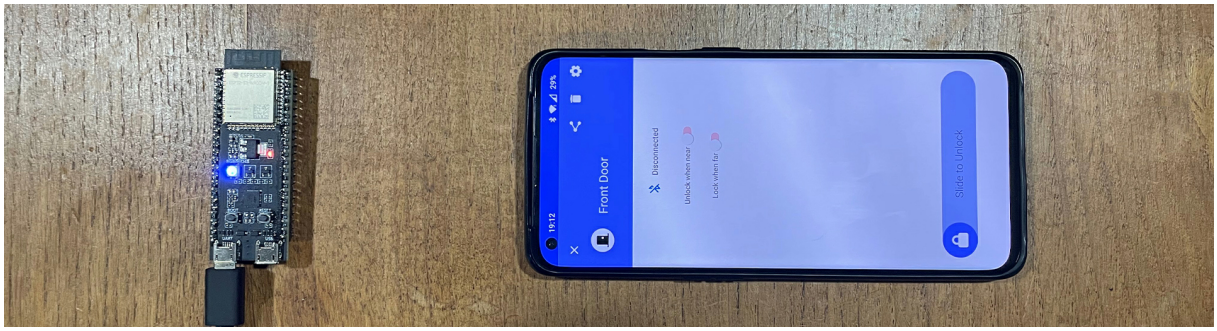
TÉCNICO LISBOA



IoT Smart Door Lock with Wireless Key Sharing for Tourism Applications

Bernardo Cabral Marques

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Rui Policarpo Duarte
Eng. Helena Cruz

Examination Committee

Chairperson: Prof. Name of the Chairperson
Supervisor: Prof. Rui Policarpo Duarte
Members of the Committee: Prof. Name of First Committee Member
Dr. Name of Second Committee Member
Eng. Name of Third Committee Member

October 2022

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible.

I would also like to acknowledge my dissertation supervisors Prof. Rui Policarpo Duarte and Helena Cruz for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life as well as everybody that contributed in some way to my academic success. Thank you.

To each and every one of you – Thank you.

Abstract

IoT technology has rapidly grown in recent years, both in terms of research and product development. The usage of IoT technology is growing in a variety of Smart Home applications, such as Smart Home Security, as a result of the development of technologies like 5G network and BLE as well as the appearance of faster and more compact embedded systems. In this project, we specified and implemented a system to work as a Door Lock capable of being controlled by the user's smartphone in a user-friendly and secure way, using BLE and IoT technology. This system focuses on capabilities that let the user remotely grant access to his Smart Door Lock, which makes it a great resource for rental housing environments. In addition to on-demand remote and close-range unlocking and locking of the Smart Door Lock, the system also contains a feature that automatically controls the lock depending on proximity without requiring user interaction with a smartphone. The final system was successfully prototyped, evaluated, and demonstrated. The results and system characteristics indicate that it is feasible to incorporate the final system in a future commercial product.

Keywords

Smart Home; Smart Door Lock; Bluetooth Low Energy; Internet of Things; Home Security; Embedded Systems

Resumo

A tecnologia IoT tem crescido rapidamente nos últimos anos, tanto em termos de investigação como de desenvolvimento de produtos. A utilização da tecnologia IoT está a crescer em várias aplicações de Smart Home, tal como a área de Smart Home Security, resultado do desenvolvimento de tecnologias como rede 5G e BLE, bem como do aparecimento de sistemas embutidos mais rápidos e compactos. Neste projecto, implementámos uma Fechadura Inteligente capaz de ser controlada pelo smartphone do utilizador de uma forma intuitiva e segura, utilizando tecnologias BLE e IoT. Este sistema está focado em capacidades que permitem ao utilizador conceder acesso remoto à sua Fechadura Inteligente, o que o torna num grande recurso para ambientes de arrendamento de habitações. Para além das funcionalidades trancar e destrancar a Fechadura Inteligente do a curta distância e remotamente, o sistema também tem uma funcionalidade que controla automaticamente a fechadura com base na proximidade, sem necessidade de qualquer interacção do utilizador com um smartphone. O sistema final foi prototipado, avaliado e demonstrado com sucesso. Os resultados e as características do sistema indicam que é viável incorporar o sistema final num futuro produto comercial.

Palavras Chave

Casa Inteligente; Fechadura Inteligente; Bluetooth Low Energy; Internet das Coisas; Segurança Doméstica; Sistemas Embebidos

Contents

List of Figures

List of Tables

Listings

Acronyms

ADB	Android Debug Bridge
AWS	Amazon Web Services
BLE	Bluetooth Low Energy
CA	Certificate Authority
EOT	End-of-Transmission character
GATT	Generic Attribute Profile
HOTP	HMAC-based one-time password
IoT	Internet of Things
MCU	Micro-Controller Unit
MTU	Maximum Transmission Unit
NFC	Near Field Communication
NVS	Non-Volatile Storage
OTP	One-Time Password
RTT	Round Trip Time
RNG	Random Number Generation
SDL	Smart Door Lock
SSL	Secure Sockets Layer
TRL	Technology Readiness Level
UART	Universal Asynchronous Receiver/Transmitter
UID	Unique Identifier
UI	User Interface
UUID	Universal Unique Identifier

1

Introduction

Contents

Internet of Things (IoT) technologies is a field that is rapidly expanding both in research and product development. The emergence of faster and more compact embedded systems and growing technologies like 5G network and Bluetooth Low Energy (BLE) make the use of IoT technology increase in multiple applications, mainly in Smart Home applications. Nearly every appliance or gadget in our home now has a Smart Home equivalent, including Smart Speakers, Smart Bulbs, Smart Plugs, Smart Fridges, and numerous others. Security-related devices, such Smart Alarm or Smart Door Lock systems, are a significant application of the Smart Home. We focused on Smart Door Lock and Keyless Entry systems for our project, in part because of the recent rise in short term rental homes, a prime application for our system.

Multiple lock systems are now starting to use IoT technologies, such as passcode locks, fingerprint locks and even smart locks using a secret code saved in a magnetic band card, Near Field Communication (NFC) card or smartphone. But most of the already existing systems have some flaws or related problems, for instance, security flaws, low power efficiency or a lack of essential features.

1.1 Motivation

There are already multiple implementations of Smart Door Locks, using a range of different technologies. However, most of these systems are commercial solutions, and therefore their implementations are opaque and secret. Another problem with the existing systems is the lack of some essential features while maintaining system security.

This project aims to create a system packed with user-friendly features, such as ”Proximity control”, ”Unlock on-demand mode”, ”Remote Control”, ”Guest access and User roles”, and others, while maintaining security and low power consumption in a compact, highly integrated and low-cost product. In this project we focus on features that will allow the user to remotely share access to his locks without meeting in person with the new user.

The final solution will be open-source, allowing the cooperation and creation of a more secure and robust system by the community. That way the users can verify that the system is in fact secure, instead of believing the claims of a black-boxed commercial solution, and even contributing to creating an even more robust and feature complete solution.

1.2 Objectives

The purpose of this project was to develop a Smart Door Lock (SDL) system, that can be remotely controlled using a smartphone application. The proposed system will solve some problems of the existing systems and add some extra features.

The main challenge of this project was to create a secure, reliable and efficient system with low power consumption and a rich feature set for the end-user. Another focus of this system was a feature that allows the user to remotely create and send keys to another user, allowing guests or tenants to use the smart lock without meeting the owner.

The proposed system was designed to be applied in multiple situations, private home locks, building or apartment complex main door, a shared building like company headquarters doors, rental houses for tourism (like Airbnb), hotel or motel rooms.

1.3 Organization of the Document

This thesis is organized as follows: Chapter 1 is an introduction to the project, with its motivation and objectives. In chapter 2 we provide a brief background overview, where we analyse related work and introduce the relevant technologies. In chapter 3 we propose an architecture for our system, gathering the requirements and designing a proposed solution. In Chapter 4 we explain and describe the multiple steps in the implementation of the proposed system, where we talk about each component and explain how they were developed. Finally, in Chapter 5 we evaluate the implemented system, validating if we meet the requirements. We conclude with some final remarks about the developed project and the possible future work in Chapter 6.

2

Background and Related Work

Contents

2.1 Existing Smart Lock solutions

This section takes a look at some types of IoT Smart Door Locks. It explores more traditional systems that require the user to manually input the authorization credentials, either by typing a code on a keypad, by swiping an ID card in the lock or using biometric sensors. And others that use wireless technologies like Wi-Fi or Bluetooth to automatically provide the credentials over the air.

2.1.1 Keypad Smart Locks

This type of smart lock is used a lot to secure a shared room and it is probably the oldest of them all. [1] In these locks, the user has a passcode to open the door. The user has to manually input the passcode into a keypad and if the code is correct the door unlocks. In some locks, it is possible to have multiple passcodes for different users. That way is possible to log the use of the door or even block some codes from working on certain days.

This solution has the advantage of being easy to share keys with a different user since it is only necessary to share a passcode. However, it lacks security, the user can use a passcode with more digits to increase the security, however, it is difficult to have a secure and usable passcode that the user can remember. It is also possible to steal a passcode just by watching the user typing it. In terms of ease of use for the end-user, it is relatively easy to memorize a passcode for a door, however, it is difficult to memorize multiple codes for multiple doors. Finally, the maintenance of this system have some disadvantages, if the user needs to change the code it normally is necessary to manually change it in the look, and then give the new code to the other users.

It is also important to mention that there are more complex types of smart locks with keypads, that use One-Time Password (OTP) in order to increase security.

2.1.2 RFID and Magnetic cards Smart Locks

This type of lock is widely used in hotels and rental rooms (figure 2.1). In this system, the user has a keycard or RFID tag with the credentials to unlock the door. [2] The user needs to swipe the card in the lock to open it. These cards can use RFID technologies or magnetic bands. With these locks, it is also possible to log and manage the use of the door because each user can have a different keycard.

This solution can be very secure because the credentials are stored in the keycard, so it is possible to use more advanced encryption algorithms. One flaw of it is the possibility to clone the keycards, or even steal them, but assuming that the user secures the keycard, it is difficult to break the system. Another advantage of these locks is the ease of use and the low power consumption. The main disadvantage of this solution is the difficulty to share keys with other users. If the user wants to share a key it is necessary to create a new keycard for the new user, and then physically give it to the user.

Figure 2.1: Hotel door lock using RFID keycard

2.1.3 Biometric Smart Locks

Biometric sensor locks work by verifying a unique physical characteristic of the user, such as fingerprints [3], face recognition [4], voice recognition [5], iris scanner or palm recognition. This type of lock is normally used in high-security facilities. They work by scanning a physical characteristic of the user and converting it to a numerical template. Then, when the user tries to unlock the door it compares his biometric feature to the saved template.

This solution provides a highly secure way to authenticate a user because these biometric features are unique to a given user. The most common way these systems are hacked is by duplicating the user template credentials and physically hacking the sensor to fake the scanning. The main disadvantage of these systems is the difficulty to share access to the door because the new user needs to register his biometric feature in loco.

2.1.4 Wireless Smart Locks

The most recent innovation in smart locks is related to the use of wireless technologies to automatically send the credentials over the air. These technologies include 5G, Wi-Fi [6], Bluetooth [7], ZigBee, Z-Waze, BLE and many others. The basic concept is to use a mobile device (i.e. smartphone) to communicate with the lock sending the necessary credentials over the air and unlocking the door. This kind of system allows the use of more complex encryption algorithms and, therefore, create more secure systems. All these can be done automatically and without any input from the user. There are already many commercial solutions using this kind of implementation, for instance, the ”Salto XS4 One”[1] and the ”Google Nest x Yale”[2].

These systems have many advantages, like the possibility to create secure systems based on mod-

[1] https://saltosystems.com/en/products/electronic-locks/xs4-one/
[2] https://store.google.com/us/product/nest_x_yale_lock?hl=en-US

ern cryptography algorithms and the ease of use to the end-user. Another advantage of these systems is the possibility to generate keys and share them with other users remotely. Finally, it is possible to fine-tune the communication algorithms and technologies to manage the power consumption and create a system that can be powered by batteries.

2.1.5 BLE Smart Locks

BLE is a technology developed by the Bluetooth Special Interest Group (Bluetooth SIG[3]). BLE focus on achieving a similar communication range as Classic Bluetooth and other widespread wireless technologies while considerably reducing the power consumption and cost. BLE is primarily designed for short-distance transmissions of small amounts of data to reduce power consumption. Unlike Classic Bluetooth, BLE is not always on, it normally remains in sleep mode until a connection is established.

These characteristics, as well as the large availability in most modern devices, make BLE an excellent candidate for many IoT applications, namely Smart Home devices.

According to research by Gomez et al., [8] the theoretical lifetime of a BLE device powered by a coin cell battery can range between 2 days and 14.1 years, depending on multiple factors like latency and throughput and the purpose of the application. So, the developer can tune these settings for better power consumption, which allows it to be embedded in small devices with low charge and small batteries and lasts a long period of time.

This section analyses some solutions that use wireless technologies to automatically provide the credentials over the air, mainly BLE. This technology was chosen since this type of lock has the most advantages, as we will show in the table 2.1.

A – Design of smart lock system for doors with special features using Bluetooth technology [9]: The authors of this paper aimed to create an automated and secure BLE security system to lock and unlock a door. This system is mainly focused on disabled and elderly people.

Beyond the automated door lock using BLE, this system also provides a doorbell that triggers a camera to capture an image of the person requesting access. The owner can provide temporary access to the person by unlocking the door remotely.

As an alternative to the automatic unlocks, the user can use a fingerprint sensor to unlock the door.

- This system uses a Passcode generated by the lock to pair the devices.
- The device is authorized using the whitelist of MAC addresses.
- No extra encryption method is mentioned on the paper to encrypt the communication between devices.

[3] https://www.bluetooth.com/

B – Design of Smart Lock System for Doors with Special Features using Bluetooth Technology [10]: The authors of this paper present a door locking system that allows the user to operate a door using only his phone, with BLE.

This system aims to solve problems present in other IoT-based locks. Such as forgetting the password or losing the device used to control the lock, preventing the user from opening it.

In this system, the mobile phone can be used to automatically open the door when the user is within a given range of it.

The system works in 2 different modes:

1. **Activating automatic mode:** This mode allows the door to be automatically open if the user comes into a 1m range of the door, using BLE.

2. **Activating fingerprint mode:** In this mode, the system will require the user to input his fingerprint into the mobile phone to open the lock.

The system also includes a webpage with two different roles, Admin and User. On this page the Admin can manage the lock, adding, editing or removing data and user credentials from the device. The User can only manage his own account.

In terms of security, this system uses Bluetooth defaults like:

- **PIN/Legacy Pairing** for paring and key exchange;

- **Legacy Authentication** in the connection process after the first paring;

- And the **E0 Encryption Algorithm** to encrypt and secure the communications.

The paper does not specify how the system checks if a certain device has access to a door.

C – Investigation and Application of Smart Door Locks based on Bluetooth Control Technology [11]: In this paper, it is presented a new design method for Bluetooth control-based intelligent door lock. The requirements of this design are economic cost, safety performance and unlocking efficiency.

In this system, the user can automatically open the door by connecting his device to the door, if the user is authorized.

Security-wise the lock in this system asks the device for a verification code after connecting, if the code is correct the user is authorized.

The paper mentions that the messages exchanged by the lock and the mobile devices are encrypted, however, it does not specify which encryption algorithm is used.

D – Digital Door-Lock using Authentication Code Based on ANN Encryption [12]: This paper presents an electronic door lock mechanism that focuses on creating a system to generate temporary access keys for guests. In this system, the door is controlled by a smartphone using BLE.

With this system, the user has a unique digital key used to control the door, this key is generated based on the guest's stay duration

When a new customer rents a room, the owner can generate the digital key for a given door on the smartphone. The owner inputs the check-in and check-out date and the application generate a digital key for the customer.

The user is authorized with an OTP, generated by a block cypher algorithm using ANN, based on the user's digital key and seed. This OTP is then sent to the lock and validated, if the OTP is valid the door unlocks and then automatically locks after some seconds.

E – Implementation and design issues for using Bluetooth low energy in passive keyless entry systems [13]: This paper aims to create a prototype of a Passive Keyless Entry System (PKES) to operate a lock using BLE. This aims to validate the use of BLE for this kind of system. The system uses a white-listing mechanism, present in the BLE feature set, based on the device Identity Resolution Key (IRK) to identify the authorized users. This system uses the security features already present in BLE to pair and encrypt the communication between devices.

The paper concludes that BLE provides a plausible solution to PKES which is both, extremely energy-efficient and also relatively secure.

2.1.6 Comparison between Smart Locks

A – Comparison between Smart Locks types: As it is shown in the previous sections and in table 2.1, all these different types of locks have advantages and disadvantages. Keypad locks are a good choice if it is necessary to share access with multiple users, but they lack security and they are hard to maintain. Both keycard technologies and biometrics, are good candidates for a secure lock, but it is hard to share access without coming into close contact. Finally, we have locks that use wireless technologies, these locks, if used correctly, can provide a high level of security while being easy to share between users, being easy to use, requiring low maintenance and having a good power efficiency.

B – Comparison between BLE systems: Most of the BLE systems presented in section 2.1.5 have a proximity control feature, where the door automatically opens when the user enters a given range. However, just the Hadis et al. system has a customizable range. Some systems have a feature to unlock the door on-demand, but none of them allows the user to do this remotely. And only Karani et

Table 2.1: Table comparing the different types of smart lock

Smart Lock Types	Security	Share access	Ease of use	Maintenance	Energy efficiency	Avg. Cost
Keypad	Poor	Excellent	Good	Poor	Good	$150
RFID or Magnetic Keycards	Excellent	Poor	Good	Good	Good	$200
Biometric	Excellent	Poor	Good	Poor	Good	$350
Wireless	Excellent	Excellent	Excellent	Excellent	Good	$250

Table 2.2: Table comparing the features of the above enumerated smart locks

System	Proximity control	Unlock on-demand	Unlock on-demand Remote	Website to manage users	Guest Access	Users Roles
Prakash et al. 2017 [9]	Yes[4]	Yes[5]	Not entirely[6]	Yes	Not entirely[7]	No
Hadis et al. 2018 [10]	Yes	Yes[8]	Not mentioned	Yes	No	No
Mu et al. 2020 [11]	Yes[4]	Not mentioned	Not mentioned	Not mentioned	No	No
Hanafiah et al. 2021 [12]	No	Yes	No	No	Yes[9]	Yes[10]
Karani et al. 2017 [13]	Yes[4]	No	No	No	No	No

al. have guest access. Table 2.2 shows a comparison of features between each of the aforementioned systems.

2.2 Traditional key sharing mechanisms

One of the main features intended for our system is its use in rental houses to facilitate key sharing between the owner and the tourist, thus, it is important to analyse the conventional ways used to share keys in rental houses situations.

- **In-person delivery** - The most basic way to share keys is to meet the guests in person and give them the key when first arriving at the house. This method has some obvious disadvantages, just as time coordination and unnecessary dislocations. In this case, the owner and the guests have

[4] Bluetooth range
[5] with fingerprint, but not implemented
[6] only if user rings the bell
[7] yes, but if requires the owner to authorize the guest in real-time
[8] with fingerprint
[9] main feature
[10] owner and guest

to arrange a specific time to meet in person, which can lead to time-wasting, and it can even be impossible if the owner is not nearby.

- **Rely on a third person** - Another usual method is to rely on a third person, like a neighbour, a cleaner or even a local coffee shop, to deliver the key to the guests. This is slightly better than the previous method, the owner can live further from the rental house, however, it shares most of its disadvantages. The third person is also time-constrained, and he might not be available 24/7 to deliver the key.

- **A Lockbox** - One of the most used solutions is to have a lockbox, a small and secure container to store the house keys inside the property. With a lockbox, the owner communicates the correct lockbox combination, and the guest can unlock the lockbox and retrieve the house key. This solves all the time constraints given that the key is accessible 24/7, without relying on an in-person meeting. However, this method has some disadvantages of its own. The owner has to trust the guests to store the key in the lockbox when leaving, without stealing the key, and if the combination to the lockbox is not changed often, someone could steal the combination, and access the house without the owner's permission.

These are the most usual solutions to share keys in a rental house, and as we can see, we have some disadvantages to all of them. We believe that our SDL system solves most of these problems, giving the easiness and comfort of the lockbox, without the need to worry if the guest will return the key since the system will automatically deny access when the rental period ends.

