Android and iOS.
The app itself offers management options for the lock, ranging from rotating passwords, adding or deleting tags, fingerprints, as well as an access log of the lock. Communication between the app and the lock happens over Bluetooth.

The app is built using the Flutter framework, as can be confirmed by decompressing the APK file. This has certain implications for the research and attack of the lock. Those implications will be discussed in the next chapter.

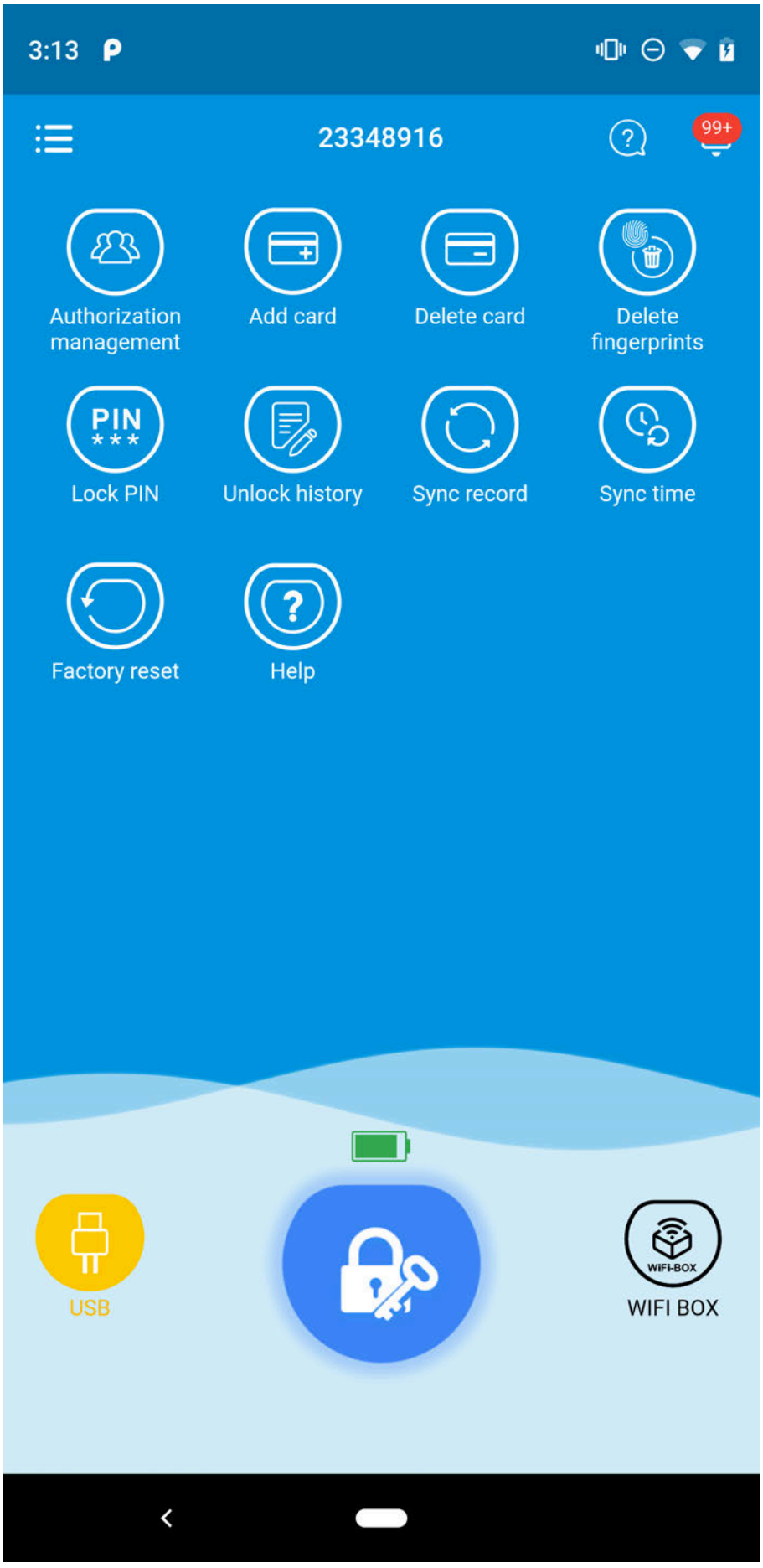

Figure 3.4: Screenshot of the Welock-app user interface
Source: Own image

3.1.2.2 What is Flutter and Dart

> "Flutter is an open source framework by Google for building beautiful, natively compiled, multi-platform applications from a single codebase."[7]

Meaning Flutter provides a framework for developers to build a single codebase for their app, that is able to deploy and run on a number of different platforms. Supported platforms include Android, iOS, Windows, Linux and the web.[25]
To achieve this portability, Flutter sets on Dart, a language designed to be "approachable, portable, and productive [...] for high-quality apps on any platform".[3] Dart offers two approaches for running/compiling applications. On one hand, it can compile to machine code, supporting x86, ARM, RISC-V, as well as WebAssembly. On the other hand, it offers a VM with Just-In-Time (JIT) compilation for fast feedback cycles during development. Both approaches do still need the Dart runtime, which manages memory, with garbage collection, performing runtime type checks and code isolation.[4] Further, Dart uses a custom calling convention[10], instead of platform specific ones. Both Flutter and Dart are open source projects.

The application shipped with Flutter consists of two parts. One being the runtime itself, called `libflutter.so` on Android, and the other being the application code, called `libapp.so` on Android. This `libapp.so` contains a so called snapshot, which is a binary representation of the application state right before the main function is executed. So it contains data structures, variables, classes and function definitions, as well as the executable code for each function.[5]
The Flutter/Dart project also provides a standard library for the projects using it. It implements a network library, including SSL handling. This is required, as Flutter runs on many different platforms, and each of those actions is a very platform specific process. In order for Flutter projects to be portable, Flutter itself has to provide methods, which abstract those platform specific actions. This also poses a challenge for reverse engineering and researching Flutter applications. A normal approach on Android would be to install a global proxy, pipe the connection out using ADB[11], and capture the traffic on a computer using something like BurpSuite. In order for that to work, the device has to trust a BurpSuite certificate, otherwise the proxied connections would fail, which would be installed in the device certificate store, which then would be used by applications using the OS API. As Flutter implements those parts itself, it does not respect proxy settings and does not use the global certificate store. Because of that, intercepting traffic from Flutter applications requires patching the engine itself, overwriting the certificate validation, as well as installing a global proxy using a rooted device.

While both codebases are not brand new, they have their first git commits on GitHub dated to 2015[12], the current state of reverse engineering tools is lackluster. Mainly two projects exist, one focusing on static analysis, Doldrums[13], and one focusing on dynamic analysis, reFlutter[14].

[10] There exists an issue in the GitHub repository to use conventional calling convention
[11] Android debug bridge
[12] Dart: https://github.com/dart-lang/sdk/commits/main/?since=2015-05-16&until=2015-05-16, Flutter: https://github.com/flutter/flutter/commits/master/?since=2015-03-06&until=2015-03-06
[13] Found at: https://github.com/rscloura/Doldrums
[14] Found at: https://github.com/Impact-I/reFlutter

Doldrums has not received a new commit in two years, only supporting versions `2.10`, while the newest stable Flutter version is `3.22`. For the versions it does support, it is able to parse the snapshot, extracting classes and function signatures together with the offset into the shared object, to where the code associated with the function lives.[13]

reFlutter does support version `3.22`, but not every version leading up to it. It works by patching the engine, so the `libflutter.so` file, directly. This enables things like dumping snapshot information like classes and function signatures, without implementing a custom snapshot parser, but by relying on the Flutter engine itself. It also supports intercepting network traffic by patching the engine to use a proxy and overwriting the certificate validation.[11]

There don't exist other prominent tools for Flutter/Dart reverse engineering next to those. The general problems those projects face, are, first, they have to cover a big base, potentially multiple platforms and architectures and a VM and native code. Second the snapshot format and engine has no standard, so is subject to change, with potentially no easy way to adapt to those changes, so the developers for reverse engineering tools are facing a constant up hill battle.

3.2 The attack

3.2.1 The planned attack: Exploiting the Bluetooth communication

The lock offers three different, attack surfaces. First, the NFC tag itself. As shown above NFC tags can be exploited to gain unauthorized access to locking systems. Second, the fingerprint sensors. While there exists attacks to bypass fingerprint readers, as shown in "A Touch of Pwn - Part I" from BLACKWING INTELLIGENCE[12], they are quite involved and therefore were not the first choice for this research. The third vector was the app.
As it allows to open the lock using Bluetooth, as well as manage fingerprints, pins, and tags, it definitely needs to contain the logic to communicate with the lock, and, as it is using Bluetooth, the chances to replicate those communications, skipping any authorization, seemed in the realm of the possible.

