

---

---

# Access Control Vulnerabilities, Cryptography, and Improvements

---

---

Project Report  
Group 19gr352

Aalborg University  
Department of Electronic Systems  
Fredrik Bajers Vej 7B  
DK-9220 Aalborg Ø





Department of Electronic Systems

Fredrik Bajers Vej 7B

DK-9220 Aalborg Ø

<https://es.aau.dk>

## AALBORG UNIVERSITY

### STUDENT REPORT

**Title:**

Access Control Vulnerabilities, Cryptography, and Improvements

**Theme:**

Microprocessor systems

**Project Period:**

Fall semester 2019

**Project Group:**

19gr352

**Participants:**

Jonathan Velgaard Sørensen

Julian Jørgensen Teule

Magnus Melgaard

Marius Frilund Hensel

Nicholas Bernth Strømgaard Hansen

Victor Büttner

**Supervisors:**

Rasmus Løvenstein Olsen

**Abstract:**

In this paper, issues with access control and vulnerabilities at Aalborg University are examined. Whilst doing this examination, vulnerabilities in various web services related to the access control system at Aalborg University were discovered. A proof of concept for a new access control system was also developed.

This proof of concept attempts to avoid trusting PICC scanners, which would allow for third party services to use said scanners safely. Additionally, the proof of concept uses unique cryptographic keys on each PICC, which reduces the risk of duplicated PICCs. This proof of concept consists of a MIFARE Classic 1K PICC, a client-side application for scanners, and a server-side API and -database that the client-side application uses.

It was found that Crypto1 can be compromised by a malicious scanner, but using unique keys on each PICC partially mitigates PICC duplication.

**Copies:**

**Number of Pages:** 67

**Date of Completion:**

December 18, 2019

*The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author. The source code to any program, which is not included in Appendices, can be found in the attached ZIP file.*



# Nomenclature

---

<i>Abbreviation</i>	<i>Name</i>
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CHF	Cryptographic Hash Function
CI	Continuous Integration
CPR-number	Civil Registration Number
CRC	Cyclic Redundancy Check
FIFO	First In First Out
GPIO	General-Purpose Input/Output
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
I/O	Input/Output
IC	Integrated Circuit
ID	Identifier
IEC	International Electrotechnical Commission
IP	Internet Protocol
IRQ	Interrupt Request
ISO	International Standards Organization
JSON	JavaScript Object Notation
LSB	Least Significant Bit
MFA	Multi-factor authentication
MISO	Master In Slave Out
MITM	Man in The Middle
MOSI	Master Out Slave In
MSB	Most Significant Bit
MUID	Manufacturing Unique Identifier
NFC	Near Field Communication
NUID	Non-unique ID
OS	Operating System
PCD	Proximity Coupling Device
PICC	Proximity Integrated Circuit Card
PUID	Personal UID
RBAC	Role Based Access Control
RFID	Radio Frequency Identifier
SBC	Single Board Computer
SHA	Secure Hash Algorithm
SPI	Serial Peripheral Interface
SQL	Structured Query Language
TLS	Transport Layer Security

UID	Unique Identifier
URL	Uniform Resource Locator

---

# Contents

---

<b>Nomenclature</b>	<b>v</b>
<b>Contents</b>	<b>vii</b>
<b>Preface</b>	<b>ix</b>
<b>Background</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Access Control . . . . .	1
1.2 Role Based Access Control Security . . . . .	4
1.3 Initial Problem Statement . . . . .	5
<b>2 Problem Analysis</b>	<b>7</b>
2.1 AAUCard Case . . . . .	7
2.2 About MIFARE Classic cards . . . . .	10
2.3 AAUCard Examination . . . . .	11
2.4 Requirement Specification . . . . .	16
2.5 Proposal of Two Systems . . . . .	17
2.6 Final Problem Statement . . . . .	18
<b>3 Development</b>	<b>19</b>
3.1 Environment and Deployment . . . . .	21
3.2 Data Store . . . . .	21
3.3 Server Communication . . . . .	28
3.4 Scanner . . . . .	35
3.5 Crypto1 in Software . . . . .	44
3.6 Admin User Interface . . . . .	48
3.7 System Sequence of Events . . . . .	49
<b>4 Test</b>	<b>51</b>
4.1 Acceptance Test . . . . .	51
<b>5 Discussion</b>	<b>61</b>
<b>6 Conclusion</b>	<b>63</b>
<b>7 Reflection</b>	<b>65</b>
<b>A Key Strengthening</b>	<b>67</b>





# Preface

---

---

Jonathan Velgaard Sørensen  
<jvsa16@student.aau.dk>

---

Julian Jørgensen Teule  
<jteule18@student.aau.dk>

---

Magnus Melgaard  
<mmelga16@student.aau.dk>

---

Marius Frilund Hensel  
<mhense15@student.aau.dk>

---

Nicholas Bernth Strømgaard Hansen  
<nbsh18@student.aau.dk>

---

Victor Büttner  
<vbattn18@student.aau.dk>

Aalborg Universitet, December 18, 2019



# Background

---

This project revolves around the experience of using the access control system place at AAU (Aalborg University).

The project was initialized through the proposal from some students at AAU who heard about a third party that had managed to duplicate an AAUCard. This peaked an interest in how the AAUCard works, which potential flaws it has and how it can be improved. As such, its current established access control system was chosen as a case study.

Newly enrolled students are granted a personalized access control card. This card has their name, photo, card number, student number, a magnetic stripe and bar-code printed onto it. It also contains an integrated circuit for the purpose of utilizing proximity scanners. The AAUCard can be used multiple ways on campus e.g. as an access card for doors and printing.



# Introduction

---

# 1

Security has been an essential concern throughout time, whether it's about providing a safe area to stay or to keep ill-intended *adversaries* away from material as well as non-material goods. As such it has come in many forms like manned guard positions, physical locks and password systems. The aforementioned adversary can be any malicious actor, for instance a cyber criminal seeking profit or a nation state seeking intelligence. When determining the security of a system, an adversary is usually assumed to have near infinite resources, to make sure all faults are discovered, even if they are not easily exploitable. If an attack is improbable due to high resource requirements, that attack can be disregarded for later reevaluation.

The improvement of security has been met with similar progression from adversaries and their means of intrusion. This has created a need to constantly keep ahead of the curve with more advanced technologies to avoid potential breaches. An example of this development can be seen with the rising fear of credit card cloning, or *skimming* [1]. Skimming prompts credit card manufacturers to outfit cards with more resilient cryptography and other potential means of rejecting card skimming devices. Similarly, existing PCDs (Proximity Coupling Device), or simply card scanners, could be updated to, for instance, requiring PIN codes more often. Another possible solution would be to not *trust* the PCD whatsoever. Trust in a security context is a complex topic. An example can be entrusting a third party with a secret, which also means that they can leak it. If sharing the secret was unnecessary, it would have been safer not to. If you are using software from a 'trusted' third party and the third party turns malicious or gets compromised, you are likely to also be compromised as a result. Decreasing the amount of trusted parties is thus desirable and being 'trustless' is optimal, though it may be impractical.

## 1.1 Access Control

Access control is a term often used in the context of security of e.g. physical access to buildings or data.

Physical access control systems are used to allow authorized people to enter an otherwise locked building without assistance from other personnel. This is used in multiple different settings, e.g. a university. Generally the most well known and used form of access control is a key and a lock, which is an analogous mechanism. In 2019, 49% of all companies with 250+ employees in Denmark used sensors for access control [2]. In terms of universities in Denmark, almost all allow students to enter certain facilities outside normal opening hours. This is typically implemented through the use of an electronic access control system. [2]

Credentials are needed from a user in order, for an electronic access control system, to work properly. A credential could be a known pass code of the user, a simple version being entered on a keypad with the numbers 0-9 as well as the \* and # symbols. Credentials can be make use of multiple factors, which are usually something you know (e.g. a code), something you have (e.g. a key card) and something you are (e.g. fingerprint). The latter category is validated by a Biometric Reader, which is a device that takes a physical attribute of the user as input.

### 1.1.1 Key Card Access Control

In terms of the aforementioned key card, there are multiple options and according to Norman [3] the following cards are the most common:

- Magnetic Stripe
- Wiegand wire
- Passive Proximity
- Active proximity
- Smart Card

The magnetic stripe was invented with the intend of easing the process of withdrawing money from an ATM. One of the benefits of this card type is that both the reader and the actual card are inexpensive. However, magnetic stripes can be copied, which can make them unfit to stop adversaries from entering. In terms of access control, magnetic stripe cards has predominantly been replaced by proximity cards.

Wiegand wire uses cards embedded with wires, the wires change polarity when swiped through a wiegand reader, which then can be translated into data.

Today, the Wiegand data protocol has become the standard for proximity cards.

Usually, passive proximity cards uses 125 KHz RFID (Radio Frequency Identifier) to transfer data to the reader. The name passive refers to the fact that the card has no stored energy on the card, but instead uses a coil from the reader to transfer an ID (Identifier) from the tag. An active proximity card has a battery attached to the tag, which allows for long-range reading. This could for example be used to open a garage door on arrival of a car. A big difference in these two types of cards are the cost.

Usually, a smart card uses 13.56 MHz RFID and it is the successor of proximity cards, since it was created with the intention of lowering expenditure and adding new features to the cards. One of the benefits of smart cards are its ability of high storage and that it can be encrypted. [3]

### 1.1.2 Biometric Access Control

Different kinds of biometric access control exist. A common benefit of biometrics is that every individual has different biometrics.

Universality of a given biometric, which means that the the physical trait being measured is common to all users, is according to Bolle and Pankanti [4] a key factor. The most commonly known biometrics are:

- Fingerprint
- Eyes
- Face
- Voice

Fingerprints, as a biometric, work by using the unique ridges and valleys that exists on the tips of an individual's fingers. The user places their finger on a fingerprint scanner, which compares the fingerprint to approved users' fingerprints. If the fingerprint matches that of an approved user, access is granted.

The fingerprint biometric is one of the most developed biometric technologies, due to the fact that it has been used in criminal investigation for the last century. Moreover, it is believed that no two humans have identical fingerprints [4]

Eyes, as a biometric, work by using retina or iris recognition. Retina recognition uses blood vessels with a scanning distance ranging from eighth centimetres to one meter. A scanner that uses near infrared light is able to see the blood vessels in the eyes and are able to compare it to already saved retinas. [5]

Iris recognition is seen as one of the strongest biometrics, due to the fact that there is high randomness in the development of the iris, and because the parameters needed for iris recognition do not change for a person, except for cases such as eye diseases or injuries. Iris recognition works by using ambient light on the person's eye, where the photons from this light is then partially reflected back to the scanner. [6]

Face recognition has the same goal as other biometrics i.e. detecting a person with one or more sensors. One of the most popular approaches to face recognition is PCA (Principal Component Analysis), which is used to create a set of eigenfaces from multiple images of a user's face. However, face recognition is generally seen as less accurate compared to other biometrics. [7]

Speaker recognition refers to identifying the individual on the voice alone. An example of this could be that the user first enters PIN code at the door, and then uses his voice to verify before the door can be opened. Many factors can cause errors in terms of speaker recognition such as: Aging, sickness, or extreme emotional states, e.g. stress. Therefore, no matter how accurate the speaker recognition algorithm is, it can be limited by the user. [8]

## 1.2 Role Based Access Control Security

The security properties of a RBAC (Role Based Access Control) system can be divided in two parts; authentication and authorization. There are also physical considerations such

as which lock and door is used. For instance, there is no benefit in having a strong lock on a paper-thin door. However, physical concerns are deemed out of scope.

### 1.2.1 Authentication

Authentication is the process of proving a claim. With regards to access control, this is usually proving a claim of identity based on credentials provided by the authenticating authority. A scenario might be that a user claims to be Alice and since the user, in this case, can provide Alice's credentials, authenticity of the claim is assessed, though not guaranteed, to be true.

In such a scenario authenticity is not strictly proven, but depending on the security of the credentials it is rather unlikely, that anyone but Alice could provide them.

The security of the credentials depends on how many of the following factors they make use of, similarly to the factors mentioned in section 1.1 on page 1.

- The **knowledge factor**, something the user knows. A knowledge factor element could be a password.
- The **ownership factor**, something the user has. An ownership factor element could be an ID card.
- The **inherence factor**, something the user is and only that user is. An inherence factor element could be a biometric, like a retinal pattern. [9]

While multiple elements of the same factor can increase security, it is usually only marginally as the work to compromise a single element of a factor is about the same as compromising an entire factor. This can be exemplified as such:

- If Alice's credentials consists of two knowledge factor elements, like a password and a security question, they would both be compromised in case of a MITM-attack (Man in The Middle), phishing attack, shoulder surfing and similar.
- If Alice's credentials consists of two ownership factor elements like a keyhanger and an ID (Identification) card, they would both be compromised in case of theft or similar.
- If Alice's credentials consists of two inherence factor elements, like a retinal pattern and a fingerprint, it could be argued that a retinal pattern is more difficult to replicate than a fingerprint, but there are still approximately the same requirements for attacks that would compromise inherence factor elements.

In general it is only beneficial to have a single element of each factor and if multiple factors are used, it is called MFA (Multi-factor authentication). [9]

An ideal RBAC should use credentials with as many factors as possible without being inconvenient. For some RBAC systems, especially the physical ones, a fourth factor can be introduced:



- The **location factor**, where the user is.

This can be used since the speed of common transportation is known. This means that, if a user within minutes of attempting to authenticate from one location, attempts to authenticate from a location far away, the attempt can be rejected. [10]

### 1.2.2 Authorization

Authorization is the process of assigning privileges. In the context of physical RBAC this is usually determining whether a user should or shouldn't be granted access to a room. These privileges should be dynamically changeable as they often change over time.

## 1.3 Initial Problem Statement

**Which issues are there with access control in a university setting?**



# Problem Analysis 2

---

*Now that general access control and role based access control have been introduced, the initial problem statement is expanded upon. This is done by researching the access control system at AAU.*

## 2.1 AAUCard Case

In order to establish security throughout the campuses and provide students and faculty members with access privileges, Aalborg University has implemented *AAUCards*. Said AAUCards are personalized PICCs (Proximity Integrated Circuit Card) which serve multiple purposes, e.g. unlocking doors of university buildings, using AAU printers and borrowing books from the library. These cards have four identifications integrated into them: RFID tag, magnetic stripe, bar-code, and visual identification, i.e. name, study number, card number, date of issuance, and picture.

- The RFID tag is used when unlocking doors and when using the printers.
- The magnetic stripe can be used on printers and in some on-campus cafés.
- The bar-code only holds the card number, and is used when borrowing books from the library.
- The visual identifications are mostly used to verify a student at course exams.

In some buildings the AAUCard is not only used to unlock doors into the building, but also to unlock doors of personal group rooms. This access is given by AAU personnel on request. At the start of study, a PIN code is chosen by the student, which is only used as a supplement to either the RFID tag or the magnetic stripe, see figure 2.1 on the next page.



**Figure 2.1:** A picture of the card reader model used to unlock doors of AAU facilities.

Moreover, third party services is built using the AAUCards. An example of this is a café on campus, which is run by volunteers, that allows for payment with the card. The aforementioned café uses magnetic stripe for transactions and no code is necessary.

Reading the manufacturing id reveals that the cards are of model NXP MIFARE Classic 1K. Introduced in 1994, these could store 1 kB of data securely by requiring key authentication to access the data [11]. This is achieved by having an onboard chip which encrypts traffic and authenticates the reader before transfer [12].

The 1 kB on the card is split up into 16 sectors (0 through 15), each containing four 16 byte blocks. The last block of each sector is called the sector trailer, containing authentication keys and permissions for each block [12]. Without the key to this sector an adversary can, in principle, not get this string, thus preventing cloning of cards.

Sector 0 on the card contains a serial number, which is also known as a MUID (Manufacturing Unique Identifier). The MUID is assigned once the card is manufactured and can only be changed on spoofable cards.

### 2.1.1 Interview

In order to achieve greater understanding concerning the AAUCard system, an interview was conducted on the 26th of September 2019. The interviewee in question was a security specialist at AAU IT Services, Finn Büttner.

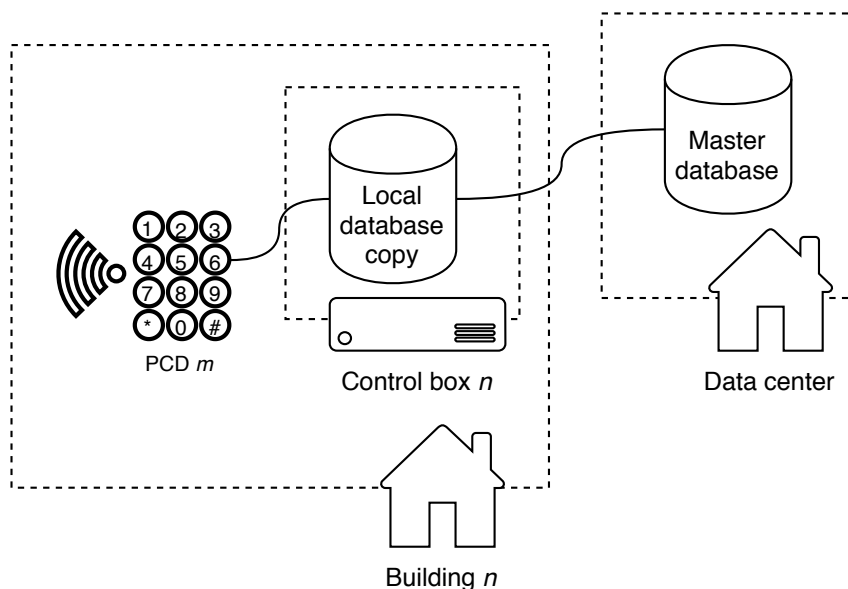
Büttner reiterated the four identifications of the AAUCard, as mentioned in section 2.1 on the preceding page, and their intended usage.