2.3 Security requirements

This project requires a high level of security, and so it is recommended to use modern encryption algorithms. In the NIST "Transitioning the Use of Cryptographic Algorithms and Key Lengths" publication [14], it is recommended the use of AES-256 for encryption and decryption of messages and RSA with keys longer than 2048 bits for key transportation and exchange.

Although it is not strictly mandatory, it is also recommended to use some kind of hardware-accelerated encryption modules. Since these encryption algorithms can be very heavy in computational requirements, these hardware-accelerated modules can help reduce power consumption and increase time efficiency, making them essential in this project.

2.4 Hardware Platforms and ESP32-S

Given the objectives of this project and the security and power consumption requirements, multiple boards were considered to build the SDL system, some of which were used in the work reviewed. We considered using the Atmel ATmega32U4, the PSoC 4 BLE and ESP32-S2/S3.

A – Atmel ATmega32U4 This board was used in [12], together with a BLE module. ATmega32U4 is a low-power microcontroller with 32K Bytes of ISP Flash and USB Controller.

B – PSoC 4 BLE This board was used in [9] and [13]. PSoC 4 BLE is a Programmable System on Chip, featuring a BLE radio. It was up to 256KB flash and 32KB SRAM.

C – ESP32-S2/S3 ESP32-S2 [15] and ESP32-S3 [16] are highly integrated, low-power, single-core Wi-Fi Microcontrollers SoC, designed to be secure and cost-effective, with high performance and a rich set of IO capabilities.

Ultimately the ESP32-S family (S2 and S3) was chosen for this project because of the low-power features and hardware-accelerated security features.

- **AES Accelerator:** ESP32-S2/S3 integrates an Advanced Encryption Standard (AES) Accelerator, which is a hardware device that speeds up AES algorithms significantly, compared to AES algorithms implemented solely in software. It is essential for this project because AES will be used to encrypt the messages exchanged between the different components.
- **RSA Accelerator:** The RSA Accelerator provides hardware support for high precision computation used in various RSA asymmetric cipher algorithms by significantly reducing their software complexity. Compared with RSA algorithms implemented solely in software, this hardware accelerator can speed up RSA algorithms significantly. This module is essential for this project because an RSA based cipher algorithm will be used to exchange the AES keys between components.
- **Random Number Generator (RNG):** The ESP32-S2/S3 contains a true random number generator, which generates 32-bit random numbers that can be used for cryptography operations. This random number generator generates true random numbers, which means random numbers generated from a physical process, rather than by means of an algorithm. This will be useful for creating a secure random seed that will be used in cryptography algorithms.
- **Low-Power Management:** ESP32-S2/S3 has an advanced Power Management Unit (PMU), which can flexibly power up different power domains of the chip, to achieve the best balance among chip performance, power consumption, and wakeup latency. ESP32-S/S3 has integrated

two Ultra-Low-Power co-processors (ULP co-processors), which allow the chip to work when most of the power domains are powered down, thus achieving extremely low power consumption.

- **BLE Modules:** Unfortunately the ESP32-S2 does not contain an integrated BLE module. This is not a problem since we can use external BLE modules that will communicate with the ESP32-S2 via the Universal Asynchronous Receiver/Transmitter (UART) protocol. We will also use the new ESP32-S3. The ESP32-S3 contains all of the above mention features and it also contains an integrated BLE module.

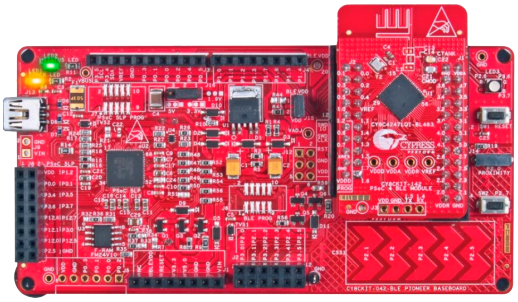


Figure 2.2: PSoC 4 BLE

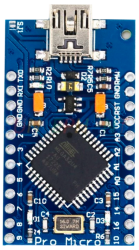


Figure 2.3: ATmega32U4

Figure 2.4: ESP32S2

3

Proposed IoT Smart Lock System

Contents

3.1 System Requirements

The smart lock system developed has some features already present in the existing systems we analysed in chapter 2. The main difference and novelty, beyond the open source nature of this system, is the focus on the key sharing aspect of the system, which allows users to share access to the smart lock with multiple people with different levels of access control, as we will explain in more detail further. We also used the brand new ESP32-S3 SoC, with BLE 5.0 and WiFi support. This feature will be very useful in home renting for tourism scenarios.

The main features of this smart lock system are the following:

- **Proximity control:** Lock or unlock the smart lock if the user enters or exits a given range. This is one of the main features of the SDL system. This feature allows the smart lock to detect a mobile phone (with BLE, Wifi and GPS support) with permission to control it, automatically unlocking the smart lock when the user enters a close range of the smart lock and then automatically locking it when the user leaves that range. This feature is customizable by the user, it is possible to change the range of control, and enable or disable it. The system uses BLE to detect and communicate between the mobile phone and the smart lock.

- **Unlock on-demand mode:** This feature allows the user to lock or unlock the smart lock on-demand through the mobile app. With this feature, a user with permission can control the smart lock without relying on the proximity control feature. This feature uses BLE to communicate between the device and the smart lock.

- **Unlock on-demand mode remote:** This feature allows the user to remotely lock or unlock the smart lock on-demand through the mobile app. With this feature, a user with permission can remotely control the smart lock from any place in the world. This feature uses an internet connection to communicate between the device and the smart lock.

- **Guest access and User roles:** This is the main focus of this system, and it gives the possibility to share access to a given lock. The user can choose between multiple user roles which can control the smart lock, but with different levels of permissions.

A – User types (Permissions levels):

- **Admin:** This role is the highest permission level a user can have. This role is meant to be used by the owner of the property or the system administrator. The admin user is used to configure and maintain the system and can add, edit and remove any users. This is the first role to be assign, and it is created upon the first configuration of the smart lock.

- **Power-User/Owner:** This role is meant to be used by the owner of the property and his co-habitants. The power-user can entirely control the smart lock, add, edit and remove any users (except admins). This is similar to the admin role in terms of permissions, but it is more focus on using the lock instead of configuring and maintaining the system.

- **Temporary user/Tenant:** This role is meant to be used by a user that will have access to the property temporary, like a tenant. This user is able to control the smart lock within a given range of dates. This user is also able to add edit or remove users to control the smart lock within the same range of dates as himself. For instance, the Owner creates a Temporary user account for his tenant. This account will be valid for a period of a year and the user will be able to control the smart lock within this year, and also create a sub-account to access the smart lock within this year (e.g. for his son). This user can only create accounts with the same or lower permissions than himself, and only within the same range of dates as himself.

- **Periodic user:** This role is meant to be used by a user that has access to the property for an uncertain time, but only periodically, like a cleaning company employee. This user is only able to control the smart lock at very strict periods of time, for instance, every Monday from 9 am to 5 pm. This user won't be able to add, edit or remove any other user.

- **One-time user:** This role is meant to be used by a user that will only have one-time access to the property, like a maintenance company employee. This user is only able to control the smart lock in a short period of time, for instance, the next Monday from 1 pm to 5 pm. This user won't be able to add, edit or remove any other user.

3.2 System Overview

The proposed system has three main components: the **IoT to control the Smart Lock**, a **central service server** and a **mobile application**. The **Smart Lock** consists of a Micro-Controller Unit (MCU) with BLE and WiFi communication capabilities which can be powered by batteries, the ESP32-S2/S3 (section 3.4). The **Central Service Server** is a Linux machine running in the cloud (AWS) which is responsible for managing access to the database and remotely controlling the Smart Locks (section 3.5). The **Mobile App** is an Android app capable of running in any smartphone that supports BLE, WiFi and GPS. The user can interact with the smart lock via the mobile application. The mobile application is used mainly to control directly the smart lock, lock and unlock when in range and lock and unlock on-demand. The mobile application also allows the owner or tenant to manage his smart locks and their users (section 3.6).

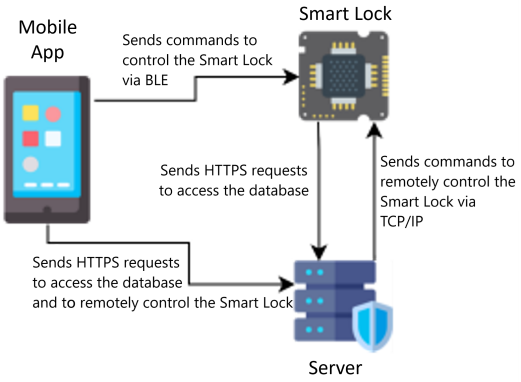


Figure 3.1: Multiple components of the system and how they interact.

As it is shown in figure 3.1 the multiple components communicate with each other using different communication protocols and technologies. The Central Service Server is an HTTPS server and so, all the components that communicate with the Central Service Server use HTTPS. Through these HTTPS requests, the Mobile App and the Smart Lock communicate with the Service Server to access the database and to request remote control operation. The Mobile App is also capable of communicating with the Smart Lock, sending string commands to request operations in the Smart Lock, like requesting to unlock it. This communication is done via BLE. Finally, the Service Server can also send remote commands to control the Smart Lock, if requested by the Mobile App, via a TCP/IP connection.

3.3 Secure Communication

Given the system's security requirements, it's critical to protect communications between each component to stop hackers from breaking into the system or listening in on conversations and obtaining sensitive data about the lock or the user.

Since an HTTPS server is used for all communications with the central service server, this connection is already secured using SSL/TLS. However, both communication channels with the Smart Lock are not secure by default. For that reason, it was necessary to implement an extra layer of security on top of the communication channels. In this secure channel, all the messages exchanged are encrypted using AES-256 in CRC mode.

In figure 3.2 it is possible to see how the creation of this secure channel works. First, the mobile phone tries to connect to the smart lock (IoT). When the connection is accepted, the phone then creates an AES-256 session key and send it to the smart lock, encrypted with the smart lock public key. Only the smart lock can decrypt this message and therefore retrieve the session key. This key is then saved by the smart lock, and it is used to encrypt all the messages in this session.

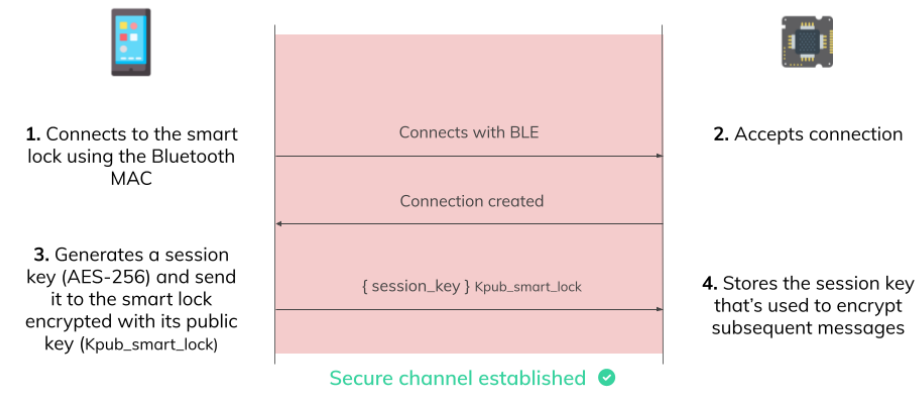


Figure 3.2: Creation of a secure channel between the mobile app and the smart lock

The creation of a secure channel between the Service Server and the IoT is similar to the one explained above, the only difference is that they communicate via TCP/IP instead of BLE.

After this initial session key exchange, all the following messages are encrypted with this session key. The messages also include timestamps and a nonce to prevent replay attacks as it is shown in figure 3.3.

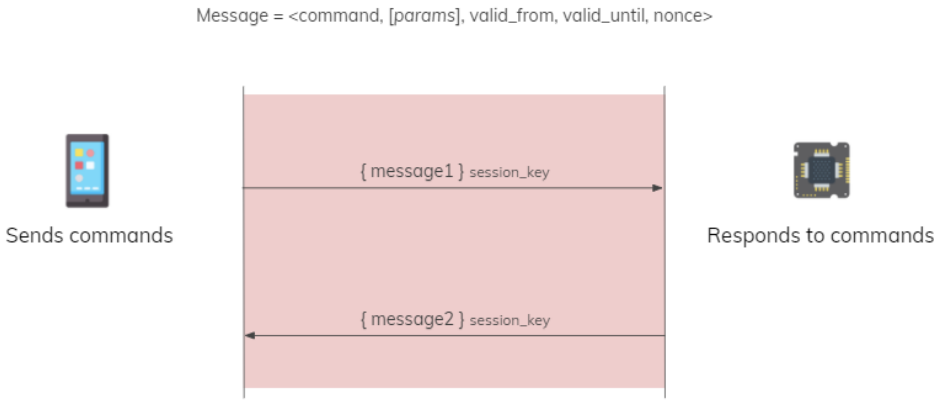


Figure 3.3: Secure communication between the phone and the smart lock

3.4 Smart Lock Hardware - ESP32-S2/S3

The smart lock is a crucial components in this system. The idea was to use a MCU with BLE and WiFi capabilities, the ESP32-S2 and ESP32-S3, to control an electronic lock mechanism or a latch with a servo motor. The ESP32-S2/S3 acts as a gateway between the client application and the lock. It is running a BLE Server (section 3.4.3) to communicate with the Mobile App and a TCP server (section 3.4.4) to communicate with the central service server.

3.4.1 Hardware

The SDL system has 2 proposed solutions, they are very similar between them, the main difference being the hardware it was used.

For the first solution, the hardware consists of the ESP32-S2 IoT and a BLE module. These components communicate through the UART protocol (GPIO15 ,GPIO16, GPIO43, GPIO44).

In the second version, the hardware consists of only the ESP32-S3 IoT, which already contains an internal BLE module.

For this project, the ESP32-S2/S3 is powered by a USB adapter, and the status of the smart lock is be simulated with an RGB led. Considering a scenario for a real implementation for the SDL system, the ESP32-S2 could be powered by batteries (e.g. 18650 lithium-ion cell batteries). To unlock and lock the door, the ESP32-S2/S3 activates one of the available GPIOs.

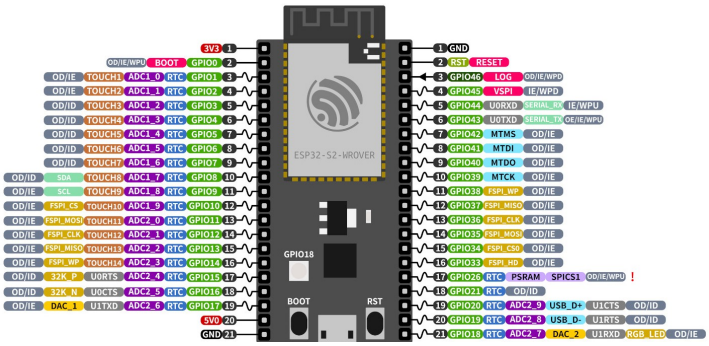


Figure 3.4: ESP32-S2 GPIO schematic

3.4.2 Message protocol and Operations Examples

To control the Smart Lock, the Mobile App Client will be able to send commands to perform certain operations, both through BLE and TCP. This communication uses a simple message protocol that consists of the commands presented in table 3.1.

In appendix A it is shown some examples of communications using the protocol shown in table 3.1.

3.4.3 BLE Server

The BLE Server handles the communication between the Moblie App client and the smart lock. This BLE Server allows the smart lock to be controlled directly by the user at close range, using the Mobile App or its proximity features.

A BLE Server does not work exactly like a TCP server or even a Classic Bluetooth server. The BLE server is called a Generic Attribute Profile (GATT) server, and it contains *services* and *characteristics*

Table 3.1: Message protocol commands

Protocol Commands	Request Result
SSC *session_key*	Send Session Credentials
RAC *seed*	Request Authorization Credentials
SAC *user_id, auth_code*	Send Authorization Credentials
RLD	Request Lock Door
RUD	Request Unlock Door
RDS	Request Door Status
SDS *status*	Send Door Status
RNI	Request New Invite
SNI	Send New Invite
RFI	Request First Invite
SFI	Send First Invite
RUI	Request User Invite
ACK	Acknowledge
NAK	Not Acknowledge

[17]. Services are basically a container that conceptually groups related attributes, while characteristics are the attributes included in a service, and each of them is used to communicate a specific type of data.

The characteristics hold the data that can be accessed by the client. Characteristics can allow different types of operations:

- **Broadcast:** this allows sending data to BLE devices using advertising packet.
- **Readable:** the client can only read the characteristic value.
- **Writable:** with this property, the client can only write a new value on the characteristic.
- **Notifiable:** the client receives a notification if the server updates the characteristic so that it can read the new value.

In this project, the server behaves as a Peripheral (or slave) and periodically sends connectable advertising packets and accepts connections initiated by the Central (or master), the client.

This server generally uses the message protocol explained in section 3.4.2 however it was adapted to work with the operations explained above.

3.4.4 TCP Server

The TCP Server handles the communication between the central service server and the smart lock. This TCP Server allows the smart lock to be controlled with requests from the central service server. The

user is able to send remote requests to the central service server, and these requests are forwarded to the smart lock (section 3.5.1). This server uses the same message protocol (section 3.4.2) as the BLE Server.To prevent unauthorized requests, this server only allows requests from the central service server.

3.4.5 Storage Database

Although most information and data are stored in the Service Server (section 3.5.2), some necessary information is also temporally stored in each smart lock, somewhat like a cache. Each smart lock has a copy of its own ”authorizations” for that smart lock. An ”authorization” is an object containing the master_key associated with the pair {phone, smart_lock}, the type of user, and the validity of the key (figure 3.5).

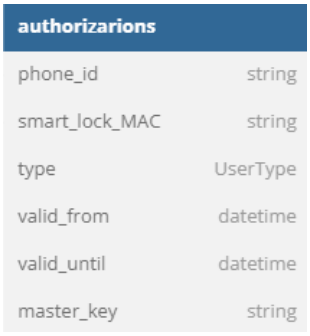


Figure 3.5: Authorization object

This information is stored in the Non-Volatile Storage (NVS) of the ESP32-S2/S3.

3.4.6 User Authorization using OTP

When the user communicates with the smart lock, it is necessary to authenticate and authorize the user, to check if the user can control the smart lock. So, when the user connects to a smart lock, before doing any request it is necessary to authorize the user. This is done with the ”**RAC** *seed*” and ”**SAC** *user_id, auth_code*” messages of the protocol explained in section 3.4.2 (figure 3.6).

The *auth_code* mentioned here is a value calculated by both the client (Mobile App) and the smart lock, using the *master_key* (known by both parties) and the seed provided by the server, with a process known as HMAC-based one-time password (HOTP).

This process consists of using a cryptographic hash function, in this case (SHA-256) and applying it to the *seed* and the *master_key*. Given that the seed is random, generated by the ESP32-S2/S3 Random Number Generation (RNG), this will create a OTP that can be compared in the server, and authenticate the user.

