BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Secure Onboarding of IoT Sensing Devices for Artwork Tracking

Mete Polat
Zürich, Switzerland
Student ID: 18-932-129

Supervisor: Thomas Grübl, Katharina Müller
Date of Submission: May 1, 2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

M. Tolat

Zürich, May 1, 2024

Signature of student

Zusammenfassung

Die schnelle Verbreitung des Internets der Dinge (IoT) hat verschiedene Branchen revolutioniert, indem sie eine verbesserte Konnektivität und Funktionalität für eine Reihe von Anwendungen ermöglicht. Allerdings wirft die Integration von IoT-Technologien in sensiblen Sektoren, wie z. B. der Verfolgung von Kunstwerken, erhebliche Sicherheitsbedenken auf. Diese Arbeit befasst sich mit den kritischen Schwachstellen, die mit dem Onboarding von IoT-Sensorgeräten verbunden sind, und schlägt einen sicheren und effizienten Prozess vor, der auf Systeme zur Verfolgung von Kunstwerken zugeschnitten ist.

Diese Arbeit konzentriert sich auf die Entwicklung eines leichtgewichtigen Onboarding-Prozesses, der robuste Sicherheitsmassnahmen integriert, ohne die Benutzerfreundlichkeit und Effizienz von IoT-Geräten zu beeinträchtigen. Unter Verwendung der Arduino-Plattform und des ESP-NOW-Kommunikationsprotokolls untersucht diese Arbeit die Implementierung fortschrittlicher kryptografischer Techniken, einschlieslich AES- und RSA-Verschlüsselung, um die Datenübertragungen zwischen Clients und Gateway zu sichern.

Es wurden umfangreiche Tests durchgeführt, um die Wirksamkeit des vorgeschlagenen Onboarding-Prozesses zu bewerten, wobei der Schwerpunkt auf seiner Anwendbarkeit lag. Die Ergebnisse zeigen, dass der sichere Onboarding-Prozess nicht nur die strengen Sicherheitsanforderungen erfüllt, die für den Schutz hochwertiger Güter erforderlich sind, sondern auch unter typischen Betriebsbedingungen effizient funktioniert. Diese Ergebnisse deuten darauf hin, dass die vorgeschlagene Methodik als Modell für die Verbesserung der IoT-Sicherheit in ähnlichen Anwendungen dienen könnte und einen skalierbaren und zuverlässigen Rahmen für die sichere Integration von IoT-Geräten bietet.

Diese Arbeit trägt zum laufenden Diskurs über IoT-Sicherheit bei und bietet eine praktische Lösung für ein dringendes Problem in einem Bereich von wachsender Bedeutung. Indem sie sich sowohl mit den theoretischen Grundlagen als auch mit der praktischen Umsetzung der IoT-Sicherheit befasst, liefert die Forschungsarbeit wertvolle Erkenntnisse, die für künftige Entwicklungen auf diesem Gebiet von Bedeutung sein könnten, insbesondere für die Verbesserung der Sicherheitsprotokolle von IoT-Geräten in verschiedenen Sektoren.

Abstract

The rapid proliferation of the Internet of Things (IoT) has revolutionized various industries by enabling enhanced connectivity and functionality across a range of applications. However, the integration of IoT technologies in sensitive sectors, such as artwork tracking, raises significant security concerns. This thesis addresses the critical vulnerabilities associated with the onboarding of IoT sensing devices, proposing a secure and efficient process tailored for artwork tracking systems.

The research focuses on the development of a lightweight onboarding process that integrates robust security measures without compromising the usability and efficiency of IoT devices. Utilizing the Arduino platform and the ESP-NOW communication protocol, the thesis explores the implementation of advanced cryptographic techniques, including AES and RSA encryption, to secure data transmissions between IoT devices and network gateways.

Extensive testing was conducted to evaluate the effectiveness of the proposed onboarding process, with a particular focus on its applicability to real-world scenarios in artwork tracking. The results demonstrate that the secure onboarding process not only meets the stringent security requirements necessary to protect high-value assets, but also operates efficiently under typical operational conditions. These findings suggest that the proposed methodology could serve as a model for improving IoT security in similar applications, providing a scalable and reliable framework for the secure integration of IoT devices.

This thesis contributes to the ongoing discourse on IoT security, offering a practical solution to a pressing issue in an area of growing importance. By addressing both the theoretical underpinnings and practical implementation of IoT security, the research provides valuable insights that could inform future developments in the field, particularly in enhancing the security protocols of IoT devices across various sectors.

Contents

Chapter 1

Introduction

The Internet of Things (IoT) has transformed our interaction with the world. An increasing number of devices are influencing different aspects of our lives, from smart home technology to cars, smart wearables, and artworks that are being connected to the Internet. However, this rapid proliferation of IoT devices has brought new and complex security challenges. Security is a major challenge for IoT devices due to their limited resources, which often prevent the implementation of robust security mechanisms. To address these challenges, new security mechanisms tailored for low-resource IoT devices must be developed, encompassing both software and hardware.

Within the art sector, museums not only exhibit pieces from their own collections, but also incorporate artworks from other institutions. This situation requires the need for logistics partners who can ensure the safety of artworks during transportation. It also offers a valuable opportunity for the deployment of IoT sensing devices, which are adept at tracking environmental conditions such as temperature, humidity, and vibrations. Regrettably, factory security configurations for these devices are often insufficient to meet these challenges. This is especially true for IoT endpoints, which are becoming smaller and therefore have less computational power and fewer security mechanisms [1]. This thesis aims to contribute to this area of research by proposing a secure and lightweight onboarding process.

1.1 CERTIFY Project

CERTIFY is a multi-partner research project, with the goal of achieving a high level of security by developing a novel framework to manage security throughout the lifecycle of IoT devices. The project is scheduled to run from 1st October 2022 for 36 months and involves 13 partners from eight European countries [2]. The Communication Systems Group within the Department of Informatics at the University of Zurich participates in CERTIFY. The work presented in this thesis represents a segment of the group's contributions to this pilot project [3].

1.2 Description of Work

1.2.1 Objective

The primary objective of this research was to enhance the security of IoT devices used in the art industry by developing a secure and lightweight onboarding process. This process was designed to mitigate risks such as unauthorized access and data interception, which are prevalent in current IoT deployments.

1.2.2 Methodology

The methodology employed in this research involved three main phases: design, implementation, and testing.

Design Phase

- Security Requirements Analysis: The initial stage involved a thorough analysis of the security requirements essential for IoT devices in the art sector. This analysis helped in identifying the key vulnerabilities and the corresponding security controls needed.
- System Architecture Design: Based on the requirements analysis, a detailed system architecture was designed. Additionally the cryptographic protocols AES and RSA integrated.
- Protocol Selection: The ESP-NOW protocol was chosen for its efficiency in handling secure communications between IoT devices. This selection was based on its low power consumption and its ability to operate independently of a Wi-Fi network.

Implementation Phase

- Hardware Setup: The implementation used ESP32-S3 boards programmed via the Arduino development environment. These boards were chosen for their robust security features and compatibility with the ESP-NOW protocol.
- Software Development: The software developed included the setup of secure communication channels, cryptographic key management, and error handling mechanisms. The Arduino IDE was used to program the devices, emphasizing the implementation of the cryptographic functions and secure data transmission.
- Integration of Components: All components, including sensors, and communication modules, were integrated to ensure seamless operation and communication between the clients and the gateway.

Testing Phase

- Field Testing: The implemented system was subjected to field testing to simulate real-world operating conditions. This testing aimed to validate the effectiveness of the onboarding process and the overall system security.
- Security Evaluation: The security of the system was rigorously evaluated through penetration testing and vulnerability assessments conducted to identify any potential security flaws.
- Performance Analysis: Performance metrics such as response time were analyzed to assess the practical viability of the implemented solution.

1.3 Thesis Outline

Chapter 1 provided an initial overview of this thesis, accompanied by background information about the CERTIFY project. Chapter 2 provides a theoretical foundation by introducing key concepts. Chapter 3 presents the current state of research in the field and offers a discussion of it. Chapter 4 presents a generic design for a secure and lightweight onboarding process. In Chapter 5 the proposed design is implemented. In order to facilitate the process of setting up the development environment, a step-by-step guide is provided, along with an explanation of the key concepts of the code. Finally, Chapter 6 evaluates the design and implementation, followed by a conclusion in Chapter 7.

Chapter 2

Background

2.1 Internet of Things

The IoT describes a broad network of connected devices that can transmit and receive information. Central to IoT is its integration of digital and physical systems, which increases efficiency, precision, and economic advantages by improving automation and control. IoT devices cover a range of uses, from domestic appliances to industrial machinery, all linked via the internet to facilitate smooth communication and compatibility. Such connections are crucial for real-time data gathering and analysis, which promotes intelligent decision making and increases operational effectiveness in various fields [4].

2.2 Onboarding

Onboarding describes the essential procedures for safe incorporating a new device into an established network, ensuring its proper authentication and secure communication capabilities. This network layer onboarding is pivotal, as it encompasses the assignment of network credentials, the safeguarding of IoT devices against unauthorized access, and the defense of the network against risks posed by newly added devices. The National Institute of Standards and Technology (NIST) underscores the need to adopt reliable and scalable onboarding methods to securely manage IoT devices over their entire lifetime. Effective onboarding techniques involve the issuance of secure credentials, verification of device integrity, and ongoing secure management, all contributing to the improved security of both devices and the networks to which they connect [5].

2.3 Bootstrapping

Bootstrapping encompasses the foundational setup and configuration tasks that allow IoT devices to securely connect and interact within a network. This initial phase is essential to build trust and ensure proper authentication and authorization of devices prior to their interaction with other network elements and services of the network [5].

The bootstrapping process generally includes critical steps such as device registration with a registration authority, provisioning of credentials, and potentially, privilege escalation. It starts with a device submitting its credentials, like certificates or tokens, for verification by a registration authority. After successful verification, the device receives credentials which might initially carry restricted privileges. Through a mechanism called privilege escalation, devices may then gain expanded access rights that are essential for their designated functions within the IoT ecosystem. This methodical process is crucial for reducing security threats by adhering to the principle of least privilege [6].

Moreover, bootstrapping frequently incorporates strategies to ensure secure management of devices throughout their lifecycle, covering situations where devices may require reconfiguration or updates without jeopardizing network security. Effective lifecycle management is vital for preserving the integrity and security of the evolving IoT system [7].

2.4 True Random Number Generator

A True Random Number Generator (TRNG) is a device or system that produces numbers by exploiting unpredictable physical processes. Unlike deterministic systems like computers, which operate on fixed algorithms (Pseudo Random Number Generators or PRNGs), TRNGs depend on naturally random physical phenomena. These include quantum mechanical effects, thermal noise, and other environmental factors that cannot be predicted [8].

TRNGs are essential for high-security applications such as cryptography, where the unpredictability of encryption keys boosts security measures. The generated random numbers are utilized for crafting encryption keys, digital signatures, and securing communications. Since these numbers cannot be duplicated by any algorithm, they provide enhanced security against hacking compared to PRNGs [8].

Nevertheless, producing true randomness presents challenges. TRNGs must transform the analog unpredictability of physical processes into digital binary outputs, often necessitating advanced hardware and meticulous calibration to ensure the randomness is unbiased and not affected by external influences. Standards like those set by NIST are employed to assess the randomness quality, confirming that TRNG outputs adhere to specific security standards [9].

2.5 Rivest-Shamir-Adleman Encryption

The Rivest-Shamir-Adleman (RSA) encryption method is a cornerstone in the realm of public-key cryptography, extensively employed to protect the transmission of sensitive information over unsecured networks such as the Internet. Central to the RSA algorithm is the complex problem of factoring large prime numbers. This algorithm employs two keys: a public key, which is openly distributed, and a private key, which remains confidential with its owner. The public key is used to encrypt the data, whereas the private key decrypts them, ensuring that only the intended receiver can view the original content [10].

The solid architecture of the RSA algorithm has positioned it as a reliable benchmark in cryptographic standards, crucial for safeguarding the confidentiality, integrity, and authenticity of digital communications across diverse technological sectors. Even with the potential threats posed by quantum computing, RSA is still considered secure against traditional challenges when implemented with sufficiently long key sizes and correct cryptographic methods. The continued reliance on RSA's security is reinforced by persistent research and professional evaluations, indicating that with appropriate applications, RSA remains a vital mechanism for secure communication [11].

Chapter 3

Related Work

3.1 Artwork Tracking

3.1.1 Artwork Preservation

Artwork conservation is impacted by several factors, notably human activities and changes in the environment [12]. The main risks to the integrity of art involve elements such as temperature, humidity, exposure to light, pollutants, and microbial growth [13]. Temperature and humidity are often the most significant concerns [13]. Monitoring these factors is essential to preserve the quality and longevity of art pieces [14]. The protection of artworks is critical not only within the confines of museums or galleries but also during their transportation.

3.1.2 Artwork Transportation

For many years, artworks have been exhibited throughout the world, frequently being transported between different venues. The hazards involved in the movement of art pieces have been thoroughly investigated, leading to the creation of sophisticated packaging techniques and safety measures [15]. Currently, numerous companies specialize exclusively in art logistics, providing cutting-edge services such as shock-resistant and climate-controlled packaging [16][17]. Despite the rigorous testing of these solutions, the dependence on consumer trust by many companies emphasizes the practical effectiveness of these strategies. This dependency also highlights the opportunity to incorporate new technological advancements in monitoring systems to further improve the artwork transportation process.

3.1.3 Artwork Monitoring

Technological advances have facilitated deeper investigations of the impact of transportation on the integrity of artwork. Numerous studies have been dedicated to this topic. For example, a specific research used a small logging device to monitor the shocks and vibrations that artworks endure during transit [18]. Even with the application of advanced packaging methods and appropriate transport techniques, considerable amounts of shock and vibration were still observed in several shipments, suggesting that ongoing monitoring is essential to evaluate the condition of the artwork after transport.

Following these observations, a proposal was made for a real-time monitoring system that tracks the environmental and safety conditions affecting artworks [12]. This system, powered by a low-energy, cost-effective IoT node, enables comprehensive and continuous surveillance, with the capability to identify problems as they arise or even preemptively.

Building further on these ideas, another research introduced a proactive approach named PACT-ART, which leverages data mining and business process intelligence to predict potential risks in handling artworks [19]. This model aims to pinpoint potential issues and recommend preventive actions.

Enhancing proactive measures, Carchiolo et al. [20] have devised a framework that supports ongoing risk evaluation throughout the storage, handling, transportation, and exhibition phases of artworks, thus improving the overall safety management of art.

Chapter 4

Architecture and Design

This chapter introduces the architecture and design of the suggested solution in a generic way so that it is independent of hardware and software. To do so, the application scenario is described in Section 4.1, followed by an overview of the technical components in Section 4.2. After that, the suggested onboarding process is explained in Section 4.3. The subsequent chapter 5 will then demonstrate the implementation of the proposed design.

4.1 Application Scenario: Artwork Tracking

The proposed solution was designed for a specific scenario, that is, artwork tracking. To be more precise, the scenario involves the necessity of monitoring the environment during the transportation of artwork from one location to another. Depending on the circumstances, different environmental conditions must be monitored, such as temperature, humidity, vibration data, GPS coordinates, etc. to ensure the safety of the artwork. In order to monitor these environmental conditions, it is necessary to affix sensors to the artwork or to distribute sensors throughout the environment. It should be noted that the proposed solution is equally applicable to the monitoring of stationary artwork.