The RFID identifier grants access at door PCDs (Proximity Coupling Device), allowing

the AAUCard holder to enter the building or room all depending on the zone (0, 1, 2, etc.) and their role. Zone 0 as an example controls access to the outer layer of a building, while zone 1 controls access to an area of the inner layer. A single room with one door is also a zone, if it uses a scanner to control access. A zone controls access by having policies for when it is locked and whom has access.

Additionally, one can use the RFID to utilize printer services. University staff members are also outfitted with an AAUCard, but it does not require PIN code for printers. Sector 15 of the RFID tag is used for door access while sector 0, the MUID, is used for printer services. The magnetic stripe, sector 15 of the RFID tag, and the bar-code contain the card number and the bar-code primarily used for AAU Library services. The final identifier, the visual information, is used to confirm the authenticity of the AAUCard holder. A PUID (Personal Unique Identifier) is the study number each individual student has.

When a PCD has read a RFID the information will get passed on to a control box which then validates and opens the door accordingly. Said control boxes connect to a database server, but a connection is not necessary to function as all of the information in the database server exists locally as well. This is illustrated in figure 2.2.



**Figure 2.2:** Illustrated AAUCard Infrastructure

Büttner addressed an issue with the current system in that it is never apparent whether someone is still inside of a room in the building. This is due to the fact that the system does not require AAUCard scans every time a room or building is entered and left, as well as how AAUCard holders may keep doors open for other students or personnel. These problems are not present at the AAU campus located in Copenhagen, as the PCDs there require each individual students to swipe their cards. This way the system is able to more reliably keep track of who is inside the buildings.

However, it is not for certain whether someone is utilizing an AAUCard belonging to another card holder than themselves, nor is their photo identifier validated.

According to Büttner, a noteworthy factor of the AAUCard was the focus on convenience.

This establishes that AAU staff and students should get access to relevant facilities with as little trouble as possible while still remaining secure. As such zone 0, 1, and 2 were established for different buildings and access intervals, where most buildings get more strict access requirements past 15:45 until next morning. This makes the necessary facilities easier to access during the most active hours of the day and locked off from unauthorized adversaries for the rest of the day.

When asked about how the current system could be improved, Büttner mentioned that it would take more than just newer cards to fix the most apparent issues, as the system is only as strong as its weakest link. This means both the relevant hardware, i.e. PICCs, PCDs, Control boxes, and processes surrounding it, i.e. cryptography, data handling, card printing, would have to be upgraded.

## 2.2 About MIFARE Classic cards

The NXP MIFARE Classic 1K cards are an example of an ISO/IEC (International Standards Organization / International Electrotechnical Commission) Type A 14443 compliant PICC [12]. They come in many varieties and have embedded ICs (Integrated Circuits) capable of performing memory and processing functions. The PICCs which are compliant with the ISO/IEC 14443 standards remain some of the most prevalent on the market, as they provide a low-cost solution to security [13]. ISO/IEC 14443 covers a number of features concerning cards and security devices for personal identification, specified in four different parts:

- 14443-1: Physical characteristics
- 14443-2: Radio frequency power and signal interface
- 14443-3: Initialization and anticollision
- 14443-4: Transmission protocol

PICCs and similar security devices are specified by ISO/IEC 14443-2 as either Type A or Type B depending on the modulation and bit coding of the embedded IC. Most PCDs (Proximity Coupling Device) will nonetheless be capable of reading either type as long as compliance with ISO/IEC 14443-2 is met.

Compliant devices typically work within a proximity of up to 10 centimetres within the PCD and with a frequency of 13.56 MHz [14]. Such proximity limitation can be seen as a benefit as it reduces the risk of multiple PICCs being caught within the range of the PCD simultaneously, also known as a collision. ISO/IEC 14443-3 compliant PICCs feature an anticollision measure to further prevent this phenomenon, ensuring that the intended PICC will always be read. Furthermore, the limited range serves to hinder adversaries from skimming the PICC and intercepting its information.

An additional feature that is commonly seen among smart cards is magnetic stripes. The magnetic stripe functions as a bit data carrier once polarized, where a long polarity reversal counts as a 0 and two short reversals count as a 1. These stripes can be laid out across three tracks meant to store data.

## 2.3 AAUCard Examination

*This section will examine the AAUCard implementation, and how cards are validated. It will also examine the services deployed around AAUCard.*

### 2.3.1 MIFARE Classic 1K Vulnerabilities

The possibility of cloning the MIFARE Classic 1K cards relies on the key authentication and encryption done on the card. The algorithm used for these operations was developed by NXP and is proprietary.

In 2008 several papers reverse engineered the algorithm used in the cards called Crypto1 [15]. Since then, many exploits have been found which enables an adversary to extract the keys from the cards without access to a genuine reader. Some of the weaknesses associated with the exploits are summarised below [16].

- The key size is only 48 bits, which may seem sufficient as every authentication request to the card takes six milliseconds. However, if an adversary knows the weaknesses of the Crypto1 algorithm, the key can be brute-forced with an offline attack.
- When authenticating, the cards sends a random challenge to the reader, which it must answer correctly. However, the pseudo-random number generator used to generate these challenges has an entropy of only 16 bits, meaning these challenges are repeated often.
- When doing nested authentication the challenge will be sent encrypted. And since the challenges repeat, an adversary can predict it and extract 32 bits of the keystream.
- The internal state of the cipher is handled by a LFSR (Linear Feedback Shift Register) which state is initially set to the sector's secret key. Given a state and the data fed to the LFSR an adversary can roll it back to its initial state, since it is deterministic, which is the secret key.

The attack described in Meijer and Verdult [16] has been implemented in multiple free/libre and open-source programs, and enables anyone to extract the secret key of a card in a couple of minutes [17]. This key can then be used to manipulate or copy the content of all cards using this key.

### 2.3.2 AAUCard Vulnerabilities

Each AAUCard has its own card number which is stored on the magnet stripe, rfid sector, bar-code, and printed on the card. This number is read out by the reader and is then sent along to be checked. [18]

Using a free/libre and open-source program named *MIFARE Classic Tool* [19] which utilizes the built-in NFC (Near Field Communication) reader in smartphones, one can read MIFARE Classic cards. An example of using this tool can be seen in figure 2.3.





for Information Interchange) encoded Sector 15 was induced.

```
GROUP 4 ;0033310
000XXXXXX0?Y0000
```

Here XXXXXX is the users' card number and Y seemed to vary between 0x30 and 0x3F. From the values of the 12 cards, it appeared that this is a check digit, determined by the following formula.

$$Y_{10} = 48 + \left( \left( \left( \sum_{n=1}^6 d_n \right) - 1 \right) \bmod 16 \right) \quad (2.1)$$

Where  $d_n$  is the  $n$ 'th digit of the card number, 6 is the length of the card number,  $Y_{10}$  is the decimal representation of the ASCII character Y and 48 is the decimal representation of 0x30.

The check digit appears to be used for verifying a valid card number and is calculated. Because the content of sector 15 can be calculated from the card number, an adversary can duplicate cards by knowing their 6 digit card number.

Experiments revealed, that when duplicating a card, both cards are valid and can be used to enter buildings.

As described, the reader is responsible for reading the card number from sector 15, meaning it has access to the key. This places trust on the card readers.

### 2.3.3 AAUCard System Challenges

*The AAUCard system faces a number of challenges as has become apparent through the problem analysis section. Following is a brief summary of the different issues to give an idea of how the system may be improved.*

First and foremost, the requirement of convenience limits the security of a potential system. Most zone 0 faculties will at least have the front doors unlocked in the day time. This is done as more strict door security during the active hours could cause a congestion e.g. when a lot of students try to access the same locked faculty at once. This convenience is generally prioritized rather highly, as such the AAU staff card does not need a PIN code to use AAU printers (see section 2.1.1 on page 8).

Skimming is a risk, but the cards will have limited usage as they still require a PIN code for zone 1 and 2 access. Since AAUCards are of the MIFARE Classic 1K type, there is also a lot of fast and reliable skimming tools available.

Büttner mentioned during the interview that the pictures submitted when a student orders a new AAUCard are not checked. This way students could order an AAUCard with a picture of someone else. The visual information on existing cards is also not verified by another human, meaning no adversary is going to be stopped from using stolen AAUCards or skimmed cards that look illegitimate. Furthermore, the system will not be able to reliably register and keep track of everyone inside the faculties. This is because

AAUCard holders can hold locked doors for each other and people who entered before a zone was automatically locked can stay inside. This way someone without the PIN code or AAUCard could have access to locked faculties.

A way of restructuring the AAUCard system and making it more secure, could be to use a different type of smart card. However, it is postulated that one of the biggest security flaws in the current system is the use of the same key on all cards. This is because even if the smart card would be upgraded, an adversary could potentially still manage to crack the key, and thereby repeat the process mentioned in section 2.3.2 on page 11. Furthermore, it is postulated that it would be cumbersome to replace all existing AAUCards.

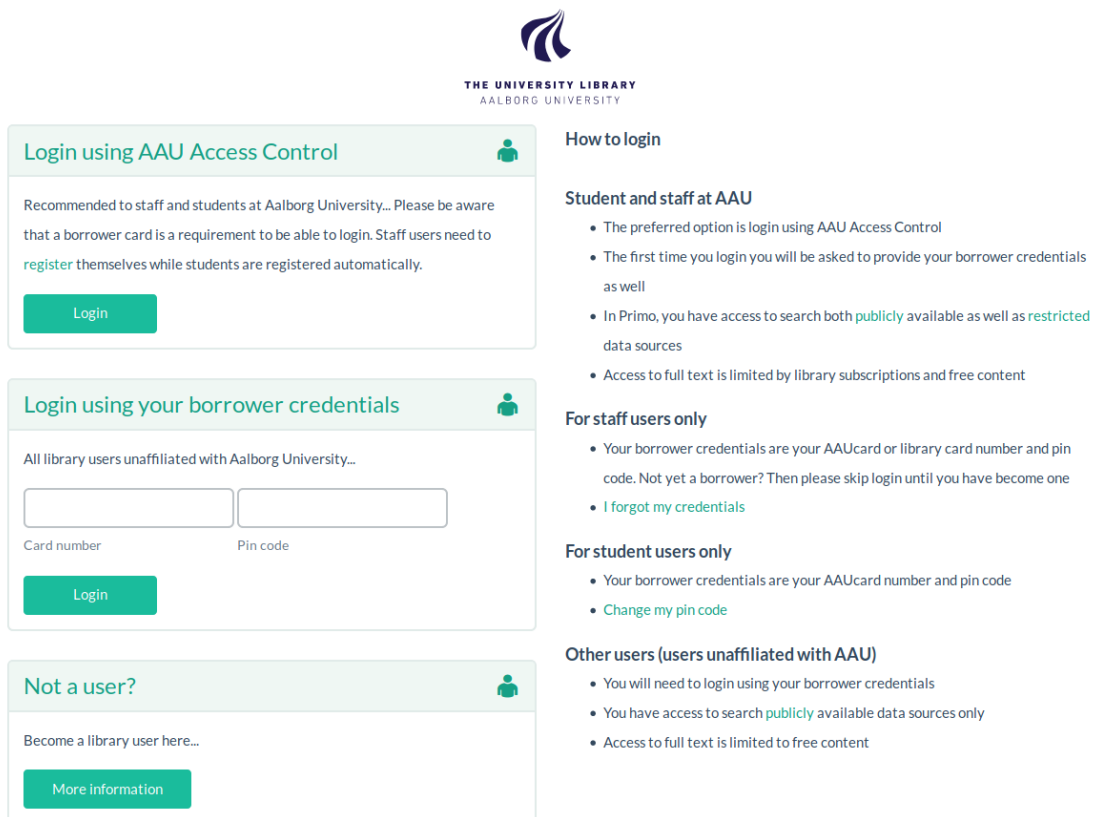
### 2.3.4 AAUCard Web Service Vulnerabilities

*The vulnerabilities described in this chapter have been disclosed to AAU Information Technology Services and they are being handled according to process.*

While examining AAUCards, a few related web services and vulnerabilities were discovered. Namely, an AAU Library login page and an AAUCard PIN code register page.

#### 2.3.4.1 AAU Library Login Page

The AAU Library login page, as seen in figure 2.5 on the next page, allows users to log in to their AAU Library account.



**THE UNIVERSITY LIBRARY**  
AALBORG UNIVERSITY

**Login using AAU Access Control**

Recommended to staff and students at Aalborg University... Please be aware that a borrower card is a requirement to be able to login. Staff users need to [register](#) themselves while students are registered automatically.

[Login](#)

**Login using your borrower credentials**

All library users unaffiliated with Aalborg University...

Card number Pin code

[Login](#)

**Not a user?**

Become a library user here...

[More information](#)

**How to login**

**Student and staff at AAU**

- The preferred option is login using AAU Access Control
- The first time you login you will be asked to provide your borrower credentials as well
- In Primo, you have access to search both [publicly](#) available as well as [restricted](#) data sources
- Access to full text is limited by library subscriptions and free content

**For staff users only**

- Your borrower credentials are your AAUCard or library card number and pin code. Not yet a borrower? Then please skip login until you have become one
- [I forgot my credentials](#)

**For student users only**

- Your borrower credentials are your AAUCard number and pin code
- [Change my pin code](#)

**Other users (users unaffiliated with AAU)**

- You will need to login using your borrower credentials
- You have access to search [publicly](#) available data sources only
- Access to full text is limited to free content

**Figure 2.5:** The AAU Library Login Page

However, an adversary can use this page to brute-force the AAUCard PIN code if the card number is known. This brute-force attack is possible since there is no rate limiting on login attempts and the PIN code is only four digits long, and can not start with a zero, leaving 9000 possible combinations. This allows an adversary to enter any building at any time on behalf of the victim, if the victim's AAUCard has access and the adversary has seen the victim's card number.

*A proof of concept implementation of this was written in Python and GNU Bash, but has been redacted from this report.*

In general, it is not clearly indicated that card numbers are supposed to be kept secret. Card numbers are, for instance, shared between group members when filling out a form to grant access to group rooms.

#### 2.3.4.2 AAUCard PIN code Register Page

The AAUCard PIN code Register Page, as seen in figure 2.6 on the following page, allows users to change their PIN code by logging in with their CPR-number (Civil Registration Number) and card number.

AALBORG UNIVERSITY  
DENMARK

Register personal key to your AAU card

DANISH

Civil register no. (\*)  Type without hyphen.

Card no.  5-6 digit code which appear on your student card.

Continue

(\*) If you do not have a Danish CPR number (Civil registration number), you must write your date, month, and year of birth in that order. Then, write the first two letters of your first given name followed by the first letter of your family name. Finally, you must indicate your gender: 1 for males and 2 for females. A male person named Claes Anders Fredrik Moen, born the 31st of August 1975, must write: 310875CLM1.

NB!!  
If you are using a public computer (i.e. in a library or at the service desk) then you should use an "Incognito mode" of your browser  
(that is how it is called in both Chrome and Firefox, but it is called "InPrivate browsing" in Internet Explorer)

Page 1 of 2

Figure 2.6: The AAUCard PIN code Register Page

However, if an adversary knows a card number and the birthdate of that card holder the CPR-number can be brute-forced. This brute-force attack is possible since there is no rate limiting on login attempts and the amount of CPR-number combinations, excluding birthdate, is 540. If the sex of the victim is known, this can be halved. [20]

If the adversary only knows a birthdate, that multiplies the amount combinations by the

amount of possible card numbers. Yet, as the card numbers seemingly are incrementally generated, the amount combinations can be reduced if the adversary can approximate when the victim was given an AAUCard.

*A proof of concept implementation of this attack was written in The Go Programming Language by a third party, but has been redacted from this report.*

Furthermore, the AAUCard PIN code Register Page does not properly validate session cookies when changing PIN codes, making it possible for an adversary to change the PIN code of any AAUCard without knowing the associated CPR-number.

*A proof of concept implementation of this attack was written in GNU Bash, but has been redacted from this report.*

## 2.4 Requirement Specification

*In this section the requirements for a proof of concept will be examined and explained.*

### 2.4.1 Vision for the AAUCard system

A focus will be placed on the cryptography used in AAUCards and how the related data is handled. It is not the aim to mitigate or prevent every single method of attack, but rather to reduce the risk of AAUCards being copied and to avoid trusting scanners. This means that the door unlocking mechanism and AAUCard theft, as well as methods like LFSR rollback and PICC sniffing, will not be considered.

To reduce the risk of cards being copied, sector 15 on each card will be locked with a unique key. This means that an adversary would have to crack a key for each AAUCard instead of a single key for all AAUCards. Furthermore, scanners will pass data to a server that will handle the cryptographic operations instead, which means active attacks are required by a potential counterfeit or otherwise malicious scanner. If the current cards are to be phased out and replaced with new ones, the system should be able to handle both card types during this period.

### 2.4.2 Requirements

- 1) Access must only be granted to users in accordance with the current roles and policies (see section 1.2 on page 4).

*A user should not be able to enter a building without proper permission given in the system.*

- 2) Each AAUCard must be *protected* with a unique key.

*As mentioned in section 2.3.2 on page 11 if an adversary manages to crack the key of the AAUCard it allows for reading all AAUCards.*

- 3) Must be able to work with more than one scanner.