As the app should have all the logic needed, the approach would be to decompile the application and intercept Bluetooth traffic, in order to find the routines handling communication with the lock, as well as available commands. If authentication is needed for communication with the lock, this approach would also reveal that, as the communication and authentication would still have to happen over Bluetooth. If the communication is encrypted, the app would contain routines to encrypt the data, possibly even encryption keys, if the implementation does not follow secure practices.
Normally the approach would unpack the APK[15], exposing all configuration files and assets shipped with the application, potentially already revealing API keys, encryption keys, used strings in the application, et cetera. Then an APK decompiler like JADX[16] could be used to further analyze the application, showing used packages, down to decompiled source code. As Java, the lingua franca for Android application[17], compiles down to a bytecode format, still relatively close to the

[15] An APK is just ZIP file containing java bytecode and assets required to run the application
[16] Found at: https://github.com/skylot/jadx
[17] There, of course, exist other languages, like Kotlin or Dart

original source code, the decompiled code is also pretty close to the original, even keeping original function names, if no further actions have been taken to obfuscate the application. From here on out the application could be reverse engineered like any standard reverse engineering project. After the communication, authentication and encryption routines have been reverse engineered and circumvention methods found, it would then be possible to write an independent implementation, giving access to arbitrary locks, not just the one linked to the app.

But before the next chapter focuses on the problems encountered with that approach, the following will be a short interlude discussing the NFC tags.

The NFC tag

As the previous chapter discussed unauthorized access using NFC tags, the NFC implementation of this lock was also examined. Scanning the tag with the MCT app revealed it to be a MIFARE Classic tag, meaning it is vulnerable to all attacks outlined in the first chapter. All sections were blank, suggesting the lock only uses the UID for authentication. This is susceptible to cloning or brute forcing attacks, as the UID can be changed freely on non-official MIFARE Classic cards.

3.2.2 Initial reverse engineering

The application was installed on a hardware Android device. The device was rooted to gain deeper access to the system, in order to facility memory layout research and traffic interception. Installed tool include Magisk[18] for root management and the Frida server for dynamic instrumentation of applications.

Decompiling Android applications normally follows a straight forward procedure. Decompress them, as they are just ZIP-archives, load the class files into a decompiler and you have the source code. Taking a look at the directory listing of the decompressed APK 3.1, it shows a pretty standard listing of files for an APK. It shows the `AndroidManifest.xml`, assets, configurations as well as classes.

```
$ ls
total 12M
-rw-r--r-- 1 zeno zeno  24K Dec 31  1979 AndroidManifest.xml
-rw-r--r-- 1 zeno zeno   53 Jan  1  1970 androidsupportmultidexversion.txt
drwxr-xr-x 1 zeno zeno   58 May 22 17:45 assets
-rw-r--r-- 1 zeno zeno 2.8M Dec 31  1979 classes2.dex
-rw-r--r-- 1 zeno zeno 7.9M Dec 31  1979 classes.dex
-rw-r--r-- 1 zeno zeno   74 Jan  1  1970 firebase-analytics.properties
-rw-r--r-- 1 zeno zeno   78 Jan  1  1970 firebase-annotations.properties
-rw-r--r-- 1 zeno zeno   76 Jan  1  1970 firebase-components.properties
-rw-r--r-- 1 zeno zeno   82 Jan  1  1970 firebase-datatransport.properties
-rw-r--r-- 1 zeno zeno   82 Jan  1  1970 firebase-encoders-json.properties
-rw-r--r-- 1 zeno zeno   72 Jan  1  1970 firebase-encoders.properties
-rw-r--r-- 1 zeno zeno   84 Jan  1  1970 firebase-encoders-proto.properties
-rw-r--r-- 1 zeno zeno   78 Jan  1  1970 firebase-iid-interop.properties
-rw-r--r-- 1 zeno zeno   98 Jan  1  1970 firebase-installations-interop.properties
-rw-r--r-- 1 zeno zeno   98 Jan  1  1970 firebase-measurement-connector.properties
drwxr-xr-x 1 zeno zeno 5.0K May 22 17:45 kotlin
drwxr-xr-x 1 zeno zeno   52 May 22 17:45 lib
-rw-r--r-- 1 zeno zeno  883 Jan  1  1970 messaging_event_extension.proto
-rw-r--r-- 1 zeno zeno 2.4K Jan  1  1970 messaging_event.proto
drwxr-xr-x 1 zeno zeno 4.4K May 22 17:45 META-INF
```

[18] Found at: https://github.com/topjohnwu/Magisk

```
drwxr-xr-x 1 zeno zeno   16 May 22 17:45 okhttp3
-rw-r--r-- 1 zeno zeno   94 Jan  1  1970 play-services-ads-identifier.properties
-rw-r--r-- 1 zeno zeno   82 Jan  1  1970 play-services-basement.properties
-rw-r--r-- 1 zeno zeno   74 Jan  1  1970 play-services-base.properties
-rw-r--r-- 1 zeno zeno   96 Jan  1  1970 play-services-cloud-messaging.properties
-rw-r--r-- 1 zeno zeno   96 Jan  1  1970 play-services-measurement-api.properties
-rw-r--r-- 1 zeno zeno   98 Jan  1  1970 play-services-measurement-base.properties
-rw-r--r-- 1 zeno zeno   98 Jan  1  1970 play-services-measurement-impl.properties
-rw-r--r-- 1 zeno zeno   88 Jan  1  1970 play-services-measurement.properties
-rw-r--r-- 1 zeno zeno  104 Jan  1  1970 play-services-measurement-sdk-api.properties
-rw-r--r-- 1 zeno zeno   96 Jan  1  1970 play-services-measurement-sdk.properties
-rw-r--r-- 1 zeno zeno   76 Jan  1  1970 play-services-stats.properties
-rw-r--r-- 1 zeno zeno   76 Jan  1  1970 play-services-tasks.properties
drwxr-xr-x 1 zeno zeno 1.1K May 22 17:45 res
-rw-r--r-- 1 zeno zeno 506K Dec 31  1979 resources.arsc
-rw-r--r-- 1 zeno zeno   62 Jan  1  1970 transport-api.properties
-rw-r--r-- 1 zeno zeno   78 Jan  1  1970 transport-backend-cct.properties
-rw-r--r-- 1 zeno zeno   70 Jan  1  1970 transport-runtime.properties
```

Listing 3.1: Directory listing of unpacked APK

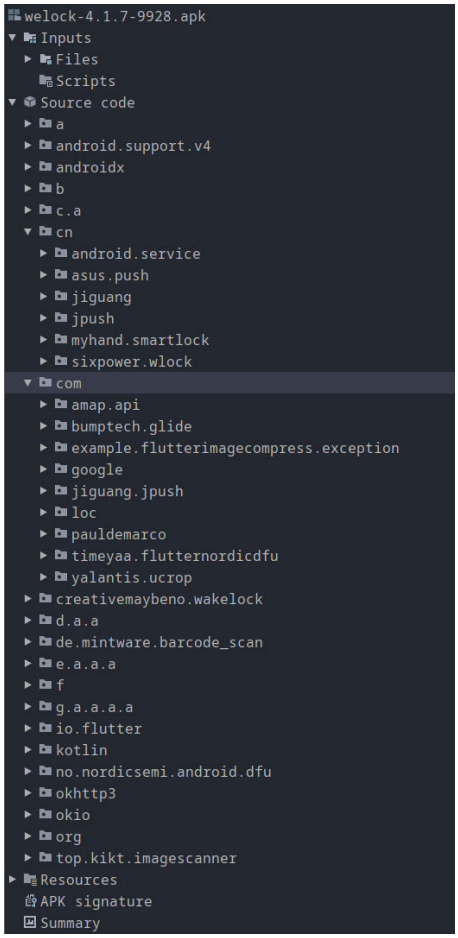

Figure 3.5: Package listing of the APK

Source: Own image

Opening the APK in JADX, an APK decompiler, the package listing, as can be seen in image 3.5, has obfuscated package names. Note, that the listing is sorted and grouped by the first part of the package name.[19] Further, the listing contains a `io.flutter` entry, suggesting it could be a Flutter application. To confirm that suspicion, there should exist a `libapp.so` and `libflutter.so` file in the APK. Showing the files in `lib/` reveals directories for different architectures, as seen in the first block of listing 3.3, containing the required files for Flutter, confirming, that at least a part of the application is implemented in Flutter. If the interesting part, the handling of the communication with the lock, is in there, is still to be determined.

Loading the `libapp.so`, which contains the application logic, into a decompiler, gives following analysis report shown in 3.6. It lists nearly 21000 functions, which is very hard to reasonably decompile and reverse engineer, especially, because the functions do not have names resembling their actual behaviour, and the analysis does not work properly. Taking a look at the following function signatures:

```
int64_t fcn.004ffc44(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4, int64_t arg5,
                     int64_t arg6, int64_t arg7, int64_t arg8);
uint64_t fcn.004ffd30(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4, int64_t arg5,
                     int64_t arg6, int64_t arg7, int64_t arg8);
int64_t fcn.004ffdb8(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4, int64_t arg5,
                     int64_t arg6, int64_t arg7, int64_t arg8);
```

[19] Java package names consist of multiple parts, concatenated by a dot (.).

```
int64_t fcn.00500444(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4, int64_t arg5,
                     int64_t arg6, int64_t arg7, int64_t arg8);
```