Reading the data from each sensor individually would not be efficient or user-friendly; therefore, the data need to be collected at a central point, called a gateway. To collect sensor data on a gateway, the sensors must first be connected to that gateway. However, different situations require different combinations of sensors; therefore, it is essential that sensors can be connected to the gateway independently of each other, based on the requirements. As discussed in [21], IoT devices have major security concerns due to the insufficient security measures implemented, as manufacturers prioritize speed over security. This thesis proposes a secure, lightweight, and user-friendly onboarding process to counteract this problem.

4.2 Technical Components

Prior to delving into the proposed solution, it is essential to introduce several key components. Figure 4.1 illustrates the application scenario described in the previous section. The scenario comprises two locations, two sensors, one gateway, and one truck as a transport medium. Furthermore, a cloud environment is included. Given that this thesis focuses on secure onboarding between the client and the gateway, the cloud will not be discussed in subsequent sections. However, for completeness, the cloud component was included in this Figure 4.1, as in most real-world applications, data will be uploaded to a cloud.

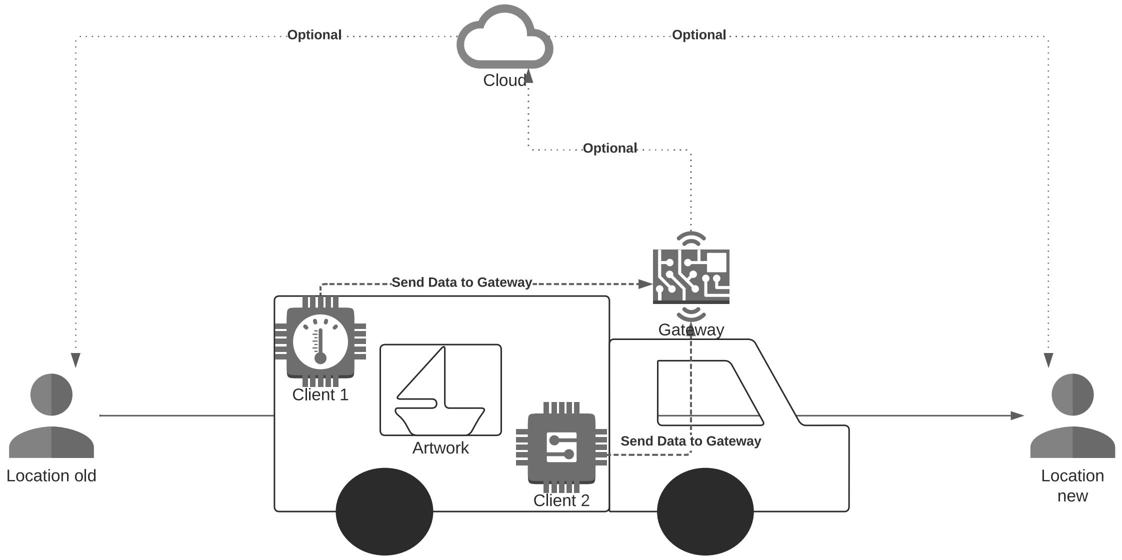

Figure 4.1: Overview of Components

4.2.1 Client

The clients, which are the sensors discussed in previous chapters, are small devices with limited computational power. Their purpose is to measure the environment and transmit data to the gateway. One client can measure several environmental conditions, but for the sake of simplicity, let us assume that one client measures just one aspect of the environment (e.g., temperature and humidity).

4.2.2 Gateway

In contrast, gateways are devices with greater computational capacity and the ability to collect, process, and forward data. Additionally, gateways can react to the data they receive, either in a basic manner (such as changing the color of the onboard LED or making a sound) or in a more complex way (such as sending push notifications).

4.2.3 Random Number Generation

The generation of secure keys depends on the utilization of the real generation of true random numbers. Unfortunately, not all IoT devices possess the hardware required to create true random numbers. In particular, smaller devices that are limited in size, cost, or power consumption, among others, often lack these capabilities [22]. It is therefore assumed that both devices have some form of hardware random number generator and are capable of generating true random numbers. In the event that a device lacks the capacity for true random number generation, a modular Trusted Platform Module (TPM) can be employed instead. It is also pertinent to note that the generated keys are not required to be stored. In accordance with the proposed design, keys are utilized on a single occasion until the session expires (reboot of the device in this case) and are subsequently generated anew for each onboarding.

4.3 Onboarding

Having introduced all the necessary components, the proposed onboarding process can now be discussed. The sequence diagram in Figure 4.2 serves as a visualization and is an exact representation of the description that follows.

The onboarding process is initiated by the gateway (after a human interaction). First, an RSA key must be generated. Although [23] states that RSA keys with a minimum length of 2048 bits are secure and [24] forecasts that RSA keys with a minimum length of 2048 bits will remain secure until 2030, this proposal employs keys with a length of 3072 bits to ensure continued security even after 2030. Following the generation of the key, the gateway continuously broadcasts a signal to let nearby clients know that it is ready for onboarding. The broadcast message is simply `Artwork Tracking Onboarding`.

The client initiates the onboarding process immediately following the start-up. Once the key pair has been created, the client begins scanning for the gateway's broadcast. Upon receiving the broadcast message `Artwork Tracking Onboarding`, the client responds with the message `Onboarding Request`. Upon receipt of the onboarding request, the gateway initiates a check of the device's eligibility for onboarding, utilizing the Media Access Control (MAC) address. Should the device in question pass the eligibility check, the gateway switches from broadcast to unicast communication. In the event that the device is not allowed to be onboard, the incident is logged appropriately and the request is dismissed.

The process is now entering the bootstrapping phase. This phase is essential and critical for secure onboarding of a device. In this phase, the session key (and sometimes other secret information) is exchanged. It is of the utmost importance that the information transferred in this phase is not leaked under any circumstances. In order to prevent attackers from reading or at least understanding the data sent between the gateway and the client, it must be encrypted. The data will be encrypted using asymmetric RSA encryption. In order to facilitate the exchange of keys, the gateway transmits its public key in the Privacy Enhanced Mail (PEM) format to the client. Upon receiving the public key, the client transmits its public key to the gateway in the PEM format.

Once the keys have been exchanged, the client must demonstrate that it is the legitimate owner of the private key. To this end, the gateway generates a random 128-bit challenge, encrypts it with the client's public key, and transmits it to the client. The client then decrypts the challenge with its own private key, encrypts it back with the public key of the gateway, and sends it back to the gateway. The gateway checks if the client decrypts the challenge correctly by decrypting and comparing the received challenge with the sent challenge. If the sent and received challenge are identical, it can be concluded that the client has proven ownership of the private key. At this juncture, the gateway is in a position to proceed with the main phase of the bootstrapping process, namely, the transmission of the secure session key. In a manner analogous to the challenge, a 128-bit session key is generated, encrypted with the RSA public key of the client, and transmitted to the client.

Upon receiving the encrypted session key, the client decrypts it with its own private key and establishes a new symmetric encrypted channel using the decrypted session key. From this point on, all communication between the client and the gateway is symmetrically encrypted. The client then begins the transmission of sensor data to the gateway. Given the application scenario, the gateway is not required to send additional data to the client following the successful transmission of the session key. Consequently, the gateway closes its sending channel and listens solely to the sensor data transmissions of the clients.

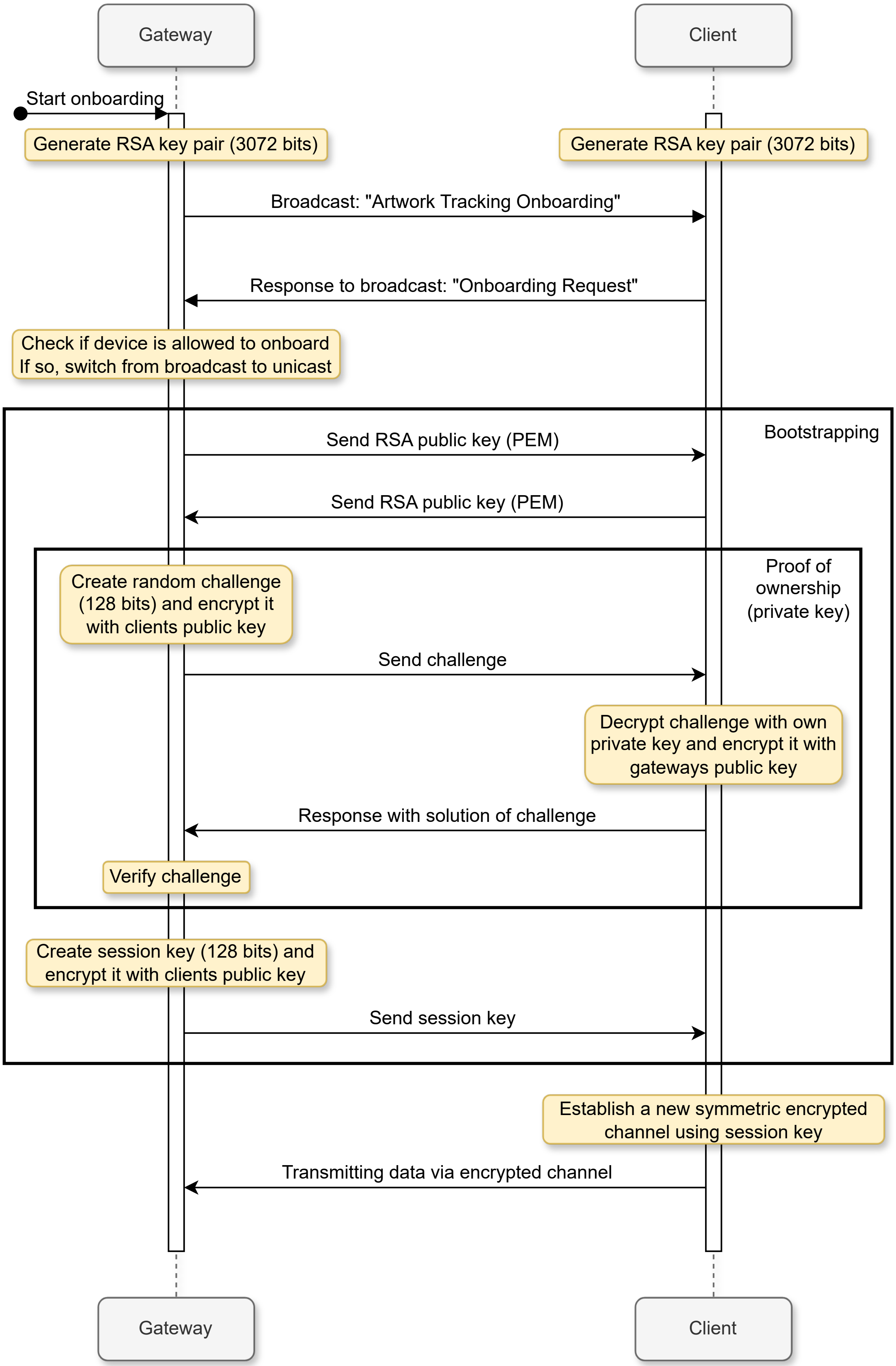

Figure 4.2: Onboarding Sequence Diagram

Chapter 5

Implementation

Having discussed the architecture and design, the implementation can now be introduced. It is important to note that this is merely one possible implementation of the proposed design and that some of the features utilized are only available on the specific board used for the implementation. However, the following implementation can be adapted depending on the needs. The complete code will be available on GitHub[1].

This chapter commences with an overview of the hardware and frameworks utilized in Section 5.1, followed by instructions on how to set up the development environment in Section 5.2 and how to secure ESP32-S3 in Section 5.3. Section 5.4 illustrates the connection of the sensors to the ESP32-S3 and the subsequent reading of the data from them. Once the fundamental concepts have been established, the primary implementation is presented in Section 5.5. Finally, in Section 5.6, this chapter is summarized.

5.1 Hardware and Frameworks

This section provides an overview of the hardware and the most important frameworks and libraries used to implement the solution.

5.1.1 ESP32-S3

The ESP32-S3 is a low-power system on a chip (SoC), based on a microprocessor. The SoC is composed of a high performance dual-core microprocessor (Xtensa 32-bit LX7), a low-power coprocessor, a Wi-Fi baseband, a Bluetooth LE baseband, a radio frequency module, and other peripherals. It has the power and storage capacity necessary to handle and process audio-visual data. The Xtensa LX7 processors are equipped to support digital signal processing for imaging and convolutional neural network processing, as well as digital signal processing for a variety of applications [25].

---

[1] https://github.com/secure-onboarding-iot

The ESP32-S3 comes with a sophisticated random number generation (RNG) system, central to which is a hardware RNG capable of generating true random numbers under certain conditions. The production of these numbers depends on the incorporation of physical noise samples into the RNG state. This incorporation requires either the activation of Wi-Fi or Bluetooth, or the activation of internal reference voltage noise. At startup, the bootloader seeds the RNG state with entropy by activating a nonradio frequency entropy source, thus guaranteeing the creation of true random numbers. Nevertheless, to maintain a steady output of such numbers, it is necessary to activate a hardware entropy source. Additionally, the ESP32-S3's RNG system features another entropy source that samples an asynchronous 8 MHz internal oscillator. This entropy source is always active and is continuously merged into the RNG state by the hardware. The comprehensive RNG system incorporated into the ESP32-S3 makes it a highly capable tool for applications that require secure and reliable random number generation [26].

After careful consideration and comparison, the ESP32-S3 board was considered suitable for this project.

5.1.2 Arduino

The Arduino platform, which emerged in the early 2000s, has profoundly impacted the community of electronic enthusiasts and educational professionals. The core of this platform is the Arduino framework, which includes a collection of software libraries and hardware specifications aimed at enabling the application of electronics in various interdisciplinary projects [27]. The primary advantages of the Arduino platform are its user-friendliness for novices and its adaptability, which makes it a powerful tool for experienced users [28].

At the core of interaction with Arduino hardware, among others, is the Arduino Integrated Development Environment (IDE), a cross-platform tool developed using functions from Wiring, based on Processing. The Arduino IDE offers a simplified setup for coding and deploying programs on microcontroller boards [29]. It supports C and C++ and provides a suite of libraries and configurations that simplify complex tasks into easier procedures, making microcontroller programming more accessible [30].

Furthermore, the integration of the Arduino framework with the IDE enables users to quickly create digital devices that can interact with their surroundings using sensors and actuators. This ease of integration allows for the development of projects ranging from basic home appliances to intricate scientific tools [31].

5.1.3 ESP-NOW

ESP-NOW is a connectionless communication protocol developed by Espressif Systems. It is designed to allow devices to communicate directly without the need for a Wi-Fi network. This protocol is particularly useful for applications where quick responses and low-power control are required, as evidenced by [32].

Its low power consumption allows for extended battery life as demonstrated by [33]. The protocol supports various control modes and device types, ensuring flexible and efficient device pairing. Furthermore, ESP-NOW's simplified data-link layer reduces transmission delays, coexists with Wi-Fi and Bluetooth LE, and offers enhanced security through ECDH and AES algorithms [34].

ESP-NOW offers several advantages over traditional Wi-Fi protocols, particularly in the context of IoT applications. The comparative performance study [35] found that ESP-NOW outperforms Wi-Fi in several key performance indicators, including maximum range, transmission speed, latency, power consumption and resistance to obstructions. The study highlighted that ESP-NOW is more efficient in terms of power consumption and offers faster data transmission, making it particularly suitable for real-time applications where speed and energy efficiency are crucial.

Despite its advantages, ESP-NOW also has some limitations. The size of the data packet is limited to 250 bytes [36]. Although this is sufficient for most sensor data, the RSA public keys and encrypted data via RSA do not fit within these 250 bytes. However, this is crucial for the bootstrapping process, necessitating the development of a solution to address this issue. A potential workaround is presented in a subsequent chapter. Furthermore, ESP-NOW is limited to a maximum of six encrypted peers [36], which should not be a significant limitation for the artwork tracking use case.