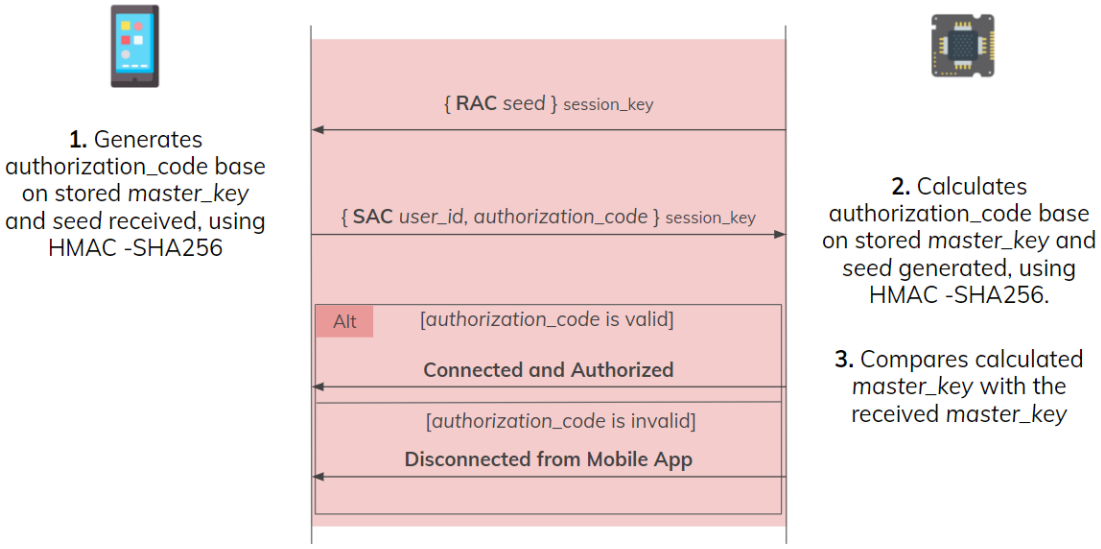


Figure 3.6: Diagram explaining the user authorization protocol

Table 3.2: Table with the main status of the smart lock and its corresponding colour

Colour	Status
Yellow	Idle - Waiting Connection
Blue	Connected - Smart Lock Locked
Green	Connected - Smart Lock Unlocked
Red (flashing)	Invalid Authentication

3.4.7 Simulation of Smart Lock Operation

This thesis only includes the development of the software side of the system, without creating any specified hardware related to a real lock or door. We program and integrate the different hardware components required for testing the full functionalities of the SDL system, but without creating or incorporating it in a lock or deadbolt. For this reason, the smart lock status is simulated using an RGB led integrated into the IoT. In the table 3.2 it is possible to see the main status of the smart lock and its corresponding colour.

3.5 Smart Lock Service Server

The Service Server has two main roles in this system:

- It serves as a **gateway between the Mobile App and the smart lock**, this gateway is used when the user wants to control the smart lock remotely and manage the users and smart locks.

- It is also used to **access the database** that stores all the information about the users and the smart locks.

This server consists of a RESTful API in a python web server built with Flask and hosted in a Linux machine provided by Amazon Web Services (AWS).

The server also interacts with the Authentication Service provided by Google Firebase with specific libraries that use HTTP requests.

3.5.1 Gateway for Remote Control

The Service Server is used as a middleman between the smart lock and the client's Mobile App. This communication channel is used when the Mobile App want to remotely connect to the Smart Lock, via the Internet. In this case, instead of directly connecting to the Smart Lock TCP server, the Mobile App will send the messages to the Service Server through the RESTful API, and the Service Server will redirect these messages to the Smart Lock via TCP/IP, as shown in figure 3.7. By using this middleman, we can improve the security of the Smart Lock by only allowing requests coming from the Service Server and preventing unauthorized access directly to the Smart Lock TCP Server.

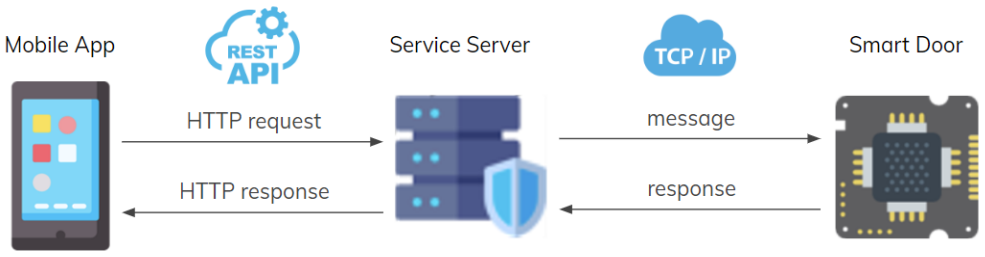


Figure 3.7: Example of communications using the Server as a gateway

3.5.2 Database

The database of the system is handled using Firebase Realtime Database[1]. Firebase Realtime Database is a cloud-hosted NoSQL database provided by Google that is used to store and sync data in real-time. This database is used to store most of the information and data necessary for the system. Figure 3.8 shows some of the objects that are stored in the database.

The Service Server (through the REST API) is used as an interface to communicate with the Firebase Realtime Database. This way it is easier to change the database structure or even provider, without the need to implement a major update in the client, and only update the server, allowing the user to keep using an older version of the app.

[1] https://firebase.google.com/products/realtime-database

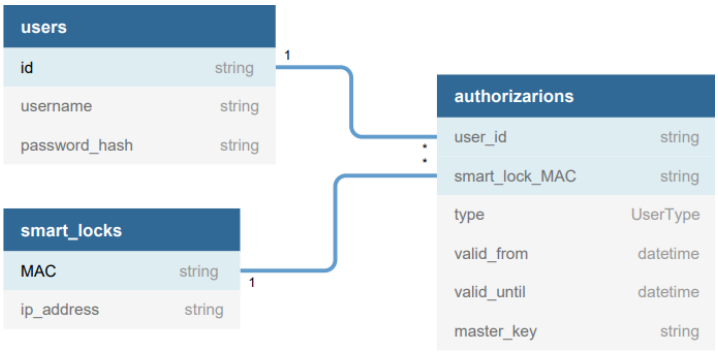


Figure 3.8: Objects to be stored in the server database

3.5.3 Authentication Service

The user login and authentication on the Mobile App are handled by Firebase Authentication[2]. Authentication is a secure and easy user authentication solution provided by Google that supports email and password accounts, phone auth, and Google, Twitter, Facebook, and GitHub login, and more. In this project, we will use email and password accounts and ”Login with Google”.

The communication between the Mobile App Client and the Authentication service is done directly through the Firebase Authentication. However, some communication can also be done through the Service Server.

3.6 Smart Lock Mobile App - Client Key

The Mobile App is the main way to interact with the smart lock, it is the ”key” to open the smart lock. The user can access and control the multiple smart locks registered in the app. It has a list of all the smart locks that the user can control, and he can interact with the smart locks through this app. There are features to remotely lock or unlock the smart lock and options to turn on or off the proximity lock or unlock. The user is also able to check if the smart lock is locked or unlocked. The owner or tenant users also have the features to add, edit or remove users for their smart locks.

3.6.1 Main features

The mobile app has two different sets of features. Features to control a smart lock, meant for every user to access a given smart lock. And features to manage a smart lock and its users meant for the Admin, Owner or Tenant of a given smart lock. A list of the main features of the mobile app is shown in table 3.3.

[2] https://firebase.google.com/products/auth

Table 3.3: System main features

Control features (all users)	
List of smart locks	The user has a list of smart locks that he can control and select one to use.
Toggle Proximity Control	The user has the option to enable or disable the feature that automatically unlocks/lock the smart lock by proximity to the smart lock.
Add a new smart lock	The user has the option to add new smart locks, by using an invite code created by an Admin, Owner or Tenant.
Manage features (Admin, Owner and Tenant)	
Invite new users	The Admin, Owner and Tenant can use this option to create an invite for a new user to control the smart lock.
Remove user	The Admin, Owner and Tenant can use this option to remove a user from the authorized users to control a smart lock.
Edit user	The Admin, Owner and Tenant can use this option to edit a given user

3.6.2 Mobile App User Interface

The mobile app was developed using Java in the Android Studio IDE[3]. For the User Interface (UI), we followed the Material Design[4] guidelines as much as possible. The application is divided into multiple pages, called Activities. Each activity represents a screen or page with a visual UI. A list of the main activities in the application are shown in table 3.4.

Table 3.4: Mobile App activities

Sign-In	An activity where the user can log in to the app.
Sign-Up	An activity where the user can create a new account.
Smart Locks List	This activity contains a list of the user's smart locks and the user can select a smart lock to interact with it.
Smart Lock Control Page	An activity where the user can control a selected smart lock.
Add a new smart lock	In this activity, the user can set up a new smart lock and add it to his account.
Create a new invite	In this activity, the user can create a new invite to add a new user to a smart lock.
User Managing	An activity where the owner of a smart lock can manage the users with permission to control their smart lock.

In the appendix B we show the initial design of the main views of the mobile app compared with the views in the final application.

[3] https://developer.android.com/studio
[4] https://material.io/design

4

Smart Door Lock Implementation

Contents

4.1 Development Methodology

The development process of this project consisted on the implementation of an initial proof of concept to validate our design before developing the whole system, which was done on a per component basis. At the end of the development, the whole system was tested for correct functioning and validation of the implementation of the initial requirements

Although all the different components were developed simultaneously, in the following sections, we decided to split the explanation of each component. In section 4.3 we explain the development of the Hardware, in section 4.4 the development of the Service Server, and in section 4.5 the development of the Mobile App.

4.2 Development Environment

This project was developed mainly on a Windows-powered computer, using multiple different tools and frameworks in the development of the different parts of the SDL system as shown in table 4.1.

Table 4.1: Tools and Frameworks used in the development process

Smart Lock Hardware	
ESP-IDF v4.4.1 [1]	ESP-IDF is a development framework for Espressif SoCs. It was used to compile and flash all the code related to the ESP32-S2/S3.
CLion IDE [2]	CLion is an IDE for C and C++ developed by JetBrains. It was used as the main IDE while coding the ESP32-S2/S3 part of the system since it is coded in C.
Smart Lock Service Server	
Flask [3]	Flask is a micro web framework written in Python. Flask was used to develop the RESTful API interface that is used to communicate with the Central Server.
Firebase Realtime Database [4]	Firebase Realtime Database is a cloud-hosted NoSQL database used to store and sync data in realtime. This Google-provided product was used as a database for our system.
Firebase Authentication [5]	Firebase Authentication is a secure and easy user authentication solution provided by Google. This product was used to authenticate the user in the system.
PyCharm IDE [6]	PyCharm is a IDE for Python developed by JetBrains. This was used as the main IDE while coding the Flask RESTful API in the Service Server.
Smart Lock Mobile App	
Android Studio [7]	Android Studio is the official IDE used to build Android apps. This IDE was used for the development of our mobile app client for Android.
Figma [8]	Figma is a collaborative browser-based interface design tool. This tool was used to create the initial design for the Android mobile app.

[1] https://github.com/espressif/esp-idf
[2] https://www.jetbrains.com/clion
[3] https://flask.palletsprojects.com/en/2.2.x
[4] https://firebase.google.com/products/realtime-database

4.3 Smart Lock Hardware

This section explains the development process of the Smart Lock Hardware, the ESP32-S2 and S3. It explains each feature, how it was done and which technologies were used. It also explains the optimizations done, and solutions found for any problem encountered.

4.3.1 Creating an Initial Proof of Concept

The first step in the development of the Smart Lock Hardware was to learn how to use the ESP-IDF and how to write and flash code for the ESP32-S2. We study and tested some of the example scripts available in the ESP-IDF Github[9]. We started with a simple hello world[10] and then began working with some of the components that we would need to use, such as the integrated RGB LED[11] and a TCP server with a Wi-Fi connection[12].

We created a simple proof of concept with a subset of the planned features using only a TCP server. We based our ESP project on the TCP server example code, and created features to handle the received messages, acting accordingly. We built the security layer for this communication channel, according to the architecture explained in section 3.3, and all the necessary exchanges of credentials for user authorization (section 3.4.6). We wrote code to encrypt and decrypt messages both in AES (CBC mode) and RSA-2048. We based this code on the examples provided in the ESP-IDF GitHub and used the ESP32-S2/S3 hardware-accelerated encryption module as much as possible. Finnaly, develop the unlock on-demand feature, one of the main features proposed for this system, to test its viability. This TCP server was tested using a basic test client created with Python.

The NVS of the ESP32-S2/S3 was used as a persistent storage for the database cache, as explained in (section 3.4.5). The RGB LED example code was adapted to create a simple version of the visual representation of the lock operation (section 3.4.7). This initial proof of concept was used in the First Thesis Report for a simple evaluation. This version of the project is saved in a GitHub Repository [13].

4.3.2 ESP-Touch to configure Wi-Fi and HTTP Client Test

It is necessary to send the WiFi credentials to setup a Wi-Fi connection on the first configuration of the ESP32-S2/S3, therefore, we used the ESP-Touch[14] protocol, created by Espressif Systems. ESP-Touch

[5] https://firebase.google.com/products/auth
[6] https://www.jetbrains.com/pycharm
[7] https://developer.android.com/studio
[8] https://www.figma.com/
[9] https://github.com/espressif/esp-idf/tree/master/examples
[10] https://github.com/espressif/esp-idf/tree/master/examples/get-started/hello_world
[11] https://github.com/espressif/esp-idf/tree/master/examples/get-started/blink
[12] https://github.com/espressif/esp-idf/tree/master/examples/protocols/sockets/tcp_server
[13] https://github.com/bernardocmarques/SmartDoorLock-IoT/releases/tag/Initial_Version
[14] https://www.espressif.com/en/products/software/esp-touch/overview

is a protocol to seamlessly configure Wi-Fi devices connecting to a router, made with the ESP32 in mind.

We decided to use the ESP-Touch protocol and implemented it into our system using as a reference the example provided in the ESP-IDF GitHub. We used the ESP-Touch Android app provided by Espressif Systems to test the code at this phase.

We tested the internet connection on the ESP32-S2/S3 using a simple HTTP client developed in the ESP32-S2/S3, that would be useful in the future to connect to our Service Server API. In this initial test, we used a free notification web service called PushingBox. With this API we managed to create a simple test project with the following behaviour:

- On the first execution of the program, it is necessary to configure the Wi-Fi credentials and the ”deviceID” (required to use the PushingBox API) using the ESP-Touch v2 Application.
- After this initial configuration, the program will immediately send a PushingBox notification to the provided ”deviceID” scenario.
- The following execution will try to use the save parameters and automatically connects to the Wi-Fi and send the notification.
- If it fails to connect to the Wi-Fi or it fails to retrieve the saved credential from the NVS it will revert to the initial behaviour, asking to configure the Wi-Fi again.

There's a demo video of this simple implementation[15], and the code can be found in this GitHub Repository [16]. The ESP-Touch protocol was then integrated with the initial version of the project and adapted to work with our Mobile Client App (section 4.5.1.A).

4.3.3 BLE server for the ESP32-S2 with an external BLE module

To develop the ability to communicate with BLE for short-range communications it was necessary to develop a BLE server, however, the ESP32-S2 doesn't have an internal BLE module, so we had to use an external solution.

The JDY-23 Ultra Low Power Bluetooth BLE Module [18] connects to the ESP32-S2 and communicates with it using the UART protocol. The JDY-23 automatically creates a GATT server (as explained in the section 3.4.3). This GATT server creates a service with the Universal Unique Identifier (UUID) equals to 0000FFE0-0000-1000-8000-00805F9B34FB (0xFFE0 for abbreviation) and a characteristic with the UUID 0000FFE1-0000-1000-8000-00805F9B34FB (0xFFE1 for abbreviation). Everything written in the characteristic with UUID 0xFFE1 is transmitted between the BLE client and the ESP32-S2. For example, when sending a message from the ESP32-S2 to the BLE client:

[15] https://drive.google.com/file/d/1665rDURe3AZMEhT5rfKFV1MRsEDc1Xjn/view
[16] https://github.com/bernardocmarques/ESP-Touch_and_PushingBox_Example

1. The ESP32-S2 writes a message to the UART interface.
2. This message is then automatically written in the GATT server in the characteristic with UUID 0xFFE1.
3. The BLE client is then notified that there is a new message to be read.
4. The client then can read the characteristic to retrieve the message

The opposite process is also valid, the BLE client can send a message to the characteristic with UUID 0xFFE1, and then the JDY-23 module sends via UART the message written in this characteristic to the ESP32-S2.

The code to receive and send data through the BLE server is very simple. To receive messages we just need to wait in a loop, and when a message is received we handle it, just like this:

```
while(1) {

    int len = uart_read_bytes(ECHO_UART_PORT_NUM, data, (BUF_SIZE - 1), 250 /
     portTICK_RATE_MS);   <------ Read data from the UART

    if (len) {
        (...)   <------------------------ If len different than 0,
    }                                     process data and handle it
}
```

Listing 4.1: Code to receive data through the BLE server

To send a message from the ESP32-S2 to the BLE client we created the function shown in listing 4.2:

```
int sendData(char* data) {  <------------ Function to send data
    size_t len = strlen(data);

    const int txBytes = uart_write_bytes(ECHO_UART_PORT_NUM, data, len);
                                  ^
    return txBytes;      Write data to the UART
}
```

Listing 4.2: Code to send data through the BLE server

One of the problems we faced was the maximum size of each message. In GATT servers, the maximum message size is defined by the Maximum Transmission Unit (MTU). In the case of the JDY-23, the MTU size was 128 bytes, so, if we sent a message with more than 128 bytes (or characters),

this message would be split into multiple messages. To solve this, we included a special character at the end of the tansmission, the End-of-Transmission character (EOT). The message is saved in a buffer until the EOT is received, and only handle it after receiving this entire message. With this change, the function to send messages is the following:

```
int sendData(char* data) {
    char EOT = '\4';
    strncat(data, &EOT, 1); <------- Add EOT character to end of data
    size_t len = strlen(data);
    const int txBytes = uart_write_bytes(ECHO_UART_PORT_NUM, data, len);
    return txBytes;
}
```

Listing 4.3: Code to send data through the BLE server with EOT

Another fix we had to implement, was related to a problem with the AT commands. To operate and change settings in the JDY-23 we use special messages called AT commands. To terminate the connection the *"AT+DISC"* message is sent to the server and, if successful, it receives the response *"+DISCONNECT"* from the JDY-23. To prevent any malfunction in the code we had to disregard any message that begins with the character '+' since it was impossible to differentiate an AT command response from a normal message received from the client.

This was a simple fix, however, it created another problem. The messages exchanged between the client and the server were encrypted using RSA or AES and then encoded using Base64. And since the character '+' is part of the base64 charset, we could receive a legit message from the client which started with a '+'. To fix this, if the message started with a '+' we changed it to the '-' character in the client, and then change it back to the '+' character in the server when decoding it.

After implementing the BLE server, we tested it using a BLE client terminal app available in the Play Store, and then with the BLE client developed during the Mobile App implementation (section 4.5.1.B).

4.3.4 HTTPS Client to communicate with the server

Although we already had a simple HTTP client implemented (used to send GET requests to the PushingBox API) we needed to implement a more robust and secure HTTPS client, that would support both GET and POST requests and could verify Secure Sockets Layer (SSL) certificates to securely communicate with the Central Service Server. We used the provided "esp_http_client.h" library to create the HTTP client and the MIT Licensed C library "cJSON"[17] to manage the JSON object used in the POST data and requests response.

[17] https://github.com/DaveGamble/cJSON

In listing 4.4 we show a simple example of a push request with some JSON data using the mentioned libraries.

```
void http_post_example() {
    char resp_buf[500] = {0};
    esp_http_client_config_t config = {(...)};
    esp_http_client_handle_t client = esp_http_client_init(&config);
                                      ^ Creation and initialization of http client with given configurations

    // POST
    char* path = "/example_path";
    char* url = malloc((strlen(base_url) + strlen(path) + 1));
    sprintf(url, "%s%s", base_url, path);

    esp_http_client_set_url(client, url);
                            ^ Creation and initialization of request path

    cJSON* post_data_json  = cJSON_CreateObject();

    cJSON_AddItemToObjectCS(post_data_json, "string", cJSON_CreateString("
string_1"));                                                              <- JSON Object creation
    cJSON_AddItemToObjectCS(post_data_json, "number", cJSON_CreateNumber(1));

    char* post_data = cJSON_PrintUnformatted(post_data_json);

    esp_http_client_set_method(client, HTTP_METHOD_POST);
    esp_http_client_set_header(client, "Content-Type", "application/json");
    esp_http_client_set_post_field(client, data, (int)strlen(data));
                                                                          <- POST configuration

    esp_err_t err = esp_http_client_perform(client); <-- Perform POST request
    if (err == ESP_OK) {
        cJSON* result_json = cJSON_Parse(resp_buf);
        (...)  <---- Handle POST response
    } else {
        // Error  <---- Handle POST error
    }
}
```

Listing 4.4: Example of code used to send a post request.

4.3.5 Invites and Share Access Mechanisms

After developing the TCP and BLE server and securing both communication channels, we continued with the implementation of the main features to operate the smart lock, specifically the invite system. Up to this point, we were testing the system with a single hardcoded user so, we needed to implement the feature to create invites and share door access. We end up developing three different types of invites, each with its purpose.