Listing 3.2: Decompiled function signatures

These signatures are just an excerpt of the almost 21000 functions. While not every function has eight function arguments, most signatures look like this. These functions do not actually use eight arguments. These erroneous analysis is caused by the fact the Dart VM uses a custom calling convention, which the decompiler does not know. Therefore the decompiler cannot reliably detect and distinguish function parameters and local variables. Dart instead of using X0-X8 for arguments[1], uses a custom stack, with the stack pointer residing in the X15 register.[20] So every argument access for a function works through a relative offset from the stack pointer, which can further be indirected to the global object pool, which again could be a pointer to a memory location holding the actual data.

```
$ ls -R lib
zeno@pleto | 16:21:00
lib:
total 0
drwxr-xr-x 1 zeno zeno 44 May 22 17:45 arm64-v8a
drwxr-xr-x 1 zeno zeno 44 May 22 17:45 armeabi-v7a
drwxr-xr-x 1 zeno zeno 44 May 22 17:45 x86_64

lib/arm64-v8a:
total 22M
-rw-r--r-- 1 zeno zeno  13M Dec 31  1979 libapp.so
-rw-r--r-- 1 zeno zeno 8.4M Dec 31  1979 libflutter.so

lib/armeabi-v7a:
total 20M
-rw-r--r-- 1 zeno zeno  14M Dec 31  1979 libapp.so
-rw-r--r-- 1 zeno zeno 6.0M Dec 31  1979 libflutter.so

lib/x86_64:
total 23M
-rw-r--r-- 1 zeno zeno  13M Dec 31  1979 libapp.so
-rw-r--r-- 1 zeno zeno 9.3M Dec 31  1979 libflutter.so
```

Listing 3.3: Directory listing of `lib/` of unpacked APK

All this makes it very hard for the decompiler to provide a proper decompilation and analysis, and it makes it very hard to determine the functionality that is actually in the Flutter code. Here the previously mentioned reverse engineering tools Doldrum and reFlutter come into play.
The snapshot analyzed here is Dart's own representation of this application. These two tools provide a way to extract classes, functions and code offsets from that file.
Running Doldrums with the following command line produces an output file containing all classes, function names and signatures, as well as the offset from the beginning of the file to where to code for said function exists.

```
python Doldrums/src/main.py app/lib/arm64-v8a/libapp.so fun-sigs.txt
```

Listing 3.4: Doldrum command line

As an example, the `StringBuffer` class and associated methods are shown:

```
class StringBuffer extends Object implements StringSink {

    StringBuffer StringBuffer.(DynamicType, DynamicType) {
```

[20] As specified by the source code here: https://github.com/dart-lang/sdk/blob/b80e682cbf313691650cc9c45302811deb710aa0/runtime/vm/constants_arm64.h#L49

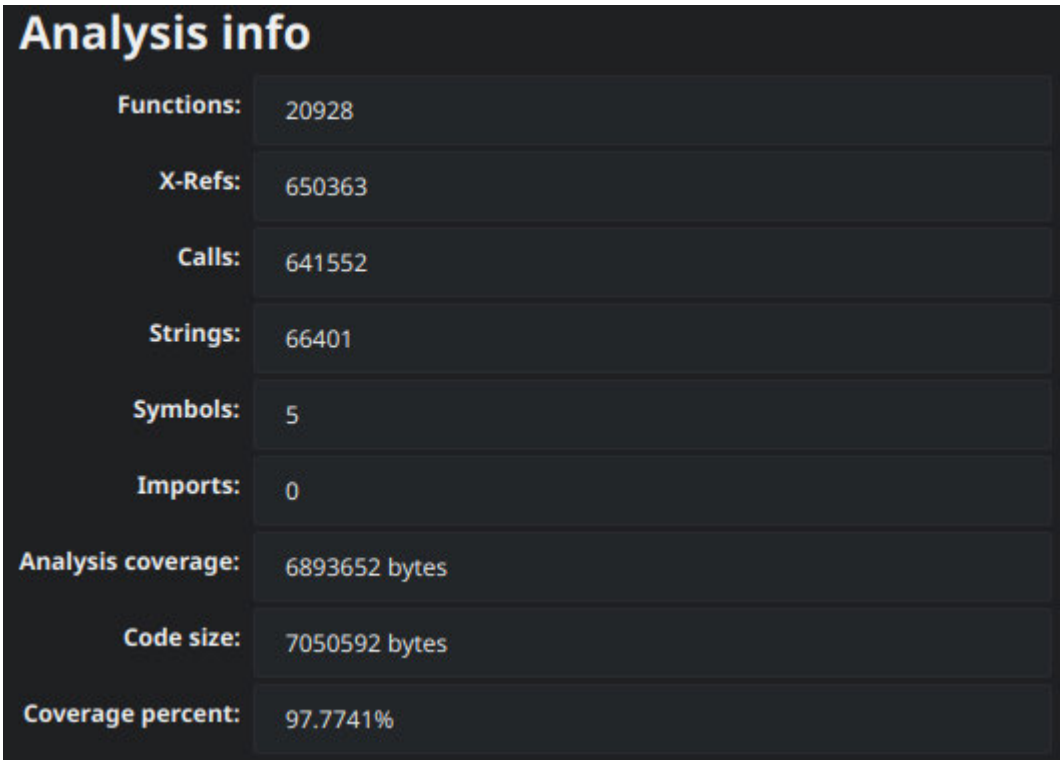

Figure 3.6: Analysis report of `libapp.so`
Source: Own image

```
    Code at absolute offset: 0x16a48
}

VoidType _ensureCapacity@0150898(DynamicType, DynamicType) {
    Code at absolute offset: 0x165bc
}

VoidType _consumeBuffer@0150898(DynamicType) {
    Code at absolute offset: 0x1683c
}

VoidType _addPart@0150898(DynamicType, DynamicType) {
    Code at absolute offset: 0x15b3c
}

VoidType _compact@0150898(DynamicType) {
    Code at absolute offset: 0x15d28
}

String _create@0150898(DynamicType, DynamicType, DynamicType) {
    Code at absolute offset: 0x16770
}

int get:length(DynamicType) {
    Code at absolute offset: 0x112d4
}

bool get:isEmpty(DynamicType) {
    Code at absolute offset: 0x15fa4
}

VoidType write(DynamicType, DynamicType) {
    Code at absolute offset: 0x1699c
}

VoidType writeCharCode(DynamicType, DynamicType) {
    Code at absolute offset: 0x163ac
}

VoidType writeAll(DynamicType, DynamicType, DynamicType) {
    Code at absolute offset: 0x160d4
}
```

```
    VoidType writeln(DynamicType, DynamicType) {
        Code at absolute offset: 0x16050
    }

    VoidType clear(DynamicType) {
        Code at absolute offset: 0x16020
    }

    String toString(DynamicType) {
        Code at absolute offset: 0x43c974
    }
}
```

Listing 3.5: Doldrum output for StringBuffer class

Comparing this to the Dart documentation[21] shows that almost all methods of the class are also present in the snapshot. The deviation might be cause by the implementation details, like the `isNotEmpty` field might be handled by the engine just negating the `isEmpty` boolean inline. Searching through the file for anything mentioning `lock`, `welock` and anything in between, with the regex `[we]*lock`, produces 218 matches, confirming, that probably all of the lock communication is handled in Flutter space. In the following only some are highlighted.

```
52410:    String get:src_lock_list_delete(DynamicType) {
52430:    String get:main_operator_reset_lock(DynamicType) {
52458:    String get:src_lock_set_transfer_tip(DynamicType) {
52478:    String get:pv_help_lock_detail_8(DynamicType) {
52502:    String get:src_lock_list_delete_tip(DynamicType) {
52510:    String get:src_scan_ble_lock(DynamicType) {
52514:    String get:src_connect_lock_failed(DynamicType) {
52542:    String get:pv_add_lock_by_gw(DynamicType) {

76862:class BleLockOperateFaceImpl extends Object implements LockOperationFace {
76864:    BleLockOperateFaceImpl BleLockOperateFaceImpl.(DynamicType, DynamicType, DynamicType) {
76884:    VoidType lockOpen(DynamicType, DynamicType) {
76968:    VoidType _sendActiveLock@1522360932(DynamicType) {
76992:    VoidType _sendResetLock@1522360932(DynamicType) {
77016:    VoidType activeLock(DynamicType, DynamicType) {
77028:    VoidType loadLockLog(DynamicType, DynamicType) {
77032:    VoidType resetLock(DynamicType, DynamicType) {
77056:    VoidType searchLock(DynamicType, DynamicType) {
77060:    VoidType lockOta(DynamicType, DynamicType) {

80756:class BleGwLockOperateFaceImpl extends Object implements LockOperationFace {
80786:    VoidType lockOpen(DynamicType, DynamicType) {
80806:    VoidType activeLock(DynamicType, DynamicType) {
80818:    VoidType loadLockLog(DynamicType, DynamicType) {
80822:    VoidType resetLock(DynamicType, DynamicType) {
80842:    VoidType setLockedDelayTime(DynamicType, DynamicType) {
```

Listing 3.6: Results for search for relevant names

The first block shows a lot of management functions, that somehow manage the locks and some lists, possibly connected to the UI. These methods are associated with a class called `S`, containing a lot of similar methods. The other two other blocks show a class and associated methods each implementing the same interface and implementing largely the same methods, handling the lock "directly", at least apparently implementing operations directly affecting the lock, like opening and resetting. Checking for those classes in the output of Doldrums, it shows a lot more methods for adding NFC tags, syncing the time, connecting, sending data, et cetera. The `BleLockOperateFaceImpl` class holds significantly more methods than

[21] Found at: https://api.flutter.dev/flutter/dart-core/StringBuffer-class.html

`BleGwLockOperateFaceImpl`. Welock also offers a gateway, making it possible to operate the lock from a distance, sending the commands over the internet to the gateway, which then relays it to the lock. The `BleGwLockOperateFaceImpl` class could be an implementation for the gateway.