In terms of security, ESP-NOW employs the CCMP method for encryption, which is described in IEEE 802.11-2012 [37]. This method involves maintaining a Primary Master Key (PMK) and several Local Master Keys (LMK), each 16 bytes in length, to safeguard the transmitted data. The PMK is used to encrypt the LMK with the AES-128 algorithm, while the LKM is used to encrypt the vendor-specific action frame within the CCMP method [36].

5.1.4 Other components

In the previous sections, all essential components required for implementation were introduced. To enhance the realism of the application, additional components will be employed for implementation. These are optional, and it is up to the reader to decide whether they wish to include these components.

To enable the measurement of certain environmental data, the DHT11, which is a temperature and humidity sensor, was chosen. In addition, a button on the gateway side is necessary to initiate the onboarding process. Finally, to obtain feedback when the serial monitor is not connected, the built-in RGB LED [38] will be used.

To focus on the primary objective of this thesis, namely secure onboarding of IoT devices, no additional components were added. In a practical application, additional components may be included, such as sensors to measure further environmental data, a GPS and 5G module to locate the artwork and facilitate real-time data transmission.

5.2 Setup development environment

Prior to commencing the implementation, it is necessary to establish an appropriate development environment. Although the Arduino IDE can be used for development purposes, it lacks the required security features. To enable Secure Boot V2 and flash encryption, it is necessary to employ the Espressif IoT Development Framework (ESP-IDF). The integration of Arduino with the ESP-IDF serves as a bridge, allowing users to leverage the simplicity of the Arduino platform while taking advantage of the comprehensive features of the ESP-IDF. This process is facilitated by the Arduino Lib Builder, a tool that customizes the default settings provided by Espressif for use in the Arduino IDE. The following section presents a summary of the official installation instructions for the Arduino ESP32 Installation Guide [39].

1. Preparation: Ensure that ESP-IDF[2] is installed, compatible with the latest Arduino Core for ESP32. Start by setting up a basic ESP-IDF project.

2. Repository setup: In the project directory, create a `components` folder and clone the Arduino-ESP32[3] repository in it. Recursively initialize and update submodules.

3. Configuration:
    - Run `idf.py menuconfig`.
    - Navigate to `Arduino Configuration`
    - Turn on `Autostart Arduino setup and loop on boot`

4. Build and Flash:
    - Modify main code in `main.ino` to include Arduino functions.
    - Use `idf.py -p <serial-port> flash monitor` to build, upload, and monitor the application.

5.3 Device hardening

As proposed by [40], several measures must be taken to enhance the security of an ESP32-S3 board. In the context of the use case, the most relevant measures are Secure Boot V2 and flash encryption. In the following sections, both measures are discussed.

---

[2] https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32/get-started/index.html#manual-installation

[3] https://github.com/espressif/arduino-esp32

5.3.1 Secure Boot V2

Secure Boot V2 on the ESP32-S3 is a security feature designed to ensure that only authenticated software can run on the device. RSA-PSS, a robust signature algorithm, is used to verify the integrity and authenticity of the bootloader and application software before execution. This process involves several key steps and configurations to protect the device from unauthorized code execution [41].

The Secure Boot process commences with the authentication of the bootloader's signature against a public key, the corresponding private key of which is securely stored and never exposed. This public key is embedded within the device's eFuses, a secure memory area, to prevent tampering or unauthorized modifications. The ESP32-S3 then compares the SHA-256 hash of the public key against those stored in the eFuses to confirm the bootloader's authenticity. If the public key matches, the device verifies the signature of the application software using the same RSA-PSS method. In the event that any of these verifications fails, the device will not boot the unverified software, thus protecting against potential security threats [41].

To implement Secure Boot V2, developers must first enable it through the ESP-IDF project configuration menu and select the RSA option for the app signing scheme. This configuration necessitates the generation of a secure RSA key pair, wherein the private key is employed to sign the bootloader and application, while the public key is incorporated into the eFuses of the ESP32-S3. Developers should utilize high-quality entropy sources for key generation to ensure the robustness of the keys [41].

In addition to setting up Secure Boot, developers can configure the device to prevent further modifications by selecting options that restrict UART ROM download modes and other similar configurations, which enhance the security by disallowing changes after initial programming. These steps are critical to maintaining the integrity of the device in production environments, where security is paramount [41].

The efficacy of Secure Boot V2 depends on maintaining the confidentiality of the RSA private key and the integrity of the public key stored in the device's eFuses. This configuration provides a robust defense against unauthorized firmware modifications, thus securing the device from a multitude of attack vectors that target the software integrity of embedded devices [41].

5.3.2 Flash Encryption

The flash encryption feature on the ESP32-S3 is a critical security measure designed to protect data stored in the off-chip flash memory of the device. When enabled, this feature automatically encrypts the firmware that is flashed as plain text during the initial boot process. This encryption is hardware-based, utilizing a key stored within the device's eFuses, rendering physical extraction of the flash content an ineffective method for retrieving the data. The process entails verifying and setting various eFuse configurations to manage encryption keys and settings, ensuring robust protection against unauthorized access [42].

5.4 Connecting Hardware

Figure 5.1 shows the configuration of the gateway. The gateway consists of an ESP32-S3 board on the left and a blue three-pin button on the right side. The button facing the reader, the left pin is connected to power, the middle pin is connected to General Purpose Input/Output (GPIO) 4 and the right pin is connected to ground. The middle pin can be connected to any digital GPIO pin, as long as the code is adjusted.

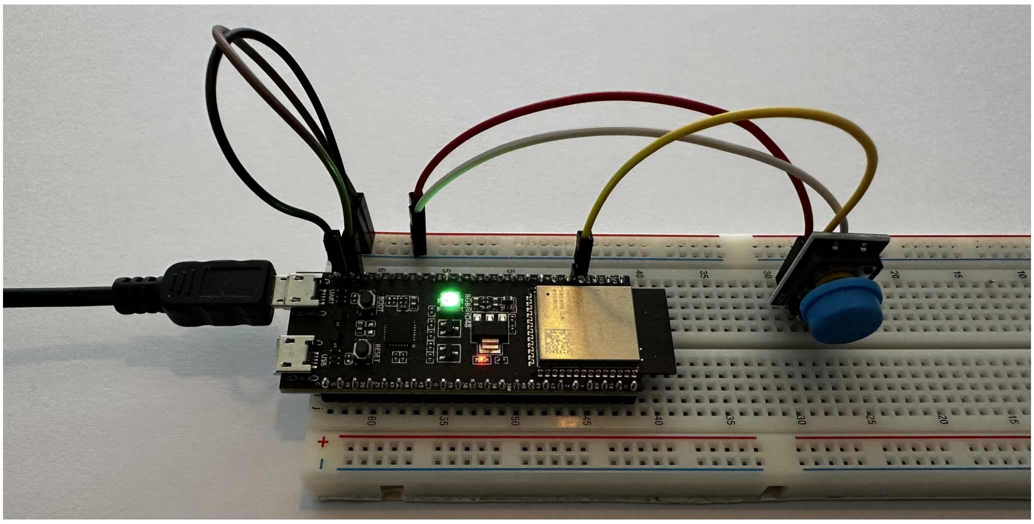

Figure 5.1: [Gateway] ESP32-S3 Board With a Connected Button

Figure 5.2 shows the client configuration. The client consists of an ESP32-S3 board on the left and a blue four-pin DHT11 sensor, that measures temperature and humidity, on the right side. The DHT11 sensor facing the reader, the first pin from the left is connected to power, the second pin is connected to GPIO 14, the third pin is not connected and remains empty, and the fourth pin is connected to ground. Here again, the second pin from the left can be connected to any digital GPIO pin, as long as the code is adjusted.

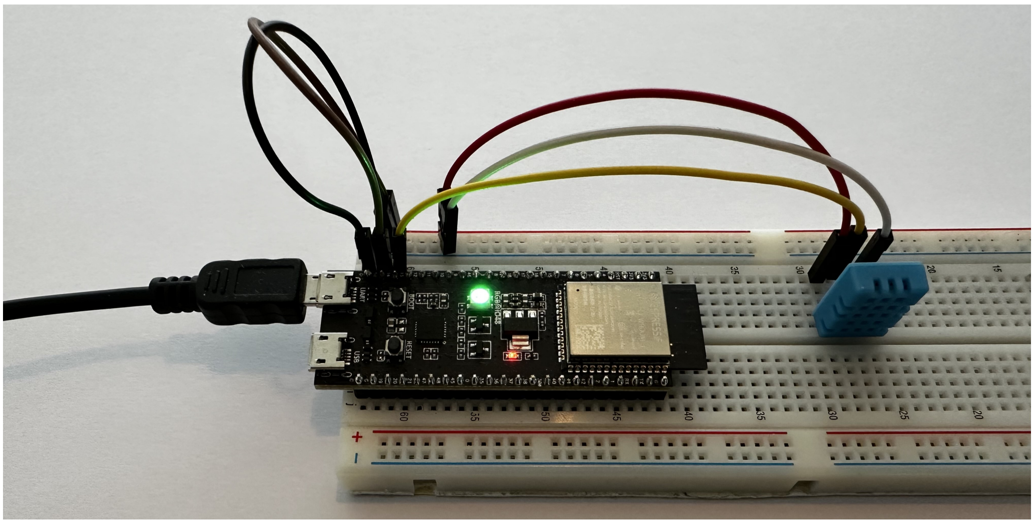

Figure 5.2: [Client] ESP32-S3 Board With a Connected DHT11 Sensor

5.5 Onboarding

This chapter introduces the programming aspect of the project. Given the limited space available, it is not possible to present the entire code in detail. Instead, each subsequent section will focus on a principal aspect of the project. As the gateway and the client share a substantial code base, the provided code and explanation are applicable to both unless otherwise specified.

5.5.1 Introduction

This project used the Arduino framework, as the objective was to develop an effective and streamlined implementation of the onboarding process outlined in Section 4.3. As previously discussed in Section 5.1.2, the Arduino framework encompasses a suite of software libraries and hardware specifications that streamline the overall implementation process, allowing developers to focus on core features and avoid the complexities associated with platform-specific aspects.

As discussed in Section 5.1.1, the ESP32-S3 is an appropriate board for this project. To simplify the process, two identical boards were employed for the gateway and the client. According to the specific requirements, it is possible to utilize an alternative board for either the gateway or the client, provided that the selected board is capable of supporting ESP-NOW and possesses the necessary capability for true random generation, as previously discussed in Section 4.2.3.

5.5.2 Initialization

Prior to the beginning of the programming process, a series of design decisions must be made. Among these are the definition of constants, global variables, and structs. This section will focus on the most crucial of these decisions.

Constants

The onboarding process is comprised of a series of distinct phases that must be accurately documented during the implementation phase. The aforementioned phases can be defined as shown in Listing 5.1.

```
#define DEFAULT 0
#define BROADCASTING 1
#define KEY_EXCHANGE 2
#define CHALLENGING 3
#define SEND_SESSION_KEY 4
```

Listing 5.1: Constants, defining the state of onboarding

The code presented in Listing 5.1 originates from the gateway. Although the meaning of the state itself is identical, for the sake of clarity, the `Broadcasting` phase is designated as `Searching` and `SEND_SESSION_KEY` is renamed `RECEIVE_SESSION_KEY` on the client.

In the default phase, the client transmits the sensor data to the gateway. This phase is the desired phase and is reached upon successful onboarding. The Broadcast / Searching mode represents the initial stage of the onboarding process. The gateway initiates the transmission of a signal indicating the client's search for a specific entity. During key exchange, the public keys of the RSA algorithm are exchanged. In the challenging phase, the client must prove to the gateway that it is the owner of the RSA private key. Finally, in the Send/Receive Session Key phase, the gateway generates a key for symmetric encryption and transmits it to the client.

Upon initialization, the client initiates the broadcast phase and attempts to establish a connection with the gateway without user intervention. In contrast, the gateway commences in its default state, as it is crucial that the user retains control over the initiation of the onboarding process. To facilitate the storage of all pertinent information, a global variable designated as `STATUS` is defined.

Structs

Depending on the sensors used, different data need to be transmitted. In this project, the temperature and humidity is measured. Additionally, the transmitted data must include an additional value, namely, a threshold indicating the point at which the artwork is in danger. To effectively process these data, a struct must be created as shown in Listing 5.2.

```c
typedef struct measure {
  int temperature;
  int humidity;
  int temperatureAlarm = 25;
  int humidityAlarm = 50;
} measure;
```

Listing 5.2: Measure Struct

As previously outlined in Section 5.1.3, ESP-NOW is constrained by a data limit of 250 bytes. In particular, during the bootstrapping process, the data that must be transmitted exceed the 250-byte limit. In order to be able to send more than 250 bytes, it is necessary to create a data structure that contains several structs. For the sake of enhanced readability, the code and the accompanying explanation have been relocated to Section 5.5.6.

5.5.3 Interacting with Sensors and LEDs

Although it is not the primary focus of this thesis, the utilization of authentic data from genuine sensors is more closely aligned with a real-world application and use case. It is assumed that the configuration described in Section 5.4 has been completed.

Reading Data from DHT11[4]

---

[4] Only for client

To read the temperature and humidity, the `DHT11`[5] library will be used. The aforementioned library facilitates the reading of data as shown in Listing 5.3.

```
measure m;
int temperature = dht11.readTemperature();
int humidity = dht11.readHumidity();
```

Listing 5.3: Reading Temperature and Humidity from DHT11 Sensor

It is possible that the sensor returns an error. This can be verified by the procedure described in Listing 5.4.

```
if (temperature == DHT11::ERROR_TIMEOUT || temperature == DHT11::ERROR_CHECKSUM) Serial.println("Temperature Reading Error: " + DHT11::getErrorString(temperature));
if (humidity == DHT11::ERROR_TIMEOUT || humidity == DHT11::ERROR_CHECKSUM) Serial.println("Humidity Reading Error: " + DHT11::getErrorString(humidity));
```

Listing 5.4: Error Handling DHT11 Sensor

Upon successful verification, the data can then be incorporated into the struct defined in Section 5.5.2 and transmitted.

```
1 m.temperature = temperature;
2 m.humidity = humidity;
3 measuresQueue.enqueue(m);
4 measure m2 = measuresQueue.peek();
5 esp_now_send(gatewayMAC, (uint8_t*)&m2, sizeof(m2));
```

Listing 5.5: Using the Measure Struct

The necessity of a queue, as illustrated in lines 3 and 4, is explained in Section 5.5.8.

Using the built-in RGB LED of ESP32-S3

The ESP32-S3 is equipped with an integrated RGB LED, which is useful to provide a straightforward indication to the user. In this example, the color of the LEDs will indicate the phase of the onboarding process and also indicate errors and measures that exceed a certain threshold value, as introduced in Section 5.5.2. The colors have the following meanings:

- White: RSA key pair is being generated.
- Blue: The onboarding process is ongoing.
- Green: The onboarding process is completed, and the transmission of measurement data has started.
- Red: Data transmission failed. In addition, for the gateway, the measured value from the client exceeds the threshold.

---

[5] https://github.com/dhrubasaha08/DHT11

The LED can be controlled as shown in Listing 5.6.

```
Adafruit_NeoPixel pixels(1, 48, NEO_GRB + NEO_KHZ800);
pixels.begin();
pixels.setPixelColor(0, pixels.Color(17, 17, 17));
pixels.show();
```

Listing 5.6: Controlling the built-in RGB LED

Listing 5.7 illustrates a specific instance of the LED's utilization within the project.

```
  if (m.temperature >= m.temperatureAlarm || m.humidity >= m.humidityAlarm || m.temperature > 250 || m.humidity > 250) {
    pixels.setPixelColor(0, pixels.Color(17, 0, 0));
    if (m.temperature > 250) m.temperature = -999;
    if (m.humidity > 250) m.humidity = -999;
  } else {
    pixels.setPixelColor(0, pixels.Color(0, 17, 0));
  }

  Serial.println("Temperature: " + (String)m.temperature + " °C");
  Serial.println("Humidity: " + (String)m.humidity + "%");
  pixels.show();
```

Listing 5.7: Example for LED Usage on Gateway Side

5.5.4 Encryption

This section presents a detailed account of the implementation of cryptographic functions, with a particular focus on the generation of AES keys, the formation of RSA key pairs, and the RSA encryption and decryption processes.