A – User created invites: These invites are the main type of invite. This invite is created by a user who wants to share his lock with another user. The process to create an invite involves the following steps:

1. The smart lock server receives a request to create a new invite from the Mobile App.
2. This request is validated by the server and it checks if the user has permission to create it. (i.e. the user needs to have a higher or equal permission level to the invite, and can only create invites for the time windows of his authorization)
3. If the invite request is valid the smart lock signs the invite with its RSA private key, and sends it to the Central Service Server, to be created and stored in the database (more in section 4.4.3).
4. After the invite is created by the Central Service Server, the smart lock sends the new invite ID to the client, to be shared and used.
5. This invite can then be redeemed and used to create a new user, with access to the lock and with the permissions specified in the invite.

B – First user creation: Now that we could create invites, it was possible to create new users without manually creating them in the ESP32-S2/S3. However, this did not work in the case of the creation of the first user for each lock. To solve this issue, we develop a special kind of invite, the First User Invite. The user created by this invite is always an admin, and it has all the permissions to control and manage the Smart Lock. To create this invite we need to follow this process:

1. When first connecting to a new Smart Door Lock, the client (Mobile App) sends a specific message to request the creation of this invite, the ”RFI” command (Request First Invite).
2. When the Smart Lock receives the ”RFI” command it checks if there are any users or invites already created for that lock.
3. If not it means that this is the first user, and the request for a First User Invite is valid.

The remaining process to create and redeem this type of invite is the same as if it was a normal invite. The Smart Lock sends the sign request to the Central Server, and it creates the invite, sending the invite ID back to the Smart Lock and then to the client.

C – Multiple phones per user: Up to this point, we were only using one phone per user account, however, we wanted to allow the user to use his account on multiple phones. The problem was that the master key used to authorize the client in the Smart Lock, was saved in the phone, and it was a unique

key for the pair {phone, smart_lock}. To solve this problem we needed to create a new key for each door, in each new phone.

To do so we created a new kind of invite, the User Invite. This kind of invite is a special invite automatically created with the same permissions and locked to the current user. The idea behind this invite was to always have a ”standing by” invite for each user, and automatically redeem this invite when the user tries to use a new phone. This works as follows:

1. When connecting to a Smart Door Lock, the client (Mobile App) will verify if it has a User Invite stored in the database.
2. If not, it will send a specific message to request the creation of this invite, the ”RUI” command (Request User Invite).
3. This invite is automatically created by the Smart Lock, like the other invites, and the client saves its ID for later use.
4. When connecting to a Smart Door Lock from a new phone, the client automatically notices that it doesn't have a master key, and redeems the User Invite, repeating then the process to create a new invite.

4.3.6 BLE server for the ESP32-S3

The ESP32-S3 has an integrated BLE 5.0 module, which would allow to use BLE 5.0, a more updated and secure version of Bluetooth, instead of the BLE 4.2 external module we were using with the ESP32-S2. To perform this change of hardware we implemented a different BLE server. This server follows the same principles as the server used by the ESP32-S2, it also is a GATT server, however, we would need to code it directly, instead of using UART to communicate with a module which had an already working GATT server.

To start this task, we look into the GATT server example[18] provided in the ESP-IDF GitHub. We used this code as a basis and adapted it to our needs, integrating it with the already developed code. However, we realized that this GATT server used the older BLE 4.2 instead of the updated BLE 5.0 version, so it didn't have the security features wanted.

We tried to find an example of a GATT server which used BLE 5.0, but we could not find any, so we looked into the implementation of a more generic BLE 5.0 secure server[19] and change it to function as the GATT server we needed. This process was not straightforward, and we run into some issues using this server both with our previous created Mobile Client and with most of the BLE client terminal

[18] https://github.com/espressif/esp-idf/tree/master/examples/bluetooth/bluedroid/ble/gatt_server
[19] https://github.com/espressif/esp-idf/tree/master/examples/bluetooth/bluedroid/ble_50/ble50_security_server

apps available in the Play Store (more about this issue in section 4.5.1.B). However, after solving this problems, we end up with a working BLE 5.0 server that worked just like the previous BLE 4.2 server, so both could be used by the same client. This server also creates a service with the UUID equals to 0000FFE0-0000-1000-8000-00805F9B34FB (0xFFE0 for abbreviation), however it created two different characteristics with the UUID 0000FFE1-0000-1000-8000-00805F9B34FB (0xFFE1 for abbreviation) and 0000FFE2-0000-1000-8000-00805F9B34FB (0xFFE2 for abbreviation). The characteristic with UUID 0xFFE1 is used to receive data in the ESP32-S3 sent from the client, and the characteristic with UUID 0xFFE2 is used to send data from the ESP32-S3 to the client.

4.3.7 Creation and exchange of RSA keys

At this point, the RSA keys were hard-coded both in the EPS32-S2/S3 code and the client code. This wasn't a valid option, so we had to implement a way to configure the keys in the ESP32-S2/S3 and then safely share the public key, in a way that it could be trusted.

To do so we created a private Certificate Authority (CA) with a self-sign certificate. This CA was used to sign the RSA public keys used by each smart lock, creating an X.509 certificate. That way, we could install our CA in the mobile app client, and it could verify each Smart Lock public key certificate.

This is an example of how to use openssl[20] in Linux to create an RSA key pair, and an X.509 certificate sign with our CA, called "rootCA".

```
1    $ openssl genrsa -out priv.pem 2048
2    $ openssl req -new -key priv.pem -out cert.csr
3    $ openssl X509 -req -in cert.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -out
         cert.crt -days 500 -sha256
```

Listing 4.5: Example of commands to create RSA keys and X.509 certificate

After running these commands we have a file called "priv.pem" which contains the private key for the Smart Lock, and the "cert.crt" file which is the X.509 certificate signed with our CA from which it is possible to retrieve the Smart Lock public key. These files are created by the developer of the smart lock and loaded to the devices upon development. The RSA private key and the X.509 certificate ("priv.pem" and"cert.crt" files) need to be stored in the ESP32-S2/S3, to do so, we used the SPIFFS Filesystem[21], with this, we can store files in the ESP-IDF project files, and when flashing the code the files are saved in the ESP32-S2/S3. To retrieve the "priv.key" file in execution we can use the code in the listing 4.6.

Then, in the client, we just need to install our CA, to verify the public key certificates. These X.509 certificates are stored in the database by the smart lock upon the first configuration and can be retrieved

[20] https://www.openssl.org/
[21] https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-reference/storage/spiffs.html

```
void init_rsa_key() {

    esp_vfs_spiffs_conf_t conf = {
                .base_path = "/spiffs",
                .partition_label = NULL,
                .max_files = 5,
                .format_if_mount_failed = false
        };

    esp_err_t ret = esp_vfs_spiffs_register(&conf);

    if (ret != ESP_OK) return;


    size_t total = 0, used = 0;
    ret = esp_spiffs_info(NULL, &total, &used);
    if (ret != ESP_OK) {
        ESP_LOGE(TAG_MAIN, "Failed to get SPIFFS partition information (%s)",
     esp_err_to_name(ret));
    } else {
        ESP_LOGI(TAG_MAIN, "Partition size: total: %d, used: %d", (int)total,
     (int)used);
    }


    // Open for reading priv.key
    FILE* f_key = fopen("/spiffs/priv.key", "r");
    if (f_key == NULL) {
        ESP_LOGE(TAG_MAIN, "Failed to open priv.key");
        return;
    }
    size_t buf_size = 2048;
    char* buf = malloc(buf_size);
    memset(buf, 0, buf_size);
    fread(buf, 1, buf_size, f_key);
    fclose(f_key);


    set_rsa_private_key(buf);
}
```

SPIFFS initialization

Retrieving information about SPIFFS partition

Reading priv.key file

Listing 4.6: Example of code to retrieve files from the SPIFFS Filesystem

from the Central Service Server for use in the mobile app client.

4.4 Smart Lock Service Server

4.4.1 RESTful flask API - First steps

A – Basic RESTful API: The first step in the server implementation was to create a simple RESTful API. To accomplish this we used Flask framework[22] and Python 3.10[23]. This was a really basic API, with only the necessary endpoints to test GET and POST requests, without any functionality for the system.

B – RSA Util functions: After creating this basic API we start developing utility functions to help us with the cryptographic operations. We only needed RSA which would be used to validate the server RSA signatures. We used the pycryptodome[24] python library to develop the required RSA-related functions. With this library, we created some basic functions that would be useful in the future, for instance, the function to verify an RSA signature in base64 (listing 4.7).

```
    def is_signature_valid(self, msg, signature_b64):
        signature = base64.b64decode(signature_b64)

        h = SHA256.new(msg.encode())
        verifier = pss.new(self.key)
        try:
            verifier.verify(h, signature)
            return True
        except (ValueError, TypeError):
            return False
```

Listing 4.7: Function to verify a RSA signature

C – Firebase integration: One of the main purposes of the Service Server is to serve as the interface to communicate with the Firebase Realtime Database. For that reason, we created a utility class to help use the Firebase database. We used the official library firebase-admin[25] and with this library we developed functions to easily get, set and delete data from Firebase database (listing 4.8).

With this initial development, we had a RESTful API capable of receiving and handling GET and POST requests, doing RSA operations, namely, verifying an RSA signature, and it could read and write to a Firebase database.

[22] https://flask.palletsprojects.com/en/2.2.x/
[23] https://www.python.org/downloads/release/python-3100/
[24] https://pypi.org/project/cryptography/
[25] https://pypi.org/project/firebase-admin/

```
def get_data(self, path):
    ref = self.db.reference(path)
    return ref.get()

def set_data(self, path, data):
    ref = self.db.reference(path)
    ref.update(data)
    return True

def delete_key(self, path):
    ref = self.db.reference(path)
    ref.delete()
    return True
```

Listing 4.8: Helper functions to use Firebase

4.4.2 Developing the basic endpoints

Most of the necessary endpoints and functions in the server are related to operations in the database, so they were somewhat easy to develop.

A – Endpoints related with the user account database:

- **GET** */get-user-locks?id_token={id_token}* - This GET request is used to get the list of locks that a given user have access. The only parameter required is the *id_token*. This parameter is used to identify and authorize the user in the Firebase database.
- **POST** */set-user-locks* - This POST request is used to register a given smart lock in the user's database of locks. This request requires a *id_token* parameter to identify and authorize the user, and a *lock* parameter, which is a JSON object with the smart lock information to be registered.
- **POST** */delete-user-locks* - This POST request is used to delete a given smart lock from the user's database of locks. This request requires a *id_token* parameter to identity and authorize the user, and a *lock_id* parameter which identifies the lock to be deleted.
- **POST** */register-phone-id* - This POST request is used to register a phone id when a user login into a new phone. This request requires a *id_token* parameter to identify and authorize the user, and a *phone_id* parameter which is the phone id to be registered in the user account.

B – Endpoints related with smart locks database:

- **POST** */register-door-lock* - This POST request is used to register a lock in the database when it first boot. This request requires 3 parameters, a *MAC* and *BLE* addresses of the lock and the X.509 *certificate* which contains the public key of the smart lock.

- **GET** */check-lock-registration-status?MAC={MAC}* - This GET request is used to check the status of registration of a given smart lock. This request requires only the smart lock *MAC* address as a parameter. It returns the registration status of the wanted smart lock, which can be ”not registered”, ”registered” and ”registered and with authorizations”.
- **GET** */get-door-certificate?id_token={id_token}&MAC={MAC}* - This GET request is used to obtain the X.509 certificate for a given smart lock. This request requires a *id_token* parameter to identify and authorize the user, and a *MAC* address for the wanted smart lock.
- **POST** */request-authorization* - This POST request is used by the smart lock to request a given authorization from the database. This request requires a *smart_lock_MAC* parameter and a *phone_id* parameter, these parameters create the pair {phone, smart_lock} that identifies the requested authorization. This request is signed with the smart lock private key (**signature** parameter) to check the authenticity of the request.

There are other general endpoints used to get publicly available information from the database or server storage, like images or icons.

4.4.3 Create and redeem invites

The server is responsible for creating and redeeming the invites used to share lock access, as explained in section 4.3.5. The smart lock can request a invite creation using a **POST** request to */register-invite*. The creation process is the following:

1. The server receives a signed invite request sent by the smart lock.
2. The invite request signature is validated using the public key of the smart lock.
3. If valid, the server proceeds to create and register the invite in the database
4. If everything was successful the server sends the new invite id to the smart lock.

The invite request (**POST** */register-invite*), requires a **data** parameter, which contains all the information necessary to create the invite and a **signature** parameter used to verify the authenticity of the request.

To redeem the invite, the mobile app can send a **POST** request to */redeem-invite*, sending as parameters the **id_token** (to identity and authorize the user) the **invite_id**, the **phone_id** (that identifies the phone which is requesting the authorization creation) and a **master_key_encrypted_lock** (the master key for the authorization, encrypted with the smart_lock public key for security reasons). The redeem process follows these steps:

1. The server receives a request to redeem an invite.
2. It checks if the invite with the provided id exists and if the user can redeem it (e.g. if it is not user locked to another user).
3. If both conditions are valid, the server proceeds to create and register, in the database, the authorization for the pair {phone, smart_lock}, with the invite information and the provided master_key.
4. If everything was successful the server responds with a success message.

A – User Invites Special Case: Even though the creation and redemption of the User Invites are almost the same as the other invites, the server has some different endpoints to perform operations regarding these invites:

- **POST** */save-user-invite* - This POST request is used to save the user invite in the user database. This request requires 3 parameters. The *id_token* parameter to identity and authorize the user, the *lock_id* parameter to identify the lock for which the invite is, and the **invite_id** to be saved.
- **POST** */redeem-user-invite* - This POST request is used to redeem the user's saved invite. This request requires 4 parameters. The *id_token* parameter to identity and authorize the user, the **phone_id** and **lock_id** parameters that create the pair {phone, smart_lock} from which the authorization will be created, and and the **master_key_encrypted_lock**, the master key for the authorization, (encrypted with the smart_lock public key for security reasons).
- **GET** */check-user-invite?id_token={id_token}&lock_id={lock_id}* - This GET request is used to check if a given user have a user invite for the given smart lock. This request requires a *id_token* parameter to identity and authorize the user, and a *lock_id* to identity the smart lock.

4.4.4 Firebase Realtime Database and Authentication

Firebase services are used to manage the database and the authentication in our system, as explained in sections 3.5.2 and 3.5.3. In this section, we will explain how we will use the necessary Firebase products, namely the Realtime Database and the Authentication.

The Firebase Realtime Database is used to store all the necessary information for the correct working of the system, this information is accessible by the Smart Door Locks and the Clients App through the Central Service Server. In the Realtime Database the information is stored in a JSON tree structure and the data is stored as JSON objects and, unlike a SQL database, there are no tables or records. When we add data to the JSON tree, it becomes a node in the existing JSON structure with an associated key. In figure 4.1 we can see the structure of the information stored in the database about each Smart Door Lock.

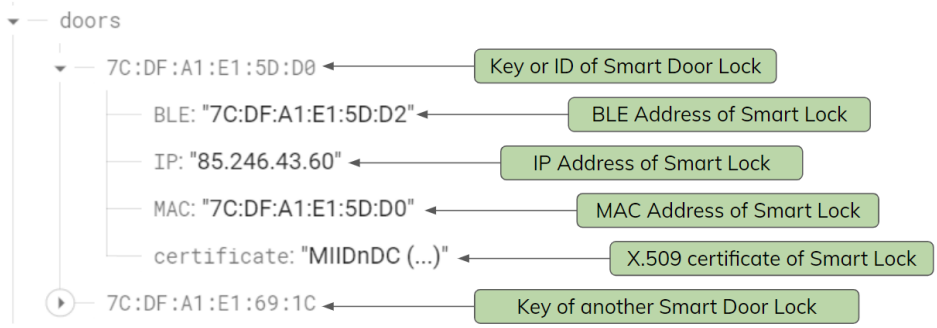


Figure 4.1: Example of information about the each Smart Door Lock

This database will be used to store information about:

- **Authorizations** - Object with the information to authorize a phone in a smart lock.
- **Doors** - Object with all the necessary information to interact with a smart door lock. Contains a *BLE* address, *IP* address, *MAC* address and X.509 *certificate*.
- **Invites** - Objects with the information of the invites used to create a new authorization.
- **Users** - Objects containing the information of each user, like a list of smart lock doors and customizable information.

Regarding Firebase Authentication, it will be used to Authenticate and manage the users in our system. The login and creation of users will be done in our Mobile App using the Firebase android library (more in 4.5.2). This service automatically creates Unique IDs for each user, that will be used to identify the user in the Database, and it also allows us to create *id_tokens* to authenticate and authorize the user when interacting with our Service Server.

4.4.5 Gateway to communicate remotely with a Smart Door Lock

The server is used as a middleman in the communication between the mobile app client and the smart door lock, as already explained in section 3.5.1. The client can send a command to a specific API endpoint in the Service Server, and this command will be forward to the respective Smart Door Lock, using a TCP connection between the Service Server and the Smart Lock. To implement this feature we developed a simple python TCP client in the Service Server. This client can establish a connection to a given smart lock and then send commands and receive responses.

To remotely control a smart lock, the client can send a **POST** request to */remote-connection*. This POST request requires 4 parameters. The *id_token* parameter to identity and authorize the user, the *lock_id* parameter which identifies the lock to whom send the messages, the *msg* parameter, which

is the message or command that the client wants to send to the smart lock, and finally a boolean parameter *close*, that indicates if this is the last message in the communication and if the connection can be terminated. In this request, the parameter *msg* is the same command the client would send to the smart lock in the BLE communication, using the protocol explained in section 3.4.2. This message or command is then sent to the Smart Lock via TCP and its response is then forward to the client via the HTTP response.

4.4.6 Central Service Server Deployment in AWS

This Service Server consists of a python application running a Flask RESTful API. This application was deployed in a Ubuntu Linux Machine in AWS. The AWS EC2 machine used was a t2.micro[26] running Ubuntu 22.04 LTS. To deploy this server we loosely followed a guide untitled "How To Serve Flask Applications with Gunicorn and Nginx on Ubuntu 22.04" [27] and used the following tools - Gunicorn [28] to run the server and Nginx [29] to act as a front-end reverse proxy. We also used Let's Encrypt [30] Certificate Authority and the tool Certbot [31] to create a SSL/TLS certificate to secure the communication to this server.

4.5 Smart Lock Mobile App

4.5.1 Android Mobile App - Initial Steps

When we first started developing the Android Client App we focused mainly on the basic functionality of the app, using a simple UI without much consideration of the design. We started by implementing the main features, like the ability to use the ESP-Touch protocol to configure the Wi-Fi credentials in the ESP32-S2/3 and the functionality to send commands through BLE to the Smart Locks. In the following subsection, we will take a look at these features, and how they were made.

4.5.1.A Integration of the ESP-Touch protocol

We started creating our application using as a template the code provided for the ESP-Touch Android app[32]. With this initial template, we developed the feature that gave us the ability to send the Wi-Fi credential when we setup a new Smart Lock. To send the Wi-Fi credential we just need to create a

[26] https://aws.amazon.com/ec2/instance-types/t2/
[27] https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-gunicorn-and-nginx-on-ubuntu-22-04
[28] https://gunicorn.org/
[29] https://www.nginx.com/
[30] https://letsencrypt.org/
[31] https://certbot.eff.org/
[32] https://github.com/EspressifApp/EsptouchForAndroid/tree/master/esptouch-v2

EspProvisioningRequest with the SSID and password of the Wi-Fi network, and then start provisioning this request. In listing 4.9 we can see an example of the function used to transmit the Wi-Fi credentials.

```
void sendWifiCredentials() {

    EspProvisioningRequest req = new EspProvisioningRequest.Builder(context)
            .setSSID(ssid)
            .setPassword(password)
            .build();

    EspProvisioningListener listener = new EspProvisioningListener() {

        @Override
        public void onStart() {
            Log.d(TAG, "Started setup!");
        }

        @Override
        public void onResponse(EspProvisioningResult result) {
            Log.d(TAG, "Done setup!");
            provisioner.stopProvisioning();
        }

        @Override
        public void onStop() {
            Log.d(TAG, "Stopped setup!");
            provisioner.close();
            connectBtn.setEnabled(true);
        }

        @Override
        public void onError(Exception e) {
            Log.d(TAG, "Error setting up!");
            e.printStackTrace();
        }

    };

    provisioner.startProvisioning(req, listener);
}
```

Create request with Wi-Fi credentials

Called when provisioning started

Called when credentials received with success

Called when provisioning stopped

Called when provisioning encounters an error

<-- Start broadcasting Wi-Fi credentials

Listing 4.9: Example of function to send Wi-Fi credentials.