*The server has to be able to handle multiple doors, since two users might need to enter two different doors at nearly the same time, without extra delay.*

- 4) The scanner must not have access to a card's secret data or unique key.  
*This would defend against an adversary installing malicious software on the scanner or replacing it with a counterfeit scanner.*
- 5) Must reuse the MIFARE Classic 1K cards.  
*As mentioned in section 2.3.3 on page 14, the use of a different smart cards is not a practical proposition.*
- 6) The time between scanning a card and the prompt for a PIN code, excluding network delay, must be within one second.  
*As mentioned in section 2.1.1 on page 8, convenience is a high priority of the AAUCard, which means that it should not take too much time to unlock a door.*

## 2.5 Proposal of Two Systems

As per requirement 4 in section 2.4 on the preceding page the scanner must not have access to a card's secret data or unique key. Due to the technical complexity of this requirement, two possible systems have been proposed; *system a* and *system b*. Both systems use MIFARE Classic 1K cards for authorization, and aim to comply with all requirements bar requirement 4.

The main difference between the two systems is that *system a* trusts the scanners, while *system b* don't. In *system a* the scanner performs Crypto1 operations as opposed to *system b*, in which the server performs Crypto1 operations, making it possible for the scanner to only pass data.

*System a* can be implemented without much difficulty, as the Crypto1 operations are already implemented on the MFRC522-chip used in the scanner. This leaves more time for development and implementation of desirable features.

In *system b* however, the proprietary Crypto1-algorithm has to be implemented on the server. This ought to be more time consuming than implementing *system a*.

**Table 2.1:** Properties of the two proposed systems

	Crypto1 operations	Amount of features	Security
<i>System a</i>	Performed by the scanner	High	Less desirable
<i>System b</i>	Performed by the server	Low	Desirable

Table 2.1 showcases how the two systems properties is a trade-off between security and features.

Due to the preferable security properties and compliance with all requirements, *system b* will be the one developed.

## 2.6 Final Problem Statement

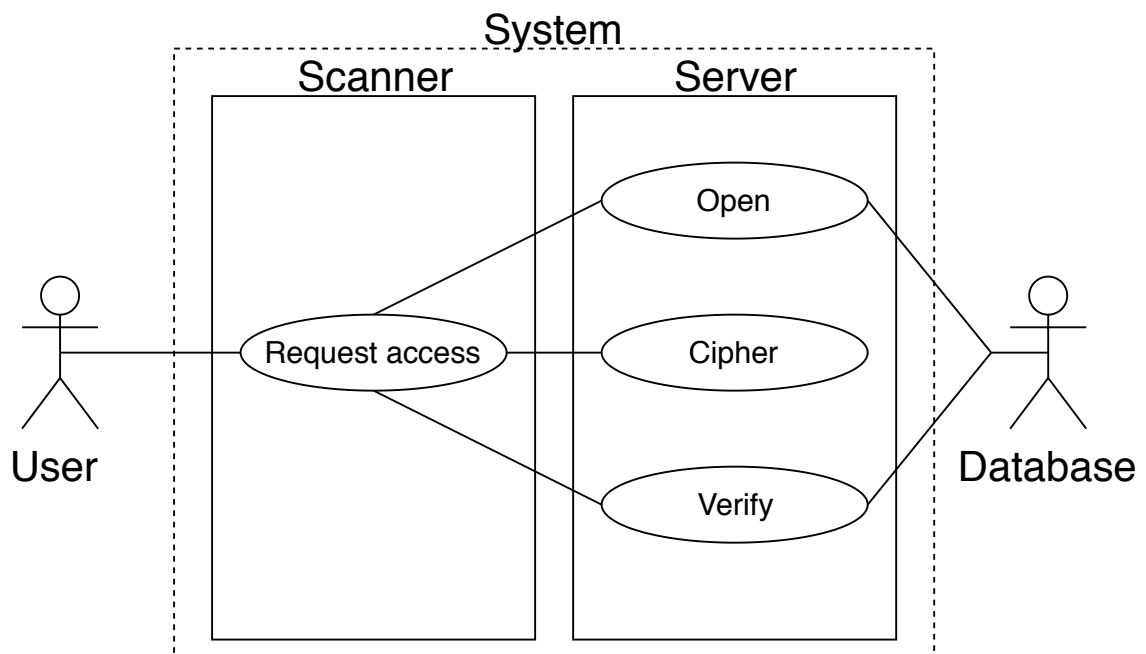
*Through the analysis done in chapter 2 on pages 7–16 it was found, how the current access control system works on AAU. The requirements for an improved system has been made based on research and an interview done at AAU. The following final problem statement has been defined:*

**What measures can be taken to reduce the risk of access control system exploitations at Aalborg University?**

# Development 3

*A system overview is conceived through the information gathered in the problem analysis and requirements set in the requirement specification.*

A distinguishing measure taken against exploitations is that the scanner is not trusted, which is possible by keeping the scanner from accessing sensitive data that could be used to duplicate a card. All verification is also done server-side. This process is visualized through the following use-case diagram:



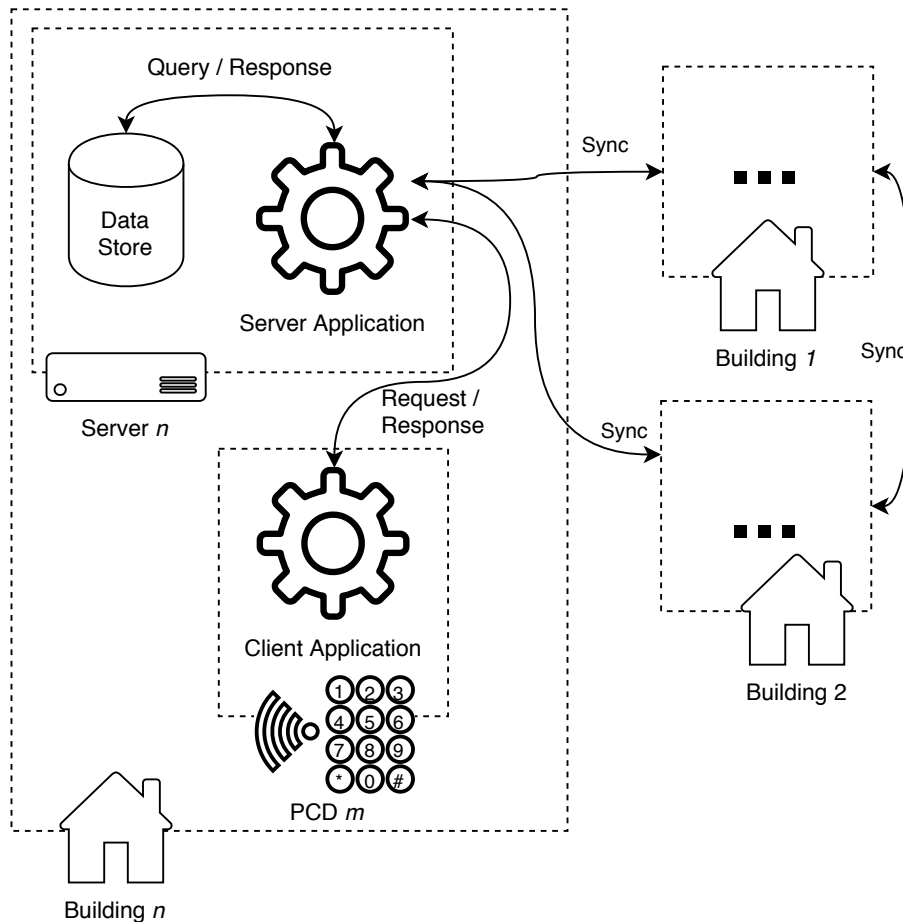
**Figure 3.1:** A use-case diagram showcasing the interactions and associations between scanner and server.

Each line in figure 3.1 on the next page represents an interaction. All possible interactions are described below. As this proof of concept will not support a physical keypad for the user to utilize, that is not included on the use-case diagram.

- **Request access:** Enables the user to scan their PICC, prompting the authentication and access protocols. The PICC will communicate with the scanner, and by proxy the server by using the scanner as a middleman, in order to gain access.
- **Open:** Enables the scanner to start a cryptography session. The validity of the scanner will also be checked. If the scanner is not valid, it will not have access to the 'Cipher' and 'Verify' use-cases.

- **Cipher:** Enables the scanner to perform various cryptographic operations on the server.
- **Verify:** Enables the scanner to verify a card. If a card is invalid, it will be denied access.

An overview of system design behind the scenes can be seen in figure 3.2 on the following page.



**Figure 3.2:** Client-server overview

The design consists of one server per building and scanners acting as clients, that request whether or not a given card should unlock a given door.

If a higher level of redundancy is needed then two servers, with a fail-over mechanism, could be placed in each building. Alternatively, if the clients have internet access they could fail-over to a server in another building.

However, the synchronization between servers will not be implemented for now, due to resource limitations. As such, fail-over will not be implemented either.

A server consists of a server application that will handle requests and query a data store accordingly.



## 3.1 Environment and Deployment

While developing, Git on GitLab was used for version control. GitLab CI (Continuous Integration) is configured with a file. The content of a sample file is shown in snippet 3.1 on the next page.

```
1 image: golang:1-alpine
2
3 test:
4   script:
5     - apk add --no-cache git
6     - go get -d -v ./...
7     - go test ./...
```

**Snippet 3.1:** GitLab CI configuration.

This configuration makes an instance of a GitLab runner install dependencies and run tests on all suggested changes. This means that it is insured that changes to the program can compile and pass all automated tests.

For developing the server locally with a database present, *docker-compose* was used. *docker-compose* allows for setting up a development environment with a single configuration file. A sample file is shown in snippet 3.2.

```
1 version: '3'
2 services:
3   db:
4     image: postgres:11.5-alpine
5     environment:
6       POSTGRES_PASSWORD: changeme
7
8   madong:
9     depends_on:
10      - db
11     image: madong:madong
12     build: .
13     ports:
14      - 4444:4444
```

**Snippet 3.2:** docker-compose file content

While production grade deployment is out of scope, the server was deployed using *docker-compose* for testing and development.

## 3.2 Data Store

*To keep track of card holders, credentials and such, persistent storage is needed. Throughout this section, the content, type, and usage of this data store is addressed.*

### 3.2.1 Table used for scanner authentication

Since only registered clients should be able to interact with the application, the application should deny any clients that are not registered scanners. This is done by having a cryptographic public key associated to each scanner, such that the application can accept and reject connections depending on whether or not the scanner can cryptographically sign its requests with the corresponding private key. Doing this also allows for revoking scanners by removing their associated public key from the database. This public key entry can simultaneously be used as a UID (Unique Identifier) if the client sends it alongside its request, such that the application knows which public key to use, if it is registered when checking the signature. An example of a table that can be used for scanner authentication is shown in table 3.1 on the next page.

**Table 3.1:** Scanner authentication information

Scanner Public Key	Zone	Location
MCowBQYDK2VuAyEAfXK7 5/+kiV1h6XoKefUWh3YL s5QkNmgy3c9LQBW11Xw=	1	Fredrik Bajers Vej 7 B1-201

This example has a single scanner, registered by its public key, which in this example is generated from Curve25519 using OpenSSL<sup>1</sup>, a location, and a zone. The location is only there to make configuration of the scanners easy. The application could for instance, make sure that two doors to the same room are not in different zones. However, it would require a consistent naming scheme, which is deemed out of scope, for locations to make those kind of checks.

The zone entry is used to look up the associated policies and authorize according to the policy table described in section 3.2.2. Zones follow the definition from section 2.1.1 on page 8, where each scanner belongs to a zone with its own access policies, i.e. the outer building door scanners are zone 0 while the inner door scanners are zone 1.

### 3.2.2 Table used for authorization policies

As mentioned in section 2.3.3 on page 14, different zones have different policies associated with them. In order to keep track of a given zone's policy, states that represent policies are introduced. The states and their corresponding abbreviation and number are shown in table 3.2.

<sup>1</sup>This may differ in the actual implementation depending on library availability etc.

**Table 3.2:** Possible states of an authorization policy

<i>State</i>	<i>Abbreviation</i>	<i>Number</i>
Unlock for noone	Noone	0
Unlock if card and PIN code are provided	PIN	1
Unlock if card is provided	Card	2
Unlocked	All	3

This can be used in cases like the example in table 3.3 on the facing page, where all cards with the "student" role, have to provide both their card and PIN code in order to gain access to zone 0 between 16:00 and 8:00.

**Table 3.3:** Authorization policies

<b>Zone</b>	<b>Role</b>	<b>Begin</b>	<b>End</b>	<b>State</b>
0	student	16:00	8:00	PIN

Policies can overlap and/or contradict each other, e.g. if two policies are identical except for the time interval. The application will need some mechanism to handle or prevent this.

Of course, authentication of cards need to be done in order to determine its roles and whether or not it is valid. This is done using the table described in the following section.

### 3.2.3 Table used for card authentication

To authenticate a card, the application needs to check some secret card data. In the case of AAUCard this is a card number and a check digit, which is not particularly secure, as discussed in section 2.3.2 on page 11. Using random data, which only use is as secret card data, is more secure as it not printed on the cards, or used during exams etc. Since one 16 byte block of a sector is read by the scanner at the time, the obvious way to implement this is by having 16 bytes of random data stored in a locked sector.

As such, the table has a column for secret card data. However, if the secret data was stored as clear text and the database was compromised, all cards could be duplicated. Therefore, a hash of the secret data is stored instead.

### Storing Secret Data

Simply hashing the secret data is not sufficient to protect it against a potential compromise as it has to be a CHF (Cryptographic Hash Function), meaning it is suited for cryptographic use (see [21]). It should be noted that if the selected CHF is too fast, an adversary could potentially reverse hashes via brute-force i.e. hash all possible combinations and check if the hash of the guessed combination and secret data match.

In fact, the selected CHF must be intentionally slow, especially if the data you wish to hash is suspected to be low entropy like passwords created by unaware victims.

Entropy is an amount of possible combinations. The possible PIN code combinations can be described with  $10^n$ , where 10 is the possible symbols 0-9 and  $n$  is the length of the PIN code. A PIN code with the length of 4 becomes  $10^4$ , meaning there's 10 000 combinations in total. Entropy is often expressed in bits, i.e. a PIN code with 10 000 possible combinations has an entropy of 14 bits, as the binary of 10 000, 10011100010000, consists of 14 bits. It is however important to note that since all bits are not 1, the entropy would instead fall somewhere between 13 and 14 bits. Note that this is assuming that the PIN code is perfectly random, otherwise, entropy would be reduced. For instance, if the PIN code was a birthday the entropy would be 365, which is between 8 and 9 bits of entropy. [22]

A CHF can be intentionally slow by using an inherently slow CHF or doing multiple rounds of a CHF. Doing multiple rounds of a CHF is called key stretching and increases the time it takes to compute a hash equally on the defending- and attacking side [21]. Additionally, to prevent an adversary from precomputing hash values, some amount of random data, called a salt, must be ap- or prepended before hashing. 16 bytes of random data has a high entropy of  $2^{16 \cdot 8} = 128$  bits and therefore will not need a slow CHF to protect.

However, a 4-digit PIN code has a low entropy. Given the requirement that opening a door does not take longer than one second and that key stretching increases time equally on the defending- and attacking side, an adversary would be able to brute-force for any PIN code hash in less than 10 000 seconds or about 2 hours and 45 minutes. Note that the 2 hours and 45 minutes is in the best case, where an attacker does not have more powerful hardware than the defender had at the time of hashing. This means that hashing with key stretching would not provide much protection for the PIN codes in the case of the database being compromised, since they do not have enough entropy.

## Key Strengthening

Another scheme called key strengthening might be useable. Key strengthening introduces the notion of a public- and private salt, where the public salt is a regular salt but the private salt is deleted immediately after its use. [23]

The scheme can be expressed as follows.

$$X = H(H(K, s_p), s_s) \quad (3.1)$$

Where

$K$ , is a PIN code or similar secret.

$s_p$ , is a public salt.

$H()$ , is a CHF.

$s_s$ , is a private salt.

$S(X)$ , is the entropy of  $X$ .

The private salt has to be brute-forced and deleted by the defender each time a PIN code is to be verified, which is an additional step with computational complexity of  $O(n)$ . However, the benefit of key strengthening is that if  $K$  is known,  $S(X) =$

$S(s_s)$ , else  $S(X) = S(s_s) \cdot S(K)$ . This means that, while the computational complexity for the attacker is still  $O(n)$ ,  $n$  is larger for the attacker by a factor of  $S(K)$ . The result of this is no better than key stretching, since key stretching also effectively puts the attacker at a disadvantage by factor of  $S(K)$ . Some example calculations were made to verify this, and the source code used to make those calculations can be found in snippet A.1 on page 67.

### Possibility of Increasing PIN Code Entropy

To increase entropy of the PIN code, a rhythm could in principle be used. In that case a sample PIN code could be 1..2.3...4 where each dot represents a delay with some tolerance. While this would add entropy, it would be required to teach card holders that rhythm is being measured and used as part of the PIN code. Furthermore, it is postulated, that it would require quite a bit of implementation work, especially if the PIN code is to be used on web pages.

The PIN codes could also be more than four digits though it is postulated that this is not worth the loss of convenience.

In conclusion, with regards to the PIN codes, as long as they do not have sufficient entropy, it does not provide much benefit to hash them before storage and they will therefore be stored in cleartext for now, even though it is suboptimal.

### Manufacturing Unique Identifier and Personal Unique Identifier

As seen in the example in table 3.4 on the next page, there are also columns for a 48 bit hash of the MUID, 48 bit key, roles, and a PUID (Personal Unique Identifier). It is assessed that a 48 bit hash of the MUID provides a reasonable balance between collision resistance and printability. The PUID is a canonical term for student- or staff number respectively. The remaining values are to be used by the application to authenticate and authorize properly.