The reverse engineering toolkit used for this project was the rizin project, with the cutter frontend[22]. The tool exposes an API to interact with a loaded project, allowing to automate everything the user is capable of doing. This includes function renaming. The script listed below uses a parser for the reFlutter dump format[23], written by Guardsquare, to extract function names and offsets, and then uses the rizin-API in order to automatically rename functions, giving them proper names, making decompilation and reverse engineering easier. This script can easily be adapted for any other decompiler offering an API.

```
#!/usr/bin/env python3

import sys
import os
from pprint import pprint
from flure.code_info import CodeInfo, ClassInfo, FunctionInfo
from flure import ReFlutterDumpParser
import rzpipe

def eprint(*args, **kwargs):
    print(*args, file=sys.stderr, **kwargs)

def main():
    if len(sys.argv) != 4:
        eprint("ERROR: wrong number of arguments")
        print(f"
                Usage: {sys.argv[0]} <path-to-function-signatures>
                <path-to-rizin-project>
                <path-to-libapp.so>
        ")
        sys.exit(1)
    dp = ReFlutterDumpParser(sys.argv[1])
    parsed = dp.code_info.dump()

    rz = rzpipe.open(sys.argv[3])
    print("Loading rizin project...")
    rz.cmd(f"Po {sys.argv[2]}")
    print("Starting rename...")

    for classo in parsed["classes"]:
        class_name = classo["name"]
        for function in classo["functions"]:
            function_name = function["name"]
            function_offset = function["offset"]
            function_signature = function["signature"]
            full_function_name = f"{class_name}::{function_name}".replace("@", "")
                .replace(">", "greater")
                .replace(">=", "geq")
                .replace("<", "less")
                .replace("<=", "leq")
                .replace(">>", "bitShiftRight")
                .replace("<<", "bitShiftLeft")
            rz.cmd(f"s sym._kDartIsolateSnapshotInstructions + {hex(function_offset)}")
            rz.cmd(f"afn {full_function_name}")

    print("Finished renaming!")
    print("Saving rizin project...")
```

[22]Found at: https://rizin.re and https://cutter.re
[23]Found at: https://github.com/Guardsquare/flutter-re-demo

```
48      rz.cmd(f"Ps {sys.argv[2]}")
49      print("Done!")
50
51  if __name__ == "__main__":
52      main()
```

Listing 3.7: Python script for function renaming

In order to perform this renaming, first the APK has to be patched, using the reFlutter project, to produce the dump with all informations for classes, functions and offsets. Here, only the `Display absolute code offset for functions` functionality of reFlutter is used. reFlutter also supports patching the Flutter runtime, in order to allow traffic inspection, but this is not a needed functionality for this procedure.

```
$ reflutter WeLock_4.2.0_APKPure.apk

 Choose an option:

 1. Traffic monitoring and interception
 2. Display absolute code offset for functions

 [1/2]? 2

 This mode is only for dump and offset output, slow application operation is possible (network patch is stil

Example: (192.168.1.154) etc.
Please enter your BurpSuite IP: 192.168.178.20

 Wait...


SnapshotHash: 8ee4ef7a67df9845fba331734198a953
The resulting apk file: ./release.RE.apk
Please sign,align the apk file

Configure Burp Suite proxy server to listen on *:8083
Proxy Tab -> Options -> Proxy Listeners -> Edit -> Binding Tab

Then enable invisible proxying in Request Handling Tab
Support Invisible Proxying -> true
```

Listing 3.8: `reFlutter` output while patching

reFlutter then produces a patched APK file, which now has to be signed, in order to be installable on an Android phone. For that the uber-apk-signer utility is used.[24] This produces a signed and aligned APK file, which now can be installed on an Android phone.

```
$ uber-apk-signer --allowResign -a release.RE.apk
source:
        /home/zeno/projects/smart-lock/app-4.2.0
zipalign location: BUILT_IN
        /tmp/uapksigner-16657619045053773144/linux-zipalign-33_0_21467853636692236868.tmp
keystore:
        [0] 161a0018 /tmp/temp_4212246617419300370_debug.keystore (DEBUG_EMBEDDED)

01. release.RE.apk

        SIGN
        file: /home/zeno/projects/smart-lock/app-4.2.0/release.RE.apk (34.36 MiB)
        checksum: bcb69b0f4ca2d0e78141ab2a6e2c2187951622fe7edab785159991fbe25602e9 (sha256)
        - zipalign success
        - sign success

        VERIFY
        file: /home/zeno/projects/smart-lock/app-4.2.0/release.RE-aligned-debugSigned.apk (34.36 MiB)
        checksum: 30d9d545ca149a4faa08a174669f419abbca846cb1a9a337d3faf7188040ce68 (sha256)
        - zipalign verified
        - signature verified [v1, v2, v3]
```

[24] Found at: https://github.com/patrickfav/uber-apk-signer

```
                53 warnings
                Subject: CN=Android Debug, OU=Android, O=US, L=US, ST=US, C=US
                SHA256: 1e08a903aef9c3a721510b64ec764d01d3d094eb954161b62544ea8f187b5953 / SHA256withRSA
                Expires: Thu Mar 10 21:10:05 CET 2044

[Fri Aug 02 11:43:29 CEST 2024][v1.3.0]
Successfully processed 1 APKs and 0 errors in 1.25 seconds.
```

Listing 3.9: `uber-apk-signer` output while signing the APK

The installation is done using ADB, the android debug bridge.

```
$ adb install release.RE-aligned-debugSigned.apk
```

Listing 3.10: `adb` command for installing the APK

The patched Flutter runtime then produces a file in `/data/data/cn.sixpower.wlock/dump.dart`, which can be extracted using ADB and then fed into the python script listed in code listing 3.7. The script takes in the dump file, a path to an already existing rizin project, where the `.so`-file of the application was already loaded into, and the path to the application Flutter code. After execution, the script should have modified the existing project, which now should have renamed functions.

```
$ adb -d shell "su -c cat /data/data/cn.sixpower.wlock/dump.dart" > dump.dart
$ python3 rename_functions.py app-4.2.0/dump.dart
        app-4.2.0/renamed-libapp-4.2.0.rzdb
        app-4.2.0/lib/arm64-v8a/libapp.so
```

Listing 3.11: Commands for retrieving and parsing the dump data

Even after the previous mentioned procedure for renaming the functions, making decompiled code a bit easier to follow, the other problems, like custom calling convention, object indirection, together with a custom stack implementation, makes reverse engineering extremely hard.
Static analysis could be supplemented by, like done above, function renaming from the parsed snapshot, and making changes to the decompiler, i.e. plugins, or the binary, i.e. patching the stack pointer, in order to make analysis more successful. This can be further improved by developing tools enabling runtime analysis and inspection, which, in turn, would require resolving the runtime inspection and well as acquiring the proper stack position for variables and return values from routines during runtime. This is outside of the scope of this thesis, but could be grounds for future work.

3.2.3 Intercepting the Bluetooth communication

The goal of intercepting/extracting the Bluetooth communication between the app and the lock with the limitations listed above, would require a lot of work, if approached from the Flutter side. The app sends the Bluetooth commands to interface with the lock, meaning opening, closing, registering a new tag, syncing the time, all from inside Flutter. But there is no way in the Flutter standard library to interface with Bluetooth devices. This functionality is provided through third party libraries.
Bluetooth also is an inherently platform-specific operation, meaning there has to be interface code somewhere in the stack, which has to be in Java, as that is the language used to interface with the Android operating system. And reverse engineering and dynamic instrumentation tooling for

Java on Android exists and is in a mature state, with lots of knowledge and tooling available. For example the JADX decompiler and Frida as a dynamic instrumentation toolkit.
The library providing Bluetooth functionality for Flutter application would then consist of at least two parts. The Flutter code, providing the API used by Flutter applications, and at least one platform-specific implementation, to pass the data from the application down to the operating system. It would be at least one implementation, as Flutter is a cross-platform toolkit, meaning libraries providing bridges between the underlying OS and Flutter could also support the platforms Flutter supports, in order to keep the codebase fully cross-platform for all platforms supported by Flutter.
Research into common Bluetooth libraries for Flutter revealed the open source FlutterBlue[25] library. Checking for related strings, like FlutterBlue, in the Doldrums output shows:

```
63760:class FlutterBlue extends Object {
63761:    FlutterBlue _instance@657317761
63763:    FlutterBlue get:_instance@657317761() {
63767:    FlutterBlue FlutterBlue._@657317761(DynamicType) {
63775:    FlutterBlue get:instance() {
```

Listing 3.12: Search results for `FlutterBlue` in Doldrums output

This reveals the usage of FlutterBlue in the Welock application. Looking at the layout of the library source code here:

```
total 32K
drwxr-xr-x 1 zeno zeno  126 Jul 30 14:04 android
-rw-r--r-- 1 zeno zeno 4.5K Jul 30 14:04 CHANGELOG.md
drwxr-xr-x 1 zeno zeno  172 Jul 30 14:04 example
-rw-r--r-- 1 zeno zeno  947 Jul 30 14:04 flutter_blue.iml
drwxr-xr-x 1 zeno zeno   92 Jul 30 14:04 ios
drwxr-xr-x 1 zeno zeno   46 Jul 30 14:04 lib
-rw-r--r-- 1 zeno zeno 1.5K Jul 30 14:04 LICENSE
drwxr-xr-x 1 zeno zeno   60 Jul 30 14:04 macos
drwxr-xr-x 1 zeno zeno   60 Jul 30 14:04 protos
-rw-r--r-- 1 zeno zeno  702 Jul 30 14:04 pubspec.yaml
-rw-r--r-- 1 zeno zeno 8.1K Jul 30 14:04 README.md
drwxr-xr-x 1 zeno zeno   60 Jul 30 14:04 site
drwxr-xr-x 1 zeno zeno   28 Jul 30 14:04 test
```

Listing 3.13: Directory listing for `FlutterBlue`

This confirms the assumptions about the layout. The `lib/` directory contains the Flutter API, and the `android/` and `ios/` directories contain the platform specific implementations.