When using this feature, the application will automatically obtain the SSID and BSSID of the connected Wi-Fi network. Then the user just needs to provide the password for that Wi-Fi network and press connect. The mobile app starts broadcasting the Wi-Fi credential using the ESP-Touch protocol. When the Smart Lock gets the credential, the app will display a success message, and the process is complete.

4.5.1.B BLE communication

The second feature developed on the client mobile app was the creation of a BLE client to establish a BLE communication with the Smart Locks. To start implementing this feature we looked at the Android BLE communications documentation [19], studied code from multiple BLE terminal apps, and used as a basis some of the code for a BLE GATT Client[33], provided in the android GitHub, mainly from the BluetoothLeService.java[34] file.

The BluetoothLeService.java have all the necessary code to manage BLE connection, find and subscribe to GATT services and read and write to GATT characteristics. We used and modified this file to have some generic methods that could be used to simplify tasks like scanning for BLE devices or even sending and receiving messages and commands.

We also created classes and methods that help us manage the BLE connection and our system-specific message protocol in a more abstract manner. For instance, we created functions to send a given command with the necessary encryption and previous authentication messages, and then revived and handle the response, all this in a single function call like:

```
bleManager.sendCommandWithAuthentication(this, cmd, callback);
```

To do so, we created multiple Util files, to help us with the AES and RSA encryption and decryption and even a BLE Message class to parse and manage BLE messages and automatically validate nonces and timestamps sent in the messages.

Although much of the code was based on the example provided in the Android BLE documentation, multiple settings and configurations were changed to work with the Smart Locks. One of the most important changes was the variable which controls the scan settings. Although the default variables were working for the ESP32-S2, which uses BLE 4.2, when testing the BLE 5.0 and the ESP32-S3, the Mobile App couldn't find that device. To solve this problem it was changed a flag in the Scan Setting variable, that disabled the Legacy Mode. With the legacy mode enabled, only Legacy advertisements were returned in scan results, which include advertisements as specified by the Bluetooth core specification 4.2 and below (as shown in listing 4.10).

4.5.2 Login and Account Creation - Firebase Authentication

To manage the user creation and logins in our Mobile App we use the Firebase Authentication service. This service allows us to create accounts using both email and password credentials, and Sign-In with Google. This feature was developed using the official Firebase Authentication library for Android[35]. Using this library we implemented functions to help us create accounts and log in with email and password,

[33] https://github.com/android/connectivity-samples/tree/main/BluetoothLeGatt

[34] https://github.com/android/connectivity-samples/blob/main/BluetoothLeGatt/Application/src/main/java/com/example/android/bluetoothlegatt/BluetoothLeService.java

[35] https://firebase.google.com/docs/auth/android/start

```
ScanSettings scanSettings = new ScanSettings.Builder()
        .setPhy(ScanSettings.PHY_LE_ALL_SUPPORTED)
        .setLegacy(false)
        .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
        .setCallbackType(ScanSettings.CALLBACK_TYPE_ALL_MATCHES)
        .setMatchMode(ScanSettings.MATCH_MODE_AGGRESSIVE)
        .setNumOfMatches(ScanSettings.MATCH_NUM_ONE_ADVERTISEMENT)
        .setReportDelay(0L)
        .build();
```

Listing 4.10: Scan Settings configurations

```
mAuth.signInWithEmailAndPassword(email, password) <--  Function to request the login with Email and Password
    .addOnCompleteListener(this, task -> {
        if (task.isSuccessful()) {

            FirebaseUser user = mAuth.getCurrentUser();        // Login Successful

            (...) <------  Update UI and application with the new logged user

        } else {

            Exception e = task.getException();                 // Failed to login with Email and Password

            if (e instanceof FirebaseAuthException) {
                String errCode = ((FirebaseAuthException) e).getErrorCode();

                (...) <------  Handle error and show error message

            }
        }
    });
```

Listing 4.11: Firebase function to login with Email and Password

and to use the ”Sign-In with Google”.

To create an account the user just needs to input an email address and a password, and the application automatically creates an account using the Firebase Authentication library. For each account, the Firebase automatically creates a user Unique Identifier (UID), that is used to identify the user in the database.

To login with an email and password, the user just needs to input these credentials, and the application will login to the account using the Firebase Authentication library. In listing 4.11 it is possible to see an example of the function used to login with email and password.

Finally, the ”Sign-In with Google” feature, is implemented almost entirely by the Firebase Authentication library. To use it, the user just needs to click a button in the login activity and then choose a Google

Account in a Dialog created by the system. If there is no account created with the selected Google Account, the Firebase Authentication will automatically create one and login into it.

4.5.3 Integration with Service Server and Database

To access the database, the mobile app needs to connect and communicate with our Central Service Server through the RESTful API, as explained in section 4.4.1. It was implemented with an HTTP Client with multiple functions to help and create an abstraction layer in the communication with the server endpoints and access the database information (section 4.4.2). All these functions are used to write or read information from the database, and 4 functions examples are provided below:

- **getUserLocks(OnTaskCompleted<ArrayList<Lock>> callback)** - A function used to obtain the list of locks that a given user have access. This function is used to communicate with the **GET** */get-user-locks* endpoint.
- **setUserLock(Lock lock, OnTaskCompleted<Boolean> callback)** - A function used to register a given smart lock in the user's database of locks. This function requires a *lock* as argument and it is used to communicate with the **POST** */set-user-locks* endpoint.
- **deleteUserLock(String lockId, OnTaskCompleted<Boolean> callback)** - A function used to delete a given smart lock in the user's database of locks. This function requires a *lock* as argument and it is used to communicate with the **POST** */delete-user-locks* endpoint.
- **registerPhoneId(Context context, OnTaskCompleted<Boolean> callback)** - A funtion used to register a phone id when a user login into a new phone. This function is used to communicate with the **POST** */register-phone-id* endpoint.

4.5.4 Control the Smart Lock

One of the main purposes of the Mobile App Client is the control the Smart Door Lock, unlocking it, locking it and changing settings. The Mobile App can control the Smart Lock in multiple different ways, it can be controlled On Demand, both closely via BLE and remotely via an internet connection, and also using an advanced proximity control feature. In this section, we explain the different ways we can control smart locks, and how these features were developed.

4.5.4.A Control the Smart Lock On Demand

A – Close Range Control: The easiest way to control the Smart Lock is to manually use the Mobile App to lock and unlock it. To do so, the user needs to select a given smart lock and then use the lock

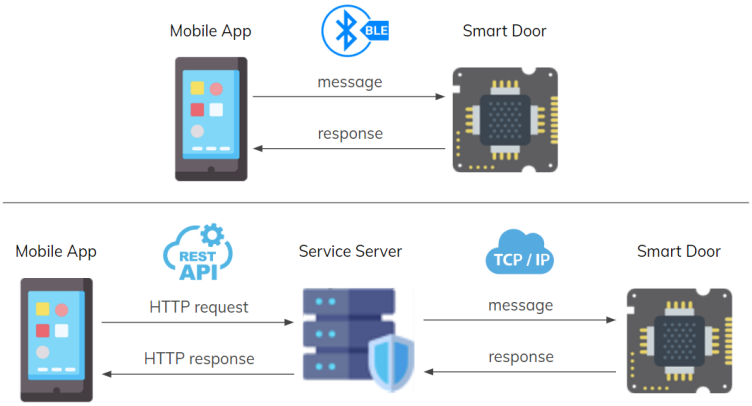


Figure 4.2: Comparison between the BLE communication and the remote communication

slider to lock or unlock the Smart Lock. If the user is a close range to the Smart Lock, this feature uses the BLE connection (explained in section 4.5.1.B) and the message protocol (explained in section 3.4.2) to directly control the smart lock.

When the user opens the activity with the lock controls, the Mobile App sends a **RDS** message to get the Smart Lock status, and when the user tries to lock or unlock the Smart Lock, it sends a **RLD** or a **RUD** message, respectively. A video demo of this feature is available at this link: `https://drive.google.com/file/d/1xswQn68peHChJIKZNKGnuyr4PtUpaPGB/view`.

B – Remote Control: When the user tries to control the Smart Lock manually the Mobile App will firstly try to connect via BLE. However, if it couldn't establish a BLE connection the Mobile App will start using the Remote Control feature. This feature uses the Service Server to communicate remotely with the Smart Lock. To send a message remotely, the Mobile App creates and encrypts the message in the same way as in the BLE connection. Then, it does a **POST** request to the */remote-connection* endpoint. In this request, the client will send, between others, the message or command for the Smart Lock, and it will receive the message response in the request response, as explained in section 4.4.5. In figure 4.2 is a schematic comparison between both communication and control methods.

From the user's point of view, this feature works in the same way as the BLE connection. On the page with the Smart Lock controls, the only difference is the indication that the app is connected remotely to the Smart Lock. A video demo of this feature is available at this link: `https://drive.google.com/file/d/1YWfGigeSVVBmq66ZDp2M_iefpfnkk2I4/view`.

4.5.4.B Proximity control of the Smart Lock

The Proximity Control feature takes advantage of the GPS localization of the user's phone and BLE of the Smart Lock to automatically unlock the Smart Lock when the user is near it and lock it when the user

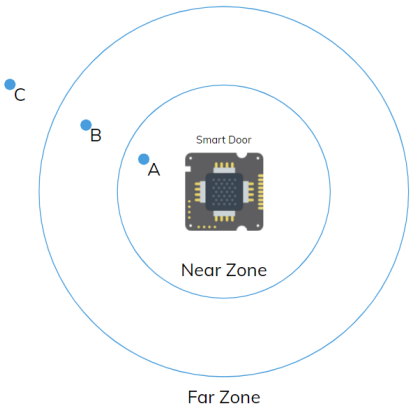


Figure 4.3: Example of the Proximity Control feature functionality

distances himself from the Smart Lock. In this feature, the user can individually activate, for each lock, the ability to unlock on proximity, and the ability to lock when far.

To use this feature the user can define 2 radius distance from the Smart Lock's location. This 2 distances create 2 different zones, a "Near Zone" and a "Far Zone", as shown in the figure 4.3. The lock will automatically unlock if the user enters the "Near Zone" and automatically unlock if the user enters the "Far Zone". For instance, in figure 4.3, if the user moves from point B to point A the Smart Lock will automatically unlock, on the other hand, if the user moves from point B to point C, the Smart Lock will automatically lock.

The initial idea for this feature was to periodically scan for BLE devices and check the BLE signal strength to see if the user was near a Smart Lock. However, periodically scanning for BLE devices is a battery-intensive task, so we end up choosing a different approach based on the GPS location of the user's phone. Even though using the GPS also consumes high amounts of battery, if done correctly and accordingly to Google's recommendations it could have a low impact on the battery life of the phone.

To implement this feature we created a Foreground service[36], that allows us to run code in the background even if the user is not using the app. With this service, the Mobile App periodically gets the phone location and checks if the user is near or far from each lock. To obtain the GPS location we used a FusedLocationProviderClient[37] which is configured to check the phone location with Priority Low Power, a trade-off that favours low power usage at the possible expense of location accuracy. With the Priority Low Power option, the app will reuse locations recently obtained by other apps or the system, preventing unnecessary uses of the GPS. When the app gets a new location update, it will check if the user entered the "Near Zone" of a Smart Lock, and if so, it will automatically, and in the background, connect to it via BLE and send the necessary messages to unlock it. On the other hand, if the user

[36] https://developer.android.com/guide/components/foreground-services
[37] https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderClient

enters the ”Far Zone” of a given lock, the app will remotely send, through the internet, the necessary messages to lock the Smart Lock.

In the following video demo it is possible to see this feature working with 2 different locks, the first lock with both unlock and lock on proximity features enabled, and the second one with only the feature to unlock on proximity enabled. Video demo link: https://drive.google.com/file/d/1q00uRN0ewzluKyNU6QcOA-hY11fVanjQ/view

4.5.5 Invites and Shared Access

Another important feature of the Mobile Client App is to allow the user to invite other users to access his Smart Lock. Through the app, the user can create invites and share them with new users. In this section, we explain how the user can create new invites and how to redeem an invite creating a new access. We also explain how the Mobile App automatically uses the special kind of invites to perform the first user creation and to allow the user to have multiple phones with the same account.

A – New Invite Creation: Using the Mobile App the user can create a new invite for his Smart Locks. To do so, the user just needs to select the wanted Smart Lock, and then click on the option to share. The user will have to input the option for the invite, (the user type, the validity of the new user and other necessary options for the selected user type). After the invite creation, the user can share the invite in multiple ways, for example, via email or SMS.

To develop this feature we created a function that interact with the existing BLE client functions (explained in section 4.5.1.B) to communicate with the Smart Lock and used the message protocol, as explained in section 4.3.5, to request the creation of the invite, with the **”RNI”** command. The invite creation can also be used remotely, using the Remote Control feature (explained in section 4.5.4.A). After the creation of the invite is done by the Smart Lock and the Service Server, the Mobile App receives the invite code, which is then converted by the app into a QR code and a link, for easy sharing. A video demo of the invite creation is available at this link: https://drive.google.com/file/d/1uuIrM1NLsU2mvxuqIxikw4ka3BIU8waz/view.

B – Redeem invites and create new Smart Lock access: To redeem an invite the user needs to create a new account or login into an existing one. Then he has multiple options to redeem the code:

- Manually inputting the invite code (a long base64 string) into the Mobile App,
- Scan the provided QR code with the Mobile App,
- Or, click on a link with the invite code, that automatically redeems the invite.

Finally, after redeeming the code with one of these options, the user will be asked to create a new Smart Lock, choosing a name and an icon to identify it. This feature simply uses the Service Server API to redeem a new invite, sending a **POST** request to the */redeem-invite* endpoint, as explained in section 4.4.3. A video demo of a user redeeming an invite is available at this link: https://drive.google.com/file/d/1N1wTNjtwqLN6j-9kjChCP8jmhns6Ym_T/view

C – Special Case - Creating First User: The creation of the First User for a given lock is a special case, when first configuring a Smart Lock, the user has 2 options to configure it:

- Scan for it using the BLE (only available in the ESP32-S3 version)
- Or, scan a QR code provided with the Smart Lock

The information scanned by the Mobile App is then used to begin the WiFi configuration process with the ESP-Touch protocol (explained in section 4.3.2), and then send a **"RFI"** message to the Smart Lock, and ask for the creation of the First User invite, as explained in section 4.3.5. This invite is then automatically redeemed by the Mobile App, and the user is asked to create a new smart lock with a name and an icon to identify it. A video demo with the creation of the first user with BLE Scan is available in https://drive.google.com/file/d/16NmpL2mWHWOg7kyWIylPbtH83_-V8DFe/view.

D – Special Case - User Invites: The User Invite is used to automatically create authorizations for new phones when a user already have access to a given Smart Lock on another phone, as explained in section 4.3.5. The creation of this type of invite is automatically managed by the Mobile App, without the need for any interaction from the user. When the user uses a phone with access to a given Smart Lock, the Mobile App automatically checks if there is a valid User Invite for that Smart Lock (using a **GET** request to the */check-user-invite* endpoint), and if not, it automatically requests the creation of a new User Invite, with the **"RUI"** message, and then instantly saves it in the database using a **POST** request to the */save-user-invite* endpoint.

On the other hand, if the user tries to access a Smart Lock with a phone without a valid authorization, the Mobile App automatically redeems the saved user invite by doing a **POST** request to the */redeem-user-invite* endpoint, creating a new authorization for that pair {phone, smart_lock} and immediately establishing the connection with the new authorization. A video demo of this feature is available in https://drive.google.com/file/d/1NkWIipAp-JaHSm1_m5TrpcSrNopK3VKG/view

4.5.6 CA and Validation of the RSA Keys

The Mobile App uses X.509 Certificates signed by our CA to validate and obtain the RSA public key of the Smart Locks, as explained in section 4.3.7. The use of these certificates with a trusted CA, allows

the mobile app to know that it is using the real RSA public key of each Smart Lock, and not a fake key sent by an imposter.

To read and validate the X.509 Certificates it was necessary to configure our CA as a trusted source in the Mobile App. This CA is then used by the Mobile App to validate the X.509 Certificates stored in the Service Server database. The code used to validate the X.509 Certificate is shown in listing the 4.12. This function reads our CA from the App resources (”R.raw.my_ca”) and registers it as a trusted anchor. Then using this trusted anchor the function will validate the X.509 Certificate received as an argument, throwing an error if not valid.

```
public static void validateCertificate(X509Certificate c1, Context context)
    throws Exception {
    try {

        InputStream  issuerCertInoutStream = context.getResources().
openRawResource(R.raw.my_ca);
        X509Certificate issuerCert = getCertFromFile(issuerCertInoutStream);
        TrustAnchor anchor = new TrustAnchor(issuerCert, null);
        Set<TrustAnchor> anchors = Collections.singleton(anchor);

        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        List<Certificate> list = Collections.singletonList(c1);
        CertPath path = cf.generateCertPath(list);
        PKIXParameters params = new PKIXParameters(anchors);
        params.setRevocationEnabled(false);
        CertPathValidator validator = CertPathValidator.getInstance("PKIX");
        PKIXCertPathValidatorResult result = (PKIXCertPathValidatorResult)
validator.validate(path, params);

    } catch (Exception e) {
        Log.e(TAG,"EXCEPTION (Certificate not valid) " + e.getMessage());
        throw e; <---- Throw exception if not valid. If valid return void without any exception
    }
}
```

Configuration of our trusted CA

Validation of Smart Lock X.509 Certificate

Listing 4.12: Android code used to validate X.509 Certificates

5

Evaluation

Contents

The SDL system was evaluated to verify if it meets the requirements and features previously described. This evaluation was performed in five different levels, 1) an evaluation of the **Hardware Platform and Peripherals**, 2) followed by the evaluation of the **Service Server** and 3) the **Mobile App**. Finally, 4) we evaluated the performance of the **Communication Between Components** and 5) did a final **System Demonstration**.

5.1 Hardware Platform and Peripherals

The Smart Door Lock hardware consists of the main platform the ESP32-S2/S3 and its peripherals. The ESP32-S2/S3 runs the Firmware that controls the Smart Lock and communicates with its peripherals. It is important to test and validate if each of these components is working as intended and show if the system as a whole is working correctly.

This section is dedicated to the details of how the ESP32-S2/3 platform and each peripheral were initialised and configured to work as intended in our system. It also show how we tested and validate the working status of the platform and each peripheral, like the ESP32-S2/3 itself, the BLE Modules (both internal and external) and the GPIO RGB Led.

5.1.1 Hardware Platform - ESP32-S2/3

The ESP32-S2/3 is the main hardware in our SDL system, since all of the peripherals necessary to control the Smart Lock are connected to the ESP32-S2/3. To develop the firmware for the ESP32-S2/3 we used the ESP-IDF v4.4. This IDF offers us a set of tools used to: a) build and flash code for the ESP32-S2/3; b) and tools to monitor the executions of the firmware. The ESP-IDF communicates with the ESP32-S2/3 using a USB to UART protocol, via the COM ports in Windows.

A – Build and Flash: To build and flash code to the ESP32-S2/3 we can use the "*idf.py flash -p [COM_PORT]*" command, specifying the **COM_PORT** used to communicate with the ESP32-S2/3. This command compiles the C code, outputting to the console the compiling process, showing any warnings or errors that might occur. Then it flashes the firmware into the ESP32-S2/3, showing the flashing process in the console. Figure 5.1 shows the end of the flash process output, indicating that the firmware was flashed with success.