AES Key Generation

The process commences with the establishment of contexts for the entropy collection (`mbedtls_entropy_context`) and the counter mode deterministic random bit generator (CTR_DRBG). These are fundamental for the generation of cryptographic quality random numbers. The entropy function collects environmental noise as a basis for randomness, which seeds the deterministic random bit generator (DRBG). The DRBG, personalized with a unique device identifier, then generates a 128-bit AES key. Listing 5.8 demonstrates this setup and key generation.

```
  mbedtls_entropy_context entropy;
  mbedtls_entropy_init(&entropy);

  mbedtls_ctr_drbg_context ctr_drbg;
  mbedtls_ctr_drbg_init(&ctr_drbg);

  uint32_t randomNumber = esp_random();
  char personalization[11];
  sprintf(personalization, "0x%08X", randomNumber);
  mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (const unsigned char*)personalization, strlen(personalization));

```

```
unsigned char key[16];
mbedtls_ctr_drbg_random(&ctr_drbg, key, sizeof(key));
```

Listing 5.8: AES Key Generation

RSA Key Generation

A similar pattern is observed in the generation of RSA keys, where the entropy and DRBG contexts are initialized. Subsequently, the RSA context (`mbedtls_rsa_context`) is configured to generate a key pair with a specified key length and public exponent. The generation of RSA keys is a computationally intensive process that is heavily based on DRBG for randomness.

```
mbedtls_rsa_init(&rsa, MBEDTLS_RSA_PKCS_V15, 0);
int ret = mbedtls_rsa_gen_key(&rsa, mbedtls_ctr_drbg_random, &ctr_drbg
  , RSA_KEY_LENGTH, RSA_EXPONENT);
```

Listing 5.9: RSA Key Pair Generation

RSA Encryption

The encryption process uses the RSA algorithm to ensure data security. Initially, a public key container is prepared and the public key is parsed from a PEM-formatted string. Once the parsing process has been successfully completed, the data is encrypted using the `mbedtls_pk_encrypt` function. This function requires the input of the public key, the data to be encrypted, and a DRBG for randomness generation. Subsequently, the encrypted data is converted into a hexadecimal string for transmission or storage. This is reflected in Listing 5.10.

```
mbedtls_pk_context pk;
mbedtls_pk_init(&pk);

if (mbedtls_pk_parse_public_key(&pk, (const unsigned char*)pem_peer.
  c_str(), pem_peer.length() + 1) != 0) {
   mbedtls_pk_free(&pk);
   return "";
}

unsigned char output[1024];
size_t olen;

mbedtls_ctr_drbg_context ctr_drbg;
mbedtls_entropy_context entropy;
mbedtls_entropy_init(&entropy);
mbedtls_ctr_drbg_init(&ctr_drbg);
uint32_t randomNumber = esp_random();
char personalization[11];
sprintf(personalization, "0x%08X", randomNumber);
mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (
  const unsigned char*)personalization, strlen(personalization));

int ret = mbedtls_pk_encrypt(&pk, (const unsigned char*)data.c_str(),
  data.length(), output, &olen, sizeof(output), mbedtls_ctr_drbg_random
  , &ctr_drbg);
```

Listing 5.10: RSA Encryption

RSA Decryption

Prior to the decryption process, the RSA private key is validated. Subsequently, the encrypted data, presented in hexadecimal format, is transformed into its binary equivalent. The `mbedtls_rsa_pkcs1_decrypt` function is employed to decrypt the binary data using the private key. Furthermore, this function is dependent on the DRBG for secure operations. Subsequently, the binary data is converted back into a readable string format, thereby indicating the successful retrieval of the original data. The relevant code is presented in Listing 5.11.

```
unsigned char encData[1024];
size_t encIndex = 0;

for (size_t i = 0; i < encHex.length(); i += 2) {
  sscanf(encHex.c_str() + i, "%02X", &encData[encIndex++]);
}

unsigned char output[1024];
size_t olen;

mbedtls_ctr_drbg_context ctr_drbg;
mbedtls_entropy_context entropy;
mbedtls_entropy_init(&entropy);
mbedtls_ctr_drbg_init(&ctr_drbg);

uint32_t randomNumber = esp_random();
char personalization[11];
sprintf(personalization, "0x%08X", randomNumber);
mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (
  const unsigned char*)personalization, strlen(personalization));

int ret = mbedtls_rsa_pkcs1_decrypt(&rsa, mbedtls_ctr_drbg_random, &
  ctr_drbg, MBEDTLS_RSA_PRIVATE, &olen, encData, output, sizeof(output)
  );
```

Listing 5.11: RSA Decryption

5.5.5 ESP-NOW

As the communication between the client and the gateway is carried out entirely through ESP-NOW, this section will introduce and explain the individual components of the ESP-NOW protocol in a general manner. Section 5.5.7 will then present concrete examples of the usage of ESP-NOW.

Initialization of ESP-NOW

In order to utilize ESP-NOW, it is necessary first to initialize the Wi-Fi driver in station mode. This is an essential step for ESP-NOW communications. Subsequently, the ESP-NOW module is initialized via the `esp_now_init()` function. This configuration is crucial for preparing the ESP32-S3 to transmit and receive ESP-NOW messages.

```
WiFi.mode(WIFI_STA);
if (esp_now_init() != ESP_OK) {
  Serial.println("Error initializing ESP-NOW. Restarting...");
  delay(3000);
  ESP.restart();
}
```

Listing 5.12: Initialization of ESP-NOW

Creating a Peer

Peers are devices that can communicate with each other using ESP-NOW. In order to add a peer, it is necessary to define its MAC address (or to use the broadcast MAC address) and any additional optional parameters, such as the Wi-Fi channel and encryption preferences. Once this has been done, the peer is added to the ESP-NOW peer list `esp_now_add_peer()` function.

```
esp_now_peer_info_t peerInfo;
memset(&peerInfo, 0, sizeof(peerInfo));
memcpy(peerInfo.peer_addr, peerAddress, 6);
memcpy(&peerInfo.lmk, localMaster, 16);
peerInfo.channel = 0;
peerInfo.encrypt = true;
esp_now_add_peer(&peerInfo);
```

Listing 5.13: Creation of a Peer

Sending Data

The `esp_now_send()` function transmits data to a specified peer. Proper error handling is crucial to ensure reliability in data transmission.

```
uint8_t data[] = { 'H', 'e', 'l', 'l', 'o' };
esp_err_t result = esp_now_send(peerAddress, data, sizeof(data));
if (result != ESP_OK) {
  Serial.println("Error sending the data");
}
```

Listing 5.14: Sending Data using ESP-NOW

Sender and Receiver Callbacks

Callbacks in ESP-NOW are used to handle events such as data sent and data received. These functions are registered right after the initialization of ESP-NOW, and they help monitor the status of the sent data and processing the received data.

```
void onDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  Serial.print("Last Packet Send Status: ");
  if (status == ESP_NOW_SEND_SUCCESS) {
    Serial.println("Delivery Success");
  } else {
    Serial.println("Delivery Fail");
  }
}

```

```
void onDataRecv(const uint8_t *mac_addr, const uint8_t *data, int
    data_len) {
  Serial.print("Bytes received: ");
  Serial.println(data_len);
}

esp_now_register_send_cb(onDataSent);
esp_now_register_recv_cb(onDataRecv);
```

Listing 5.15: Sender and Receiver Callbacks

Limitations

Using ESP-NOW, only 250 bytes of data can be sent. For the onboarding process, especially key exchanges, more than 250 bytes are needed. Section 5.5.6 proposes a solution that includes splitting the message.

5.5.6 Sending and Receiving Long Messages

As previously stated in Section 5.5.5, ESP-NOW has a limitation of 250 bytes of data that can be sent during a single transmission. The primary issue arises during the RSA public key exchange, where the keys can be up to 3072 bits, which is equivalent to 384 bytes. Moreover, there may be instances where a client is required to transmit more than 250 bytes of measurement data, although this would be a rare occurrence.

Message Structuring

The handling of long messages in ESP-NOW begins with the struct `message`, which serves to structure the data for transportation. This struct is designed to carry a segment of data, along with metadata that is used to manage the segmentation and reassembly process. The metadata includes a unique message ID (`id`), the sequence number of the current packet (`count`), and the total number of packets (`total`). The field designated as `data` is where the actual payload of the segment is located.

```
typedef struct message {
  char id[37];
  byte count;
  byte total;
  char data[MAX_DATA_SIZE];
} message;
```

Listing 5.16: Struct for handling splitted messages

Sending Long Messages

Prior to the transmission of data, the `sendLongMessage` function first calculates the total number of messages required, based on the maximum data size allowed by ESP-NOW. Subsequently, a unique identifier is assigned to each message. This identifier ensures that the receiving end can distinguish between segments belonging to the same original message. For each segment, the function adjusts the length to fit within the packet size limit, populates the structure, and sends the packet using the `esp_now_send()` function.

```
void sendLongMessage(const char* input_data, const uint8_t* macAddr) {
  int total_messages = (strlen(input_data) + MAX_DATA_SIZE - 1) /
   MAX_DATA_SIZE;
  int attempts = 1;
  char buffer[37];
  sprintf(buffer, "%u", esp_random());

  for (int i = 0; i < total_messages; i++) {
    message msg;
    strncpy(msg.id, buffer, sizeof(msg.id));
    msg.count = i;
    msg.total = total_messages;

    int length = strlen(input_data) - i * MAX_DATA_SIZE;
    if (length > MAX_DATA_SIZE) length = MAX_DATA_SIZE;
    strncpy(msg.data, &input_data[i * MAX_DATA_SIZE], length);
    if (length < MAX_DATA_SIZE) msg.data[length] = '\0';

    esp_err_t result = esp_now_send(macAddr, (const uint8_t*)&msg,
   sizeof(msg));

    if (result != ESP_OK) {
      if (attempts++ >= 3) {
        Serial.println("Could not send long message. Abbort...");
        reset();
        break;
      }
      --i;
    } else {
      attempts = 1;
    }
  }
}
```

Listing 5.17: Sending Long Messages

Receiving and Reassembling Messages

Upon reception, the `receiveLongMessage` function takes the incoming data packet and casts it to a `message` struct. Subsequently, the system searches for an existing record of a multipart message with the same ID or, in the absence of such a record, initializes a new one. This record keeping process is facilitated by the `MessageRec` struct, which stores each received part, as well as the total number of parts expected and the number received so far.

```
String receiveLongMessage(const uint8_t* macAddr, const uint8_t* data,
    int len) {
  message* msg = (message*)data;

  MessageRec* fullMessage = NULL;
  for (auto& message : messages) {
    if (strcmp(message.id, msg->id) == 0) {
      fullMessage = &message;
      break;
    } else if (message.received == 0) {
      strcpy(message.id, msg->id);
```

```
      message.total = msg->total;
      fullMessage = &message;
      break;
    }
  }

  if (fullMessage == NULL) return "";

  strcpy(fullMessage->parts[msg->count].data, msg->data);
  fullMessage->parts[msg->count].index = msg->count;
  fullMessage->received++;

  if (fullMessage->received == fullMessage->total) {
    String fullMessageStr = "";
    for (int i = 0; i < fullMessage->total; i++) fullMessageStr += fullMessage->parts[i].data;

    fullMessage->received = 0;
    return fullMessageStr;
  }

  return "";
}
```

Listing 5.18: Receiving and Reassembling Long Messages

Error Handling and Flow Control

Error handling is crucial during the transmission of message segments. If a transmission fails, the function will retry sending the segment a specified number of times before aborting the process. This ensures that temporary issues do not permanently disrupt the message flow.

5.5.7 Bootstrapping

During the bootstrapping phase, a considerable number of messages are transmitted in both directions. The sequence of messages is of paramount importance and must be strictly adhered to, as each step in the bootstrapping process is contingent upon the preceding step. The nature of the bootstrapping process determines the required actions.

As previously outlined in Section 5.5.5, ESP-NOW offers an interface for callback functions that can be registered dynamically during run-time. This feature was employed in the development process. Consequently, instead of employing a single, extensive, and opaque callback function comprising numerous if statements for each state, a distinct callback function was devised for each state. The active callback function is registered during runtime, dependent on the state. Listing 5.19 illustrates the callback function associated with the receipt of the public key, followed by a supplementary method to transmit the subsequent message.

```
void receivePublicKey(const uint8_t* macAddr, const uint8_t* data, int dataLen) {
```

```
  if (!deviceAllowed(macAddr) || memcmp(clientMAC, macAddr, 6) != 0) return;

  Serial.println("Receiving Public Key from " + formatMacAddress(macAddr));
  String message = receiveLongMessage(macAddr, data, dataLen);

  if (!message.equals("")) {
    pem_peer = message;
    sendChallenge(macAddr);
  }
}

void sendChallenge(const uint8_t* macAddr) {
  STATE = CHALLENGING;
  esp_now_register_recv_cb(receiveChallengeResponse);
  Serial.println("Sending Challenge to " + formatMacAddress(macAddr) + " started.");
  challengePhrase = generateAESKey();
  sendLongMessage(encryptRSA(challengePhrase).c_str(), macAddr);
}
```

Listing 5.19: Receiving Public Key and Sending Challenge (Gateway)

On the client side, the transmission is handled in a manner that is analogous to that described above. Listing 5.20 illustrates the manner in which the client handles the incoming challenge, transmits it to the gateway, and establishes the new callback function for subsequent transmission.

```
void receiveChallenge(const uint8_t* macAddr, const uint8_t* data, int dataLen) {
  if (memcmp(gatewayMAC, macAddr, 6) != 0) return;
  Serial.println("Receiving Challenge from " + formatMacAddress(macAddr));
  String message = receiveLongMessage(macAddr, data, dataLen);
  if (!message.equals("")) sendChallenge(message, macAddr);
}

void sendChallenge(String data, const uint8_t* macAddr) {
  STATE = CHALLENGING;
  esp_now_register_recv_cb(receiveSessionKey);
  Serial.println("Sending Challenge Solution to " + formatMacAddress(macAddr));

  sendLongMessage(encryptRSA(decryptRSA(data)).c_str(), macAddr);
}
```

Listing 5.20: Receiving Challenge and Sending Solution of Challenge (Client)

The remaining stages of the bootstrapping process are handled in a comparable manner. As the implementation is in accordance with the proposed design in Section 4.3 and Figure 4.2, a further explanation of the code and the process has been omitted.

Once the bootstrapping process is complete, the client and gateway terminate the communication channel and establish a new encrypted communication channel. As the onboarding process is concluded with the final step of the bootstrapping, the gateway removes the send callback function from its registration list and the client removes the receive callback function from its registration list. From this point onward, communication is unidirectional.

5.5.8 Error Handling

During the implementation phase, a number of techniques were developed for error handling. This section presents the most significant and effective of these techniques.