**Table 3.4:** Card authentication information

MUID Hash	Key	Secret card data	PIN	Role	PUID
8ae508aae8c6	22 45 33 F1 A2 01	\$5\$L/qwDQcBUa\$é0 /qD5qoJmHFC/4pW Meq5JQXzOUV0bST bBaeettSDTC	1337	student	123456

The hash of the secret card data is in the format  $\$<ID>\$<SALT>\$<PWD>$  and in the example on table 3.4, the ID of the hashing algorithm is 5, which corresponds to SHA-256 (Secure Hash Algorithm) from the SHA-2 family as implemented in the GNU C Library<sup>2</sup>. The salt is random data as discussed previously. Lastly, PWD corresponds to the hashed password. [24]

<sup>2</sup>This may differ in the actual implementation depending on library availability etc.

### 3.2.4 Database

PostgreSQL was chosen as the database system, due to it being a RDBMS (Relational Database Management System), which allows for records to have attributes with relations to other tables.

PostgreSQL supports a subset of the American National Standards Institute's SQL (Structured Query Language) standard, which is a standard for SQL commands, where PostgreSQL uses similar syntax with some additional functionalities. One of which is native JSON (JavaScript Object Notation) support, which enables unstructured data.

#### 3.2.4.1 Table Relations

The sections above describe the different content types and their structure. This section will describe its implementation using PostgreSQL.

Figure 3.3 on the next page describes the different tables and their references. A key icon is used to mark a table's primary key, which must be a unique value in each row.

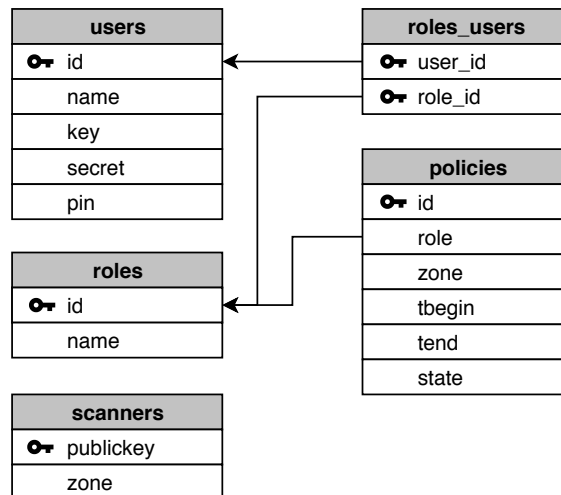


Figure 3.3: Database structure

Because a role can contain multiple policies it employs a *many-to-one* relationship, meaning each policy references its parent role. As seen in snippet 3.3 on the facing page this reference is implemented with a simple integer value *role* referencing *roles.id*.

Snippet 3.3 on the next page shows the SQL definition of the policies table. The SQL **REFERENCES** keyword constraints the integer value to a valid *roles.id* [25].

```
1 CREATE TABLE IF NOT EXISTS policies(  
2     id        SERIAL PRIMARY KEY,  
3     role     INT REFERENCES roles(id) ON DELETE CASCADE,  
4     zone     INT,  
5     tbegin  TIME WITHOUT TIME ZONE,  
6     tend    TIME WITHOUT TIME ZONE,  
7     state   INT  
8 );
```

**Snippet 3.3:** Creation of policies table

The roles\_users table functions as a *many-to-many* relationship between roles and users. Here users can reference multiple roles and roles can reference multiple users.

When a user joins a role a new row will be added to roles\_users referencing the user and role. The primary key of this table is a pair of user\_id and role\_id, meaning no two rows can share user\_id **and** role\_id.

#### 3.2.4.2 SQL statements

In order to manipulate tables and retrieve data SQL statements was used.

The library sqlx was used to facilitate the communication with the database in Golang [26]. sqlx is an extension to the standard SQL library and integrates well with Golang structures. The sqlx library has multiple methods to make queries to the database, where all takes SQL statements as strings.

Furthermore, as previously mentioned these methods integrate well with Golang in that the data that is either passed or retrieved can either be passed or stored with Golang structures parameter.

An example of a SQL statement that is used in the system can be seen in snippet 3.4 on the facing page. This statement was used to find all users with a specific role.

```
1 SELECT users.id, users.name FROM users  
2 INNER JOIN roles_users ON users.id = roles_users.user_id  
3 INNER JOIN roles ON roles.id = roles_users.role_id  
4 WHERE roles.name = $1
```

**Snippet 3.4:** An example of a SQL statement used in the system

In snippet 3.4 a SQL statement is shown, which starts with a SQL command **SELECT** that will try to find the **user.id** and **user.name**. The SQL command **INNER JOIN** is used to linking tables, by specifying which values they share. When using **INNER JOIN** it creates a new extended table which can be seen below:

**Table 3.5:** Users table

id	name
1	John Doe

**Table 3.6:** Roles\_users table

user_id	role_id
1	5

**Table 3.7:** Roles table

id	name
5	student

**Table 3.8:** INNER JOIN on the tables above

id	name	user_id	role_id	id	name
1	John Doe	1	5	5	student

Finally, the last SQL command used in snippet 3.4 is **WHERE**, which is filtering all records with the \$1 parameter, which in this case is **student**.

### 3.3 Server Communication

According to figure 3.2 on page 20 communications between server and clients use the request-response model. This dictates that clients sends a request followed by a response by the server. A request could be "get user information", "change password", and a response can be a list of users or "password changed".

This communication is one-way, meaning a server can not directly address a client. The server can therefore only send information when a client requests it.

#### 3.3.1 Hypertext Transfer Protocol

Communication will be implemented using HTTP (HyperText Transfer Protocol) which defines request types and how these are exchanged. HTTP has been chosen because it is a simple protocol while still being flexible. It also has extensive support in most programming languages and software. [27]

An example of a HTTP request can be seen below;

```
GET /welcome.html HTTP/1.1
Accept-Language: da-DK
```

**Method** The method describes the type of the request. The example above is a **GET** request, meaning it will retrieve data from the server [28].



**Path** Path to access, which is used to describe the resource on the server to be accessed. The meaning varies depending on the context, but the path is often used to identify a page or file on a server. In this case it points to the file `welcome.html` in the root of the server.

**Headers** Additional headers can be used to send additional information to the server. They are in the format `<NAME>: <VALUE>` and contain information like cookies, authentication, etc. In aforementioned example a header is included to request the page in a specified language. [29]

**Body** After the headers an optional request body can be included. Whether it is included or not depends on the request method. In the example above, the body is empty because a `GET` request will retrieve data and not send anything.

When the server receives a request it processes it and sends back the result. In the example above, the server could read the danish version of the file `welcome.html` from disk, optionally add some information, and send it. Continuing that example, the HTTP response would look like the following;

```
HTTP/1.1 200 OK
Server: julian/2.0.1
Content-Type: text/html
Content-Length: 83

<html>
<h1>Velkommen til den seje server</h1>
<p>Klokken er nu 15:43</p>
</html>
```

The first line of the response is different as it includes a http version followed by a status code. A status code describes if the request was successful or not and how.

Similarly to the HTTP request, the response contains headers and a body. In this case the body is filled out with a Danish version of `website.html` and the current time as determined by the server. It has also two headers included which describe the body's type and length.

### 3.3.2 Application Programming Interface

On top of HTTP, an API (Application Programming Interface) is implemented. Instead of a visual HTML (HyperText Markup Language) page, which is usually associated with HTTP, the API will define requests and responses which can easily be parsed by computers. This makes interactions with scanners simple.

The API consists of a set of endpoints, each represented by a URL (Uniform Resource Locator), that allows for various HTTP requests.

The API was written in Golang and was created with a goal of being a RESTful (REpresentational State Transfer) API. That was accomplished with the use of a library named Gorilla mux, which is a HTTP router and URL matcher. When the user accesses a resource through a URL the router will find the corresponding function and execute it.

All the API endpoints are prefixed with */api*, to make it clear that a resource is associated with the API. Furthermore, there are multiple sub routes under the API: */users*, */scanners*, */crypto*, */roles*, and */policies*. These sub routes allow for a more organized API, where all endpoints that operate on users are prefixed with */api/users/*.

Therefore, an example of an endpoint would be: `http://localhost:4444/api/roles/{role}/users`, which only accepts GET requests, and would return a list of all users that has the specific *{role}*.

The full list of endpoints and what requests they take can be seen on a GET request to `http://localhost:4444/api`, and the response:

- 1) { "href": "/api", "method": "GET" }
- 2) { "href": "/api/git", "method": "GET" }
- 3) { "href": "/api/scanners", "method": "POST" }
- 4) { "href": "/api/scanners/zone", "method": "GET" }
- 5) { "href": "/api/users", "method": "GET" }
- 6) { "href": "/api/users", "method": "POST" }
- 7) { "href": "/api/users/{card}", "method": "GET" }
- 8) { "href": "/api/users/search/{query}", "method": "GET" }
- 9) { "href": "/api/users/{card}", "method": "DELETE" }
- 10) { "href": "/api/users/{card}/roles", "method": "GET" }
- 11) { "href": "/api/users/{card}/roles", "method": "POST" }
- 12) { "href": "/api/users/{card}/roles", "method": "DELETE" }
- 13) { "href": "/api/users/{card}/updatepin", "method": "PATCH" }
- 14) { "href": "/api/users/{card}/genkey", "method": "GET" }
- 15) { "href": "/api/roles", "method": "GET" }
- 16) { "href": "/api/roles", "method": "POST" }
- 17) { "href": "/api/roles/{role}", "method": "PATCH" }
- 18) { "href": "/api/roles/{role}", "method": "DELETE" }
- 19) { "href": "/api/roles/{role}", "method": "GET" }

- 20) { "href": "/api/roles/{role}/users", "method": "GET" }
- 21) { "href": "/api/roles/{role}/policies", "method": "POST" }
- 22) { "href": "/api/policies/{policy}", "method": "PATCH" }
- 23) { "href": "/api/policies/{policy}", "method": "DELETE" }
- 24) { "href": "/api/encrypt/open/{card}", "method": "GET" }
- 25) { "href": "/api/encrypt/cipher", "method": "POST" }
- 26) { "href": "/api/encrypt/verify", "method": "POST" }

A REST API is an architectural style of writing an API, and in order for the API to be completely 'RESTful' it must follow some constraints. The API designed in this system is not fully 'RESTful', because it violates the constraint that the API should be stateless. This means that all state needed to handle a request should be included in the request itself. [30]

Because of the nature of the Crypto1 cryptographic operations (see section 3.4.2 on page 36) this constraint is hard to implement and will therefore be ignored.

As previously mentioned, when a request is made to an endpoint, a respective handler function for that endpoint is called. An example of this could be if a GET request were made to `/api/git` the URL matcher will find this router: `api.HandleFunc("/git", md.httpGitGet).Methods("GET")`, where `httpGitGet` is the handler function that is called.

Handler functions manages both decoding JSON and all SQL commands, and an example of a function that handles both is the creation of a new user, which is a POST request to `/api/users` and can be seen in snippet 3.5 on the next page.

```
1 func (md *Madong) httpUsersNew(w http.ResponseWriter, r *http
  .Request) {
2
3     var card Card
4
5     // Read user input
6     err := json.NewDecoder(r.Body).Decode(&card)
7     if err != nil {
8         http.Error(w, err.Error(), http.
          StatusBadRequest)
9         return
10    }
11
12    _, err = md.db.NamedExec('INSERT INTO users (name, id
      ) VALUES (:name, :id)', &card)
13    if err != nil {
14        http.Error(w, err.Error(), http.
          StatusInternalServerError)
15        return
16    }
17
18    // Set the header to StatusCreated to indicate that a
      new resource was created
19    w.WriteHeader(http.StatusCreated)
20 }
```

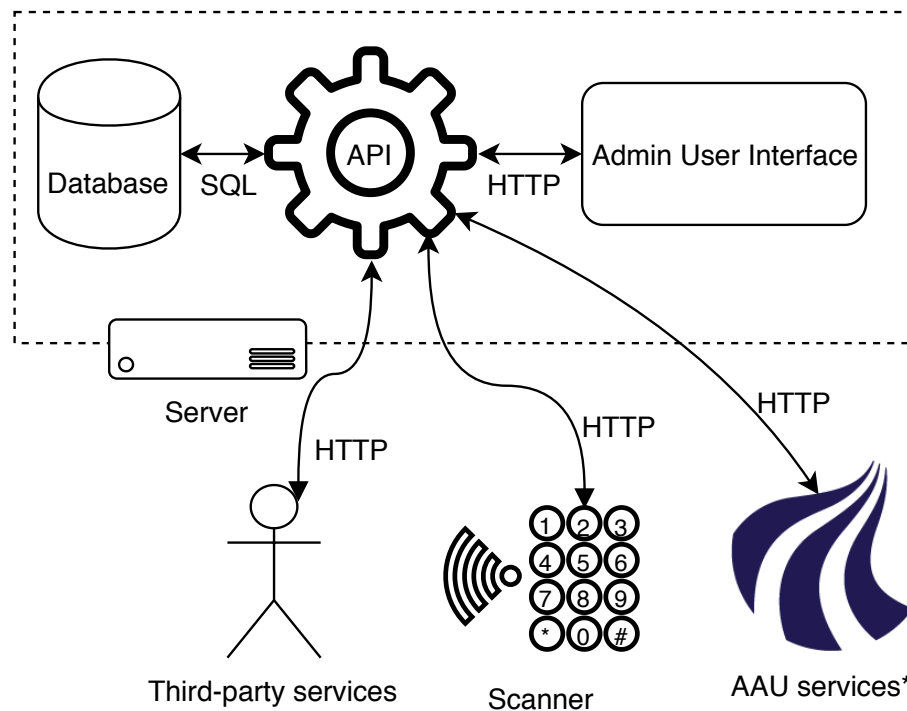
Snippet 3.5: Create new user handler

The code in snippet 3.5 on the following page decodes JSON, which happens in `json.NewDecoder(r.Body).Decode(&card)` where `r.Body` is the request input in JSON format, this data is then put into a Golang struct `&card`. A method `NamedExec` from the `sqlx` library is also used, which takes two arguments: *A query string* and *Interface{}* i.e. the data that is to be inserted. Afterwards, if no error, the header for the `http.ResponseWriter` is set to the HTTP status code 201, indicating a new resource was created.

All responses from GET requests give: Error, data, and links. *Error* is left empty in case of a successful request, *data* is empty in case of an unsuccessful request, and *links* is a list of other endpoints in that subrouter, where the addition of links is in order to make it more 'REST-like'.

One of the reasons for building an API was to give ease of implementation of third-party services, a case could be the aforementioned study café at AAU which uses the AAUCard. All that would be needed is for the user to generate a public- and private key, and then for the administrators of the system to insert the public key into the database. Note that authentication for administrators is not implemented in this proof of concept.

An illustration of the intended use of the API can be seen in figure 3.4 on the next page.



**Figure 3.4:** An illustration of the intended vision for the API. \*AAU services refers to all extra usages of the AAUcard on campus e.g. printers.

In figure 3.4 it can be seen that the API is located on the same server as both the Admin UI (User Interface) and the Database, which allows for a portable system, that would ease the deployment elsewhere than AAU.

Furthermore, modern browsers contain a security feature called CORS (Cross-Origin Resource Sharing) which limits the sites the Admin UI can request. If a request was to be made outside of the server where the Admin UI runs, the server would need to white-list this other server with a *Access-Control-Allow-Origin: http://domain.example* header [31].

### 3.3.3 Scanner authentication

Currently, endpoints can be accessed from every device on the network. This is an issue on critical endpoints which set or retrieve sensitive information. This section will only focus on scanner authentication and not administrator access.

When a scanner connect to the server, the server must be able to verify that it's a trusted scanner. This is achieved using public key cryptography, which uses an asymmetric key pair instead of the traditional symmetric key.

Traditionally encryption and signing of messages could be done using a shared key. Decryption and encryption is then done using the same key and should therefore be held secret. This greatly complicates distribution and storage of such a key.

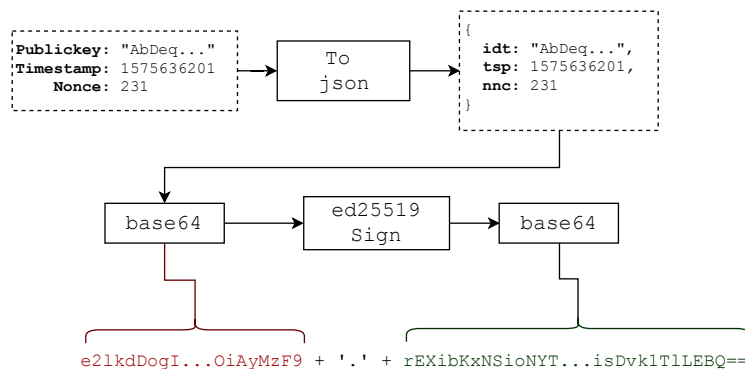
Public key uses a pair of two keys; one that can be shared publicly (*public key*) and one that is kept secret (*private key*). One can then encrypt messages using the public key which can only be decrypted by the secret key. [32]

This utilises a *trap-door one-way permutation* function, which has the following properties. [32]

One-way	A function that is easy to calculate one way but hard the other way.
Trap-door	A <i>one-way</i> function which is easy to reverse once private information is known.
Permutation	Output of the method is itself a valid input and vice versa. This property enables message signing, which is the feature used to authenticate scanners.

When a scanner wants to access an endpoint it will prepare an authentication token. This includes the public key of the scanner, unix timestamp and a random nonce, which is serialized using JSON (JavaScript Object Notation).