B – Monitor execution: To monitor the execution of the firmware running in the ESP32-S2/3 we used the *idf.py monitor -p [COM_PORT]*" command. The monitor command establishes a connection between the computer and the ESP32-S2/3, outputting the executing status of its firmware, and all the logs printed by the firmware code. Using the monitor we were able to debug the firmware code using

Figure 5.1: Output of ESP-IDF flash

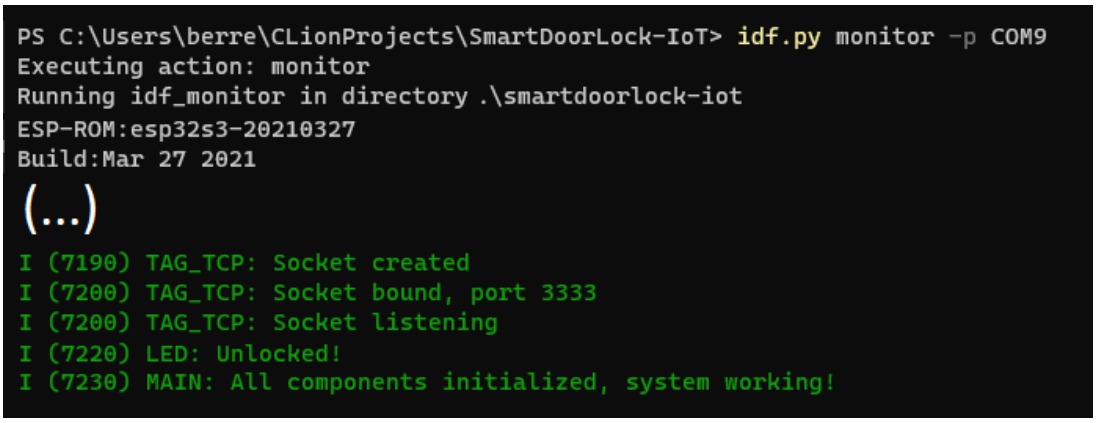


Figure 5.2: Output of ESP-IDF monitor

the output logs, and also verify the correct initialization and configuration of each peripheral. Figure 5.2 shows the initialization of the ESP32-S2/3 firmware obtained using the monitor command, allowing us to confirm that the system is running correctly and all components are initialized, as it is possible to see by the log message ”All components initialized, system working!”.

C – ESP-IDF Configuration Menu: Given the complexity the MCU systems, it is normal to have some middleware used to help in the initialization, configuration and use of the peripherals. In the case of the ESP32-S2/S3, we can use the ESP-IDF Configuration Menu to help initialize and configure the necessary middleware to use the wanted peripherals.

To configure the ESP-IDF we used the ”*idf.py menuconfig*” command. This command is used to configure the current project files and enable, disable or configure each component necessary for the project, for instance, enabling the BLE and WiFi components. Figure 5.3 is a screenshot of the Menu used to configure the ESP-IDF project, accessed through the menuconfig command. It is possible to see multiple options and sub-menus used to configure the ESP32, for example, the ”Security Features” menu, where we can configure the security of the device, or the ”Component config” where we can enable or disable each component.

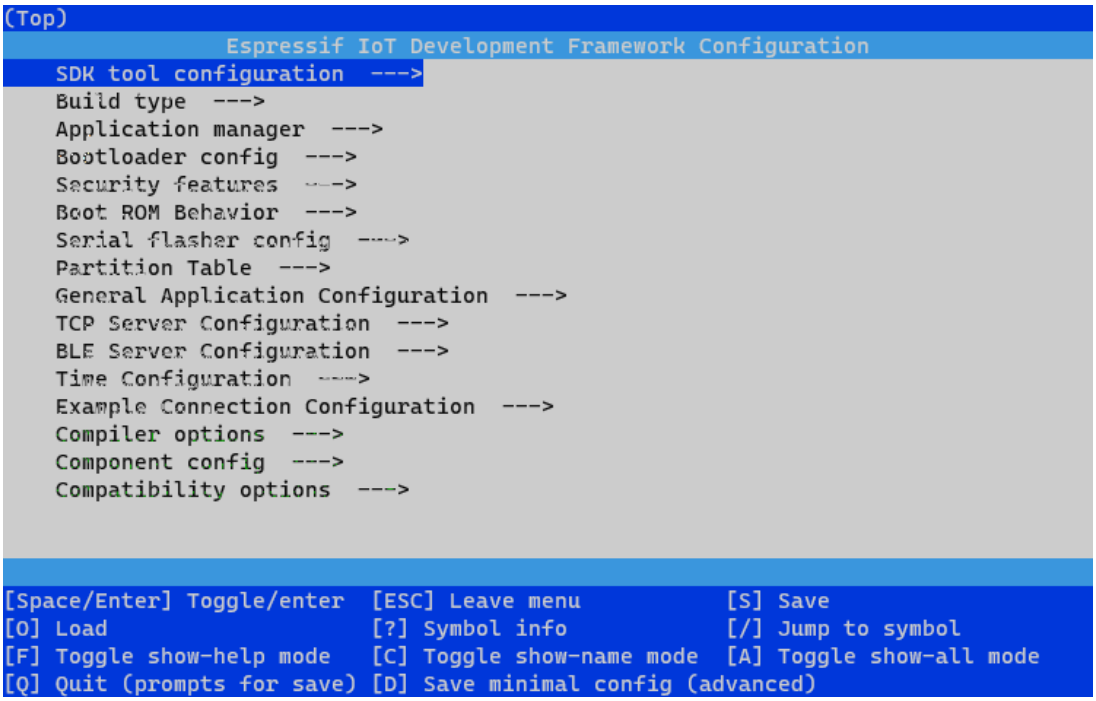


Figure 5.3: Output of ESP-IDF menuconfig

5.1.2 BLE modules

The BLE module is an important part of the system, as all of the close-range communication is done using BLE. Different versions of BLE are used in the different ESP32. The ESP32-S3 have an internal BLE module, and so we just need to activate it in the ESP-IDF project configuration. However, the ESP32-S2 needs an external BLE module, in the case of our project it is the JDY-32, which needs to be connected to the UART GPIO pins of the ESP32-S2.

A – ESP32-S3 internal BLE module: To activate the BLE 5.0 module of the ESP32-S3, we use the ESP-IDF project configuration accessed through the "*idf.py menuconfig*" command. In "Component config -> Bluetooth" the "Bluetooth" option is enabled, in "Component config -> Bluetooth -> Bluedroid Options" we enabled "Bluetooth Low Energy" and "Enable BLE 5.0 features". In the Firmware code the BLE is configure and initialize to work as intended using the default ESP32-S3 BLE library, "esp_bt_controller", and start the GATT server, as explained in section 4.3.6.

With the BLE component active we can scan for the ESP32-S3 (as shown in figure 5.4, captured in the BLE Scanner Android app, by Bluepixel Technologies) and then connect to it and exchange messages, which allow us to confirm that the BLE module is working as intended.

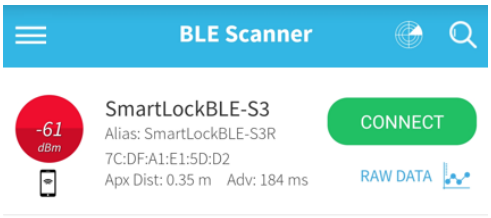


Figure 5.4: ESP32-S3 scanned by BLE Scanner app on Android

```
(Top) → BLE Server Configuration
      Espressif IoT Development Framework Configuration
(1) UART port number
(9600) UART communication speed
(17) UART RXD pin number
(16) UART TXD pin number
```

Figure 5.5: ESP32-S2 UART configurations

B – ESP32-S2: JDY-23 BLE module: To configure the JDY-23 BLE module we use AT commands. The most important AT commands are: "*AT+BAUD<Param>*", used to change the baud rate in the UART communication; the "*AT+NAME<Param>*", used to change the name displayed by the BLE module; and the "*AT+DISC*", used to disconnect from a client (the full list of AT commands can be found at the JDY-23 User Manual [18]). To use these commands we need to connect to the BLE module via Bluetooth and send these commands as a message.

After configuring the JDY-23 module we have to configure the ESP32-S2 UART GPIO pins to receive and transmit data (RXD and TXD). In the ESP-IDF project configuration we can change the "BLE Server Configuration" and configure the "UART communication speed" (baud rate), the "UART RXD pin number" and the "UART TXD pin number" (figure 5.5). With these values set, we can initialize the UART in the Firmware code as shown in listing 5.1.

With the JDY-23 and the UART configured, we can scan for the JDY-23, connect to it and send BLE messages, which are forwarded to the ESP32-S2 via the UART protocol, as explained in section 4.3.3.

C – RGB LED GPIO: To manipulate an electronic door lock we use the signal sent to a GPIO, however, in this project this signal is used to manipulate an RGB LED. In the project configuration it is possible to change the GPIO pin for the LED, in the menu "General Application Configuration", option "Led GPIO pin number" the default GPIO pin for the ESP32-S3 is 48. After configuring the pin number, we can manipulate the LED color using the default LED controller, "led_strip.h". Using the first function in the listing 5.2 it is possible to configure and initialize the LED and with the second one, it is possible to change its color using an RGB value.

Figure 5.6 shows the ESP32-S3 with the LED, on the left, turned on blue, indicating the lock state, and on the right, turned on green, indicating the unlock state, assuring us that both the firmware code

```
void init(void) {
    const uart_config_t uart_config = {
            .baud_rate = ECHO_UART_BAUD_RATE,
            .data_bits = UART_DATA_8_BITS,
            .parity = UART_PARITY_DISABLE,
            .stop_bits = UART_STOP_BITS_1,
            .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
            .source_clk = UART_SCLK_APB,
    };

    uart_driver_install(ECHO_UART_PORT_NUM, BUF_SIZE * 2, 0, 0, NULL, 0);
    uart_param_config(ECHO_UART_PORT_NUM, &uart_config);
    uart_set_pin(ECHO_UART_PORT_NUM, TXD_PIN, RXD_PIN, UART_PIN_NO_CHANGE,
    UART_PIN_NO_CHANGE);
}
```

Listing 5.1: Function to initialize UART

```
static void configure_led() {
    if (led_configured != 0) {
        return;
    }
    led_configured = 1;
    /* LED strip initialization with the GPIO and pixels number*/
    pStrip_a = led_strip_init(0, LED_GPIO, 1);
    /* Set all LED off to clear all pixels */
    pStrip_a->clear(pStrip_a, 50);
}

static void change_led_color(int rgb[3]) {
    pStrip_a->set_pixel(pStrip_a, 0, rgb[0], rgb[1], rgb[2]);
    pStrip_a->refresh(pStrip_a, 100);
}
```

Listing 5.2: Functions to initialize and manipulate the RGB LED

and the RGB LED are working properly.

5.1.3 Power Optimization

In a real-world scenario, the ESP32-S2/S3 could be powered by batteries, for that reason, it is important to optimize its power consumption to the lowest consumption possible. Both the ESP32-S2/S3 and the JDY-23 BLE module are already configured and prepared to have low power consumption, however, it is possible to optimize even more using sleep modes in the ESP32-S2/S3. The ESP32-S2/S3 contains the following power-saving modes, a) Light-sleep, and b) Deep-sleep.

- **In Light-sleep mode,** the digital peripherals, most of the RAM, and CPUs are clock-gated and

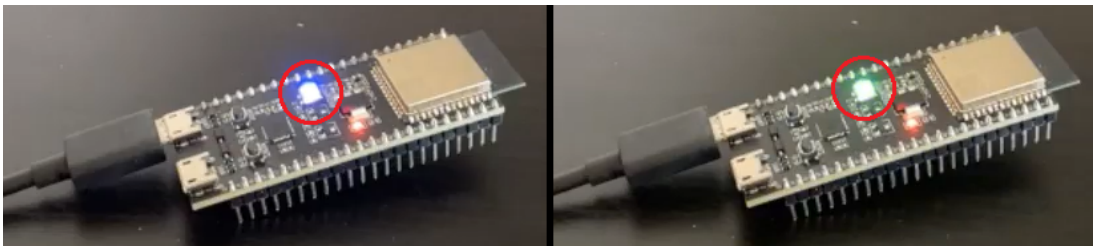


Figure 5.6: ESP32-S3 RGB LED showing the lock and unlock state

their supply voltage is reduced. Upon exit from Light-sleep, the digital peripherals, RAM, and CPUs resume operation and their internal states are preserved.

- **In Deep-sleep mode,** the CPUs, most of the RAM, and all digital peripherals are clocked or powered off.

In the case of our system, we decided to take advantage of the Deep-sleep mode to save power. We propose a system that, upon initialization, it works in a cycle between the wake mode and the Deep-sleep mode. It is awake for a short period of time to check if the client or the server is trying to communicate with it. If so, it would stay awake and perform all the requested operations. After a while, if inactive, the ESP32-S2/S3 would enter Deep-sleep and after a timeout wake and resume the cycle. This approach would theoretically help to reduce power consumption to a fraction.

After some testing and planning, we settled into a Deep-sleep period of 25 seconds and a wake period of 2 seconds (if no operation was requested). These timings made sense since no communication took longer than 2 seconds to reach the ESP32-S2/S3, allowing it to successfully check any pending request, and the 25 seconds in Deep-sleep is perfectly acceptable for a user waiting to connect and operate the Smart Lock.

To evaluate the power saved by this approach we measured the power consumption of the ESP32-S3 with the UM24C USB power meter and the Keithley Model 2001 Multimeter[1]. We measured the instant power consumption in multiple scenarios, a) in initialization, b) running (waiting for communication), c) deep-sleep and d) while receiving/decryption communication.

Table 5.1: Power consumption of the ESP32-S3 in different states of execution

Status	Power Consumption (mA)
Initialization	≈ 75
Running	≈ 100
Deep-sleep	≈ 0.87
Communication	≈ 120

[1] https://www.tek.com/en/products/keithley/digital-multimeter/2001-series

While the ESP32-S3 is in ”Stand-By” (waiting for commands), it cycles through the following states: it runs for 2 seconds, it then enters a Deep-sleep state for 25 seconds and then it takes approximately 3 seconds to initialize. In this ”Stand-By” scenario the ESP32-S3 consumes, on average, between 14mAh and 15mAh which corresponds to a power consumption of approximately 63mWh when using 5V. When requesting the Smart Lock to Unlock, using the RUD operation, it consumes, on average 0.205mAh or 1.025mWh.

If we consider two 18650 lithium-ion cells with a combined capacity of 26640mWh, our system operates for about 18 days between battery changes. To create a commercial product it is necessary to increase this autonomy, either by adding more battery power or optimizing the system to consume less power. An alternative is to add a power supply through a 5V adapter instead of batteries.

5.2 Service Server

The Service Server is responsible for all the communication with the Firebase database, and also for the remote communication with the Smart Door Lock. For this reason, it is important to test and verify the correctness of its code.

To do this evaluation it was implemented a suite of unit tests that checks the validity of the server code. These unit tests were made using the python library ”unittest”, and tested each function both with correct and incorrect inputs. The files tested were the a) ”rsa_util.py”; the b) ”firebase_util.py”; and the main file with the RESTful API c) ”app.py”.

A – RSA Util file: The main purpose of this file is to help in the RSA encryption and decryption of messages and also verify the signatures made by the Smart Lock with the private keys. The tests for this file were simple, testing the initialization function, the encrypt and decrypt functions and the sign and verify the signature. The details of these tests are in table 5.2.

Table 5.2: RSA util file Unit Tests

Test	Description	Expected Result
test_rsa_init_with_filename	Test the initialization of RSA util object with the name of the file containing the RSA key	Successful initialization of the RSA util object with the key in the file provided
test_rsa_init_with_key_str	Test the initialization of RSA util object with RSA key provided as a string	Successful initialization of the RSA util object with the key provided
test_get_rsa_key_from_x509_cert	Test the extraction of a RSA public key from a provided X.509 certificate	Successful extraction of the RSA public key from the provided X.509 certificate
test_encrypt_and_decrypt_msg_ok	Test the RSA encryption and decryption of a given message	Successful encryption and decryption of the given message
test_decrypt_msg_with_pub_key	Test the RSA decryption of a given message, only with the public key	Failed to decrypt message, returning a null object.

test_sign_and_is_signature_valid_ok	Test the RSA message signing of a message, and the correct signature validation	Successful message signing and signature validation with True result
test_sign_and_is_signature_valid_invalid_message	Test the RSA message signing and signature validation, with an altered message	Successful message signing and signature validation with False result

B – Firebase Util file: This file is used to help with the access and manipulation of the Firebase Database, we tested mainly the functions used to read and write in the database. The details of these tests are in table 5.3.

Table 5.3: Firebase util file Unit Tests

Test	Description	Expected Result
test_set_data_ok	Test the function used to write data in the Firebase Database	Successful write operation
test_get_data_ok	Test the function used to read data from the Firebase Database	Successful read operation, returning the requested data
test_delete_data_ok	Test the function used to remove data from the Firebase Database	Successful delete operation
test_add_data_to_path_ok	Test the function used to write data in the Firebase Database, automatically creating a random id for the data	Successful add operation
test_generate_random_id_small_n	Test the function used to create random alphanumeric IDs of size n, with n=5.	Successful creation of a random ID of type string, with length 5
test_generate_random_id_big_n	Test the function used to create random alphanumeric IDs of size n, with n=100.	Successful creation of a random ID of type string, with length 100
test_generate_random_id_n_equal_to_1	Test the function used to create random alphanumeric IDs of size n, with n=1.	Successful creation of a random ID of type string, with length 1
test_generate_random_id_n_equal_to_0	Test the function used to create random alphanumeric IDs of size n, with n=0.	Return an empty string
test_generate_random_id_negative_n	Test the function used to create random alphanumeric IDs of size n, with n=-1.	Return an empty string

C – Main RESTful API file: The most important file in the python Server is the app.py, which contains the endpoints and the main code for the RESTful API. This file used functions from both the util files mentioned above and given its importance, this file was tested more intensively. We tested all the endpoints available to the client, with multiple tests for each function, inputting both valid and invalid parameters to test all the possible scenarios. Due to the large number of tests done for this file, the table with its details is available in Appendix C.

5.3 Mobile App

The Mobile App Client is the user interface to interact with the SDL system. Through the Mobile App, the user can control the Smart Lock and change its settings, he can also create invites to share access to his Smart Locks. To develop the Android App we used the Android Studio IDE. The Android Studio

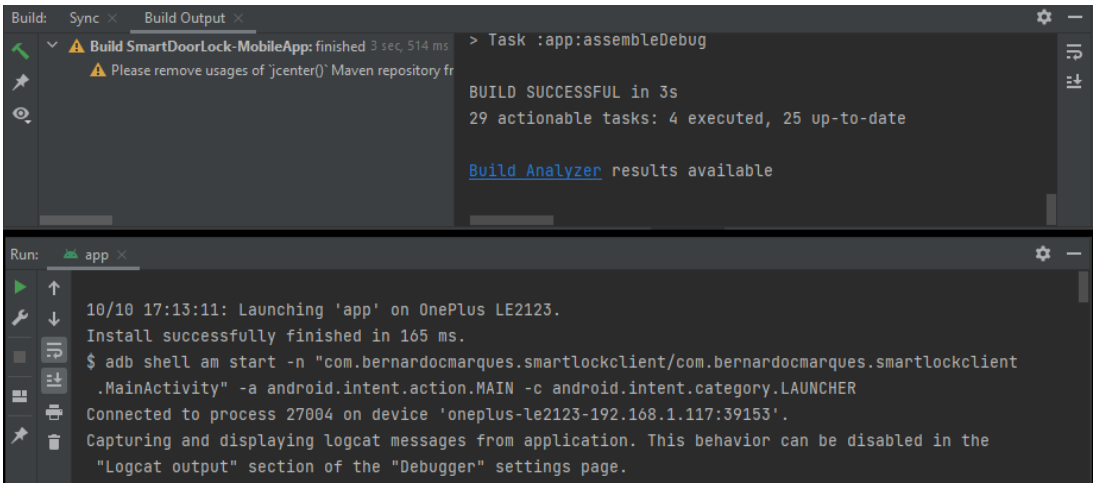


Figure 5.7: Android Studio Build and Run example

offers us a set of tools used to: a) Build and Run the Mobile App and b) monitor and debug the Mobile App code using the **Logcat** window. The Android Studio communicates with the Android Device for debugging using the Android Debug Bridge (ADB) programming tool, through USB or TCP.

A – Build and Run: The Android Studio easily build the Mobile App with a push of a button, automatically downloading and compiling the necessary extract external libraries and our code. After building the app, the Android Studio automatically installs the application in the debugging Android Device and runs it. Figure 5.7 it is shown the output of the **Build** window, where it is possible to check the success of the build operation, followed by the output of the **Run** window where it is shown the app being installed and run on the debugging device.