Check if Onboarding is Stuck

A procedure was implemented to determine whether the onboarding process was progressing or encountering difficulties across all stages. Following the initial handshake, the program commences the counting of elapsed seconds. In the event that the onboarding process is still ongoing for a period exceeding 60 seconds, the board will be reset. Using the built-in loop function of Arduino, the implementation is simple as shown in Listing 5.21.

```
  if (STATE != DEFAULT) {
    sleep(3);
    if (++time_elapsed >= 20) reset();
  }
```

Listing 5.21: Checking if Onboarding is Stuck

Using RGB LED as Medium

As previously stated in Section 5.5.3, the integrated RGB LED can be used to transmit different error messages to the user. At this time, only one type of error results in a red color, namely, if the send fails on the client site. This can be readily expanded to accommodate different error types, with each error having a different color. Listing 5.22 illustrates the implementation of this feature.

```
void sendHandler(const uint8_t* macAddr, esp_now_send_status_t status) {
  if (status == ESP_NOW_SEND_FAIL) {
    Serial.println("Package sent to " + formatMacAddress(macAddr) + "
   FAILED ");
    pixels.setPixelColor(0, pixels.Color(17, 0, 0));
  } else {
    pixels.setPixelColor(0, pixels.Color(0, 17, 0));
    if (STATE == DEFAULT) {
      messageQueue.dequeue();
      if (!messageQueue.isEmpty()) {
        measure m = messageQueue.peek();
        esp_now_send(gatewayMAC, (uint8_t*)&m, sizeof(m));
      }
    }
  }
  pixels.show();
}
```

Listing 5.22: Using built-in RGB LED and Queuing System

Queue System to Overcome Temporary Signal Loss

Finally, a queueing system was implemented on the client side to overcome temporary signal loss. After measuring data, the measure struct is placed into the queue using the `enqueue()` function. Subsequently, the function `peak()` is used to retrieve the first element of the queue without deleting it. The first element is then attempted to be sent. Upon successful transmission, the `dequeue()` function is executed to delete the first element (which was just transmitted) from the queue. This process is repeated until the queue is empty, as shown in Listing 5.22. If the queue is full, which would indicate that the signal has been lost for a period of five minutes with the default settings, the board resets itself, and the onboarding process must be repeated.

Listing 5.23 illustrates the implementation of the queue system. This code is located in a separate file designated `MessageQueue.h` which is imported into the client code as a library.

```
#ifndef MESSAGE_QUEUE_H
#define MESSAGE_QUEUE_H

typedef struct measure {
  int temperature;
  int humidity;
  int temperatureAlarm = 25;
  int humidityAlarm = 50;
} measure;

class MessageQueue {
private:
  measure *queueArray;
  int capacity;
  int front;
  int rear;
  int count;

public:
  MessageQueue(int size = 100) {
    capacity = size;
    queueArray = new measure[size];
    front = 0;
    rear = -1;
    count = 0;
  }

  ~MessageQueue() {
    delete[] queueArray;
  }

  void enqueue(measure item) {
    if (!isFull()) {
      rear = (rear + 1) % capacity;
      queueArray[rear] = item;
      count++;
    }
  }

```

```
  measure dequeue() {
    measure item;
    if (!isEmpty()) {
      item = queueArray[front];
      front = (front + 1) % capacity;
      count--;
    }
    return item;
  }

  measure peek() {
    measure item;
    if (!isEmpty()) {
      item = queueArray[front];
    }
    return item;
  }

  int size() {
    return count;
  }

  bool isEmpty() {
    return (count == 0);
  }

  bool isFull() {
    return (count == capacity);
  }
};

#endif
```

Listing 5.23: MessageQueue.h Library

5.6 Summary

This chapter outlines the practical implementation of the proposed secure onboarding system, introduced in Section 4.3. The initial section provides an overview of the hardware and software frameworks utilized, with a particular focus on the ESP32-S3 and its capabilities. Subsequently, the development environment setup is described, with the objective of ensuring the system's security through the implementation of measures such as Secure Boot V2 and flash encryption. In addition, the chapter provides practical guidance on connecting hardware and configuring the device. This chapter outlines the process of integrating ESP-NOW for communication, addressing limitations such as data packet size and the number of encrypted peers. Furthermore, the chapter discusses the challenges encountered during the implementation phase, particularly in managing long message transmissions within the constraints of ESP-NOW.

Chapter 6

Evaluation

This chapter will focus on general security considerations, which are discussed in Section 6.1. Subsequently, the functionality of the whitelist is discussed in Section 6.2. Finally, a field test was performed, which is discussed in Section 6.3.

6.1 General Security considerations

Security was a primary consideration during the design phase. In order to minimize the attack surface, solutions that utilize pre-shared keys or credentials were deemed unsuitable. The ESP-NOW protocol's broadcast feature enables the transmission of an onboarding signal to nearby devices in a connection-less manner. A traditional connection, such as WiFi, would not have allowed this functionality. For each device undergoing onboarding, new RSA key pairs and new session keys are generated. Consequently, even if an adversary were to successfully extract a key, only the single communication would be compromised, and only for a brief period, until the next reboot of either the client or gateway.

In addition, security measures have been implemented on the hardware side. As previously discussed in Section 5.3, the security of the ESP32-S3 has been enhanced by enabling Secure Boot V2 and Flash encryption.

With regard to RSA encryption, the key length of 3072 bits has been selected. Although 2048-bit keys would also be secure, as previously stated in Section 4.3, a key with a length of 3072 bits is being used, with consideration for future developments. However, a compromise between security and efficiency was necessary. Although 3072-bit RSA keys are considered more secure, they require a longer generation time. In particular, in constrained environments such as the ESP32-S3, where performance is limited, this can result in a greater computational time for key generation. In contrast, a 3072-bit key takes, on average, 17.59 seconds to generate, whereas a 2048-bit key takes just 5.21 seconds on average to generate. The measurement was carried out using the built-in clock of the ESP32-S3, with the key generated 100 times. No significant discrepancy was observed between the two runs.

6.2 Bypassing Whitelist

In order to enhance the security of the system, a whitelist was implemented. Any device (client) that is not included in the whitelist, which is stored on the gateway, is rejected upon receiving an onboarding request. To assess the efficacy of this measure, attempts have been made to onboard devices that are not permitted. Various scenarios have been tested, including instances where no client is connected, one client is already connected, and during the onboarding of another device. The security mechanism was found to be effective in all tests.

6.3 Field Test

In this field test, the onboarding process was conducted in a real-world setting, rather than within the laboratory. Various settings were used to assess the efficacy of the implementation in its entirety. Firstly, the range of ESP-NOW was tested. During the onboarding process and when transmitting data over a distance of 50 meters with minimal obstructions between, there were no issues or delays in communication. Subsequently, the time component was tested. Even after a period of two hours, the communication remained operational and the expected functionality was maintained.

Figure 6.1 shows the gateway with a red LED, indicating that the threshold of the measured data has been exceeded.

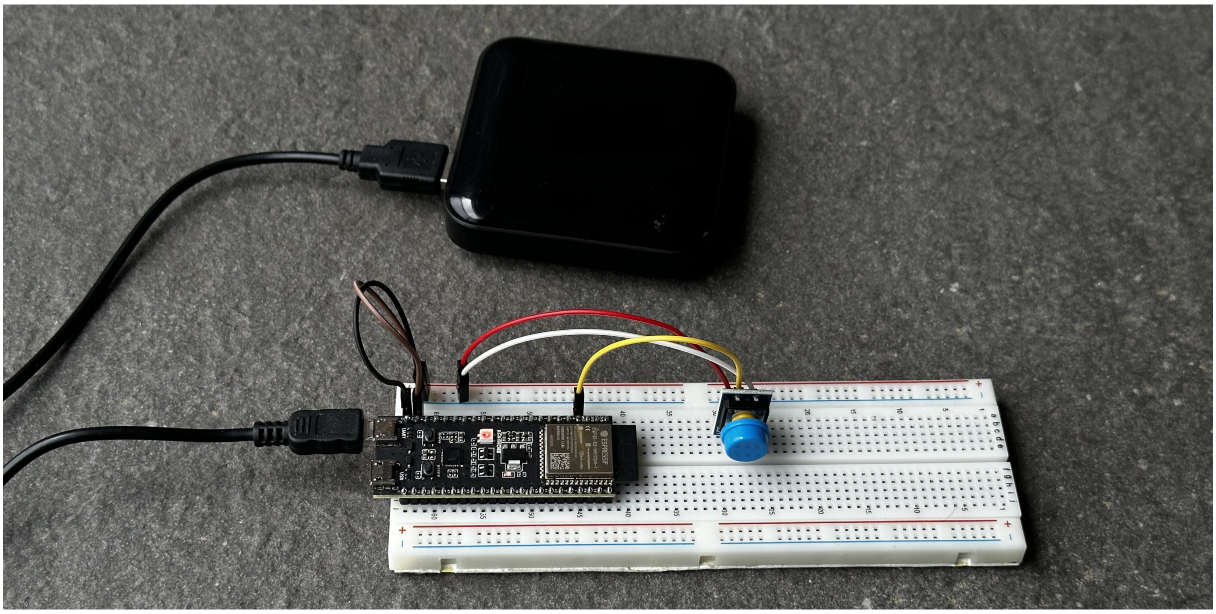

Figure 6.1: [Gateway Outdoor] Threshold of Measured data Exceeded

Figure 6.2 depicts the scenario in which the client is unable to transmit any data to the gateway. This is indicated by the red LED. From this point onward, the messages are queued. Once the client is able to send the data to the gateway again, all previous measures are also transmitted. This was also tested in the field test and the results were as expected.

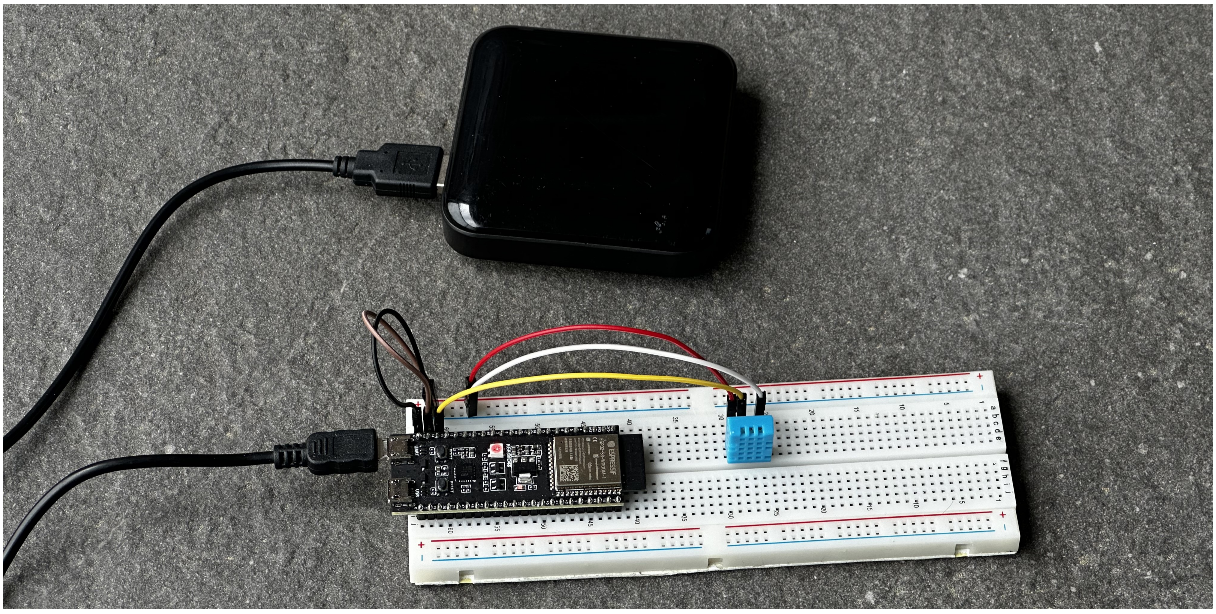

Figure 6.2: [Client Outdoor] Sent Messages are not Received by the Gateway

Chapter 7

Summary and Conclusions

7.1 Summary

In this thesis, the security vulnerabilities inherent in IoT devices were investigated, with a specific focus on their application in artwork tracking. This research was motivated by the increasing integration of IoT technologies in sensitive sectors, where security often lags behind functionality, posing significant risks to valuable assets like artwork.

The primary objective was to develop a secure and lightweight onboarding process for IoT devices tasked with monitoring artworks during transportation and storage. Utilizing the Arduino platform and ESP-NOW protocol, the proposed solution enhances security through encrypted communication channels and streamlined device authentication processes. The methodology adopted involved the design and implementation of cryptographic functions, including AES and RSA encryption, to safeguard data transmissions between IoT devices and gateways.

Key findings from the research underscore the effectiveness of the proposed onboarding process in mitigating common security threats such as unauthorized access and data breaches. Field tests demonstrated that the implementation not only adheres to security best practices but also operates efficiently under real-world conditions. These outcomes suggest that the strategic integration of robust encryption mechanisms can significantly enhance the security posture of IoT systems in the art sector.

The implications of these findings are profound, offering viable pathways to fortify the security frameworks of IoT devices across various applications. The enhanced onboarding process developed in this thesis could be adapted for broader IoT security applications, marking a step forward in balancing functionality with stringent security needs.

7.2 Conclusions

Conclusively, this thesis contributes to the crucial discourse on IoT security, particularly within the context of high-value asset tracking. The development of a secure, efficient onboarding process for IoT devices in artwork tracking addresses a significant gap in the current security measures, providing a scalable model that can be adapted across similar IoT applications.

The research highlights the potential of integrating advanced cryptographic techniques with conventional IoT communication protocols to enhance security. Despite these advancements, the study acknowledges limitations, including the scalability of the proposed methods across different IoT platforms and the potential for increased system complexity.

7.3 Future Work

The proposed implementation provides a solid foundation for the artwork tracking use case. Depending on the specific requirements, the implementation can be further developed and optimized.

At this stage, the setup comprises only two entities: the gateway and the client. All measurements made by the client are transmitted to the gateway, where they are stored. The user receives minimal feedback based on the color of the built-in LED. Although this may be sufficient for some basic use cases, it is now evident that most scenarios would benefit from remote monitoring by integrating cloud systems [43]. The implementation of a 5G module would facilitate this [44].

Currently, the measure structure, which is needed to transport the data in a meaningful manner, is hard coded and must be known in advance by the client and the gateway. To enhance the system's flexibility, a novel generic measure structure could be devised. This structure could encapsulate all the information necessary for the gateway to process the measured data correctly. This would confer the advantage that new clients could have sensors that are not known to the gateway, yet the gateway would still be able to process the data.

Finally, the implementation began as a proof of concept and was subsequently developed further. Consequently, there may be instances where more efficient or elegant solutions could be implemented.

Bibliography

[1] C. Kelly, N. Pitropakis, S. McKeown, and C. Lambrinoudakis, „Testing and hardening iot devices against the mirai botnet“, in *2020 International conference on cyber security and protection of digital services (cyber security)*, IEEE, 2020, pp. 1–8.

[2] *About - certify project*, Web Page, Accessed: 2024-04-30. [Online]. Available: `https://certify-project.eu/about/`.

[3] E. Commission, „Certify: Horizon europe (horizon) description of the action (doa)“, Tech. Report, Jun. 2022.

[4] A. Whitmore, A. Agarwal, and L. Da Xu, „The internet of things—a survey of topics and trends“, *Information systems frontiers*, vol. 17, pp. 261–274, 2015.

[5] S. Symington, W. Polk, and M. Souppaya, „Trusted internet of things (iot) device network-layer onboarding and lifecycle management (draft)“, US Department of Commerce, Tech. Rep., 2020.

[6] *Device bootstrap*, Web Page, Accessed: 2024-04-30. [Online]. Available: `https://iotatlas.net/en/patterns/device_bootstrap/`.

[7] Industrial Internet Consortium, *Automated onboarding and device provisioning best practices*, Accessed: 2023-04-30, Sep. 2022. [Online]. Available: `https://www.iiconsortium.org/wp-content/uploads/sites/2/2022/09/Automated-Onboarding-and-Device-Provisioning-Best-Practices-2022-10-04-v0.3.11.pdf`.