The token is signed using the ED25519 signature scheme, that provides secure and fast signing. Its public- and private keys are also short, thus making delivery easy [33]. Figure 3.5 on the next page shows visually how the tokens are generated.



**Figure 3.5:** Token generation

The token is then sent along with a request requiring authentication. The server will then verify it by checking the following;

- 1) Public key exists in database
- 2) Nonce has not been used in the last two minutes
- 3) Timestamp is not more than one minute old
- 4) Signature is valid

Tokens are therefore only valid for one minute and because the server keeps track of used nonces, tokens cannot be repeated. This means that an attacker cannot act on behalf on a scanner without its public key.

However, this implementation does not sign request content which can be exploited by changing it underway. This is possible if the content is sent as plain text.

Because the scanners public key can be shared publicly, it can also easily be used for identification. This means that to add a new scanner one would only have to copy its public key into the database.

## 3.4 Scanner

*This section details the implementation of the physical PCD (Proximity Coupling Device) for the purpose of reading PICCs (Proximity Integrated Circuit Card) and enabling secure data transmission between the PCD and server.*

### 3.4.1 Hardware

To create a working physical PCD, the Raspberry Pi 3 SBC (Single Board Computer) was chosen. This is because the Raspberry Pi 3 supports the TLS (Transport Layer Security) cryptographic protocol, providing security of data in transit. Another important feature is the Debian-based OS (Operating system) of the Raspberry Pi 3, Raspbian, providing useful Linux capabilities such as networking. Furthermore, the Raspberry Pi 3 already has a number of relevant libraries available that assist in the implementation of PCD functionalities. The Raspberry Pi 3 was also readily available to borrow. Two nearly identical Raspberry Pi 3 PCDs were made in order to test how the system handles more than one PCD.

Each Raspberry Pi 3 is connected to a breadboard through a flat ribbon cable, where a RC522 RFID Module is seated. The module is based on the MIFARE RC522 IC (Integrated Circuit), and serves as an ISO 14443 compliant reader/writer, capable of communicating with PICCs such as the MIFARE Classic 1K [34].

The connections between the RC522 module and Raspberry Pi 3 pins are shown in table 3.9 on the facing page.

**Table 3.9:** Pin connection between RC522 and Raspberry Pi 3. IRQ is set to "none", as the feature is not used.

RC522	Raspberry	Description
SDA	Pin 24	Serial Data (Input/Output)
SCK	Pin 23	SPI Serial Clock (Input)
MOSI	Pin 19	SPI Master Out Slave In
MISO	Pin 21	SPI Master In Slave Out
IRQ	None	Interrupt Request
GND	Pin 6	Ground
RST	Pin 22	Reset
3.3V	Pin 1	DC Power

The IRQ (Interrupt Request) function allows the processor to temporarily stop a program. It will not be implemented in this proof of concept PCD, as the library already handles the process. If it were to be implemented, it would provide support for *wait for* types

of programs, allowing for better control under certain events. An example of this can be how the scanner waits for a new card or handles card I/O (input/output) scenarios.

Moreover, the proof of concept PCD is not fitted with a physical keypad to enter the PIN code, and as such the implementation instead takes advantage of PIN codes being stored in the database. In a future iteration of the system the PCD should be responsible for handling and passing the PIN code.

### 3.4.2 Communication between components

The RC522 module supports the SPI (Serial Peripheral Interface) bus, allowing for high-speed serial communication up to data speeds of 10 Mbit/s. During SPI communication, the RC522 module acts as a *slave* while the host, in this case the Raspberry Pi 3, acts as a *master*. It is possible for the master to support multiple slaves, such as if the Raspberry Pi 3 was connected to multiple modules. [34]

The master is responsible for the SPI clock signal, which is a clock edge that oscillates between high and low during data transfers. This ensures synchronization, meaning the master and slave(s) read and write the data lines at the right time, so every single bit is received as intended. The MOSI (Master Out Slave In) line lets the slave receive data from the master, while the MISO (Master In Slave Out) line is used to send data to the master from the slave. With the RC522 module, both lines must have stable data as the clock oscillates from low-to-high, where the lines typically are read, while the data can be changed during high-to-low transitions. Furthermore, the MSB (Most Significant Bit) is always sent first on either line. In other scenarios, the SPI protocol may require the LSB (Least Significant Bit) first, but the RC522 module specifically requires the MSB first. The MSB is the left-most bit in a binary number representing the highest digit, while the LSB is the right-most bit in a binary number representing the lowest digit. [34]

To read data from the RC522, following byte order must be used:

**Table 3.10:** Byte order when using SPI to read data. First byte defines mode and register address. The MISO line is set to "undefined" when content doesn't matter. [34]

Line	Byte 0	Byte 1	Byte 2	Byte ...	Byte $n$	Byte $n+1$
MOSI	address 0	address 1	address 2	address ...	address $n$	00
MISO	undefined	data 0	data 1	data ...	data $n-1$	data $n$

To write data to the RC522, following byte order must instead be used:

**Table 3.11:** Byte order when using SPI to write data. First byte defines mode and register address. The MISO line is set to "undefined" when content doesn't matter. [34]

Line	Byte 0	Byte 1	Byte 2	Byte ...	Byte $n$	Byte $n+1$
MOSI	address 0	data 0	data 1	data ...	data $n-1$	data $n$
MISO	undefined	undefined	undefined	undefined ...	undefined	undefined

The address bytes as they appear in table 3.10 on the next page and table 3.11 are defined as following:

SPI functionality on the Raspberry Pi 3 is managed through the *go-rpio* Go library, which allows for access to the GPIO (General-Purpose Input/Output) pins. [35] This way the



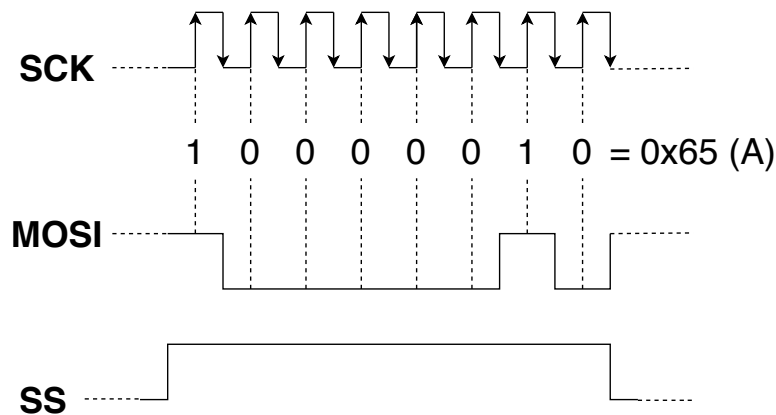
**Table 3.12:** Structure of an address byte as used in SPI read and SPI write. LSB is always a logical 0. [34]

Bit	Contains
7 (MSB)	1 = read from RC522, 0 = write to RC522
6 - 1	Address
0 (LSB)	0

Raspberry Pi 3 can freely use the GPIO pins for I/O with other hardware. Except for the SPI clock speed which is set to 4 Mhz, the implementation on the Raspberry Pi 3 scanner utilizes the default values from the go-rpio library. This means the rest state of the clock is low, and the first clock transition happens at the middle of a data bit.

As an example of the go-rpio library, the *rpio.SpiExchange* function is used to read a single value from a register in the card (MISO), while the *rpio.SpiTransmit* function writes to a specified register (MOSI). When the slave is sent data through MOSI, it will respond with arbitrary data as shown by the 'undefined' cells in table 3.10. The *rpio.SpiBegin* function serves as a *SS (Select Slave)* line, which initializes the SPI bus with a specified slave, allowing them to send and/or receive data. To disconnect a slave again, *rpio.SpiEnd* is used.

Following SPI timing diagram shows the MOSI line sending data from the Raspberry Pi 3 to the RC522 module. It is displayed how MOSI is read when the clock transitions from low-to-high, and how the data can be changed when the clock transitions from high-to-low:



**Figure 3.6:** Example of an SPI communication where the master transmits the ASCII symbol 'A'. [34]

A FIFO (First In First Out) buffer is used with the RC522 in order to manage longer data streams at a time. This allows for buffering of I/O data streams up towards 64 bytes long between the RC522 and host. Preparing data streams like this bypasses potential timing constraints. When using the FIFO buffer, the FIFODataReg register will be written to.

In order to access blocks on the MIFARE Classic 1K, it's necessary to run the proprietary NXP IC (Integrated Circuit) authentication protocol. This authentication requires at least one of two keys stored in the *sector trailer*, which is the fourth and last block in each sector. Of the two keys, key A is required while key B is optional, both of which are used in this proof of concept. Along with the keys is the access conditions for the sector,

setting the rights during memory access with the secret keys. The data blocks have the following possible rights: read, write, increment, decrement, transfer and restore, while the sector trailer can only be set to read or write. These access conditions are set by the access bits between the keys.

The 16 bytes in the sector trailer are partitioned as follows. Three access bits for each of the four blocks are stored in the bytes 6 - 8, while byte 9 contains general user data. Each block has an associated set of 3 access bits: C1, C2 and C3. This is illustrated in the following figure.

**Table 3.13:** The 16 bytes of the sector trailer and how they're used. [34]

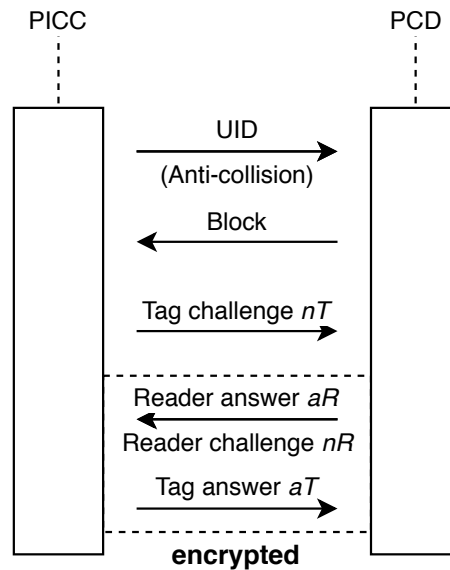
Key A					Access bits				Key B						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Additionally, MIFARE Classic 1K utilizes the proprietary NXP stream cipher *Crypto1* in order to achieve encryption during authentication. Crypto1 consists of a 48-bit LFSR (Linear Feedback Shift Register) alongside a filter function and a feedback function [16]. A LFSR is a type of *shift register*, which is a collection of single bit binary storage circuits known as *flip-flops*. These flip-flops are either shifted to the left or to the right every clock tick. What sets the LFSR variant apart is that the input of each operation is an offset of the previous internal state. In addition, the LFSR uses the following generating polynomial [36]:

$$x^{48} + x^{43} + x^{39} + x^{38} + x^{36} + x^{34} + x^{33} + x^{31} + x^{29} + x^{24} + x^{23} + x^{21} + x^{19} + x^{13} + x^9 + x^7 + x^6 + x^5 + 1$$

The LFSR shifts the bits to the left every clock tick and discards the leftmost bit, the MSB. In order to fill the LFSR again, the feedback function generates a new bit, which becomes the rightmost bit. When creating the keystream necessary for encrypted ciphertext, the filter function takes 20 bits from the LFSR as its input, creating a singular keystream bit every clock tick. The 20 bits used in the filter function consist of the odd bits 9, 11, 13 ... 47. [36]

When initializing an authentication, the reader must first specify a sector and key. This prompts the three-pass authentication protocol, visualized below:



**Figure 3.7:** Diagram over the three-pass authentication between the PICC and PCD.  $nT$ ,  $nR$ ,  $aT$ ,  $aR$  respectively stand for tag challenge nonce, reader challenge nonce, tag answer and reader answer. The square with dotted lines signify encryption with the Crypto1 cipher. Diagram inspired by Garcia, Gans, Muijers, Rossum, Verdult, Schreur, and Jacobs [36].

The authentication starts after the PICC is scanned, reading the UID. The PCD requests a block to access, which prompts the PICC to send a challenge  $nT$  in the form of a 32-bit pseudo random number, generated using the 16-bit LFSR (first pass). At this point the communication gets XORed with the keystream from the Crypto1 stream cipher, creating ciphertext [16]. The reader calculates an answer  $aR$  with the secret key, which is transmitted back to the card alongside a new challenge from the reader  $nR$  (second pass).  $nT$  and  $aR$  are compared by the reader, where if  $aR$  is a valid answer, the card will calculate and transmit an answer to  $nR$  (third pass). If the reader receives the correct answer, it will know that the authentication was successful.

### 3.4.3 Software

#### RC522

Control of the RC522 is as previously mentioned done using the go-rpio library, but not singlehandedly. Functions to control the RC552 were implemented in the Go programming language for the Raspberry Pi named *Gofare*, inspired by an Arduino library for the RC522 written in C++ (see [37]). When using *Gofare*, the first step is usually to initialize a scanner, which is done using the following function.

```
func NewScanner(rstPin uint) (*Scanner, error)
```

**Snippet 3.6:** Declaration of the function used to initialize a RC522

The function in snippet 3.6 on page 39 takes an unsigned integer, representing the RST pin, and returns a pointer to a *Scanner* alongside an error. The declaration of the *Scanner*

type is shown in snippet 3.7 on the next page.

```
type Scanner struct {
    rst  rpio.Pin
    Nuid []uint8 // nuid of last selected card
}
```

**Snippet 3.7:** Declaration of the scanner type

This type contains a value, representing the RST pin, of the type *rpio.Pin*, which is a type provided by the previously mentioned *go-rpio* library, alongside a NUID (Non-unique ID). Most other functions rely on the RST pin and NUID to operate, and if that is the case, they are implemented as methods on the *Scanner* type.

This means that these two variables will be available to those methods, but it also means that it is rather simple to have a single Raspberry Pi controlling multiple RC522s if they share the SPI bus that is hard-coded in this proof of concept. Note that these methods operate directly on a given instance of the type *Scanner* by using pointers, which is faster than each method receiving a copy of a given instance and returning that copy [38].

While initializing the scanner, it is necessary to read and write some registers. This is done with a few private methods such as *setupTimeout()*, which are abstractions of *ReadRegister()* and *WriteRegisterMult()*.

```
func (s *Scanner) ReadRegister(reg uint8) uint8
```

**Snippet 3.8:** Declaration of the method used to read a register on the RC522

As shown in snippet 3.8 and mentioned previously, this function is a method of the type *Scanner*. It takes an unsigned integer, representing the register to read, and returns another unsigned integer, representing the response.

```
func (s *Scanner) WriteRegister(reg uint8, data uint8)
```

**Snippet 3.9:** Declaration of the method used to write registers on the RC522

As shown in snippet 3.9 and mentioned previously, this function is a method of the type *Scanner*.

It takes two unsigned integers, one representing a register, the other representing data to be written to that register. This method, writing a single unsigned integer, is an abstraction on top of another function which writes multiple unsigned integers to a register. This is mainly done since the *rpi.SpiTransmit()* function takes a slice<sup>3</sup> of unsigned integers as

<sup>3</sup>A slice is a dynamic reference to an array.

data, but it is often only needed to send a single value. As it would be cumbersome to create a slice with a single value every time a single value is to be sent, this method does that instead.

Lastly, a scanner connection can be closed using the method, which declaration is shown in snippet 3.10 on the facing page.

```
func (s *Scanner) Close()
```

**Snippet 3.10:** Declaration of the method used to close a connection with a RC522

### MIFARE Classic 1K

Interacting with a MIFARE Classic 1K is also done in a program named *madong-client*, mainly consisting of methods and functions that are abstractions of the *CardIO* method from *Gofare*, which itself is an abstraction of *ReadRegister* and *WriteRegister*. *CardIO*'s declaration is shown in snippet 3.11.

```
func (s *Scanner) CardIO(cmd uint8, data []uint8, bitsInLast
    uint8) ([]uint8, error)
```

**Snippet 3.11:** Declaration of the method used transmit and receive data from a MIFARE Classic 1K

This method is used to transmit and receive data to and from a MIFARE Classic 1K. It takes an unsigned integer representing a command which can be; transmit, receive, or transceive. It also takes some data which is put in the FIFO buffer on the RC522, but the content of that buffer is not written to the a MIFARE Classic 1K unless the command is transmit or transceive. Moreover, it takes another unsigned integer representing how many bits of the last byte in the FIFO buffer should be used. This is needed since the FIFO buffer is written to one byte at a time, but one might wish to write a certain amount of bits to the card.

An example of a method which is an abstraction of *CardIO* is *ReadBlockEnc* and its declaration is shown in snippet 3.12.

```
func (c *Context) ReadBlockEnc(ks crypt.Cipher, addr uint8)
    ([]uint8, error)
```

**Snippet 3.12:** Declaration of the method used transmit and receive data from a MIFARE Classic 1K

This method is on the type *Context*, which is shown in snippet 3.13 on the next page.

```

type Context struct {
    Scanner *rc522.Scanner
}

```

**Snippet 3.13:** Declaration of the method used transmit and receive data from a MIFARE Classic 1K

The *Context* type only holds a scanner for now. It could have been a global variable, but a global variable would not allow for as many features that could be useful in the future such as separate configuration, for multiple RCC522s.