The lock uses not just standard Bluetooth communication, but rather Bluetooth Low Energy, a lower power version of Bluetooth, providing (new) features like mesh networking, broadcasting, point-to-point and lower power consumption.[24] Bluetooth Low Energy[26] uses a concept of services, characteristics and descriptors.
Services are a kind of container used to group characteristics. They often correspond to a certain feature a device offers.
Characteristics represent specific values of that service. Each value can have attributes associated to it, like is it writeable.
Descriptors describe a characteristic more closely, containing metadata. They are also required for a characteristic to send asynchronous notifications to peers, if the value changes. This is the

[25] Found at: https://github.com/pauldemarco/flutter_blue
[26] From here on out referred to as BLE

`Client Characteristic Configuration` descriptor.[24]
For example, a thermostat sensor could contain a service called "Temperature", which contains a characteristic called "get" for getting the temperature, which is read only, and a characteristic called "set" for setting the temperature, which is write only. Both contain a descriptor, containing a specific name for that action, like "Set the temperature" for the "set" characteristic and "Get the temperature" for the "get" characteristic. The "get" characteristic also supports sending notifications, if the value changed, which requires it to have the `client Characteristic Configuration` descriptor.

The assumption now is, that the app, in order to tell the lock to open, pairs with the lock, gets the appropriate service and writes to an endpoint, accepting commands, the open lock command. The interesting part now is extracting the data to be sent to the characteristic somewhere in the call chain.
The FlutterBlue documentation [27] outlines, that consumers of the API should call the `write`-method on a characteristic, in order to send data to the characteristic. Checking the source code for that method gives the following code (note, that this is only an excerpt, the method does not end here):

```
1  Future<Null> write(List<int> value, {bool withoutResponse = false}) async {
2      final type = withoutResponse
3          ? CharacteristicWriteType.withoutResponse
4          : CharacteristicWriteType.withResponse;
5
6      var request = protos.WriteCharacteristicRequest.create()
7          ..remoteId = deviceId.toString()
8          ..characteristicUuid = uuid.toString()
9          ..serviceUuid = serviceUuid.toString()
10         ..writeType =
11             protos.WriteCharacteristicRequest_WriteType.valueOf(type.index)!
12         ..value = value;
13
14     var result = await FlutterBlue.instance._channel
15         .invokeMethod('writeCharacteristic', request.writeToBuffer());
16
17     // ...
```

Listing 3.14: `FlutterBlue` source code for the Characteristic `write` method

As can be seen, the method takes in an array of integers, representing the binary data to be sent to the lock, and an optional bool, whether or not to wait for an answer. This mostly is a wrapper around the platform native implementations, creating a request, which is then passed down to the native implementation. This passing down happens on line 14, where calls the `writeCharacteristic` with the previously build request of the native implementation, represented by the `FlutterBlue.instance` object. The `request` object contains information about which characteristic to write to, under which service it is grouped, whether or not to wait for an answer and what data to write. This then gets passed down to the platform implementation, which implement the `FlutterPlugin`-class, which provides a `invokeMethod` method, which then calls the `onMethodCall` method of the platform implementation. The method signature for that functions looks like the following:

[27] Found at: https://github.com/pauldemarco/flutter_blue#usage

```
public void onMethodCall(MethodCall call, Result result);
```

Listing 3.15: Method signature for `onMethodCall`[28]

It takes in a `MethodCall` object and a `Result` object. The `Result` object is used to communicate results back to the caller, while the `MethodCall` object contains the context and arguments that were build before calling the platform implementation. Before the data was not extractable, because it was in Flutter-land, not easily parse-able, but now they have to be Java objects, in order to be usable by the platform implementation. Now it is possible to extract them using normal Android dynamic runtime instrumentation tools, like Frida.

Frida is a tool, that allows reverse engineers, security researchers and pentesters to instrument native applications, hooking, intercepting function calls, extracting and changing parameters and arguments, as well as intercepting variables and return values. Users can specify own routines using JavaScript, which then gets interpreted and injected into the target application. It supports multiple platform, i.e. Android, Linux, Windows, et cetera.[29]

The below script uses Frida's Java API, overloading the `onMethodCall` method of the FlutterBlue native platform implementation, checking whether the method is `writeCharacteristic`, the method actually sending the data over Bluetooth and then casting the data returned by the `arguments()` call on the `MethodCall` object to a byte array. This approach was taken directly from the FlutterBlue source code, from the `writeCharacteristic` implementation.[30] The data then is written to the console, using a helper function not included in this excerpt, taking in the byte buffer and generating a hexdump. Finally the call is passed down to the actual implementation.

```
1  Java.perform(function () {
2      var flutterBluePlugin = Java.use("com.pauldemarco.flutter_blue.FlutterBluePlugin");
3
4      flutterBluePlugin.onMethodCall.overload(
5                  'io.flutter.plugin.common.MethodCall',
6                  'io.flutter.plugin.common.MethodChannel$Result')
7          .implementation = function(methodCall, result) {
8                  const method = methodCall.method.value;
9                  if (method === "writeCharacteristic") {
10                         const bl_payload = Java.array('byte', methodCall.arguments());
11                         console.log('calling writeCharacteristic with:');
12                         printHexdump(bl_payload);
13                 }
14
15                 return this.onMethodCall(methodCall, result);
16     };
17
18     console.log();
19     console.log("[*] Hooked onMethodCall");
20 });
```

Listing 3.16: `frida` script to intercept `onMethodCall`, dumping the data passed in

[28] Found at: https://github.com/pauldemarco/flutter_blue/blob/7c7a86c95a3a2537bfb7e3c368716cdb26cf66b0/android/src/main/java/com/pauldemarco/flutter_blue/FlutterBluePlugin.java#L183C1-L184C63

[29] More at: https://frida.re

[30] Found here: https://github.com/pauldemarco/flutter_blue/blob/7c7a86c95a3a2537bfb7e3c368716cdb26cf66b0/android/src/main/java/com/pauldemarco/flutter_blue/FlutterBluePlugin.java#L443

This script can then be executed using the following command line. It invokes `frida`, spawning the app `cn.sixpower.wlock`, using the script given by path.

```
$ frida -U -f cn.sixpower.wlock -l flutter-blue-jacker/hook.js
```

Listing 3.17: `frida` command line to spawn the app and execute the intstrumentation script