[8] B. Sunar, „True random number generators for cryptography“, in Dec. 2008, pp. 55–73, ISBN: 978-0-387-71816-3. DOI: `10.1007/978-0-387-71817-0_4`.

[9] V. Fischer, „A closer look at security in random number generators design“, vol. 7275, May 2012, pp. 167–182, ISBN: 978-3-642-29911-7. DOI: `10.1007/978-3-642-29912-4_13`.

[10] R. L. Rivest, A. Shamir, and L. Adleman, „A method for obtaining digital signatures and public-key cryptosystems“, *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[11] D. J. Bernstein and T. Lange, „Post-quantum cryptography“, *Nature*, vol. 549, no. 7671, pp. 188–194, 2017.

[12] E. Landi, L. Parri, R. Moretti, A. Fort, M. Mugnaini, and V. Vignoli, „An iot sensor node for health monitoring of artwork and ancient wooden structures“, in *2022 IEEE International Workshop on Metrology for Living Environment (MetroLivEn)*, IEEE, 2022, pp. 110–114.

[13] E. Schito and D. Testi, „Integrated maps of risk assessment and minimization of multiple risks for artworks in museum environments based on microclimate control“, *Building and Environment*, vol. 123, pp. 585–600, 2017.

[14] M. Cannistraro, G. Cannistraro, and R. Restivo, „Environmental monitoring of sacred artworks–a case study for the search for an index of correlation between particle concentration and mass of fine dust“, *Thermal Science and Engineering Progress*, vol. 14, p. 100 405, 2019.

[15] M. F. Mecklenburg, „Art in transit: Studies in the transport of paintings“, 1991.

[16] *High-quality packaging for sensitive valuables*, Web Page, Accessed: 2024-04-30. [Online]. Available: `https://hasenkamp.com/en/fineart/packaging`.

[17] *Made-to-measure for perfect protection*, Web Page, Accessed: 2024-04-30. [Online]. Available: `https://kraft-els.ch/en/packing/`.

[18] N. G. (Britain), *National gallery technical bulletin.* Publications Department, National Gallery, 1988, vol. 12.

[19] R. Mousheimish, Y. Taher, K. Zeitouni, and M. Dubus, „Pact-art: Adaptive and context-aware processes for the transportation of artworks“, in *2015 Digital Heritage*, IEEE, vol. 2, 2015, pp. 347–350.

[20] V. Carchiolo, M. P. Loria, M. Toja, and M. Malgeri, „Real time risk monitoring in fine-art with iot technology.“, in *FedCSIS (Communication Papers)*, 2018, pp. 151–158.

[21] E. Schiller, A. Aidoo, J. Fuhrer, J. Stahl, M. Ziörjen, and B. Stiller, „Landscape of iot security“, *Computer Science Review*, vol. 44, p. 100 467, 2022.

[22] K. Seyhan and S. Akleylek, „Classification of random number generator applications in iot: A comprehensive taxonomy“, *Journal of Information Security and Applications*, vol. 71, p. 103 365, 2022.

[23] E. Barker and A. Roginsky, „Transitioning the use of cryptographic algorithms and key lengths“, National Institute of Standards and Technology, Tech. Rep., Mar. 2019.

[24] A. K. Lenstra, „Key lengths“, *The Handbook of Information Security*, 2006.

[25] *Esp32-s3 technical information*, Web Page, Accessed: 2024-04-22. [Online]. Available: `https://www.espressif.com/en/products/socs/esp32-s3/`.

[26] *Random number generation*, Web Page, Accessed: 2024-04-28. [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32s3/api-reference/system/random.html`.

[27] M. Banzi and M. Shiloh, *Getting started with Arduino.* Maker Media, Inc., 2022.

[28] M. Margolis, B. Jepson, and N. R. Weldin, *Arduino cookbook: recipes to begin, expand, and enhance your projects.* O'Reilly Media, 2020.

[29] B. Craft, *Arduino projects for dummies.* John Wiley & Sons, 2013.

[30] S. Monk, *Programming Arduino: getting started with sketches*, vol. 176.

[31] P. Scherz, *Practical electronics for inventors.* McGraw-Hill, Inc., 2006.

[32] *Arduino-esp32 esp-now api*, Web Page, Accessed: 2024-04-28. [Online]. Available: `https://docs.espressif.com/projects/arduino-esp32/en/latest/api/espnow.html`.

[33] D. Urazayev, A. Eduard, M. Ahsan, and D. Zorbas, „Indoor performance evaluation of esp-now“, in *2023 IEEE International Conference on Smart Information Systems and Technologies (SIST)*, IEEE, 2023, pp. 1–6.

[34] *Esp-now*, Web Page, Accessed: 2024-04-28. [Online]. Available: `https://www.espressif.com/en/solutions/low-power-solutions/esp-now`.

[35] D. Eridani, A. F. Rochim, and F. N. Cesara, „Comparative performance study of esp-now, wi-fi, bluetooth protocols based on range, transmission speed, latency, energy usage and barrier resistance“, in *2021 international seminar on application for technology of information and communication (iSemantic)*, IEEE, 2021, pp. 322–328.

[36] *Esp-now*, Web Page, Accessed: 2024-04-28. [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32s3/api-reference/network/esp_now.html`.

[37] *Ieee 802.11-2012*, Web Page, Accessed: 2024-04-30. [Online]. Available: `https://standards.ieee.org/ieee/802.11/4523/`.

[38] *Esp32-s3-devkitc-1 v1.1*, Web Page, Accessed: 2024-04-29. [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html#description-of-components`.

[39] *Arduino as an esp-idf component*, Web Page, Accessed: 2024-04-24. [Online]. Available: `https://docs.espressif.com/projects/arduino-esp32/en/latest/esp-idf_component.html`.

[40] *Security*, Web Page, Accessed: 2024-04-29. [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32s3/security/security.html`.

[41] *Secure boot v2*, Web Page, Accessed: 2024-04-29. [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32s3/security/secure-boot-v2.html`.

[42] *Flash encryption*, Web Page, Accessed: 2024-04-29. [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32s3/security/flash-encryption.html`.

[43] A. Botta, W. De Donato, V. Persico, and A. Pescapé, „Integration of cloud computing and internet of things: A survey“, *Future generation computer systems*, vol. 56, pp. 684–700, 2016.

[44] *News meet walter, our new esp32-s3-based friend!*, Web Page, Accessed: 2024-04-30. [Online]. Available: `https://www.espressif.com/en/news/Walter`.

Abbreviations

CTR_DRBG	Counter Mode Deterministic Random Bit Generator
DRBG	Deterministic Random Bit Generator
ESP-IDF	Espressif IoT Development Framework
GPIO	General Purpose Input/Output
IDE	Integrated Development Environment
IoT	Internet of Things
LMK	Local Master Key
NIST	The National Institute of Standards and Technology
PEM	Privacy Enhanced Mail
PMK	Primary Master Key
PRNG	Pseudo Random Number Generator
RNG	Random Number Generation
RSA	Rivest-Shamir-Adleman
SoC	System on a Chip
TPM	Trusted Platform Module
TRNG	True Random Number Generator