*ReadBlockEnc* takes a variable, *ks* which is of the type *crypt.Cipher* originating from *Gofare*. Its declaration is shown in snippet 3.14 on the following page.

```

type Cipher interface {
    Cipher(feedin []uint8, input []uint8) ([]uint8, []uint8,
        error)
}

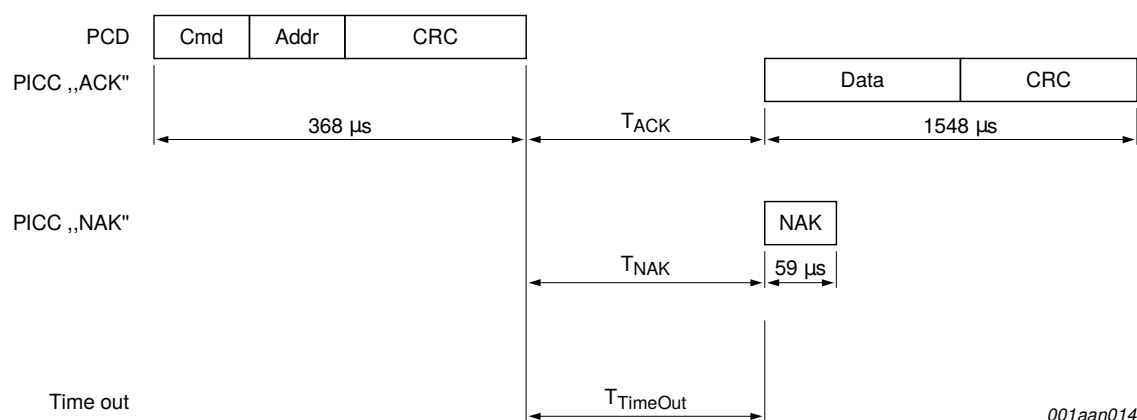
```

**Snippet 3.14:** Declaration of the method used transmit and receive data from a MIFARE Classic 1K

The type *crypt.Cipher* is an interface to the function *Cipher*. Working with an interface allowed for authenticating cards locally on the client using the default key, which is FF FF FF FF FF FF in hex, making sure various functions and methods worked properly, before using server-side crypto. On a related note, large parts of the Crypto1 code created on the client could simply be moved to the server since the interface was already implemented. Thus, it was just a matter of creating a method mapping the interface to server-side crypto instead of client-side crypto.

*ReadBlockEnc* also takes an unsigned integer representing the block to be read.

Transmitting to the MIFARE Classic 1K requires adding a CRC (Cyclic Redundancy Check) as shown in figure 3.8 illustrating a read command.



**Figure 3.8:** MIFARE read. [12]

The CRC is calculated using code which was translated from C to Golang (see [39]).

When the ReadBlockEnc function is running it temporarily turns off parity on the RC522, since parity needs to be calculated on the server before encrypting a packet which should be passed to a MIFARE Classic 1K.

Parity bits are calculated with the function shown in snippet 3.15 on the facing page.

```

1 func OddParity(bt byte) byte {
2     var temp uint16 = (0x9669 >> ((bt ^ (bt >> 4)) & 0xF))
3     & 1;
4     return uint8(temp);
5 }

```

**Snippet 3.15:** The function used to calculate parity bits. Translated to Golang from C (see [40]).

It takes a byte which it splits into two halves (i.e. two nibbles <sup>4</sup>). The first half is determined by bitshifting said byte to the right by four, for instance  $1001\ 0011 \gg 4 = 1001$ . The second half is calculated with a logical AND  $0xF$ .  $0xF$  is  $0000\ 1111$  in binary, which means an example calculation is:

$$\begin{array}{r}
 1001\ 0011 \\
 \wedge\ 0000\ 1111 \\
 \hline
 0011
 \end{array}$$

Those two halves are then XORed, for instance:

$$\begin{array}{r}
 1001 \\
 \oplus\ 0011 \\
 \hline
 1010
 \end{array}$$

$0x9669$ ,  $0110\ 1001\ 1001\ 0110$  in binary, is then bitshifted right by the value of that XOR, which in this case is a decimal 10, which means an example calculation is  $0110\ 1001\ 1001\ 0110 \gg 10 = 0001\ 1010$ . Bitshifting  $0x9669$  has a similar effect to looking up in a 16-bit parity table, the result is 6 bit long in this case, but that is ANDed with 1, to only keep the relevant digits which is the last one, as shown below.

$$\begin{array}{r}
 0001\ 1010 \\
 \wedge\ 0000\ 0001 \\
 \hline
 0
 \end{array}$$

(ANDing to only keep the relevant bits is also called masking.)

Thus, the parity bit of  $1001\ 0011$  is 0. [40]

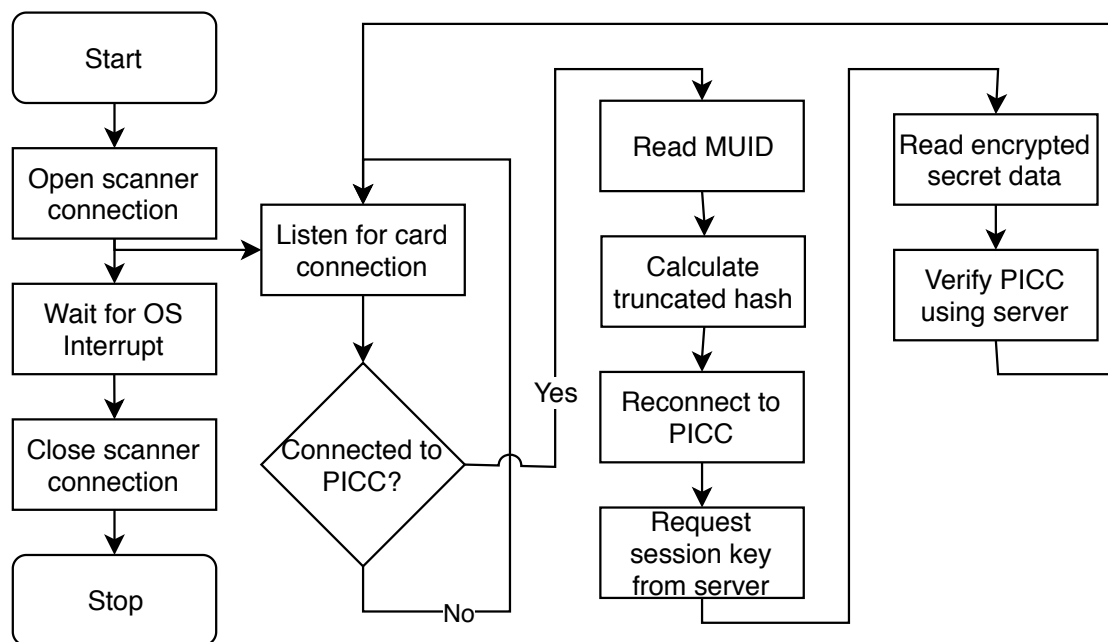
<sup>4</sup>A nibble is any 4 bits

Once parity bits are calculated they are inserted in a slice alongside the data which the parity bits are based on. These insertions are done using code which was translated from C++ to Golang (see [37]).

When *ReadBlockEnc* is finished, it returns a slice of unsigned integers representing encrypted data of that block alongside a potential error.

## Summary

In summary, the previously explained functions and methods are used to have the scanner listen for a card connection and handle it appropriately as shown in figure 3.9 on the next page



**Figure 3.9:** A flowchart illustrating the program running on the scanner.

There are other functions, methods, and error handling which has been left out for simplicity's sake.

## 3.5 Crypto1 in Software

*This section will cover the software implementation of crypto1 described in section 3.4.2 on page 36, where instead of the calculations being done on the MFRC522, it will be done in software. This means data sent to the MFRC522 will already be encrypted and is thus harder to extract.*

### 3.5.1 Implementation

The crypto1 implementation in *Gofare* was created by translating another crypto1 implementation to Golang, the crypto1 C library [41]. This means the client and server can share parts of *Gofare*, since they are both written in Golang. The following functions were translated.



<code>NewCrypt</code>	Creates a new internal state from a sector key
<code>Filter</code>	Simulates the crypto1 mathematical filter function
<code>CipherBit</code>	Get a single bit while changing the internal state using the <code>Filter</code> function
<code>CipherByte</code>	Calls <code>CipherBit</code> sequentially for each bit in a byte
<code>CipherWord</code>	Same as <code>CipherByte</code> but with a 32bit word
<code>Parity</code>	Used in <code>Filter</code> to calculate new state
<code>PrngSuccessor</code>	Used in authentication

This was done by running the C library and generating sample input and output for each of the needed functions. Subsequently this could then be loaded into go lang as a unit test, and used to check when the translation was accurate. An example of such a test can be seen in snippet 3.16 on the next page.

```

1 func TestFilter(t *testing.T) {
2     testFilter := func(in uint32, out uint8) {
3         rest := Filter(in)
4         if rest != out {
5             t.Errorf("Filter(%X) -> %X expected %X", in, rest, out)
6         }
7     }
8
9     testFilter(0, 0)
10    testFilter(0x12344556, 1)
11    testFilter(0x0000000A, 0)
12    testFilter(0xFFFFFFFF, 1)
13    testFilter(0x00005349, 1)
14    testFilter(0x0D70F684, 0)
15 }

```

**Snippet 3.16:** Testing the filter function

Internal state specifies the LFSR value as described in section 3.4.2 on page 36 and is initialized by `NewCrypto` using the users unique key. The functions `CipherBit`, `CipherByte`, and `CipherWord` depend on the internal state of the crypto, while the rest only operate on given input.

### 3.5.2 Moving Internal State to Remote Server

Section 2.4.2 on page 17 specifies that the scanner must not have access to a card secret data or unique key. All operations using the unique key or secret data should therefore run on the server.

The only operations needing the unique key is `NewCrypto` which uses the key directly and `Cipher*` which uses it indirectly through the internal state. These are therefore the only

functions which should be moved to the server.

Instead of having one server endpoint for each of the functions, a simpler interface to Crypto1 is used. Other than the gained simplicity, this also reduces the amount of calls needed during operations.

The table below describes how the Cipher\* functions are used during authorization and block reading.

Ciphering a slice of bytes while calculating parity	Used to encrypt command and decrypt blocks during reading
Ciphering a 32 bit word	Used when de- and encrypting challenges nonces during authorization

The first case was abstracted into the following function. It takes a slice as input which is to be encrypted and a feed-in slice used to alter the internal state. If either input or feed-in is not defined it will replace it by zeroes.

```

1 func (cs *CryptState) Cipher(feedin []uint8, input []uint8)
  ([]uint8, []uint8, error) {
2   l := len(input)
3   if input == nil {
4     l = len(feedin)
5   }
6   output := make([]uint8, l)
7   parity := make([]uint8, l)
8   for i := 0; i < l; i++{
9     var feed, in uint8
10    if feedin != nil {
11      // Check if feedin is nil
12      feed = feedin[i]
13    }
14    if input != nil {
15      in = input[i]
16    }
17    output[i] = cs.CipherByte(feed, false) ^ in
18    parity[i] = Filter( cs.Odd ) ^ OddParity( in )
19  }
20  return output, parity, nil
21 }

```

**Snippet 3.17:** Testing the filter function

The second case can be derived from the Cipher function because a 32 bit word is 4 bytes. Its implementation can be seen in snippet 3.18 on page 47 which uses the golang binary package to convert a 32 bit value to a slice of bits. Because cipherWord does not need

parity, input can be excluded when calling `Cipher`.

```

1 func cipherWord(c crypt.Cipher, feed, input uint32) (uint32,
   error) {
2   tmp := make([]uint8, 4)
3   binary.BigEndian.PutUint32(tmp, feed)
4   tmp, _, err := c.Cipher(tmp, nil)
5   if err != nil {
6     return 0, err
7   }
8   // Reuse to feedvalue
9   feed = binary.BigEndian.Uint32(tmp)
10
11  return feed ^ input, nil
12 }

```

**Snippet 3.18:** Testing the filter function

Using this multi-purpose `Cipher` function it can be mapped to a single API endpoint on the server as mentioned in section 3.4.3 on page 39. The 3 crypto endpoints are summarized below.

**Table 3.14:** API calls for cryptol operations

<code>/api/crypt/open/{card}</code>	Creates a new internal crypto state for the specified <i>card</i> and returns a session ID for reference. Also checks if the user has access to the requesting scanner.
<code>/api/crypt/cipher</code>	Direct map to the <code>Cipher</code> function. Scanner must specify valid session ID.
<code>/api/crypt/verify</code>	Accepts an encrypted secret block and verifies it against the users hash in the database. Scanner must specify valid session ID.

### 3.5.3 Creating a Generic Cipher Type

As mentioned in section 3.4.3 on page 39 Gofare was written to handle setup, authorization, and read operations with the MIFARE Classic 1Ks. To keep this library simple it does not handle calls to the remote API described in table 3.14.

Instead the library calls a generic `Cipher` function which can be passed as a parameter. This was done using Golang interfaces as described in section 3.4.3 on page 39. Interfaces are a collection of method definitions required to implement said interface. [42]

If a variable is defined by an interface, it can hold all types which implement its interface. In snippet 3.14 on page 42 one can see the interface defining the `Cipher` function.

The function `cipherWord` in snippet 3.18 takes the `Cipher` interface and calls the `Cipher` function. Because it uses the interface one can pass different types as long as they implement the `Cipher` function. This enables multiple implementations of `Cipher`, see snippet 3.19 on the next page for an example.

```
1 // Create standard implementation of Cipher with known key
2 ks := crypt.NewCrypt(zerokey)
3
4 // Authenticate sector zero
5 err := c.Scanner.ManualAuth(ks, rc522.CardAuthA, 0)
6 if err != nil {
7     return "", fmt.Errorf("auth on block 0 %v", err)
8 }
9 ...
10 // Create api implementation of Cipher from user id
11 ks, err := NewCrypt("http://10.42.0.28:4444", id)
12 if err != nil {
13     return err
14 }
15
16 // Auth using the thing
17 err = c.Scanner.ManualAuth(ks, rc522.CardAuthA, 4)
18 if err != nil {
19     return err
20 }
```

**Snippet 3.19:** Using the `ManualAuth` method with two different implementations of the `Cipher` interface

The `Cipher` implementation returned from `NewCrypt` on line 11, redirects all calls to the API endpoint `/api/crypt/cipher`.

## 3.6 Admin User Interface

As part of testing the API (Application Programming Interface), a GUI (Graphical User Interface) was made. This gave the opportunity of creating/retrieving records in different tables on the database with HTTP (HyperText Transfer Protocol) requests. The GUI was made with HTML (HyperText Markup Language), CSS (Cascading Style Sheets), and JavaScript. An example of one of the functions can be seen in figure 3.10 on the following page.

## Roles & Policies

**Figure 3.10:** Screenshot of the function for roles and policies in the admin UI.

When the page with the function in figure 3.10 is loaded, a GET request is called that fills the drop-down bar with all current roles in the database. When a user selects a different role in the drop-down menu, another GET request is made to retrieve all users with that role. The requests to the API was made with jQuery which is a library for JavaScript. An example of a POST request to *policies* can be seen in snippet 3.20 on the facing page.

```

1  $.ajax({
2      url: "http://localhost:4444/api/roles/"+currentRole+"/
      policies",
3      type: "POST",
4      data: JSON.stringify({
5          zone: zoneInput,
6          begin: tBegin,
7          end: tEnd,
8          state: 0,
9      }),
10     contentType: "application/json; charset=utf-8",
11     dataType: "json",
12     success: function() {
13         alert("success");
14     }

```

**Snippet 3.20:** An example of a POST request on the admin UI

In snippet 3.20 a POST request is made to the endpoint "http://localhost:4444/api/roles/"+currentRole+"/policies", where currentRole is the selected role as previously mentioned. The data is sent in JSON (JavaScript Object Notation) format, which is why the data is converted from a Javascript object with **JSON.stringify()**.

## 3.7 System Sequence of Events

Figure 3.11 on the following page is summarization of the card scanning progress described in the sections above. It shows the sequence of the events happening when a user uses a scanner. Actions with PICC are in reality done on the MFRC522 however it has been abstracted away.