This now gives access to all Bluetooth data being send to the lock, intercepting it without any external hardware. This now allows runtime analysis of the data send, presenting a large step in circumventing the access controls of the lock.
The above approach produces following output, when pressing the "Unlock"-button:

```
Started tracing 1 function. Press Ctrl+C to stop.
           /* TID 0x2576 */
  5927 ms  calling writeCharacteristic with:
  5927 ms
0x0a 0x11 0x46 0x35 0x3a 0x44 0x30 0x3a 0x34 0x38 0x3a 0x43 0x30 0x3a 0x31 0x42   ..F5:D0:48:C0:1B
0x3a 0x34 0x39 0x12 0x24 0x36 0x65 0x34 0x30 0x30 0x30 0x30 0x32 0x2d 0x62 0x35   :49.$6e400002-b5
0x61 0x33 0x2d 0x66 0x33 0x39 0x33 0x2d 0x65 0x30 0x61 0x39 0x2d 0x65 0x35 0x30   a3-f393-e0a9-e50
0x65 0x32 0x34 0x64 0x63 0x63 0x61 0x39 0x65 0x1a 0x24 0x36 0x65 0x34 0x30 0x30   e24dcca9e.$6e400
0x30 0x30 0x31 0x2d 0x62 0x35 0x61 0x33 0x2d 0x66 0x33 0x39 0x33 0x2d 0x65 0x30   001-b5a3-f393-e0
0x61 0x39 0x2d 0x65 0x35 0x30 0x65 0x32 0x34 0x64 0x63 0x63 0x61 0x39 0x65 0x28   a9-e50e24dcca9e(
0x01 0x32 0x07 0x55 0x30 0x00 0x00 0x00 0x00 0x00                                 .2.U0.....

  6316 ms  calling writeCharacteristic with:
  6316 ms
0x0a 0x11 0x46 0x35 0x3a 0x44 0x30 0x3a 0x34 0x38 0x3a 0x43 0x30 0x3a 0x31 0x42   ..F5:D0:48:C0:1B
0x3a 0x34 0x39 0x12 0x24 0x36 0x65 0x34 0x30 0x30 0x30 0x30 0x32 0x2d 0x62 0x35   :49.$6e400002-b5
0x61 0x33 0x2d 0x66 0x33 0x39 0x33 0x2d 0x65 0x30 0x61 0x39 0x2d 0x65 0x35 0x30   a3-f393-e0a9-e50
0x65 0x32 0x34 0x64 0x63 0x63 0x61 0x39 0x65 0x1a 0x24 0x36 0x65 0x34 0x30 0x30   e24dcca9e.$6e400
0x30 0x30 0x31 0x2d 0x62 0x35 0x61 0x33 0x2d 0x66 0x33 0x39 0x33 0x2d 0x65 0x30   001-b5a3-f393-e0
0x61 0x39 0x2d 0x65 0x35 0x30 0x65 0x32 0x34 0x64 0x63 0x63 0x61 0x39 0x65 0x28   a9-e50e24dcca9e(
0x01 0x32 0x14 0x55 0x30 0x01 0x58 0x21 0x50 0x66 0x41 0x38 0x41 0x76 0x64 0x4d   .2.U0.X!PfA8AvdM
0x64 0x38 0x41 0x41 0x41 0x41 0x42                                                d8AAAAB

  6898 ms  calling writeCharacteristic with:
  6898 ms
0x0a 0x11 0x46 0x35 0x3a 0x44 0x30 0x3a 0x34 0x38 0x3a 0x43 0x30 0x3a 0x31 0x42   ..F5:D0:48:C0:1B
0x3a 0x34 0x39 0x12 0x24 0x36 0x65 0x34 0x30 0x30 0x30 0x30 0x32 0x2d 0x62 0x35   :49.$6e400002-b5
0x61 0x33 0x2d 0x66 0x33 0x39 0x33 0x2d 0x65 0x30 0x61 0x39 0x2d 0x65 0x35 0x30   a3-f393-e0a9-e50
0x65 0x32 0x34 0x64 0x63 0x63 0x61 0x39 0x65 0x1a 0x24 0x36 0x65 0x34 0x30 0x30   e24dcca9e.$6e400
0x30 0x30 0x31 0x2d 0x62 0x35 0x61 0x33 0x2d 0x66 0x33 0x39 0x33 0x2d 0x65 0x30   001-b5a3-f393-e0
0x61 0x39 0x2d 0x65 0x35 0x30 0x65 0x32 0x34 0x64 0x63 0x63 0x61 0x39 0x65 0x28   a9-e50e24dcca9e(
0x01 0x32 0x0f 0x55 0x31 0x41 0x38 0x41 0x76 0x64 0x4d 0x64 0x38 0x41 0x41 0x41   .2.U1A8AvdMd8AAA
0x41 0x42                                                                         AB
```

Listing 3.18: `frida` output, containing the sent Bluetooth data

As is apparent, the messages follow a similar layout, each starting with the same two bytes, followed by a MAC-address, the characteristic ID and the service ID. The first message then ends, padded with zeros. The second message contains 8 undetermined bytes, followed by a copy of the last 16 bytes of the third message. The third message display another anomaly in comparison to the first two, the byte before the payload, is different. In message one and two it is `0x30`, in message three `0x31`. Because those bytes are at the end of the regions that are the same across messages, it might signal a op-code, which is later used by the lock to process the message. So the suspected message layout for now looks like this:

```
header = 0x0a 0x11 <MAC-address> 0x12 <characteristic-UUID> 0x1a <service-UUID> 0x28 0x01 0x32

payload_1 = <header> 0x07 0x55 <opcode> 0x00 0x00 0x00 0x00 0x00
payload_2 = <header> 0x14 0x55 <opcode> 0x01 0x.. 0x.. 0x.. 0x.. <payload_3 body>
payload_3 = <header> 0x0f 0x55 <opcode> <payload_body>
```

Listing 3.19: Suspected message layout

3.2.4 Extracting the encryption passphrase

The interesting part is the for unlocking the lock without prior authentication is the payload body of `payload_3`. Looking at the hex dump the data does seem to be encrypted, or at least encoded. The payload body does change from unlock to unlock, suggesting either the contents change, if this is encoded, or the encryption IV/key changes, if it is encrypted. Now, with such short messages analysis is difficult. Assuming it is encrypted, the IV/passphrase, or some kind of key, has to be encoded in the binary, or calculated, retrieved. Since the application logic mostly lives in the snapshot, from which functions names could be retrieved, it could be possible to search for such functions in the dump. Further the UI exposes a button to change the lock pin.

Searching for functions with names relating to password or passphrase yields a multitude of results, with the most relevant listed here:

```
src_ble_key_pin
main_change_ble_pwd_pcb
main_change_ble_pwd_pro
main_operator_modify_ble_pwd
_modifyLockPwd
randomPassword
randomPasswordStr
```

Listing 3.20: Relevant functions in regards to password and passphrase

Trying to analyze the call chain by hooking those functions, together with the function names in the decompiler, reveals following call chain:

```
[*] called src_ble_key_pin (0x1e1688)
[*] called main_change_ble_pwd_pcb  (0x1e6d10)
[*] called main_change_ble_pwd_pro   (0x1e9750)
[*] called main_operator_modify_ble_pwd   (0x1e7a10)
[*] called src_add_password_ble_gw    (0x1ea588)
[*] called src_lock_detail_sending_open    (0x1e1758)
[*] called src_code_sending    (0x1e4e30)
[*] called _modifyLockPwd    (0x2e1a64)
[*] called bluetooth_pin_info_from_pointer    (0x2e94ac)
[*] called randomPassword    (0x2f768c)
[*] called _randomBit    (0x2f7368)
[*] called randomPasswordStr    (0x2f732c)
[*] called randomNextInt    (0x2fcb0)`
```

Listing 3.21: "Lock pin" call chain

This means, that when the "Lock pin" button is pressed, these functions get executed in order[31]. Looking at the last 4 functions they all seem concerned with generating a random password, with the last 3 likely facilitating the `randomPassword` function, meaning access to the password send to the lock can probably obtained by hooking this function, extracting the return argument.
This approach does show one problem. It is not possible to use the same process as above, because there is no native implementation for the `randomPassword` functionality. It seems to happen completely in Flutter space, exposing all problems listed, such as their own calling convention and own stack implementation.

[31] At least these, there could be function calls missing

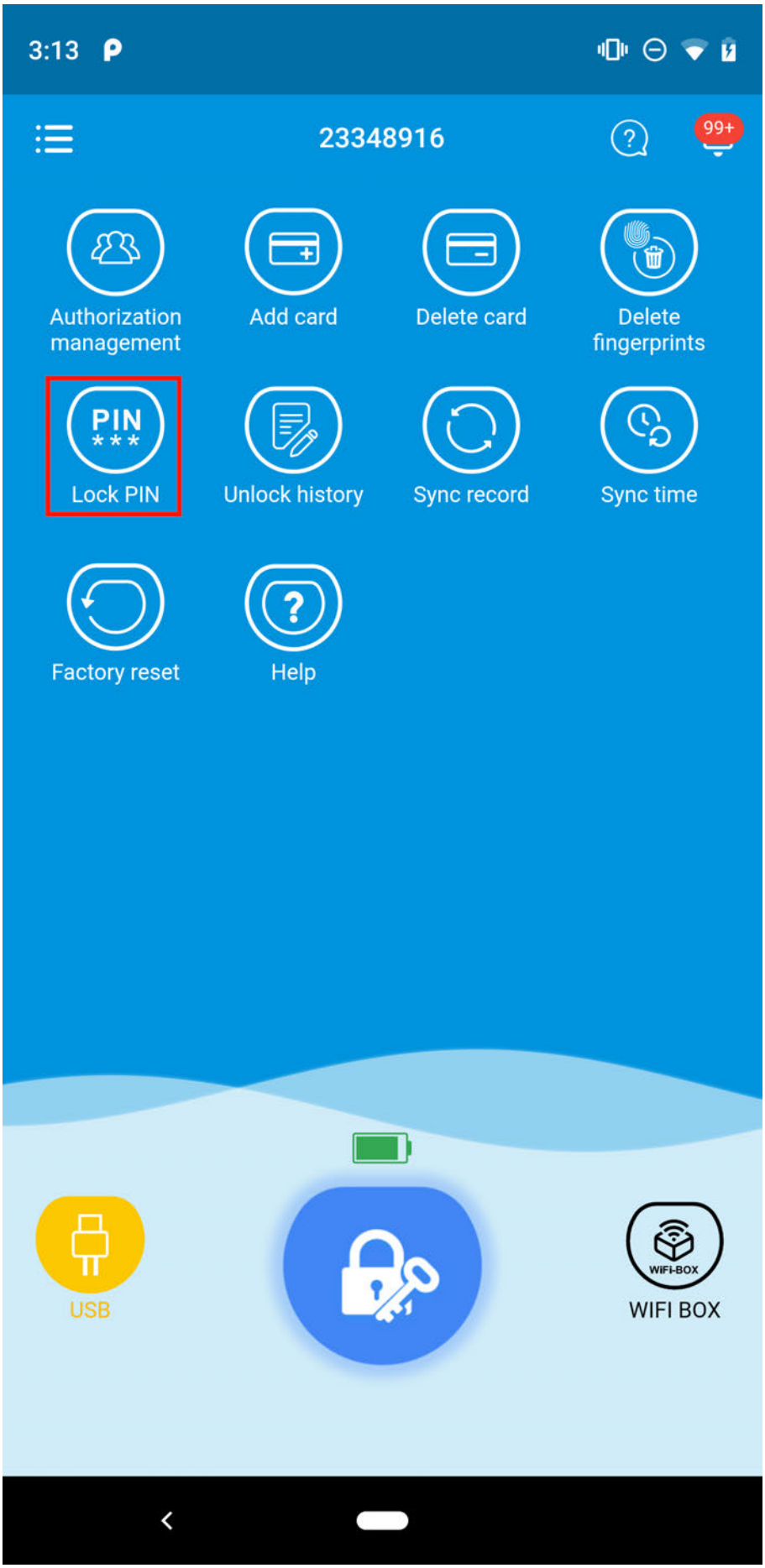

Figure 3.7: Screenshot of the Welock-app user interface with the "Lock Pin" button highlighted
Source: Own image