List of Figures

List of Listings

Appendix A

Gateway Code

```
#include <WiFi.h>
#include <esp_now.h>
#include <Adafruit_NeoPixel.h>
#include <mbedtls/entropy.h>
#include <mbedtls/ctr_drbg.h>
#include <mbedtls/rsa.h>
#include <mbedtls/pk.h>

#define BUTTON_PIN 7
#define RSA_KEY_LENGTH 3072
#define RSA_EXPONENT 65537
#define DEFAULT 0
#define BROADCASTING 1
#define KEY_EXCHANGE 2
#define CHALLENGING 3
#define SEND_SESSION_KEY 4
#define MAX_DATA_SIZE 211
#define MAX_MESSAGES 10
#define MAX_PARTS 10
#define PMK "#ArtworkTracking"

Adafruit_NeoPixel pixels(1, 48, NEO_GRB + NEO_KHZ800);
int STATE = DEFAULT;
int time_elapsed = 0;
mbedtls_rsa_context rsa;  // Initialize a global RSA context
uint8_t clientMAC[6];
String pem_peer;  // public key of peer
String challengePhrase;
const uint8_t broadcastAddress[] = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
   };
// Populate whitelist with allowed mac addresses
const uint8_t allowedMacAddresses[][6] = {
  { 0xf4, 0x12, 0xfa, 0xe6, 0x56, 0xe4 },
};

typedef struct message {      // Structure for long messages
  char id[37];                // Unique message id
  byte count;                 // Number of packets sent yet
  byte total;                 // Total number of packages to be sent
```

```
  char data[MAX_DATA_SIZE];  // Data
} message;

struct MessagePart {
  int index;
  char data[MAX_DATA_SIZE];
};

struct MessageRec {
  char id[37];
  MessagePart parts[MAX_PARTS];
  int total;
  int received;
};
MessageRec messages[MAX_MESSAGES];


typedef struct measure {  // Structure for sensor data
  int temperature;
  int humidity;
  int temperatureAlarm;
  int humidityAlarm;
} measure;


void setup() {
  // General initialization
  Serial.begin(115200);
  pixels.begin();
  pixels.setPixelColor(0, pixels.Color(17, 17, 17));
  pixels.show();
  pinMode(BUTTON_PIN, INPUT);

  // Setup ESP-NOW
  WiFi.mode(WIFI_STA);
  if (esp_now_init() == ESP_OK) {
    uint8_t pmk[16];
    hexStringToByteArray(PMK, pmk, 16);
    esp_now_set_pmk(pmk);                          // Change the PMK for an extra layer of security
    esp_now_register_recv_cb(receiveSensorData);  // Register the default recieve callback
    esp_now_register_send_cb(sendHandler);         // Register the default send callback
  } else {
    Serial.println("ESP-NOW Init Failed. Retry...");
    delay(3000);
    ESP.restart();
  }

  pixels.setPixelColor(0, pixels.Color(0, 17, 0));  // Set the led to green
  pixels.show();
}

void loop() {
```

```
  if (STATE == BROADCASTING) {
    sendBroadcast();
  } else if (STATE == DEFAULT && digitalRead(BUTTON_PIN) == HIGH) {
    STATE = BROADCASTING;
    esp_now_register_recv_cb(receiveOnboardRequest);  // Register the onboarding callback
    esp_now_unregister_send_cb();                     // Unregister the send callback

    pixels.setPixelColor(0, pixels.Color(17, 17, 17));
    pixels.show();

    initializeRSAKey();  // RSA key pair generation. Can take up to 25s

    pixels.setPixelColor(0, pixels.Color(0, 0, 17));
    pixels.show();

    Serial.println("Sending Broadcast to " + formatMacAddress(broadcastAddress));
  }

  if (STATE != DEFAULT) {
    sleep(3);
    if (++time_elapsed >= 20) reset();  // Reset after 60 seconds (3s * 20)
  }
}


// Called when data is sent
void sendHandler(const uint8_t* macAddr, esp_now_send_status_t status) {
  if (status == ESP_NOW_SEND_FAIL) {
    Serial.println("Package sent to " + formatMacAddress(macAddr) + " FAILED ");
    pixels.setPixelColor(0, pixels.Color(17, 0, 0));
  } else {
    pixels.setPixelColor(0, pixels.Color(0, 17, 0));
  }
  pixels.show();
}

// Called when sensor data is received (default)
void receiveSensorData(const uint8_t* macAddr, const uint8_t* data, int dataLen) {
  if (!deviceAllowed(macAddr)) return;
  measure m;
  memcpy(&m, data, sizeof(m));

  if (m.temperature >= m.temperatureAlarm || m.humidity >= m.humidityAlarm || m.temperature > 250 || m.humidity > 250) {  // Check if alarm is triggered
    pixels.setPixelColor(0, pixels.Color(17, 0, 0));
    if (m.temperature > 250) m.temperature = -999;
    if (m.humidity > 250) m.humidity = -999;
  } else {
    pixels.setPixelColor(0, pixels.Color(0, 17, 0));
```

```
  }

  Serial.println("Temperature: " + (String)m.temperature + " °C");
  Serial.println("Humidity: " + (String)m.humidity + "%");
  pixels.show();
}

void sendBroadcast() {
  esp_now_peer_info_t peerInfo = {};          // Create peer
  memcpy(&peerInfo.peer_addr, broadcastAddress, 6);          // Add mac adress of peer (in this case broadcast to everyone)
  if (!esp_now_is_peer_exist(broadcastAddress)) esp_now_add_peer(&peerInfo);  // Add the peer to the list

  // Send message
  const String message = "Artwork Tracking Onboarding";
  esp_err_t result = esp_now_send(broadcastAddress, (const uint8_t*)message.c_str(), message.length());

  if (result != ESP_OK) Serial.println("Broadcast Failed... Trying again.");
}

// Called when onboarding request is received
void receiveOnboardRequest(const uint8_t* macAddr, const uint8_t* data, int dataLen) {
  if (!deviceAllowed(macAddr)) return;
  for (int i = 0; i < 6; i++) clientMAC[i] = macAddr[i];
  Serial.println("Receiving Onboarding Request from " + formatMacAddress(macAddr));
  time_elapsed = 0;

  char buffer[dataLen + 1];  // Only allow a maximum of 250 characters in the message + a null terminating byte
  strncpy(buffer, (const char*)data, dataLen);
  buffer[dataLen] = 0;  // Make sure we are null terminated

  if (strcmp(buffer, "Onboarding Request") == 0) {
    esp_now_register_send_cb(sendHandler);
    sendPublicKey(macAddr);
  } else {
    Serial.println("Bad request from " + formatMacAddress(macAddr) + ".");
  }
}

void sendPublicKey(const uint8_t* macAddr) {
  STATE = KEY_EXCHANGE;
  Serial.println("Sending Public Key to " + formatMacAddress(macAddr));
  esp_now_register_recv_cb(receivePublicKey);  // Register the recieve callback

  esp_now_del_peer(broadcastAddress);                              // Remove broadcast since it is not necessary any more.
```

```
  esp_now_peer_info_t peerInfo = {};                              // Create peer
  memcpy(&peerInfo.peer_addr, macAddr, 6);                        // Add mac adress of peer (in this case broadcast to everyone)
  if (esp_now_is_peer_exist(macAddr)) esp_now_del_peer(macAddr);  // Remove if there is an old connection
  esp_now_add_peer(&peerInfo);                                    // Add the peer to the list

  sendLongMessage(exportPublicKey().c_str(), macAddr);
}

void receivePublicKey(const uint8_t* macAddr, const uint8_t* data, int dataLen) {
  if (!deviceAllowed(macAddr) || memcmp(clientMAC, macAddr, 6) != 0) return;  // Check if data comes from correct client

  Serial.println("Receiving Public Key from " + formatMacAddress(macAddr));
  String message = receiveLongMessage(macAddr, data, dataLen);

  if (!message.equals("")) {  // Message completely received.
    pem_peer = message;
    sendChallenge(macAddr);
  }
}

void sendChallenge(const uint8_t* macAddr) {
  STATE = CHALLENGING;
  esp_now_register_recv_cb(receiveChallengeResponse);  // Register the recieve callback
  Serial.println("Sending Challenge to " + formatMacAddress(macAddr) + " started.");
  challengePhrase = generateAESKey();
  sendLongMessage(encryptRSA(challengePhrase).c_str(), macAddr);
}

void receiveChallengeResponse(const uint8_t* macAddr, const uint8_t* data, int dataLen) {
  if (!deviceAllowed(macAddr) || memcmp(clientMAC, macAddr, 6) != 0) return;  // Check if data comes from correct client
  Serial.println("Receiving Challenge Response from " + formatMacAddress(macAddr) + " received");

  String message = receiveLongMessage(macAddr, data, dataLen);

  if (!message.equals("")) {  // Message completely received.
    if (decryptRSA(message).equals(challengePhrase)) {
      challengePhrase.clear();
      sendSessionKey(macAddr);
    } else {
      Serial.println("Challenge not passed. Abbord...");
      reset();
    }
  }
}
```

```

void sendSessionKey(const uint8_t* macAddr) {
  STATE = SEND_SESSION_KEY;
  esp_now_register_recv_cb(receiveSensorData);  // Register the recieve callback
  Serial.println("Sending Session Key to " + formatMacAddress(macAddr) + " started.");
  String sessionKey = generateAESKey();

  sendLongMessage(encryptRSA(sessionKey).c_str(), macAddr);

  if (esp_now_is_peer_exist(macAddr)) esp_now_del_peer(macAddr);
  uint8_t lmk[16];
  hexStringToByteArray(sessionKey, lmk, 16);
  esp_now_peer_info_t peerInfo = {};                            // Create peer
  memcpy(&peerInfo.peer_addr, macAddr, 6);                      // Add mac adress of peer (in this case broadcast to everyone)
  memcpy(&peerInfo.lmk, lmk, 16);                               // Add Local Master Key (LMK) of peer
  peerInfo.encrypt = true;                                      // Enable encryption
  if (!esp_now_is_peer_exist(macAddr)) esp_now_add_peer(&peerInfo);  // Add the peer to the list

  sessionKey.clear();
  Serial.println("Onboarding of " + formatMacAddress(macAddr) + " complete\n");
  reset();
}

// Util for long messages
void sendLongMessage(const char* input_data, const uint8_t* macAddr) {
  int total_messages = (strlen(input_data) + MAX_DATA_SIZE - 1) / MAX_DATA_SIZE;
  int attempts = 1;
  char buffer[37];
  sprintf(buffer, "%u", esp_random());

  for (int i = 0; i < total_messages; i++) {
    message msg;
    strncpy(msg.id, buffer, sizeof(msg.id));
    msg.count = i;
    msg.total = total_messages;

    int length = strlen(input_data) - i * MAX_DATA_SIZE;
    if (length > MAX_DATA_SIZE) length = MAX_DATA_SIZE;
    strncpy(msg.data, &input_data[i * MAX_DATA_SIZE], length);
    if (length < MAX_DATA_SIZE) msg.data[length] = '\0';  // Ensure null termination

    esp_err_t result = esp_now_send(macAddr, (const uint8_t*)&msg, sizeof(msg));  // Send message

    if (result != ESP_OK) {
      if (attempts++ >= 3) {
```

```
        Serial.println("Could not send long message. Abbort...");
        reset();
        break;
      }
      --i;
    } else {
      attempts = 1;
    }
  }
}

// Util for long messages
String receiveLongMessage(const uint8_t* macAddr, const uint8_t* data, int len) {
  message* msg = (message*)data;  // Cast the data to a message

  // Find or create the message in the messages array
  MessageRec* fullMessage = NULL;
  for (auto& message : messages) {
    if (strcmp(message.id, msg->id) == 0) {
      fullMessage = &message;
      break;
    } else if (message.received == 0) {
      strcpy(message.id, msg->id);
      message.total = msg->total;
      fullMessage = &message;
      break;
    }
  }

  if (fullMessage == NULL) return "";  // If message couldn't be found or created, return an empty string

  // Store this part of the message
  strcpy(fullMessage->parts[msg->count].data, msg->data);
  fullMessage->parts[msg->count].index = msg->count;
  fullMessage->received++;

  // If received all parts of the message, combine them into a single string
  if (fullMessage->received == fullMessage->total) {
    String fullMessageStr = "";
    for (int i = 0; i < fullMessage->total; i++) fullMessageStr += fullMessage->parts[i].data;

    fullMessage->received = 0;  // Reset the message
    return fullMessageStr;      // Return the full message
  }

  return "";  // If not all parts of the message have been received, return an empty string
}

// Reset STATE
void reset() {
```

```
  esp_now_register_recv_cb(receiveSensorData);  // Register the recieve callback
  esp_now_unregister_send_cb();
  freeRSAKey();
  memset(clientMAC, 0, sizeof(clientMAC));
  STATE = DEFAULT;
  time_elapsed = 0;
  pixels.setPixelColor(0, pixels.Color(0, 17, 0));
  pixels.show();
}

// Helpers
// Check if request from device is allowed
bool deviceAllowed(const uint8_t* macAddr) {
  for (int i = 0; i < sizeof(allowedMacAddresses) / sizeof(allowedMacAddresses[0]); i++) {
    if (memcmp(macAddr, allowedMacAddresses[i], sizeof(allowedMacAddresses[i])) == 0) {
      return true;
    }
  }
  Serial.println("Transmission from unauthorized device (" + formatMacAddress(macAddr) + ") rejected.");
  return false;
}

// Formats MAC Address for prints
String formatMacAddress(const uint8_t* macAddr) {
  char res[18];
  snprintf(res, sizeof(res), "%02x:%02x:%02x:%02x:%02x:%02x", macAddr[0], macAddr[1], macAddr[2], macAddr[3], macAddr[4], macAddr[5]);
  return String(res);
}

void hexStringToByteArray(const String& hexString, uint8_t* byteArray, int byteArrayLength) {
  for (int i = 0; i < byteArrayLength; i++) {
    String hexByte = hexString.substring(i * 2, i * 2 + 2);
    byteArray[i] = (uint8_t)strtol(hexByte.c_str(), nullptr, 16);
  }
}

// ENCRYPTION
// Function to generate a 128-bit AES key and return it as a string
String generateAESKey() {
  mbedtls_entropy_context entropy;  // Context for entropy collection
  mbedtls_entropy_init(&entropy);   // Initialize entropy context to gather entropy used for random number generation

  mbedtls_ctr_drbg_context ctr_drbg;  // Context for the CTR_DRBG random number generator
  mbedtls_ctr_drbg_init(&ctr_drbg);   // Initialize the CTR_DRBG context

  uint32_t randomNumber = esp_random();
  char personalization[11];  // Personalization string for the DRBG seeding
```

```
  sprintf(personalization, "0x%08X", randomNumber);
  // Seed the CTR_DRBG context with entropy collected plus a personalization string for additional randomness
  mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (const unsigned char*)personalization, strlen(personalization));

  unsigned char key[16];  // Buffer to hold the 128-bit key (16 bytes)
  // Generate a random 128-bit key using the seeded CTR_DRBG context
  mbedtls_ctr_drbg_random(&ctr_drbg, key, sizeof(key));

  String keyHex = "";  // String to hold the hexadecimal representation of the key
  for (unsigned char i : key) {
    char hex[3];                // Temporary buffer to hold each byte in hex format
    sprintf(hex, "%02X", i);  // Format each byte of the key as two hexadecimal characters
    keyHex += hex;              // Append the hex string to the keyHex string
  }

  mbedtls_ctr_drbg_free(&ctr_drbg);  // Free the CTR_DRBG context to release any associated resources
  mbedtls_entropy_free(&entropy);    // Free the entropy context to release any associated resources

  return keyHex;  // Return the hexadecimal string representation of the key
}

// Function to initialize and generate RSA keys
void initializeRSAKey() {
  mbedtls_entropy_context entropy;  // Context for entropy collection
  mbedtls_entropy_init(&entropy);   // Initialize entropy context

  mbedtls_ctr_drbg_context ctr_drbg;  // Context for random number generator
  mbedtls_ctr_drbg_init(&ctr_drbg);   // Initialize CTR_DRBG context

  uint32_t randomNumber = esp_random();
  char personalization[11];  // Personalization string for the DRBG seeding
  sprintf(personalization, "0x%08X", randomNumber);
  mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (const unsigned char*)personalization, strlen(personalization));
  // Seed the DRBG

  mbedtls_rsa_init(&rsa, MBEDTLS_RSA_PKCS_V15, 0);  // Initialize RSA context
  int ret = mbedtls_rsa_gen_key(&rsa, mbedtls_ctr_drbg_random, &ctr_drbg, RSA_KEY_LENGTH, RSA_EXPONENT);
  // Generate RSA key pair

  if (ret != 0) {
    Serial.print("Failed to generate RSA key with error code: ");
    Serial.println(ret);
```

```
  }
  if (mbedtls_rsa_check_privkey(&rsa) != 0) {
    Serial.println("Generated RSA private key is not valid.");
  }

  mbedtls_ctr_drbg_free(&ctr_drbg);  // Free the DRBG context
  mbedtls_entropy_free(&entropy);    // Free the entropy context
}

// Function to encrypt data using an external RSA public key
String encryptRSA(const String& data) {
  mbedtls_pk_context pk;  // Public key container
  mbedtls_pk_init(&pk);   // Initialize the public key container

  // Parse the public key from provided PEM string
  if (mbedtls_pk_parse_public_key(&pk, (const unsigned char*)pem_peer.c_str(), pem_peer.length() + 1) != 0) {
    mbedtls_pk_free(&pk);
    return "";  // Return empty if public key parsing fails
  }

  // Encrypt the data
  unsigned char output[1024];  // Buffer to hold encrypted data
  size_t olen;

  mbedtls_ctr_drbg_context ctr_drbg;
  mbedtls_entropy_context entropy;
  mbedtls_entropy_init(&entropy);
  mbedtls_ctr_drbg_init(&ctr_drbg);
  uint32_t randomNumber = esp_random();
  char personalization[11];  // Personalization string for the DRBG seeding
  sprintf(personalization, "0x%08X", randomNumber);
  // Seed the CTR_DRBG context with entropy collected plus a personalization string for additional randomness
  mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (const unsigned char*)personalization, strlen(personalization));

  int ret = mbedtls_pk_encrypt(&pk, (const unsigned char*)data.c_str(), data.length(), output, &olen, sizeof(output), mbedtls_ctr_drbg_random, &ctr_drbg);

  mbedtls_pk_free(&pk);
  mbedtls_ctr_drbg_free(&ctr_drbg);
  mbedtls_entropy_free(&entropy);

  if (ret != 0) return "";  // Return empty if encryption fails

  String encHex = "";
  for (size_t i = 0; i < olen; i++) {
    char hex[3];
    sprintf(hex, "%02X", output[i]);
    encHex += hex;
  }

  return encHex;  // Return the hex string of the encrypted data
```

```
}

// Function to decrypt data using RSA
String decryptRSA(const String& encHex) {
  if (mbedtls_rsa_check_privkey(&rsa) != 0) {
    Serial.println("RSA private key is not valid.");
  }

  unsigned char encData[1024];  // Buffer to store the encrypted data in binary form
  size_t encIndex = 0;          // Index for filling the encData buffer

  // Convert hexadecimal string back to binary data
  for (size_t i = 0; i < encHex.length(); i += 2) {
    sscanf(encHex.c_str() + i, "%02X", &encData[encIndex++]);  // Parse two hexadecimal characters at a time_elapsed
  }

  unsigned char output[1024];  // Buffer to hold the decrypted data
  size_t olen;                 // Variable to store the length of the decrypted data

  mbedtls_ctr_drbg_context ctr_drbg;  // Context for the CTR_DRBG random number generator
  mbedtls_entropy_context entropy;    // Context for entropy collection
  mbedtls_entropy_init(&entropy);     // Initialize the entropy context
  mbedtls_ctr_drbg_init(&ctr_drbg);   // Initialize the CTR_DRBG context

  uint32_t randomNumber = esp_random();
  char personalization[11];  // Personalization string for the DRBG seeding
  sprintf(personalization, "0x%08X", randomNumber);
  // Seed the CTR_DRBG context with entropy collected plus a personalization string for additional randomness
  mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (const unsigned char*)personalization, strlen(personalization));

  // Decrypt the data using the private key
  int ret = mbedtls_rsa_pkcs1_decrypt(&rsa, mbedtls_ctr_drbg_random, &ctr_drbg, MBEDTLS_RSA_PRIVATE, &olen, encData, output, sizeof(output));

  mbedtls_ctr_drbg_free(&ctr_drbg);  // Free the CTR_DRBG context
  mbedtls_entropy_free(&entropy);    // Free the entropy context

  if (ret != 0) {
    Serial.print("Decryption failed with error: ");
    Serial.println(ret);
    return "";
  }

  if (mbedtls_rsa_check_privkey(&rsa) != 0) {
    Serial.println("RSA private key is not valid.");
  }

```

```
  return String((char*)output);  // Convert the decrypted binary data back to a string and return it
}

String exportPublicKey() {
  char buf[626];  // Ensure buffer is large enough for the key
  mbedtls_pk_context pk;
  mbedtls_pk_init(&pk);  // Initialize the PK context

  // Setup the PK context to hold an RSA key
  if (mbedtls_pk_setup(&pk, mbedtls_pk_info_from_type(MBEDTLS_PK_RSA)) != 0) {
    mbedtls_pk_free(&pk);
    return "";
  }

  // Copy the RSA context to the PK context
  mbedtls_rsa_context* rsa_copy = mbedtls_pk_rsa(pk);
  mbedtls_rsa_copy(rsa_copy, &rsa);  // Correctly copy RSA context

  // Check if the public key can be written into buffer
  if (mbedtls_pk_write_pubkey_pem(&pk, (unsigned char*)buf, sizeof(buf)) < 0) {
    mbedtls_pk_free(&pk);
    return "";  // Return empty string on failure
  }

  mbedtls_pk_free(&pk);  // Free the PK context
  return String(buf);    // Return the public key in PEM format
}

// Function to clean up RSA context when no longer needed
void freeRSAKey() {
  volatile char* p = const_cast<char*>(pem_peer.c_str());  // Access the underlying character array of the string
  size_t len = pem_peer.length();                          // Get the length of the string
  while (len--) *p++ = 0;                                  // Overwrite each character with zero
  pem_peer.clear();                                        // Clear the string to remove all content and reduce its size to zero

  secureZeroMemory(&rsa, sizeof(rsa));
  mbedtls_rsa_free(&rsa);  // Free the RSA context and all associated resources
}

void secureZeroMemory(void* ptr, size_t size) {
  volatile uint8_t* p = (volatile uint8_t*)ptr;
  while (size--) *p++ = 0;
}
```