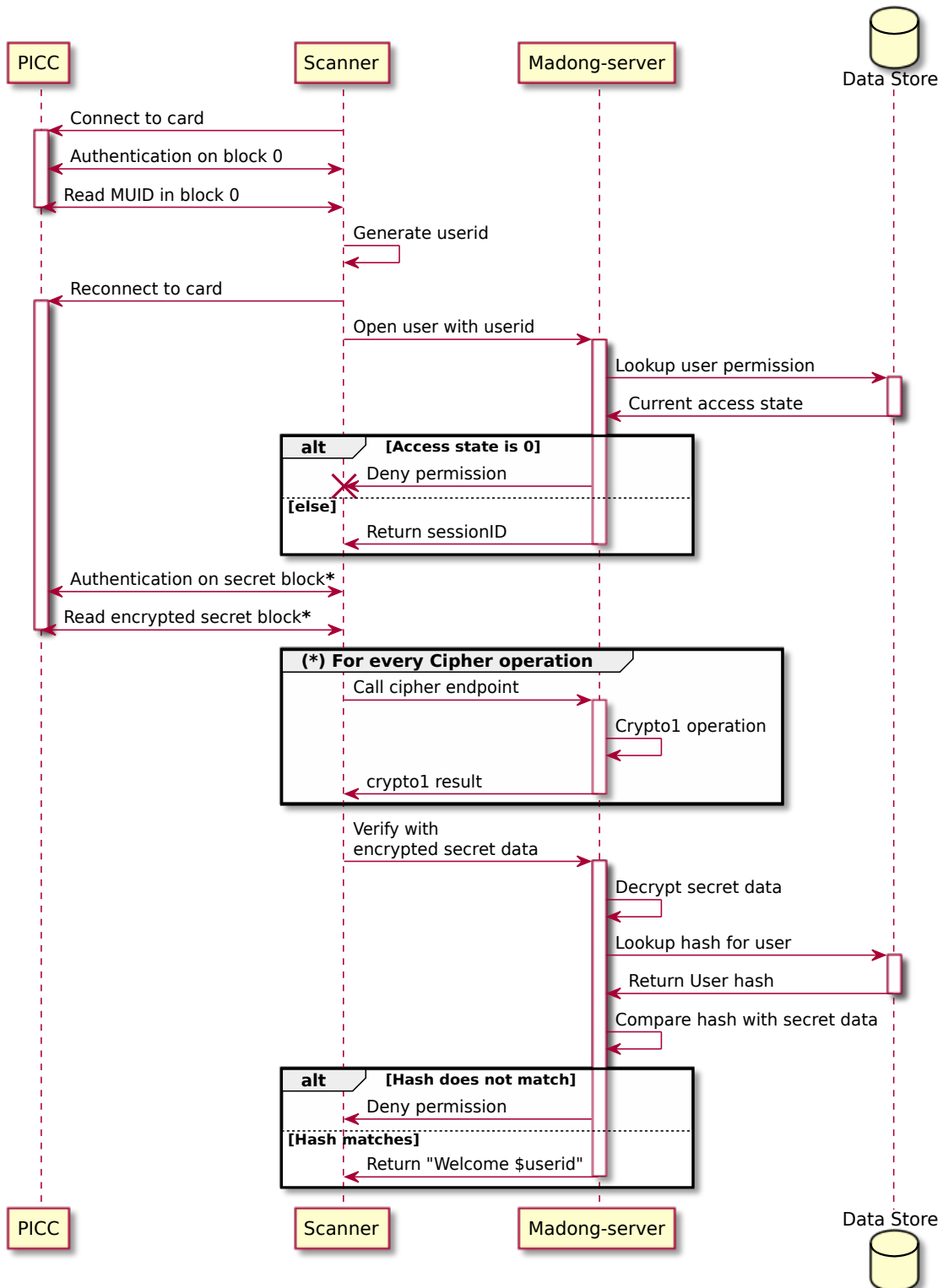


Figure 3.11: Sequence diagram depicting a card scan

## 4.1 Acceptance Test

In order to validate the functionality of the proposed proof of concept security system and its individual components, a number of acceptance tests are conducted. These tests will cover the requirements specified in section 2.4 on page 16. The procedure for each test as well as the definitive test success criteria will be documented in the following subsections.

Therefore black box testing will be the focal point of the acceptance tests. As such, expected output of each test as well as the actual output is the deciding factor in whether the requirement can be accepted as successful or not.

All of the tests take place in the same test environment, where the Raspberry Pi 3 PCD (Proximity Coupling Device) is connected to the server and being prompted to read one or more different PICCs (Proximity Integrated Circuit Card). For the test of requirement 3 specifically, two identical Raspberry Pi 3 PCDs will be used between the specified time frame.

### 4.1.1 Structure of tests

*In order to verify whether the individual requirements are met, a number of tests will be run. Following is a description of each requirement, which test will be run for them, as well as both the test steps and success criteria. Furthermore, a number of other features will also be tested.*

#### 4.1.1.1 Test of Requirement 1

*Access must only be granted to users in accordance with the current roles and policies.*

The goal of requirement 1 is to ensure that the user gets access to the facilities they should, based on their role and the policies of the zone. This also constitutes that specific locations that should not grant access to certain users need to have the scanners deny their entry.

When the user has their PICO read, the server should match their role with the policies for the zone of the scanner. If a policy states that a user of that role should have access to the door during the current time interval, it should grant access to the user.

The test follows these steps:

- 1) A PICO is given a role policy that will grant it access to the zone in accordance to

the current time.

- 2) The PICC is scanned on the PCD.
- 3) The server scans the database for the role policies of the PICC based on its MUID.
- 4) The server solves the Mifare Classic 1K challenge.
- 5) The PCD transfers card data to the server.
- 6) The PICC gives the server access to reading its sector.
- 7) The server decrypts the encrypted data on the sector.
- 8) Based on the current time and policy, the PICC will be granted access which is written in the terminal on the server with 'Access granted'.

In order for the test to be deemed successful, the user must be granted access with a matching role policy for the current point in time.

Furthermore, a scenario where the user will not be granted access will also be tested:

- 1) A PICC is given a role policy that will deny it access to the zone in accordance to the current time.
- 2) The PICC is scanned on the PCD.
- 3) The server scans the database for the role policies of the PICC based on its MUID.
- 4) The PCD transfers card data to the server.
- 5) The server solves the Mifare Classic 1K challenge.
- 6) The PICC gives the server access to reading its sector.
- 7) The server decrypts the encrypted data on the sector.
- 8) Based on the current time and policy, the PICC will be denied access which is written in the terminal on the server with 'Access denied'.

This test will be deemed successful if a user without a relevant role policy is denied entry.

**Result of tests for requirement 1** Two different PICCs registered in the system were scanned individually, one with the corresponding role policies i.e. matching the scanner's policies, and one PICC without the correct role policies. In figure 4.1 on page 53 a PICC with the correct policies setting is used, and figure 4.2 on the next page is a response in the terminal from a PICC without the correct policies.



```
Auth success
cipher rsp: {Resp:[255 5 184 0] Parity:[0 0 0 0]}
input 09032D86984A45B35A
output 09818B10A92ACDB5
input 39891F6356555F93A2
output 39C4C7CC55FA4D45
input 438302
output 4341
Secret: 09818B10A92ACDB539C4C7CC55FA4D454341
verify rsp: Welcome a862753
```

**Figure 4.1:** Response in terminal with a PICC with correct policy setting.

```
output 89D890F08782CAB6
input 928901
output 92C4
MUID Truncated Hash: ecb154e
La
2019/12/11 13:35:47 Part 2
2019/12/11 13:35:47 nuid: [38 196 65 131]
2019/12/11 13:35:47 Created token eyJpZHQiOiIxNHExT3VyNFBJZy9lVFI2eFNCYlBwUHJvUDZ2Zk50ZE8zMTE4f0o=.rF+DxYgwTcsiNI9qBRAiWnf/1CW/IsqvATDcLgrQaL0jWJtYuDnzQIHdiCNjBpZy8bVfZ5b3V/VyqIG177
2019/12/11 13:35:47 handlecon: Server returned code 403 and error: user not allowed in zone
```

**Figure 4.2:** Response in terminal with a PICC with-out correct policy setting.

#### 4.1.1.2 Test of Requirement 2

*Each AAUCard must be protected with a unique key.*

The goal of this requirement is for each AAUCard to be protected with a unique key, rather than all AAUCards sharing the exact same key. The test can be made as an extension of the first test from requirement 1, where the system will have read two different cards. Each of the cards will be identical except for the secret key used during authentication, which will be unique.

The test follows these steps:

- 1) Two PICCs, A and B, with different keys are used.
- 2) A smartphone application is given the same key as PICC A.
- 3) PICC A is scanned on the aforementioned smartphone.
- 4) The smartphone application will be able to read the locked sector in PICC A.
- 5) The application will not be able to read the same sector in PICC B.

The test is regarded as successful if only PICC A's locked sector can be read.

#### Result of test for requirement 2

Two different PICCs were attempted to be read, one with a matching key in smartphone application, and the other without a matching key. The key used for this test can be seen in figure 4.3 on the facing page.



1DADE8089620

**Figure 4.3:** The key file used to read the two PICCs

As seen in figure 4.4 on the next page the sector in PICC A is open to read with the key and the same attempt to read PICC B can be seen in figure 4.5.

```

Sector: 0
EB140C798A0804006263646566676869
00000000000000000000000000000000
01020304050607080901020304050607
FFFFFFFFFFFFF078069FFFFFFFFFFFF

Sector: 1
68656A206D6564206469672068766F72
48656A206D6564206469670000000000
00000000000000000000000000000000
1DADE8089620FF0780001DADE8089620

```

Figure 4.4: The attempt of reading PICC A

```

Sector: 0
01B2332EAE0804006263646566676869
00000000000000000000000000000000
00000000000000000000000000000000
FFFFFFFFFFFFF078069FFFFFFFFFFFF

Sector: 1
No keys found (or dead sector)

```

Figure 4.5: The attempt of reading PICC B

This means that PICC A and PICC B does not have the same key.

#### 4.1.1.3 Test of requirement 3

*Must be able to work with more than one scanner.*

Through this requirement the server must be able to support more than one PCD at once.

The test follows these steps:

- 1) Two PCDs are used.
- 2) Two PICCs are scanned, one for each scanner.
- 3) The two PICCs are scanned relatively synchronous.
- 4) The PCDs pass information to server.

- 5) The server respond with access granted for both scanners.

The test is regarded as successful if no errors or noticeable delay occurs as the cards are scanned.

### Result of test for requirement 3

The test was connected using a server on the same local network. On this server a packet capture was running during the test, which wrote to a file that was opened afterwards in a software called Wireshark. A screendump of this can be seen in figure 4.6 on the next page.

No.	Time	Source	Destination	Protocol	Length	Info
18	4.047555	10.42.0.168	10.42.0.1	HTTP	427	GET /api/crypt/open/a862753 HTTP/1.1
20	4.053844	10.42.0.1	10.42.0.168	HTTP	407	HTTP/1.1 200 OK (application/json)
22	4.058478	10.42.0.168	10.42.0.1	HTTP	444	POST /api/crypt/cipher HTTP/1.1 (application/json)
23	4.059183	10.42.0.1	10.42.0.168	HTTP	252	HTTP/1.1 200 OK (application/json)
24	4.062282	10.42.0.168	10.42.0.1	HTTP	450	POST /api/crypt/cipher HTTP/1.1 (application/json)
25	4.062518	10.42.0.1	10.42.0.168	HTTP	252	HTTP/1.1 200 OK (application/json)
26	4.064414	10.42.0.168	10.42.0.1	HTTP	444	POST /api/crypt/cipher HTTP/1.1 (application/json)
27	4.064738	10.42.0.1	10.42.0.168	HTTP	252	HTTP/1.1 200 OK (application/json)
28	4.068011	10.42.0.168	10.42.0.1	HTTP	444	POST /api/crypt/cipher HTTP/1.1 (application/json)
29	4.068397	10.42.0.1	10.42.0.168	HTTP	252	HTTP/1.1 200 OK (application/json)
30	4.070127	10.42.0.168	10.42.0.1	HTTP	444	POST /api/crypt/cipher HTTP/1.1 (application/json)
31	4.070462	10.42.0.1	10.42.0.168	HTTP	252	HTTP/1.1 200 OK (application/json)
32	4.076028	10.42.0.168	10.42.0.1	HTTP	446	POST /api/crypt/verify HTTP/1.1 (application/json)
38	4.284317	10.42.0.102	10.42.0.1	HTTP	427	GET /api/crypt/open/32a1309 HTTP/1.1
40	4.294390	10.42.0.1	10.42.0.102	HTTP	407	HTTP/1.1 200 OK (application/json)
42	4.298435	10.42.0.102	10.42.0.1	HTTP	444	POST /api/crypt/cipher HTTP/1.1 (application/json)
43	4.298547	10.42.0.1	10.42.0.102	HTTP	252	HTTP/1.1 200 OK (application/json)
44	4.300570	10.42.0.102	10.42.0.1	HTTP	450	POST /api/crypt/cipher HTTP/1.1 (application/json)
45	4.300720	10.42.0.1	10.42.0.102	HTTP	252	HTTP/1.1 200 OK (application/json)
46	4.302577	10.42.0.102	10.42.0.1	HTTP	444	POST /api/crypt/cipher HTTP/1.1 (application/json)
47	4.302685	10.42.0.1	10.42.0.102	HTTP	252	HTTP/1.1 200 OK (application/json)
48	4.306815	10.42.0.102	10.42.0.1	HTTP	444	POST /api/crypt/cipher HTTP/1.1 (application/json)
49	4.306934	10.42.0.1	10.42.0.102	HTTP	252	HTTP/1.1 200 OK (application/json)
50	4.309098	10.42.0.102	10.42.0.1	HTTP	444	POST /api/crypt/cipher HTTP/1.1 (application/json)
51	4.309207	10.42.0.1	10.42.0.102	HTTP	252	HTTP/1.1 200 OK (application/json)
52	4.314405	10.42.0.102	10.42.0.1	HTTP	446	POST /api/crypt/verify HTTP/1.1 (application/json)
67	4.338330	10.42.0.1	10.42.0.168	HTTP	231	HTTP/1.1 200 OK (application/json)
70	4.567281	10.42.0.1	10.42.0.102	HTTP	231	HTTP/1.1 200 OK (application/json)

Figure 4.6: Wireshark screendump of PICC scans.

In figure 4.6 the two scanner clients can be seen with the IP-addresses (Internet Protocol): **10.42.0.102 (scanner 1)** and **10.42.0.168 (scanner 2)**, and the server IP-address: **10.42.0.1**. Furthermore, it can be seen that before scanner 2 gets a final response from the server, the server handles requests from scanner 1.

#### 4.1.1.4 Test of requirement 4

*The scanner must not have access to a card's secret data or unique key.*

To test this requirement an attempted attack will be done from the scanner. This assumes an adversary has full control of the attacking scanner and said scanner has permission to access the API. Two possible attacks are speculated to be possible and can be seen below.

- 1) Because of the low level crypto1 API (see section 3.5 on page 44) it is possible to extract internal crypto1 state from server. Because a scanner knows every input to the crypto (feed-in argument to `api/crypto/cipher`) it can do LFSR rollback (see section 2.3.1 on page 11) and thus recover the user unique key.

- 2) Instead of calling `api/crypto/verify` with the encrypted secret data and get an okay from the server, it can call `api/crypto/cipher` and get the unencrypted secret data.

Exploit 2. will be used for this test and will be done as described below:

- 1) One PCD is patched to call `api/crypto/cipher` instead of `api/crypto/verify`.
- 2) One PICC is setup to have access to the PCD, and its secret data is set to some known value.
- 3) PICC is scanned and result is read out on PCD log.

### Result of test for requirement 4

Currently after a successful authentication with the card it reads the encrypted content of the secret sector. This is sent along to the `verify` which calls the corresponding API call on the server, and can be seen below.

```
1 | ...
2 | // Contact server and request sessionkey
3 | ks, err := NewCrypt(urlAPI, id)
4 | ...
5 | // Read the secret
6 | blk, err := c.ReadBlockEnc(ks, 4)
7 | if err != nil {
8 |     return err
9 | }
10 | // Verify it with the server
11 | _, err = ks.Verify(blk)
12 | if err != nil {
13 |     return err
14 | }
15 | ...
```

**Snippet 4.1:** Original scanner software verifying secret block

Instead of calling the special `ReadBlockEnc` function the patched version will call `ReadBlockManual` which in this case decrypts the block using the `api/crypto/cipher` API.

```
1 | ...
2 | // Contact server and request sessionkey
3 | ks, err := NewCrypt(urlAPI, id)
```

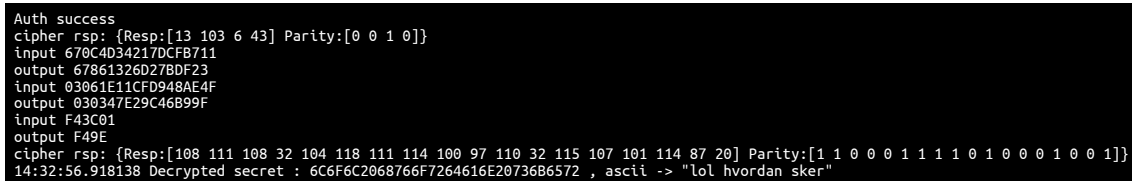
```

4  ...
5  // Read the secret
6  blk, err := c.Scanner.ReadBlockManual(ks, 4)
7  if err != nil {
8      return err
9  }
10 // Haha stupid server
11 log.Printf("Decrypted secret: %X, ascii -> %q", blk, blk)
12 ...

```

**Snippet 4.2:** Patched scanner software exploiting the server

The output of the patch can be seen in figure 4.7 on page 57. Here the decrypted secret data can be seen printed out at the bottom.



```

Auth success
cipher rsp: {Resp:[13 103 6 43] Parity:[0 0 1 0]}
input 670C4D342170CFB711
output 67861326D278DF23
input 03061E11CFD948AE4F
output 030347E29C46899F
input F43C01
output F49E
cipher rsp: {Resp:[108 111 108 32 104 118 111 114 100 97 110 32 115 107 101 114 87 20] Parity:[1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0 1]}
14:32:56.918138 Decrypted secret : 6C6F6C2068766F7264616E2073686572 , ascii -> "lol hvordan sker"

```

**Figure 4.7:** Log output of patched scanner

Because this is running on a patched scanner it shows that a compromised scanner can get access to scanned cards secret data.

#### 4.1.1.5 Test of requirement 5

*Must reuse the MIFARE Classic 1K cards.*

Requirement 5 constitutes that the cards must not be replaced by a different type of card or IC (Integrated Circuit) device, such as the MIFARE Classic EV1.

By extension, the previous tests already reuse the MIFARE Classic 1K cards, so this requirement will be considered successful if any of the previous tests are successful.

#### Result of test for requirement 5

Requirement 1 was deemed successful in the first test of the chapter. As that test uses the MIFARE Classic 1K type of PICCs, the result further completes the success criteria of requirement 5.

#### 4.1.1.6 Test of requirement 6

*The time between scanning a card and the prompt for a PIN code, excluding network delay, must be within one second.*

For this requirement the time between when the user scans the card and receives access must not exceed one second. One second was chosen in order to keep the service fast and convenient, while still giving the system enough time to communicate.

Timing starts once a PICC is connected to the scanner and stops when access is granted. That means time will be measured whenever the server and scanner process data. Total time elapsed will be the sum of the two different timed processes. As the connection on the network can have a high variance it will be negated from the final results.