Taking a look at the decompiled version of `randomPassword` in `libapp.so`, the return statement, which is supposed to return a string, according to the function signature as provided by Doldrums, looks like the following:

```
return *(undefined8 *)(in_x15 + -0x20);
```

Listing 3.22: Return statement of `randomPassword`

According to the Dart source code, the X15 register serves as the stack pointer in dart code[2], meaning the return value lives at position 0x20 of the Dart stack. Since the object probably is not a small integer, which can be passed by value, it probably is a pointer to the actual value, which is then stored in some kind of more flexible memory. This also tracks with the analyzed return type of a pointer to a pointer to an object of unknown type. The script shown below finds the loaded `libapp.so` module and then installs a hook for the address of the `randomPassword` function. On function exit it retrieves the stack pointer, calculates the position of the return value, with the value from the decompiled return statement. At that positions lies a pointer, which is stored in reverse fashion, which is then flipped in order to produce a usable pointer. Then the memory at the location of the pointer stored on the stack is dumped, using the `hexdump` helper function.

```
function hook_random_password() {
        let app_mod = Process.findModuleByName("libapp.so");
        if (app_mod == null) {
                setTimeout(hook, 1000);
                return;
        } else {
                console.log("found libapp.so at", app_mod.base, "with size", app_mod.size);
        }

        Interceptor.attach(app_mod.base.add(0x002cf288), {
                onLeave: function (args, ret) {
                        console.log(`leaving randomPassword`);
                        const stack_pointer = ptr(this.context.x15);
                        const return_value = stack_pointer.add(-0x20);

                        const rev_pointer = new Uint8Array(Memory.readByteArray(return_value, 8));
                        const part_1 = Array.from(rev_pointer.slice(4)).reverse();
                        const part_2 = Array.from(rev_pointer.slice(0, 4)).reverse();

                        let pointer = new Uint8Array(part_1.concat(part_2));
                        pointer = Number(new DataView(pointer.buffer).getBigUint64(0));

                        console.log(`the return value lives at 0x${pointer.toString(16)}`);
                        console.log(hexdump(ptr(pointer)));
                }
        });
}
```

Listing 3.23: Frida script to extract return value from a function

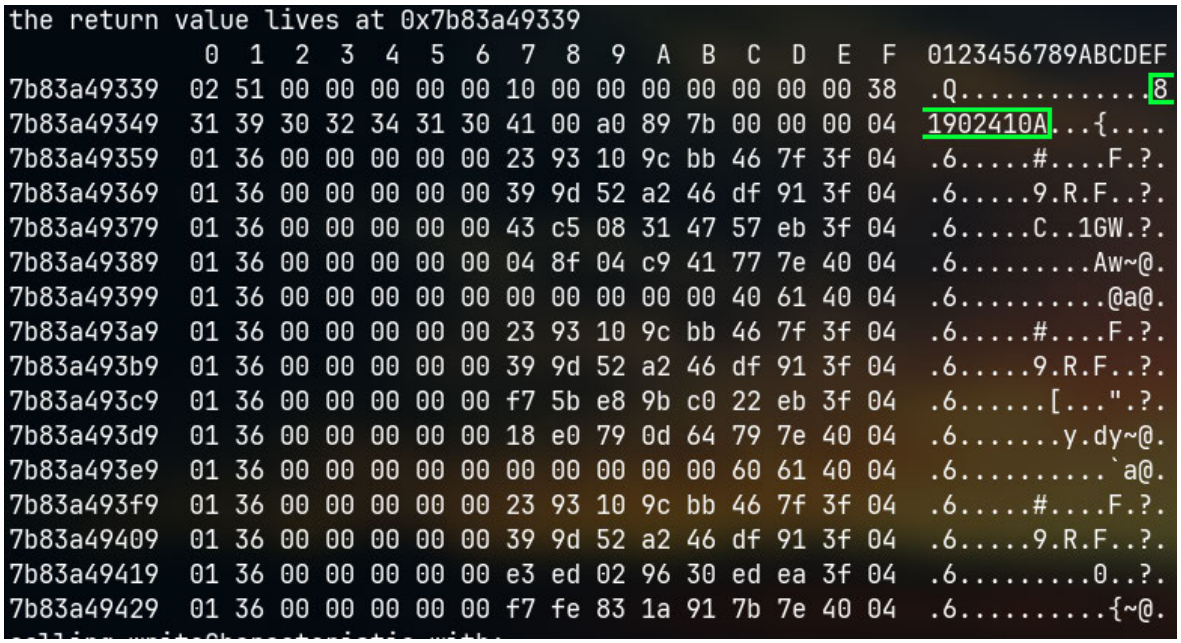

Figure 3.8: The dumped, potential, encryption passphrase

Source: Own image

This produces a hexdump, which seems to contain the password directly, as the memory region does not contain a further pointer. The string in the hexdump seems to be 8 byte, 64 bit, long. Now that a potential key was found, how does one decrypt an encrypted message?

To decrypt a encrypted message four things are required:

1. The algorithm
2. The key
3. The initialization vector (if used by the algorithm)
4. The message

At the current point, two things are known. The (possible) key, extracted using Frida, and the message, as intercepted by hooking `FlutterBlue`. Without the algorithm that was used during encryption, it is not possible to decrypt the message. Even knowing all other 3 would not yield a decrypted message, if the algorithm uses an initialization vector.
Determining the encryption algorithm could be done in different ways. On one hand, searching through the decompiled Flutter code in order to find the encryption routines. On the other hand, analysis of the encrypted messages and encryption key.
The first approach can be quite time intensive, because the function signatures retrieved using Doldrums does not indicate any encryption routines, suggesting they are rolling their own cryptographic in their own routines, which makes finding them and reverse engineering them quite time intensive. But this might also open the door for vulnerabilities in the encryption, as rolling your own cryptographic is a challenge riddled with edge cases and generally frowned upon in the developer community.
The second approach proves to be difficult because of the small sample size of messages and keys. While it is possible to generate an arbitrary number of messages and encryption keys with the methods outline above, their length is too short to meaningfully run any analysis on them.

Taking a look at the two most common symmetric encryption algorithms, AES and DES, only DES is really relevant, because the smallest key usable with AES is 128 bit. While DES uses a 56-bit key, this key is derived from a 64-bit key and therefore might be a possible contender.

At the time of writing, it was not possible to decrypt the payload. This might suggest a non-"standard" encryption algorithm, no encryption at all, or a custom cryptographic algorithm.

3.2.5 Viability and conclusion

Given the techniques above, the data collected and further research, it should be possible to reverse engineer the message format as well as the encryption algorithm used by Welock to secure their system, in order to reimplement the authentication flow to open the lock, without prior authorization.
Reverse engineering the message format should be possible, as a lot of data is available and dynamic analysis is possible by intercepting and modifying the Bluetooth traffic, of course barred problems that could arise down the road.
Locating and reverse engineering the encryption algorithm proved to be more of a challenge, as no easily identifiable functions implementing such an algorithm could be found, at it could not be unambiguously identified, if encryption is used at all. Solving this problem would require closely following the call chain, reverse engineering possible algorithms, understanding the Dart calling convention and control flow and then identifying the used encryption algorithm from very low level instructions.

If it is possible to circumvent all those challenges, crack the encryption, figure out the message format and build a custom implementation, it would be conceivable to build a tool, that is able to pair to any Welock adhering to the format, and unlock it with the click of a button, with the only mitigation possible being an update of every lock, if an update process is integrated, or plain replacing all the hardware.

3.3 Results

The above chapter shows the process of intercepting and extracting the Bluetooth traffic of the app by hooking native platform implementations, as well as a possible encryption passphrase used to secure communication with the device. It demonstrates the process of decompiling a Flutter Android application showing ways to make reverse engineering easier by automatically renaming functions from a dump, as well as demonstrates a way to intercept Bluetooth traffic without dedicated hardware. It shows how to dump the possible encryption passphrase by using the decompiled application code to build a Frida script, dumping the memory, pointed to by a pointer returned from the associated function ,directly from the Dart runtime memory.

The original goal of gaining unauthorized access to the electronic lock from Welock could not be achieved, being hindered by the possibly encrypted payload, which could not be deciphered. This might be caused by encryption but could also have other causes, such as being encoded or using a own "command"-language to communicate with the lock, prompting further research.

4 Conclusion and recommendations

This thesis's goal was to show different implementations of electronic locking systems, discussing a locking system often seen in commercial installations and a single electronic door lock, often used by consumers. It aimed to discuss whether they are implemented secure and show the process during investigation, possibly demonstrating attacks on the locks.