B – Logcat window: To monitor the execution of the Android App we used the **Logcat** window. In this window, it is displayed all the system messages and debug messages logged by our code or external libraries. It displays messages in real-time and it keeps a history, so it is possible to view older logs and help in the debugging process. Figure 5.8 shows a log of the messages sent to the Smart Lock device, through BLE (on the right) and these messages (underlined in red) being received on the monitor console of the ESP-IDF (on the left).

5.4 Communication Between Components

The multiple components in the SDL system communicate with each other using BLE and TCP/IP technologies, as explained in section 3.2. It is important to test and evaluate the communications between each component. In this section, we will evaluate the performance of the communication between the

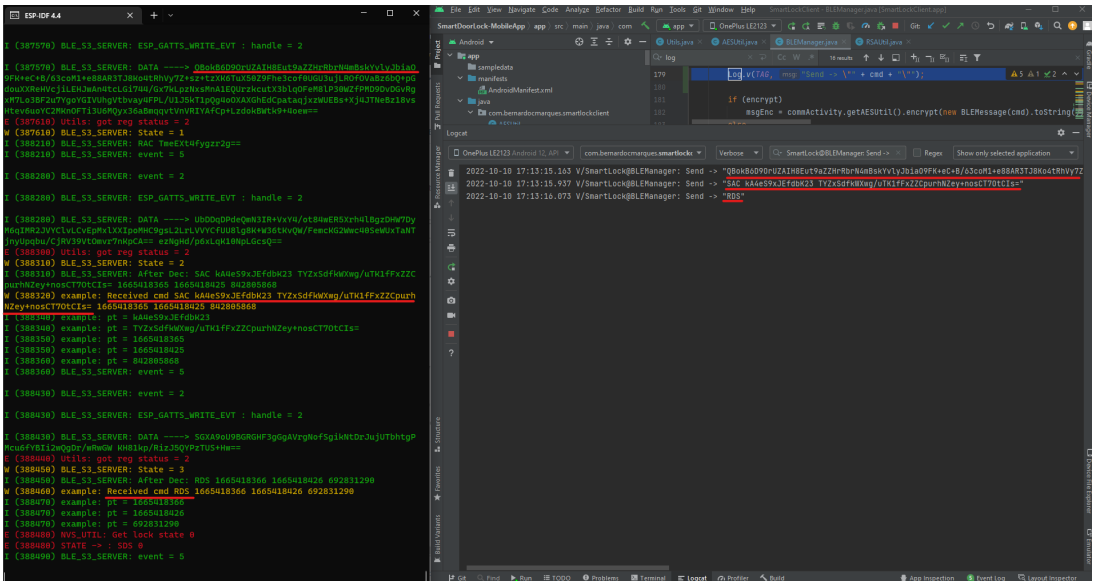


Figure 5.8: Android Studio Logcat example

Mobile App and the Smart Lock, and we will demonstrate the security of this communication channel.

5.4.1 Communication Performance

Measuring the time it takes to open the Smart Lock is very important to validate if the system is useful for the user, no one will use a SDL system if it took too long to open or close the door.

To evaluate the time performance of these communications we perform 4 different tests, 2 using direct BLE communication, and the other 2 using remote communication through the Service Server.

In the BLE communication channel, the Mobile App speaks directly with the Smart Lock using BLE. To measure the time performance of this channel we perform 2 different tests. In the first one we measured the Round Trip Time (RTT) of one message sent through BLE. And in the second, we measure the time it took to request the unlock of the door, the RUD operation, as shown in figure 5.9.

In the remote communication channel, the Server is used as an intermediary between the Mobile App and the Smart Lock. The Mobile App speaks with the Server using the RESTful API, and then the Server redirects the commands to the Smart Lock using a TCP/IP connection. To measure the time performance of this channel we perform 2 different tests. In the first one, we measured the RTT of one message sent through this communication channel as shown in figure 5.10. And in the second, we measure the time it took to remotely request the unlock of the door, the RUD operation, as shown in figure 5.11.

We performed each of the aforementioned tests 2000 times and calculated the average time each operation took, as well as the average of the worst 95% and 99% results.

Looking at the results in table 5.4, it is possible to conclude that the time performance of the com-

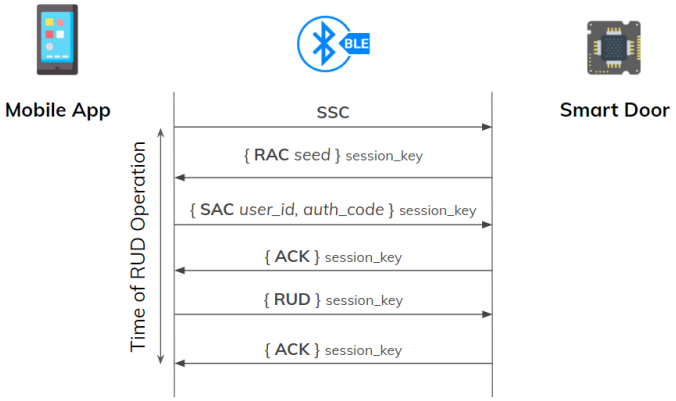


Figure 5.9: Test to measure the time of the RUD operation with BLE communication

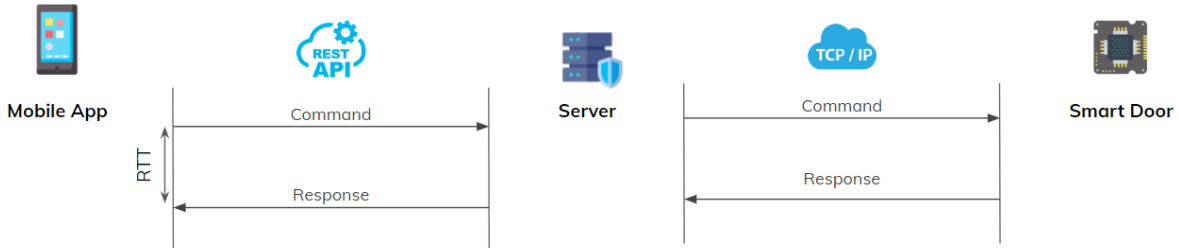


Figure 5.10: Test to measure the RTT with remote communication

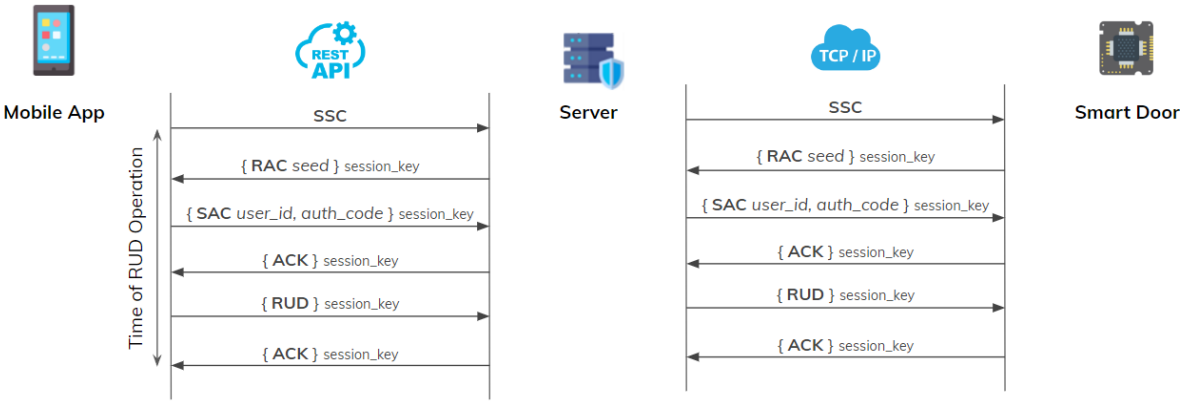


Figure 5.11: Test to measure the time of the RUD operation with remote communication

Table 5.4: Results of time performance of the communication.

RTT Tests	**Average (ms)**	**Worst 95% (ms)**	**Worst 99% (ms)**
Single command with BLE	77.67	111.62	117.05
RUD operation with BLE	911.01	1024.15	1067.60
Single command remotely	335.91	1493.34	2425.30
RUD operation remotety	1418.94	2586.04	3379.10

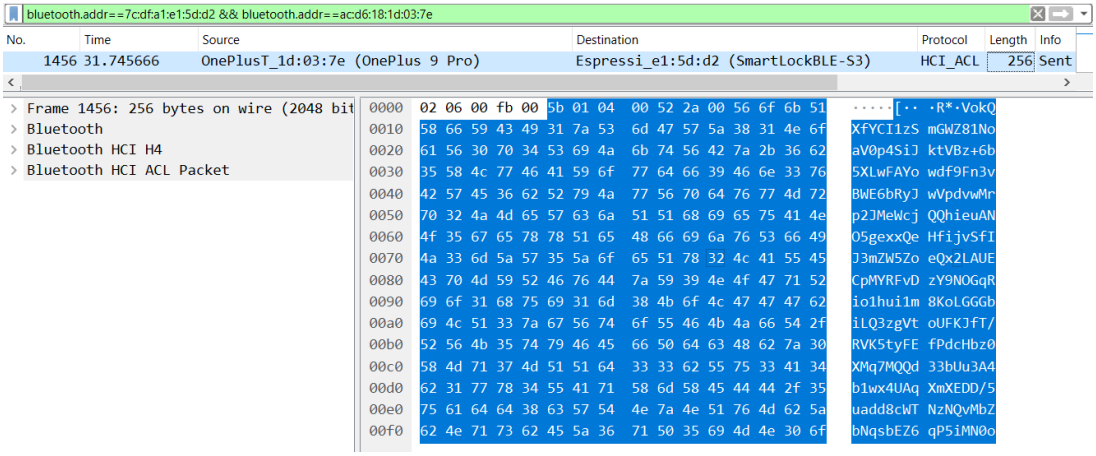


Figure 5.12: Wireshark Capture showing the correct encryption of the messages

munication between the Mobile App and the Smart Lock is good and acceptable for the use case. In the RTT of the RUD operation, we have an average result lower than a second, and even when looking at the worst 95% and 98% we have times near to a second, which is more than acceptable to a user requesting the unlock of the Smart Door Lock. When looking at the RUD operation remotely, we have an average time of less than a second and a half, and in the worst cases this time is around 3.3 seconds, also an acceptable time to wait for the remote unlock of the Smart Lock. Looking at the RTT of a single message using BLE we can also conclude that most of the time during the RUD operation is used in the encryption and decryption of the messages, as well as, calculating the authorization code. Finally, we can conclude that the worst test results in remote communication are probably related to the instability of the internet connection, since the worst results of a single message are similar to the worst results of the RUD operation.

5.4.2 Communication Security

The security of the communication channel between the Mobile App and the Smart Lock is necessary to prevent intruders from eavesdropping on the communication and getting hold of private information about the lock or the user. All the messages exchanged between these two devices are encrypted using RSA or AES, as explained in section 3.3.

To verify the successful encryption of these messages and that it is impossible to eavesdrop, we used Wireshark [2] and Android HCI Snoop Log to intercept the messages exchanged via BLE and check the correct encryption of these messages. Figure 5.12 shows a screenshot of Wireshark intercepting messages sent from the Mobile App to the Smart Lock. As it is possible to see that the message (highlighted in blue) is encrypted, and the user eavesdropping can't read the contents of the message.

[2] https://www.wireshark.org/

5.5 System Demonstration

For a final evaluation and validation of the system, it is important to show the system working as a whole, to check if the different components interact as intended. For that reason, we did a video demonstration of the system working in a situation near the real world, and we can verify if everything works as intended, and in a user-friendly way.

In the following demo, the user uses the Mobile App to create and login to a new account, using ”Login with Google”, and then configures a new Smart Lock. Once the lock is configured he unlocks and locks it, edit the lock name and icon and finally create a new invite for the lock, sharing it via email. In the video it is possible to see the ESP32-S3 Monitor console, the ESP32-S3 RGB led, the Firebase database and the Mobile App. Throughout this demo, it is possible to see the multiple components working with each other when the user interacts with the Mobile App.

Demo Video: https://drive.google.com/file/d/1DsX27aJCN70dJIFdr-0tSk9-tjE1fzTj/view

Although this video shows multiple of the system features, it doesn't show all the developed features, for that reason we included a link to a folder with multiple small video demos of each feature, filmed during the development process of the system.

Video Repository: https://drive.google.com/drive/folders/1dkLh4pij-du4HPkIwg8WJwYqWif1iakr

6

Conclusions

Contents

6.1 Conclusions

In this project, we proposed the design and implementation of a proof of concept for a SDL system that could be controlled by a smartphone. We focused on the development of a secure, reliable and efficient system with low power consumption, and we set ourselves to solve some of the problems of the already existing smart lock systems.

At the end of this project, we successfully developed a SDL system based on the ESP32-S2/S3 embedded system which uses BLE and a TCP server to communicate with the other components. In a real-world working system, the ESP32-S2/S3 would be used to control an electronic lock mechanism or a latch with a servo motor.

We also implemented a Service Server with two main purposes, a) to serve as a gateway between the Mobile App and the Smart Locks, allowing for secure remote communication between the two, and b) to give access to the Firebase Database, where the users and locks data are stored. The Service Server consists of a RESTful API in a python web server.

Finally, we built a client Mobile App for Android devices, through this app the user can control the Smart Lock and share access to his locks. The Mobile App is the main way for the user to interact with the SDL system and it is equivalent to a key to open the Smart Lock.

In summary, we created a proof of concept for a SDL system with all the necessary firmware and software for a real-world working system. The developed system could be classified with a level 4 on the Technology Readiness Level (TRL) scale which corresponds to a technology validated in lab, and it would be easily upgraded to a level 5, a technology validated in relevant environment. The final system has a useful set of features that allow the user to share access to his locks, creating multiple sub-users, features to control the lock remotely via Internet, and an automatic unlocking system based on proximity.

With this thesis, we proved the possibility to create a SDL system with a rich feature set, using small and low-power embedded systems and low-power communication technologies as BLE.

6.2 Future Work

There are multiple possible paths of future work to follow from this project. One of which would be to integrate it with a mechanism to electronically control a door, creating a system that could work in the real world. Another way to improve this system is to further develop the Mobile client App, developing more features useful for the end user, and creating a more straightforward user interaction. Finally, it would be worth some work to streamline the process to deploy the system and install a smart lock. As it stands now, the user needs to deploy a python server, and configure both the ESP32-S2/S3 and his home network to allow exterior communication with the lock via the Internet. All this work is hard for a non-expert user, creating an unnecessary barrier to using this system.

Another valid approach for future work would be to experiment with other embedded systems and communication technologies. We used the ESP32-S2/S3 and BLE for this project, however, there are multiple other wireless technologies to experiment with, like Z-Wave, Zigbee, NFC, and multiple others. It would be interesting to test other combinations of embedded systems and communication technologies and evaluate if would be possible to create a more efficient system or a better user experience.

In our system, we used the recommended encryption algorithms and key sizes for the master key used in the Smart Lock security. However, it would be interesting to change the system to allow the use of different key sizes for locks with different security requirements. It would be possible to have smaller keys for low-security locks, improving the unlocking time, and have more advanced keys for high-security locks.

Finally, to deploy this technology as a commercial product with batteries, it would be necessary to reduce the system's power consumption. Although the system can be powered by batteries, the end user may only expect to change the batteries twice or three times a year, hence the final power consumption results need to be improved in this scenario.

Bibliography

[1] R. Martin, “Electronic combination door lock with dead bolt sensing means,” US Patent US4 148 092A, 8, 1977. [Online]. Available: https://patents.google.com/patent/US4148092A

[2] P. S. Lee, “Smart card access control system,” US Patent US5 204 663A, 10, 1991. [Online]. Available: https://patents.google.com/patent/US5204663A

[3] J. C. Schmitt and D. R. Setlak, “Access control system including fingerprint sensor enrollment and associated methods,” US Patent US5 903 225A, 5, 1997. [Online]. Available: https://patents.google.com/patent/US5903225A

[4] M. Sahani, C. Nanda, A. K. Sahu, and B. Pattnaik, “Web-based online embedded door access control and home security system based on face recognition,” *IEEE International Conference on Circuit, Power and Computing Technologies, ICCPCT 2015*, 7 2015. [Online]. Available: https://ieeexplore.ieee.org/document/7159473

[5] R. A. Rashid, N. H. Mahalin, M. A. Sarijari, and A. A. A. Aziz, “Security system using biometric technology: Design and implementation of voice recognition system (vrs),” *Proceedings of the International Conference on Computer and Communication Engineering 2008, ICCCE08: Global Links for Human Development*, pp. 898–902, 2008.

[6] N. Hashim, N. Azmi, F. Idris, and N. Rahim, “Smartphone activated door lock using wifi,” *ARPN Journal of Engineering and Applied Sciences*, vol. 11, pp. 3309–3312, 3 2016. [Online]. Available: http://www.arpnjournals.org/jeas/research_papers/rp_2016/jeas_0316_3803.pdf

[7] K. Patil, N. Vittalkar, P. Hiremath, and M. Murthy, “Smart door locking system using iot,” *International Journal of Engineering and Technology*, vol. 7, pp. 2395–0056, 5 2020. [Online]. Available: https://www.researchgate.net/publication/341508373

[8] C. Gomez, J. Oller Bosch, and J. Paradells, “Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology,” *Sensors (Basel, Switzerland)*, vol. 12, pp. 11 734–53, 12 2012. [Online]. Available: https://www.mdpi.com/1424-8220/12/9/11734

[9] Y. W. Prakash, V. Biradar, S. Vincent, M. Martin, and A. Jadhav, “Smart bluetooth low energy security system,” pp. 2141–2146, 3 2017. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8300139

[10] M. S. Hadis, E. Palantei, A. A. Ilham, and A. Hendra, “Design of smart lock system for doors with special features using bluetooth technology,” pp. 396–400, 3 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8350767

[11] Z. Mu, W. Li, C. Lou, and M. Liu, “Investigation and application of smart door locks based on bluetooth control technology,” pp. 68–72, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9115189

[12] N. Hanafiah, C. P. Kariman, N. Fandino, E. Halim, F. Jingga, and W. Atmadja, “Digital door-lock using authentication code based on ann encryption,” *Procedia Computer Science*, vol. 179, pp. 894–901, 2021, 5th International Conference on Computer Science and Computational Intelligence 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050921001095

[13] R. Karani, S. Dhote, N. Khanduri, A. Srinivasan, R. Sawant, G. Gore, and J. Joshi, “Implementation and design issues for using bluetooth low energy in passive keyless entry systems,” pp. 1–6, 12 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7838978

[14] E. B. Barker and A. L. Roginsky, “Transitioning the use of cryptographic algorithms and key lengths,” 3 2019. [Online]. Available: https://www.nist.gov/publications/transitioning-use-cryptographic-algorithms-and-key-lengths

[15] Espressif, *ESP32-S2 Technical Reference Manual*, 9 2022. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-s2_technical_reference_manual_en.pdf

[16] Espressif, *ESP32-S3 Technical Reference Manual*, 9 2022. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-s3_technical_reference_manual_en.pdf

[17] J. Tosi, F. Taffoni, M. Santacatterina, R. Sannino, and D. Formica, “Performance evaluation of bluetooth low energy: A systematic review,” *Sensors*, vol. 17, p. 2898, 12 2017. [Online]. Available: https://www.researchgate.net/publication/321800210

[18] Shenzhen Jindouyun Electronic Technology Co., *Ultra Low Power Bluetooth 5.0 BLE Module - User Manual of JDY-23 Slave Bluetooth Module*, 8 2018. [Online]. Available: https://fcc.report/FCC-ID/2AXM8-JDY-23/4936741.pdf

[19] Google. (2021) Bluetooth Low Energy - Android Developers. Accessed 17-Aug-2022. [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth/ble-overview