The test is deemed successful if less than one second elapses through test steps 2 - 6.

### Result of test for requirement 6

In order to negate the network time from the results, 3 different time processes were made.

- Server request handling time (Server)
- Scanner time on request (Server+Network)
- Total scanner time (Server+Network+Scanner)

First, the *Server* was subtracted from the *Server+Network*, which then gave the result of time of the *Network*, that was then subtracted from the *Scanner+Network+Server*, which gave the final result. Ten different readings of the processes were made and averaged to a final result which can be seen in table 4.1 on the following page.

**Table 4.1:** Results of authentication time in ms.

Server	Network+Server+Scanner	Server+Network	Network	Scanner+Server
252.68	640.46	257.98	5.3	635.16
258.16	661.25	266.85	8.69	652.56
262.74	657.15	271.42	8.68	648.47
252.57	643.9	264.64	12.07	631.83
260.77	649.56	267.95	7.18	642.38
260.97	654.56	269.28	8.31	646.25
260.06	657.07	270.75	10.69	646.38
260.81	655.22	270.07	9.26	645.96
260.72	654.43	268.83	8.11	646.32
260.86	652.11	268.04	7.18	644.93
259.034	652.571	267.581	8.547	<b>644.024</b>

The scanner authentication, which checks if the scanner used is registered, was also tested with an average of 100  $\mu$ s and was therefore ignored from the results due to its low influence.

#### 4.1.1.7 Summary of tests

*A table with results from tests run in this chapter has been created in order to reach a summary.*

**Table 4.2:** Table of the test results gathered through the acceptance test section.

<b>Test</b>	<b>Conclusion</b>
Requirement 1	Accepted - This test was deemed successful as both individual tests, which call for 'access granted' and 'access denied' subsequently, returned the expected output.
Requirement 2	Accepted - This test was deemed successful as the card encrypted with a known key could be read by Mifare Classic Tool, while a unknown card could not be read.
Requirement 3	Accepted - This test was deemed successful as multiple scanners were handled without issues.
Requirement 4	Failed - This test was deemed a failure, as a malicious scanner can sniff secret data.
Requirement 5	Accepted - This test was deemed successful as requirement 1, which reuses the MIFARE Classic 1K PICCs, was also successful.
Requirement 6	Accepted - This test was deemed successful as the average of 10 tests showed a processing time of 644.024 ms.





# Discussion 5

---

In test 1 the roles and policies of the system were tested with two different PICCs, where each PICC had different roles in the database and therefore different policies. It was found that the result of this test was sufficient and therefore complied with requirement 1.

The second test were conducted with the intention of validating that two different PICCs were able to have unique keys, which was found to be possible. Furthermore, in section 2.3.3 on page 14 it was postulated that one of the biggest flaws in the current AAU access control system, is the lack of a unique key on each card. This was later found to be confirmed, since the manufacturer of the MIFARE Classic 1K cards strongly suggest using unique keys on each card. Furthermore, NXP also suggests to have a fraud detection in the system, which refers to a possibility of finding the existence of fraudulent cards, and also employing a mechanism to block these cards from entering [43]. This is implemented in the system with the usage of secret data and the MUID. In section 2.3.3 on page 14, the possibility of upgrading the cards was also discussed, however due to limited resources this was not investigated thoroughly.

The motive of conducting test 3 was to demonstrate if the concept of the system was feasible in a production environment, in that it needs to be able to handle multiple scanner. The system showed promise since it was able to handle overlapping requests from two scanners.

Disregarding the requirements, the most essential components of the proof of concept are firstly the unique key on each card and secondly avoiding trust in the scanners. However, the latter component was not fulfilled given that test 4 (see section 4.1.1.4 on page 55) showed at least two different possible attacks on the system. The first attack, as mentioned in section 2.3.2 on page 11, is based on an inherent flaw in Crypto1, since a LFSR rollback attack is possible. Furthermore, test 4 showed that the proposed system is also flawed in that an exploit was found on one of the endpoints in the API, which allows for a registered scanner to make a request where the response will be unencrypted secret data.

The latter exploit is caused by the low level `api/crypto/cipher` endpoint trusting the scanner too much. This can be naively mitigated by checking the response before sending and checking if it contains the secret data. However a malicious scanner could then spread the decryption across multiple `api/crypto/cipher` calls or exploit the check to brute-force the secret data.

It was not possible to test all the desired facets of the AAU access control system due to limited resources. One of such facets is a precise measurement of the process between scanning a card and prompting a user to enter a PIN code. This made requirement 6 (see

section 2.4.2 on page 17) non testable. It was only tested through empirical data from using the AAUCard day to day, rather than more substantial testing. One way of conducting a more adequate test of the requirement would be to get more access to the whole system, and thereby doing precise time measurements of the different components in the system. Moreover, a user experience test could be conducted to see how long the scanning process can take before it becomes inconvenient or annoying. It could also be that feedback on scan could be desired for the user, as mentioned in test 6 (see section 4.1.1.6 on page 57) the process took 642.024 ms disregarding the network time. Furthermore, in a production environment, the time of data transmission on the network could be higher due to more network latency, which would slow the whole process between scanning and prompting for PIN code. Therefore, the result of test 6 needs to be revisited and reevaluated in order to see if it is sufficient in a production environment.

# Conclusion 6

---

This project examined the current access control system at Aalborg University in Aalborg. This examination is comprised of both AAUCards and Web Services built onto these cards. It can be concluded that the current system has vulnerabilities in both web services and on the PICCs. Via the web services it was possible to extract PIN codes and CPR-numbers of AAU staff and -students. Furthermore, it can be concluded that the fundamentals of the access control system is flawed by not following guidelines from the manufacturer of the PICCs alongside outdated cryptography.

Therefore, a proof of concept was developed with the aim of addressing these flaws. It can be concluded, that a system which reuses the current technology of AAUCard and uses unique keys on each card is possible. Be that as it may, if this is done, it is not possible to solve the flaws that are inherent to Crypto1. As such, it was not possible to completely avoid trusting the scanners in this proof of concept, even if authentication for administrative API endpoints was implemented. These issues need to be addressed before the proof of concept is used in a production environment.



# Reflection 7

---

Throughout the implementation process it became apparent that the MIFARE Classic 1K cards do not provide adequate cryptographic security despite the proposed attempts to improve the existing system. While it may be possible to mitigate select types of attacks or prevent a single cracked key from working on every single card, the additional underlying vulnerabilities in Crypto1 make the cards a less desirable choice in this day and age. As such, the authors of this paper propose that any future secure access control systems do not utilize the MIFARE Classic 1K cards, and that any current implementations aim to replace the cards with a more up to date alternative as soon as possible. For a future development of the proof of concept system, it could be beneficial to conduct a cost-benefit analysis of the different alternatives.

In terms of improving the proof of concept as it is, it would be ideal to try to establish more empirical requirements. This could entail figuring out how much time a user has to wait while getting their card scanned for it to become inconvenient, which could be conducted through a usability test. To measure the time to scan, having total access to the control system in order to log scan prompts and access grants would provide more clear data. Furthermore the current system lacks any visual or auditory feedback when scanning, meaning the user will not know that their card is getting scanned or when access is granted. This should also be iterated upon in the future.

The proof of concept also has a number of available technical improvements that could be implemented. There are also no admin authentication required to access admin features, which would be quite problematic in a real life use-case scenario. However, even if this was fixed, request bodies from scanners are not signed alongside the request header, which means a potential MITM could impersonate scanners. Moreover, while writing new keys to a MIFARE Classic 1K is supported by *Gofare*, no automatic transition mechanism was implemented, leaving a transition to new keys up the issuing organization.

Additionally, only the most prevalent side-channel attacks on the software implementation of Crypto1 were considered.

Finally, the production-ready iteration of the security system should investigate the possibility of utilizing public key cryptography, where each card has its own set of asymmetric keys. This would allow for secure authentication on the scanner as opposed to the server.



# Bibliography

---

- [1] Nolen Scaife, Christian Peeters, and Patrick Traynor. “Fear the Reaper: Characterization and Fast Detection of Card Skimmers”. In: *27th USENIX Security Symposium* (2018-08), pp. 1–1.
- [2] Danmarks Statistik. *ITAV7: Virksomhedernes brug af avancerede teknologier*. 2019-09. URL: <https://www.statistikbanken.dk/ITAV7> (visited on 2019-12-17).
- [3] Thomas Norman. *Electronic Access Control*. Butterworth-Heinemann, 2012. ISBN: 978-0-12-382028-0.
- [4] Ruud Bolle and Sharath Pankanti. *Biometrics, Personal Identification in Networked Society: Personal Identification in Networked Society*. Ed. by Anil K. Jain. Norwell, MA, USA: Kluwer Academic Publishers, 1998. ISBN: 0792383451.
- [5] Stan Li and Anil Jain. *Encyclopedia of Biometrics*. 2009-01, pp. 1128–1131. DOI: 10.1007/978-0-387-73003-5.
- [6] Stan Li and Anil Jain. *Encyclopedia of Biometrics*. 2009-01, pp. 760–777. DOI: 10.1007/978-0-387-73003-5.
- [7] M. A. Turk and A. P. Pentland. “Face recognition using eigenfaces”. English. In: (1991), pp. 586–591. DOI: 10.1109/CVPR.1991.139758.
- [8] Joseph P. Campbell and Jr. *Speaker recognition: A tutorial*. 1997. DOI: 10.1109/5.628714.
- [9] Dawn M. Turner. *Digital Authentication - the basics*. 2016-08. URL: <https://www.cryptomathic.com/news-events/blog/digital-authentication-the-basics> (visited on 2019-12-17).
- [10] Shraddha D. Ghogare, Swati P. Jadhav, Ankita R. Chadha, and Hima C. Patil. “Location Based Authentication: A New Approach towards Providing Security”. In: *International Journal of Scientific and Research Publications* 2.4 (2012-04).
- [11] *Mifare Classic*. URL: [https://www.nxp.com/products/rfid-nfc/mifare-hf/mifare-classic:MC\\_41863](https://www.nxp.com/products/rfid-nfc/mifare-hf/mifare-classic:MC_41863) (visited on 2019-09-24).
- [12] *MIFARE Classic EV1 1K - Mainstream contactless smart cardIC for fast and easy solution development*. URL: [https://www.nxp.com/docs/en/data-sheet/MF1S50YYX\\_V1.pdf](https://www.nxp.com/docs/en/data-sheet/MF1S50YYX_V1.pdf) (visited on 2019-09-24).
- [13] Wolfgang Issovits and Michael Hutter. “Weaknesses of the ISO/IEC 14443 Protocol Regarding Relay Attacks”. English. In: *Conference on RFID-Technologies and Applications - RFID-TA 2011, IEEE International Conference, Barcelona, Spain, September 15-16, 2011, Proceedings*. United States: Institute of Electrical and Electronics Engineers, 2011, pp. 335–342. ISBN: 978-1-4577-0028-6. DOI: 10.1109/RFID-TA.2011.6068658.

- [14] René Habraken, Peter Dolron, Erik Poll, and Joeri de Ruiter. “An RFID Skimming Gate Using Higher Harmonics”. In: (2015-11). URL: [https://doi.org/10.1007/978-3-319-24837-0\\_8](https://doi.org/10.1007/978-3-319-24837-0_8).
- [15] Starbug Karsten Nohl David Evans and Henryk Plötz. “USENIX Association 17th USENIX Security Symposium 185Reverse-Engineering a Cryptographic RFID Tag”. In: (2008-02). URL: [https://www.usenix.org/legacy/events/sec08/tech/full\\_papers/nohl/nohl.pdf](https://www.usenix.org/legacy/events/sec08/tech/full_papers/nohl/nohl.pdf).
- [16] Carlo Meijer and Roel Verdult. “Ciphertext-only Cryptanalysis on Hardened Mifare Classic Cards”. In: (2015-10), pp. 18–30. DOI: 10.1145/2810103.2813641. URL: [http://www.cs.ru.nl/~rverdult/Ciphertext-only\\_Cryptanalysis\\_on\\_Hardened\\_Mifare\\_Classic\\_Cards-CCS\\_2015.pdf](http://www.cs.ru.nl/~rverdult/Ciphertext-only_Cryptanalysis_on_Hardened_Mifare_Classic_Cards-CCS_2015.pdf).
- [17] Aram Versteegen. *crypto1\_bs*. URL: [https://github.com/aczyd/crypto1\\_bs](https://github.com/aczyd/crypto1_bs) (visited on 2019-12-17).
- [18] *Betjeningsvejledning - SMS 8500 V5*. 2009. URL: <https://ipaper.ipapercms.dk/G4S/kundeservice-erhverv/brugervejledning-g4s-sms8500> (visited on 2019-11-04).
- [19] Gerhard Klostermeier. *MifareClassicTool*. URL: <https://github.com/ikarus23/MifareClassicTool> (visited on 2019-12-17).
- [20] *Personnummeret i CPR-systemet*. URL: <https://cpr.dk/media/17534/personnummeret-i-cpr.pdf> (visited on 2019-11-04).
- [21] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. “Secure Applications of Low-Entropy Keys”. In: *Information Security*. Vol. 1396. Lecture notes in computer science. ISBN 3-540-64382-6. Springer, 1998-09, pp. 121–134.
- [22] William E. Burr, Donna F. Dodson, and W. Timothy Polk. *Electronix Authentication Guideline*. Tech. rep. Appendix A. National Institute of Standards and Technology, 2004-06.
- [23] Udi Manber. *A Simple Scheme to Make Passwords Based on One-Way Functions Much Harder to Crack*. Tech. rep. University of Arizona, 1994-11.
- [24] Ulrich Drepper. *Unix crypt using SHA-256 and SHA-512*. 2016-08. URL: <https://akkadia.org/drepper/SHA-crypt.txt> (visited on 2019-12-17).
- [25] *Foreign Keys*. URL: <https://www.postgresql.org/docs/11/tutorial-fk.html> (visited on 2019-12-11).
- [26] *sqlx golang library*. URL: <https://github.com/jmoiron/sqlx> (visited on 2019-12-11).
- [27] *An overview of HTTP*. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview> (visited on 2019-11-20).
- [28] *HTTP GET method*. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET> (visited on 2019-11-20).
- [29] *HTTP headers*. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers> (visited on 2019-11-20).
- [30] *What Is Rest*. URL: <https://www.restapitutorial.com/lessons/whatisrest.html> (visited on 2019-12-09).



- [31] MDN web docs. *Cross-Origin Resource Sharing (CORS)*. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> (visited on 2019-12-17).
- [32] R.L. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: (1977-09).
- [33] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. “High-speed high-security signatures”. In: *Journal of Cryptographic Engineering* 2.2 (2012-09), pp. 77–89. ISSN: 2190-8516. DOI: 10.1007/s13389-012-0027-1. URL: <https://doi.org/10.1007/s13389-012-0027-1>.
- [34] *MFRC522 - Standard performance MIFARE and NTAG frontend*. URL: <https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf> (visited on 2019-11-25).
- [35] Stian Eikeland. *go-rpio*. URL: <https://github.com/stianeikeland/go-rpio> (visited on 2019-12-17).
- [36] Flavio Garcia, Gerhard Gans, Ruben Muijers, Peter van Rossum, Roel Verdult, Ronny Schreur, and Bart Jacobs. “Dismantling mifare classic”. In: *Lect. Note. Comput. Sci.* 5283 (2008-10), pp. 97–114. DOI: 10.1007/978-3-540-88313-5\_7.
- [37] Miguel Balboa. *Arduino Library for MFRC522*. URL: <https://github.com/miguelbalboa/rfid> (visited on 2019-12-17).
- [38] *Effective Go: Methods*. URL: [https://golang.org/doc/effective\\_go.html#methods](https://golang.org/doc/effective_go.html#methods) (visited on 2019-11-04).
- [39] Libnfc contributors. *nfc-tools/libnfc/iso14443-subr.c*. URL: <https://github.com/nfc-tools/libnfc/blob/04ef5ca90225aa3eb230d66bbd0717469d9cdf3d/libnfc/iso14443-subr.c> (visited on 2019-12-17).
- [40] Sean Eron Anderson. *Bit Twiddling Hacks*. URL: <http://graphics.stanford.edu/~seander/bithacks.html#ParityParallel> (visited on 2019-12-12).
- [41] *Proxmark3 source version of Crpto1*. URL: <https://github.com/Proxmark/proxmark3/tree/master/common/crpto1> (visited on 2019-12-13).
- [42] *Interfaces in Go (part I)*. URL: <https://medium.com/golangspec/interfaces-in-go-part-i-4ae53a97479c> (visited on 2019-12-13).
- [43] NXP. *AN10969 - System level security measures for MIFARE installations*. URL: <https://www.nxp.com/docs/en/application-note/AN10969.pdf> (visited on 2019-12-17).



# Key Strengthening

# A

```
1 #!/usr/bin/env python3
2
3 tidForskel = []
4 for t in range (1, 5000):
5     for s in range (0, 5000):
6         tidForsvar = (s*(2*t))/2
7         if tidForsvar > 500:
8             continue
9         tidAngriber = (s*(10**4)*2*t)/2
10        diff = tidAngriber - tidForsvar
11        parameters = [diff, t, s, tidForsvar, tidAngriber]
12        tidForskel.append(parameters)
13
14 def by_first_entry(entry):
15     return entry[0]
16
17 tidForskel.sort(reverse = True, key=by_first_entry)
18
19 print("diff    t    s    T_f    T_a ")
20 for i in range(0, 20):
21     print(tidForskel[i])
```

**Snippet A.1:** Source code used for calculating effect of key strengthening parameters