It shows the vulnerability of the commercial system, a MIFARE Classic system, to cloning attacks, as well as abusing the weak encryption to build a master key giving access to almost all of the system. It demonstrates the techniques and tools needed to exploit that weakness and documents the process of building the master key. By doing so, it manages to show how trivial that attack is, demonstrating the risks that amount from using that system without further hardening, as the lock relies on outdated cryptography, NXP's proprietary `CRYPTO1`, and assumptions of a unique UID, which are not granted anymore. The cryptographic implementation has long been broken, by exploiting a few implementation issues and weak random number generation. The UID was changeable as soon as aftermarket cards were produced.
It then discusses the successor and alternative to that system, MIFARE Desfire, showing their implementation addressing those issues, namely the encryption, but also highlights possible issues with that implementation. These issues could arise, if system integrators do not adhere to secure practices, using the weaker encryption algorithms offered by the system, like 3DES, or by using default/hard-coded keys. It comes to the conclusion, that while the first system is insecure, as all security measures could be circumvented, it's successor could fall victim to the same problems, if implemented incorrectly.

Then it focuses on a very different product, targeted at consumers, with the same goal of showing possible vulnerabilities, as well as the techniques and tools used during the research. It demonstrates how to decompile the application in order to identify communication routines, how to intercept the smartphone-lock communication without additional hardware. It shows existing Flutter reverse engineering tooling and how to use it to make the decompiled code easier to understand and shows a unique approach to extracting a possible encryption passphrase from the application runtime memory. It explains the workings of the Dart VM and Flutter application on a surface level, showing grounds for further research. Given more time, it might be possible to crack the encryption and message format, allowing unauthorized access to the lock. The greatest obstacle proves to be determining, whether it is encryption or encoding, and reverse engineering the process. Unauthorized access to the system could not be achieved at the time of writing.

Recommendations for the system above, in order to be implemented securely, are at least:

- Do not trust the environment you operate in, not even your own application runtime. Everything can and will be intercepted, inspected and reverse engineered. Like the smart lock from Welock showed, defense in depth, such as encrypting the traffic, significantly increases the effort, resources and skills needed, potentially even rendering the attempt impossible. But as shown by the MIFARE Classic system: use strong and established, open encryption algorithms, ideally recommended by NIST or the BSI.

- Follow Kerckhoffs's principle
  Do not rely on the confidentiality of your data. Even if the encryption algorithm is known, the encrypted data should still be safe. Meaning, that even if somebody can get to your data, be it encrypted or not, have additional safe guards in place, like authentication and authorization. This, of course, does not apply to all data. For example, data required to implement authorization and authentication might rely on data exchanged in encrypted form. Data like that cannot be open, as not to defeat its purpose. That data should then be protected by proper encryption, as stated above.
- Implement secure and tested authentication and authorization methods to establish identity and permissions.
  To secure access to the system, implement authentication and authorization protocols, which are based on secure cryptographic concepts and validate their correctness.

5 Discussion

This research shows, that it is possible to exploit an electronic locking system, with few amounts of tools or deep knowledge about the problem domain required, in order to gain access to restricted areas. It shows, that those systems rely on system integrators to implement necessary changes, like knowing to encrypt data with proper encryption algorithms, in order for the system to be secure, or risk causing supposed secure areas to be vulnerable to unauthorized entry.
While it was not possible to exploit the Welock smart lock, it did show techniques, that can be used with Flutter and Dart applications, even producing a new approach to extracting value from memory. This field is relatively small, with not too much resources, mostly consisting of a few blog posts, two reverse engineering projects and the source code for the engine and language. It brings new light to this field, laying grounds for further research, which could advance the field.

The research revealed a lot of different aspects and interesting fields of research during, which would have drastically increased the amount of effort and complexity of the work. All of those aspects, like the development of reverse engineering and dynamic instrumentation tooling for the Dart/Flutter ecosystem, actual hardware-level engagements of the locks, the Welock as well as a door lock that form the electronic locking system under testing in chapter 2, would not have fit the scope of this work, giving opportunity for future work.
Missing data proved mostly to be a problem for the testing of the encryption of the Welock Bluetooth lock, as it hindered analysis of the encrypted/encoded parts, as well as the memory layout during reverse engineering. Test applications were built in order to aid reverse engineering and understanding the Dart runtime memory layout, but together with the growing scope, potential gains were left on the table.

Still, more sophisticated tooling and understanding of the Flutter/Dart ecosystem, runtime and execution were developed and gained during the writing of this thesis and the research process, but were not in the scope of this thesis. So while they were not mentioned in this thesis, they lay the grounds for further research and publishing on this topic. This research could take the form of developing the theoretical ground work, or a full practical implementation, for a dynamic instrumentation toolkit, similar to the Frida toolkit used here, but for the Dart and Flutter ecosystem.

Bibliography

[1] ARM. *ARM Cortex-A Series Programmer's Guide for ARMv8-A - Parameters in general-purpose registers*. July 29, 2024. URL: https://developer.arm.com/documentation/den0024/a/The-ABI-for-ARM-64-bit-Architecture/Register-use-in-the-AArch64-Procedure-Call-Standard/Parameters-in-general-purpose-registers.

[2] *constants_arm64.h*. Aug. 4, 2024. URL: https://github.com/dart-lang/sdk/blob/b80e682cbf313691650cc9c45302811deb710aa0/runtime/vm/constants%20%5C_arm64.h.

[3] *Dart*. July 22, 2024. URL: https://dart.dev/.

[4] *Dart overview*. July 22, 2024. URL: https://dart.dev/overview.

[5] Slava Egorov. *Introduction to Dart VM*. July 22, 2024. URL: https://mrale.ph/dartvm/.

[6] Allen Roginsky Elaine Barker. *Transitioning the Use of Cryptographic Algorithms and Key Lengths*. NIST Special Publication 800-131A Revision 2 (NIST).

[7] *Flutter*. July 22, 2024. URL: https://flutter.dev/.

[8] Rory Flynn. "An investigation of possible attacks on the MIFARE DESFire EV1 smartcard used in public transportation". PhD thesis. July 2019. DOI: 10.13140/RG.2.2.19591.42401.

[9] OWASP Foundation. *Fuzzing*. July 16, 2024. URL: https://owasp.org/www-community/Fuzzing#.

[10] Flavio D. Garcia et al. "Wirelessly Pickpocketing a Mifare Classic Card". In: *2009 30th IEEE Symposium on Security and Privacy*. 2009, pp. 3–15. DOI: 10.1109/SP.2009.6.

[11] Impact-I. *reFlutter*. July 23, 2024. URL: https://github.com/Impact-I/reFlutter.

[12] Timo Teräs Jesse D'Aguanno. *A Touch of Pwn - Part I*. July 24, 2024. URL: https://blackwinghq.com/blog/posts/a-touch-of-pwn-part-i/.

[13] Ricardo Loura. *Doldrums*. July 23, 2024. URL: https://github.com/rscloura/Doldrums.

[14] *MIFARE Classic EV1 1K - Mainstream contactless smart card IC for fast and easy solution development*. 279232. Rev. 3.2. NXP Semiconductors. May 2018.

[15] *MIFARE Classic EV1 4K - Mainstream contactless smart card IC for fast and easy solution development*. 279332. Rev. 3.2. NXP Semiconductors. Nov. 2017.

[16] David Oswald and Christof Paar. "Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World". In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 207–222. ISBN: 978-3-642-23951-9.

[17] *Part 1: Physical characteristics*. ISO/IEC 14443-1:2018 (ISO). 2018.

[18] *Part 2: Radio frequency power and signal interface*. ISO/IEC 14443-2:2020 (ISO). 2020.

[19] *Part 4: Transmission protocol*. ISO/IEC 14443-4:2018 (ISO). 2018.

[20] Hratch G. Semerjian. *Federal Register / Vol. 70, No. 96*. URL: https://www.govinfo.gov/content/pkg/FR-2005-05-19/pdf/05-9945.pdf.

[21] NXP Semiconductors. *MIFARE Classic*. July 1, 2024. URL: https://www.nxp.com/products/rfid-nfc/mifare-hf/mifare-classic:MC_41863.

[22] NXP Semiconductors. *MIFARE Product Families*. July 4, 2024. URL: https://www.mifare.net/#products.

[23] NXP Semiconductors. *MIFARE® DESFire® EV3: High-Security IC for Contactless Smart City Services*. July 17, 2024. URL: https://www.nxp.com/products/rfid-nfc/mifare-hf/mifare-desfire/mifare-desfire-ev3-high-security-ic-for-contactless-smart-city-services:MF3DHx3.

[24] Bluetooth SIG. *The Bluetooth® Low Energy Primer*. Version 1.2.0. Mar. 15, 2024.

[25] *Supported deployment platforms*. July 22, 2024. URL: https://docs.flutter.dev/reference/supported-platforms.

Statutory Declaration in Lieu of an Oath

I – Sönke Adrian Eichelbaum – do hereby declare in lieu of an oath that I have composed the presented work independently on my own and without any other resources than the ones given.

All thoughts taken directly or indirectly from external sources are correctly acknowledged.

This work has neither been previously submitted to another authority nor has it been published yet.

Mittweida, 19. August 2024

Location, Date

Sönke Adrian Eichelbaum