Appendix A - Protocol Communications Example

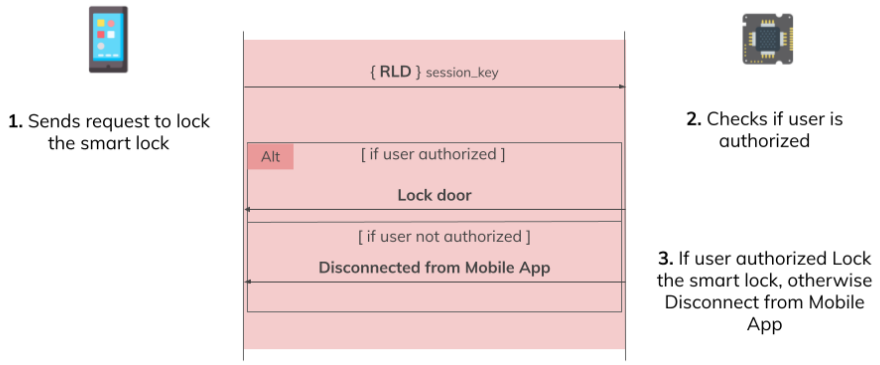


Figure A.1: Example of protocol to request the smart lock to lock

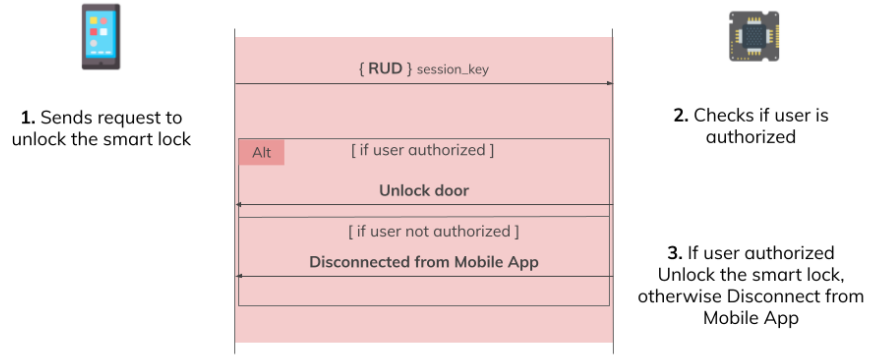


Figure A.2: Example of protocol to request the smart lock to unlock

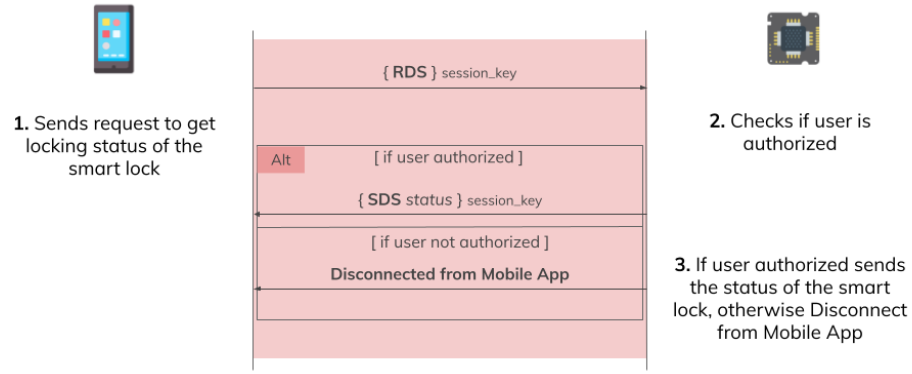


Figure A.3: Example of protocol to request the smart lock current status

B

Appendix B

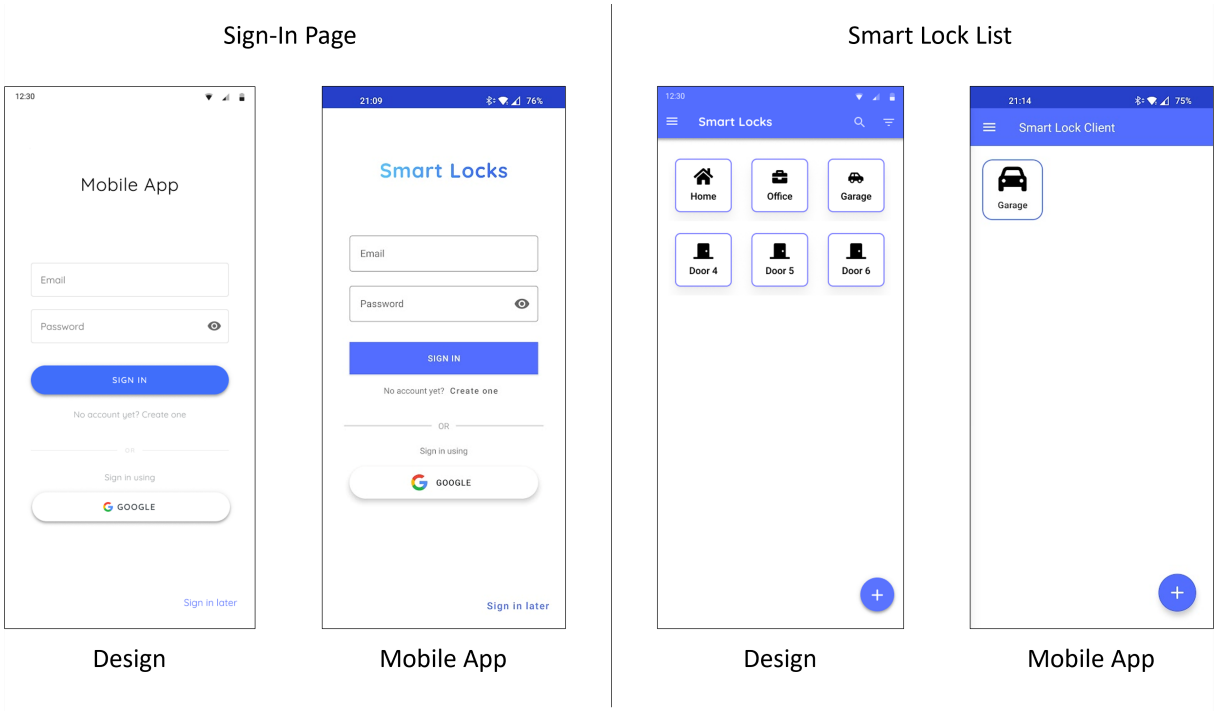


Figure B.1: Design and final implementation of login page and smart locks list page

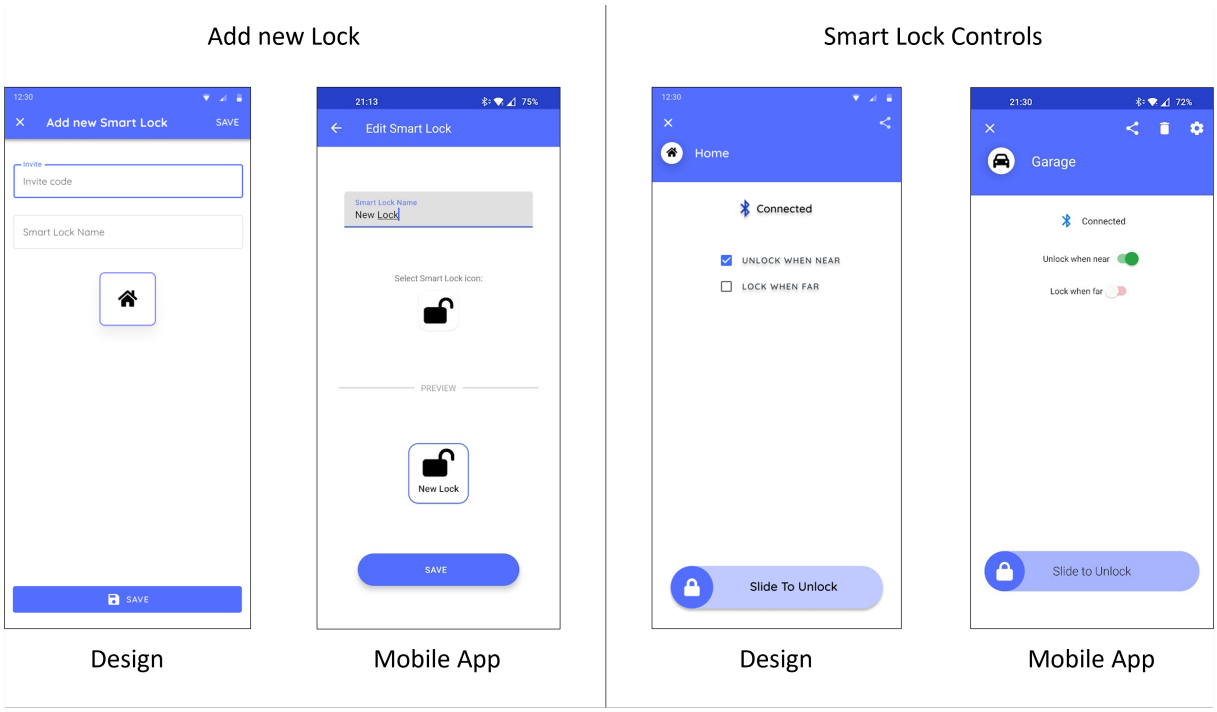
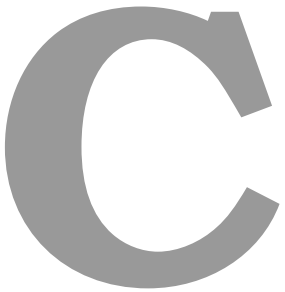


Figure B.2: Design and final implementation of add smart lock page and smart lock control page



Appendix C - Table Unit Tests Server

Table C.1: RESTfull API file Unit Tests

Test	Description	Conditions	Expected Result
test_get_all_icons	Test the endpoint ”/get-all-icons” which returns a list of all the icon IDs available on the server.	All parameters correct	List of available icons
test_get_icon_ok	Test the endpoint ”/get-icon?icon_id={icon_id}” which should return the image file of the requested icon.	All parameters correct	Image in png format of the request icon
test_get_icon_invalid_icon_id		Invalid ”icon_id” parameter	Returns an error with the message ”Icon with ID {icon_id} does not exist”
test_get_icon_no_icon_id		Does not provide ”icon_id” parameter	Returns an error with the message ”icon_id not provided”
test_register_phone_id_ok	Test the endpoint ”/register-phone-id”, used to register a new phone id in the user’s database.	All parameters correct	Successfully register new phone
test_register_phone_id_no_id_token		Does not provide ”id_token” parameter	Fails to register new phone, and returns the error message ”No Id Token”
test_register_phone_id_invalid_id_token		Invalid ”id_token” parameter	Fails to register new phone, and returns error message the ”Invalid Id Token”
test_register_phone_id_no_phone_id		Does not provide ”phone_id” parameter	Fails to register new phone, and returns the error message ”No Phone Id”
test_register_door_lock_ok	Test the endpoint ”/register-door-lock”, used to register a new Smart Door Lock in the database.	All parameters correct	Successfully register the Smart Door Lock
test_register_door_lock_invalid_post_data		Any of the request parameters missing	Fails to register the Smart Door Lock and returns the error messege ”Missing arguments.”
test_door_get_cert_ok	Test the endpoint ”/get-door-certificate?id_token={id_token}smart_lock_mac={smart_lock_mac}”, used to get the X.509 Certificate of a given Smart Lock.	All parameters correct	Returns the X.509 certificate of the request Smart Lock
test_door_get_cert_invalid_id_token		Invalid ”id_token” parameter	Returns an error with the message ”Invalid Id Token”
test_door_get_cert_no_id_token		Does not provide ”id_token” parameter	Returns an error with the message ”No Id Token”
test_door_get_cert_no_smart_lock_mac		Does not provide ”smart_lock_mac” parameter	Returns an error with the message ”No smart_lock_mac”
test_door_get_cert_invalid_smart_lock_mac		Invalid ”smart_lock_mac” parameter	Returns an error with the message ”Invalid smart_lock_mac”
test_register_invite_ok_admin	Test the endpoint ”/register-invite”, used to create and register a new invite into the database.	All parameters correct. Invite type Admin	Successfully creates invite of type Admin
test_register_invite_ok_owner		All parameters correct. Invite type Owner	Successfully creates invite of type Owner
test_register_invite_ok_tenant		All parameters correct. Invite type Tenant	Successfully creates invite of type Tenant
test_register_invite_ok_periodic_user		All parameters correct. Invite type Periodic User	Successfully creates invite of type Periodic User
test_register_invite_ok_one_time_user		All parameters correct. Invite type One Time User	Successfully creates invite of type One Time User
test_register_invite_not_signed		Does not provide ”signature” parameter, which is used to validate the invite creation request	Fails to create invite and returns the error message ”Message not signed”

test_register_invite_no_data		Does not provide the parameters of the requested invite	Fails to create invite and returns the error message ”Invalid data”
test_register_invite_invalid_signature		Invalid ”signature” parameter	Fails to create invite and returns the error message ”Invalid signature”
test_redeem_invite_ok_admin	Test the endpoint ”/redeem-invite”, used to redeem a given invite, and create a new authorization.	All parameters correct. Invite type Admin	Successfully redeems invite of type Admin
test_redeem_invite_ok_owner		All parameters correct. Invite type Owner	Successfully redeems invite of type Owner
test_redeem_invite_ok_tenant		All parameters correct. Invite type Tenant	Successfully redeems invite of type Tenant
test_redeem_invite_ok_periodic_user		All parameters correct. Invite type Periodic User	Successfully redeems invite of type Periodic User
test_redeem_invite_ok_one_time_user		All parameters correct. Invite type One Time User	Successfully redeems invite of type One Time User
test_redeem_invite_no_id_token		Does not provide ”id_token” parameter	Fails to redeem invite and returns the error message ”No Id Token”
test_redeem_invite_invalid_id_token		Invalid ”id_token” parameter	Fails to redeem invite and returns the error message ”Invalid Id Token”
test_redeem_invite_no_invite_id		Does not provide ”invite_id” parameter	Fails to redeem invite and returns the error message ”No invite id”
test_redeem_invite_invalid_invite_id		Invalid ”invite_id” parameter	Fails to redeem invite and returns the error message ”Invalid invite”
test_redeem_invite_invalid_phone_id		Invalid ”phone_id” parameter	Fails to redeem invite and returns the error message ”Invalid Phone Id!”
test_redeem_invite_email_locked_ok		Provide invite locked to user that is requesting the invite	Successfully redeems invite
test_redeem_invite_email_locked_not_ok		Provide invite locked to a different user from the one that is requesting the invite	Fails to redeem invite and returns the error message ”No permissions. This invite is user locked!”
test_save_user_invite_ok	Test the endpoint ”/save-user-invite”, used to save the user invite in the database.	All parameters correct.	Successfully saves used invite
test_save_user_invite_no_id_token		Does not provide ”id_token” parameter	Fails to save user invite and returns the error message ”No Id Token”
test_save_user_invite_invalid_id_token		Invalid ”id_token” parameter	Fails to save user invite and returns the error message ”Invalid Id Token”
test_save_user_invite_no_invite_id		Does not provide ”invite_id” parameter	Fails to save user invite and returns the error message ”No invite id”
test_save_user_invite_no_id_lock		Does not provide ”id_lock” parameter	Fails to save user invite and returns the error message ”No lock id”
test_save_user_invite_invalid_invite_id		Invalid ”invite_id” parameter	Fails to save user invite and returns the error message ”Invalid invite”

test_save_user_invite_locked_wrong_user		Provide invite locked to a different user from the one that is requesting the invite	Fails to save user invite and returns the error message ”No permissions. This invite is user locked!”
test_check_user_invite_ok_got_invite	Test the endpoint ”/check-user-invite?id_token={id_token}lock_id={lock_id}”, used to check if the user have a saved user invite in the database.	All parameters correct. User have saved invite in database	Retuns ”got_invite” = True
test_check_user_invite_ok_not_got_invite		All parameters correct. User does not have saved invite in database	Retuns ”got_invite” = False
test_check_user_invite_no_id_token		Does not provide ”id_token” parameter	Returns error messages ”No Id Token”
test_check_user_invite_invalid_id_token		Invalid ”id_token” parameter	Returns error messages ”Invalid Id Token”
test_check_user_invite_no_lock_id		Does not provide ”lock_id” parameter	Returns error messages ”No lock id”
test_redeem_user_invite_ok	Test the endpoint ”/redeem-user-invite”, used to redeem the user-saved invite in the database.	All parameters correct.	Successfully redeems user invite
test_redeem_user_invite_no_token_id		Does not provide ”id_token” parameter	Fails to redeem user invite and return the error messages ”No Id Token”
test_redeem_user_invite_no_phone_id		Does not provide ”phone_id” parameter	Fails to redeem user invite and return the error messages ”No Phone Id”
test_redeem_user_invite_no_lock_id		Does not provide ”lock_id” parameter	Fails to redeem user invite and return the error messages ”No lock id”
test_redeem_user_invite_no_master_key		Does not provide ”master_key” parameter	Fails to redeem user invite and return the error messages ”No Master Key”
test_redeem_user_invite_invalid_id_token		Invalid ”id_token” parameter	Fails to redeem user invite and return the error messages ”Invalid Id Token”
test_redeem_user_invite_invite_not_saved		User does not have a saved invite	Fails to redeem user invite and return the error messages ”Can’t get user saved invite.”
test_redeem_user_invite_no_invite		Invite does not exist	Fails to redeem user invite and return the error messages ”Invalid invite”
test_redeem_user_invite_email_locked_nok		Invite locked to a different user	Fails to redeem user invite and return the error messages ”No permissions. This invite is user locked!”
test_redeem_user_invite_invalid_phone_id		Invalid ”phone_id” parameter	Fails to redeem user invite and return the error messages ”Invalid Phone Id!”
test_request_auth_ok	Test the endpoint ”/request-authorization”, used to get an authorization object from the database.	All parameters correct.	Returns the requested authorization
test_request_auth_not_signed		Does not provide ”signature” parameter, which is used to validate the request	Returns the error message ”Message not signed”
test_request_auth_no_data		Does not provide the parameters of the requested invite	Returns the error message ”Invalid data”
test_request_auth_invalid_signature		Invalid ”signature” parameter	Returns the error message ”Invalid signature”

test_check_lock_reg_status_ok_registered _with_auths	Test the endpoint "/check-lock-registration-status?MAC={MAC}", used to get the current registration status of a Smart Lock	All parameters correct. Smart Lock registered and with authentications	Returns 'status' = 2, meaning the lock is registered and with authentications
test_check_lock_reg_status_ok _registered		All parameters correct. Smart Lock registered	Returns 'status' = 1, meaning the lock is registered
test_check_lock_reg_status_ok_not_registered		All parameters correct. Smart Lock not registered	Returns 'status' = 0, meaning the lock is registered
test_check_lock_reg_status_no_mac		Does not provide "mac" parameter	Missing argument MAC.
test_get_user_locks_ok_one_lock	Test the endpoint "/get-user-locks?id_token={id_token}", used to get the list of Smart Locks of a given user	All parameters correct. Only one lock in the database	Returns a list with the user saved lock
test_get_user_locks_ok_multiple_locks		All parameters correct. Multiple lock in the database	Returns a list with the user saved locks
test_get_user_locks_ok_no_locks		All parameters correct. No lock in the database	Returns an empty list
test_get_user_locks_invalid_id_token		Invalid "id_token" parameter	Returns error message "Invalid Id Token"
test_get_user_locks_no_id_token		Does not provide "id_token" parameter	Returns error message "No Id Token"
test_set_user_locks_ok	Test the endpoint "/set-user-locks?id_token", used to add a lock to the list of Smart Locks of a given user	All parameters correct	Successfully add lock to the user's list
test_set_user_locks_ok_add_to_existing		All parameters correct. Some Locks already registed	Successfully add lock to the user's list
test_set_user_locks_invalid_id_token		Invalid "id_token" parameter	Returns error message "Invalid Id Token"
test_set_user_locks_no_id_token		Does not provide "id_token" parameter	Returns error message "No Id Token"
test_delete_user_lock_ok	Test the endpoint "/delete-user-locks", used to delete a lock from the list of Smart Locks of a given user	All parameters correct	Successfully deletes the lock from the user's list
test_delete_user_lock_ok_multiple_phone_id		All parameters correct. Some Locks already registed	Successfully deletes the lock from the user's list
test_delete_user_lock_no_id_token		Invalid "id_token" parameter	Returns error message "Invalid Id Token"
test_delete_user_lock_invalid_id_token		Does not provide "id_token" parameter	Returns error message "No Id Token"
test_delete_user_lock_no_lock_id		Does not provide "lock_id" parameter	Returns error message "No lock id"