Listing A.1: Complete Code of Gateway

Appendix B

Client Code

```
#include <WiFi.h>
#include <esp_now.h>
#include <Adafruit_NeoPixel.h>
#include <DHT11.h>
#include <mbedtls/entropy.h>
#include <mbedtls/ctr_drbg.h>
#include <mbedtls/rsa.h>
#include <mbedtls/pk.h>
#include "MeasureQueue.h"

#define RSA_KEY_LENGTH 3072
#define RSA_EXPONENT 65537
#define DEFAULT 0
#define SEARCHING 1
#define KEY_EXCHANGE 2
#define CHALLENGING 3
#define RECEIVE_SESSION_KEY 4
#define MAX_DATA_SIZE 211
#define MAX_MESSAGES 10
#define MAX_PARTS 10
#define PMK "#ArtworkTracking"

Adafruit_NeoPixel pixels(1, 48, NEO_GRB + NEO_KHZ800);
DHT11 dht11(14);
int STATE = SEARCHING;
int time_elapsed = 0;
mbedtls_rsa_context rsa;  // Initialize a global RSA context
String pem_peer;          // public key of peer
uint8_t gatewayMAC[6];
MessageQueue messageQueue(100);  // Queue with capacity for 100 measures

typedef struct message {     // Structure for long messages
  char id[37];               // Unique message id
  byte count;                // Number of packets sent yet
  byte total;                // Total number of packages to be sent
  char data[MAX_DATA_SIZE];  // Data
} message;

struct MessagePart {
```

```
  int index;
  char data[MAX_DATA_SIZE];
};

struct MessageRec {
  char id[37];
  MessagePart parts[MAX_PARTS];
  int total;
  int received;
};
MessageRec messages[MAX_MESSAGES];

void setup() {
  // General initialization
  Serial.begin(115200);
  pixels.begin();
  pixels.setPixelColor(0, pixels.Color(17, 17, 17));
  pixels.show();

  // Setup ESP-NOW
  WiFi.mode(WIFI_STA);
  if (esp_now_init() == ESP_OK) {
    uint8_t pmk[16];
    hexStringToByteArray(PMK, pmk, 16);
    esp_now_set_pmk(pmk);
    initializeRSAKey();
    esp_now_register_recv_cb(receiveBroadcast);  // Register the recieve broadcast callback
    esp_now_register_send_cb(sendHandler);       // Register the default send callback
  } else {
    Serial.println("ESP-NOW Init Failed. Retry...");
    delay(3000);
    ESP.restart();
  }

  pixels.setPixelColor(0, pixels.Color(0, 0, 17));
  pixels.show();
}

void loop() {
  if (STATE == DEFAULT) sendSensorData();

  sleep(3);
  if (STATE != DEFAULT && STATE != SEARCHING) {
    if (++time_elapsed >= 20) reset();
  }
}

// Called when data is sent (default)
void sendHandler(const uint8_t* macAddr, esp_now_send_status_t status) {
  if (status == ESP_NOW_SEND_FAIL) {
    Serial.println("Package sent to " + formatMacAddress(macAddr) + " FAILED");
    pixels.setPixelColor(0, pixels.Color(17, 0, 0));
  } else {
```

```
    pixels.setPixelColor(0, pixels.Color(0, 17, 0));
    if (STATE == DEFAULT) {
      measuresQueue.dequeue();
      if (!measuresQueue.isEmpty()) {
        measure m = measuresQueue.peek();
        esp_now_send(gatewayMAC, (uint8_t*)&m, sizeof(m));
      }
    }
  }
  pixels.show();
}

void sendSensorData() {
  if (measuresQueue.isFull()) {
    Serial.println("Gateway connection lost. Resetting board...");
    ESP.restart();
    return;
  }

  measure m;
  int temperature = dht11.readTemperature();
  int humidity = dht11.readHumidity();

  if (temperature == DHT11::ERROR_TIMEOUT || temperature == DHT11::ERROR_CHECKSUM) Serial.println("Temperature Reading Error: " + DHT11::getErrorString(temperature));
  if (humidity == DHT11::ERROR_TIMEOUT || humidity == DHT11::ERROR_CHECKSUM) Serial.println("Humidity Reading Error: " + DHT11::getErrorString(humidity));

  m.temperature = temperature;
  m.humidity = humidity;
  measuresQueue.enqueue(m);
  measure m2 = measuresQueue.peek();
  esp_now_send(gatewayMAC, (uint8_t*)&m2, sizeof(m2));
}

// Called when broadcast is received
void receiveBroadcast(const uint8_t* macAddr, const uint8_t* data, int dataLen) {
  for (int i = 0; i < 6; i++) gatewayMAC[i] = macAddr[i];
  Serial.println("Receiving Broadcast from " + formatMacAddress(macAddr));
  char buffer[dataLen + 1];  // Only allow a maximum of 250 characters in the message + a null terminating byte
  strncpy(buffer, (const char*)data, dataLen);
  buffer[dataLen] = 0;  // Make sure we are null terminated

  if (strcmp(buffer, "Artwork Tracking Onboarding") == 0) {
    sendOnboardigRequest(macAddr);
  } else {
    Serial.println("Bad request from " + formatMacAddress(macAddr));
  }
}

void sendOnboardigRequest(const uint8_t* macAddr) {
```

```
  STATE = SEARCHING;
  Serial.println("Sending Onboarding request to " + formatMacAddress(macAddr));

  esp_now_peer_info_t peerInfo = {};                                 // Create peer
  memcpy(&peerInfo.peer_addr, macAddr, 6);                           // Add mac adress of peer (in this case broadcast to everyone)
  if (esp_now_is_peer_exist(macAddr)) esp_now_del_peer(macAddr);  // Remove if there is an old connection
  esp_now_add_peer(&peerInfo);                                       // Add the peer to the list

  const String message = "Onboarding Request";
  esp_err_t result = esp_now_send(macAddr, (const uint8_t*)message.c_str(), message.length());  // Send message

  if (result == ESP_OK) {
    esp_now_register_recv_cb(receivePublicKey);
    time_elapsed = 0;
  } else {
    Serial.println("Onboarding request could not be sent. Restart the board.");
  }
}

// Called when public key is received
void receivePublicKey(const uint8_t* macAddr, const uint8_t* data, int dataLen) {
  if (memcmp(gatewayMAC, macAddr, 6) != 0) return;  // Check if data comes from correct gateway
  Serial.println("Receiving Public Key from " + formatMacAddress(macAddr));

  String message = receiveLongMessage(macAddr, data, dataLen);

  if (!message.equals("")) {  // Message completely received.
    pem_peer = message;
    sendPublicKey(macAddr);
  }
}

void sendPublicKey(const uint8_t* macAddr) {
  STATE = KEY_EXCHANGE;
  esp_now_register_recv_cb(receiveChallenge);  // Register the recieve callback
  Serial.println("Sending Public Key to " + formatMacAddress(macAddr));

  mbedtls_pk_context pk;  // Public key container
  mbedtls_pk_init(&pk);   // Initialize the public key container

  sendLongMessage(exportPublicKey().c_str(), macAddr);
}

// Called when challenge is received
void receiveChallenge(const uint8_t* macAddr, const uint8_t* data, int
```

```
    dataLen) {
  if (memcmp(gatewayMAC, macAddr, 6) != 0) return;  // Check if data comes from correct gateway
  Serial.println("Receiving Challenge from " + formatMacAddress(macAddr));
  String message = receiveLongMessage(macAddr, data, dataLen);
  if (!message.equals("")) sendChallenge(message, macAddr);  // Message completely received.
}

void sendChallenge(String data, const uint8_t* macAddr) {
  STATE = CHALLENGING;
  esp_now_register_recv_cb(receiveSessionKey);  // Register the recieve callback
  Serial.println("Sending Challenge Solution to " + formatMacAddress(macAddr));

  sendLongMessage(encryptRSA(decryptRSA(data)).c_str(), macAddr);
}

// Called when session key is received
void receiveSessionKey(const uint8_t* macAddr, const uint8_t* data, int dataLen) {
  if (memcmp(gatewayMAC, macAddr, 6) != 0) return;  // Check if data comes from correct gateway
  STATE = RECEIVE_SESSION_KEY;
  Serial.println("Receiving Session Key from " + formatMacAddress(macAddr));
  String message = receiveLongMessage(macAddr, data, dataLen);

  if (!message.equals("")) {  // Message completely received.
    esp_now_unregister_recv_cb();
    if (esp_now_is_peer_exist(macAddr)) esp_now_del_peer(macAddr);
    uint8_t lmk[16];
    hexStringToByteArray(decryptRSA(message), lmk, 16);
    esp_now_peer_info_t peerInfo = {};        // Create peer
    memcpy(&peerInfo.peer_addr, macAddr, 6);  // Add mac adress of peer (in this case broadcast to everyone)
    memcpy(&peerInfo.lmk, lmk, 16);           // Add Local Master Key (LMK) of peer
    peerInfo.encrypt = true;                  // Enable encryption
    esp_now_add_peer(&peerInfo);              // Add the peer to the list
    done();
  }
}

// Util for long messages
void sendLongMessage(const char* input_data, const uint8_t* macAddr) {
  int total_messages = (strlen(input_data) + MAX_DATA_SIZE - 1) / MAX_DATA_SIZE;
  int attempts = 1;
  char buffer[37];
  sprintf(buffer, "%u", esp_random());

  for (int i = 0; i < total_messages; i++) {
```

```
    message msg;
    strncpy(msg.id, buffer, sizeof(msg.id));
    msg.count = i;
    msg.total = total_messages;

    int length = strlen(input_data) - i * MAX_DATA_SIZE;
    if (length > MAX_DATA_SIZE) length = MAX_DATA_SIZE;
    strncpy(msg.data, &input_data[i * MAX_DATA_SIZE], length);
    if (length < MAX_DATA_SIZE) msg.data[length] = '\0';  // Ensure null termination

    esp_err_t result = esp_now_send(macAddr, (const uint8_t*)&msg, sizeof(msg));  // Send message

    if (result != ESP_OK) {
      if (attempts++ >= 3) {
        Serial.println("Could not send long message. Abbort...");
        reset();
        break;
      }
      --i;
    } else {
      attempts = 1;
    }
  }
}

// Util for long messages
String receiveLongMessage(const uint8_t* macAddr, const uint8_t* data, int len) {
  message* msg = (message*)data;  // Cast the data to a message

  // Find or create the message in the messages array
  MessageRec* fullMessage = NULL;
  for (auto& message : messages) {
    if (strcmp(message.id, msg->id) == 0) {
      fullMessage = &message;
      break;
    } else if (message.received == 0) {
      strcpy(message.id, msg->id);
      message.total = msg->total;
      fullMessage = &message;
      break;
    }
  }

  if (fullMessage == NULL) return "";  // If message couldn't be found or created, return an empty string

  // Store this part of the message
  strcpy(fullMessage->parts[msg->count].data, msg->data);
  fullMessage->parts[msg->count].index = msg->count;
  fullMessage->received++;

  // If received all parts of the message, combine them into a single string
```

```
  if (fullMessage->received == fullMessage->total) {
    String fullMessageStr = "";
    for (int i = 0; i < fullMessage->total; i++) fullMessageStr += fullMessage->parts[i].data;

    fullMessage->received = 0;  // Reset the message
    return fullMessageStr;      // Return the full message
  }

  return "";  // If not all parts of the message have been received, return an empty string
}

void reset() {
  STATE = SEARCHING;
  time_elapsed = 0;
  esp_now_register_recv_cb(receiveBroadcast);  // Register the recieve callback
  pixels.setPixelColor(0, pixels.Color(0, 0, 17));
  pixels.show();
}

void done() {
  STATE = DEFAULT;
  time_elapsed = 0;
  esp_now_unregister_recv_cb();
  freeRSAKey();
  Serial.println("Onboarding to " + formatMacAddress(gatewayMAC) + " complete\n");
  pixels.setPixelColor(0, pixels.Color(0, 17, 0));
  pixels.show();
}

// Helpers
// Formats MAC Address for prints
String formatMacAddress(const uint8_t* macAddr) {
  char res[18];
  snprintf(res, sizeof(res), "%02x:%02x:%02x:%02x:%02x:%02x", macAddr[0], macAddr[1], macAddr[2], macAddr[3], macAddr[4], macAddr[5]);
  return String(res);
}

void hexStringToByteArray(const String& hexString, uint8_t* byteArray, int byteArrayLength) {
  for (int i = 0; i < byteArrayLength; i++) {
    String hexByte = hexString.substring(i * 2, i * 2 + 2);
    byteArray[i] = (uint8_t)strtol(hexByte.c_str(), nullptr, 16);
  }
}

// ENCRYPTION
// Function to generate a 128-bit AES key and return it as a string
String generateAESKey() {
  mbedtls_entropy_context entropy;  // Context for entropy collection
  mbedtls_entropy_init(&entropy);   // Initialize entropy context to gather entropy used for random number generation
```

```

  mbedtls_ctr_drbg_context ctr_drbg;  // Context for the CTR_DRBG random number generator
  mbedtls_ctr_drbg_init(&ctr_drbg);   // Initialize the CTR_DRBG context

  uint32_t randomNumber = esp_random();
  char personalization[11];  // Personalization string for the DRBG seeding
  sprintf(personalization, "0x%08X", randomNumber);
  // Seed the CTR_DRBG context with entropy collected plus a personalization string for additional randomness
  mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (const unsigned char*)personalization, strlen(personalization));

  unsigned char key[16];  // Buffer to hold the 128-bit key (16 bytes)
  // Generate a random 128-bit key using the seeded CTR_DRBG context
  mbedtls_ctr_drbg_random(&ctr_drbg, key, sizeof(key));

  String keyHex = "";  // String to hold the hexadecimal representation of the key
  for (unsigned char i : key) {
    char hex[3];              // Temporary buffer to hold each byte in hex format
    sprintf(hex, "%02X", i);  // Format each byte of the key as two hexadecimal characters
    keyHex += hex;            // Append the hex string to the keyHex string
  }

  mbedtls_ctr_drbg_free(&ctr_drbg);  // Free the CTR_DRBG context to release any associated resources
  mbedtls_entropy_free(&entropy);    // Free the entropy context to release any associated resources

  return keyHex;  // Return the hexadecimal string representation of the key
}

// Function to initialize and generate RSA keys
void initializeRSAKey() {
  mbedtls_entropy_context entropy;  // Context for entropy collection
  mbedtls_entropy_init(&entropy);   // Initialize entropy context

  mbedtls_ctr_drbg_context ctr_drbg;  // Context for random number generator
  mbedtls_ctr_drbg_init(&ctr_drbg);   // Initialize CTR_DRBG context

  uint32_t randomNumber = esp_random();
  char personalization[11];  // Personalization string for the DRBG seeding
  sprintf(personalization, "0x%08X", randomNumber);
  mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (const unsigned char*)personalization, strlen(personalization));
  // Seed the DRBG

  mbedtls_rsa_init(&rsa, MBEDTLS_RSA_PKCS_V15, 0);  // Initialize RSA
```

```
   context
  int ret = mbedtls_rsa_gen_key(&rsa, mbedtls_ctr_drbg_random, &ctr_drbg
   , RSA_KEY_LENGTH, RSA_EXPONENT);
  // Generate RSA key pair

  if (ret != 0) {
    Serial.print("Failed to generate RSA key with error code: ");
    Serial.println(ret);
  }
  if (mbedtls_rsa_check_privkey(&rsa) != 0) {
    Serial.println("Generated RSA private key is not valid.");
  }

  mbedtls_ctr_drbg_free(&ctr_drbg);  // Free the DRBG context
  mbedtls_entropy_free(&entropy);    // Free the entropy context
}

// Function to encrypt data using an external RSA public key
String encryptRSA(const String& data) {
  mbedtls_pk_context pk;  // Public key container
  mbedtls_pk_init(&pk);   // Initialize the public key container

  // Parse the public key from provided PEM string
  if (mbedtls_pk_parse_public_key(&pk, (const unsigned char*)pem_peer.
   c_str(), pem_peer.length() + 1) != 0) {
    mbedtls_pk_free(&pk);
    return "";  // Return empty if public key parsing fails
  }

  // Encrypt the data
  unsigned char output[1024];  // Buffer to hold encrypted data
  size_t olen;

  mbedtls_ctr_drbg_context ctr_drbg;
  mbedtls_entropy_context entropy;
  mbedtls_entropy_init(&entropy);
  mbedtls_ctr_drbg_init(&ctr_drbg);
  uint32_t randomNumber = esp_random();
  char personalization[11];  // Personalization string for the DRBG
   seeding
  sprintf(personalization, "0x%08X", randomNumber);
  // Seed the CTR_DRBG context with entropy collected plus a
   personalization string for additional randomness
  mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (
   const unsigned char*)personalization, strlen(personalization));

  int ret = mbedtls_pk_encrypt(&pk, (const unsigned char*)data.c_str(),
   data.length(), output, &olen, sizeof(output), mbedtls_ctr_drbg_random
   , &ctr_drbg);

  mbedtls_pk_free(&pk);
  mbedtls_ctr_drbg_free(&ctr_drbg);
  mbedtls_entropy_free(&entropy);

  if (ret != 0) return "";  // Return empty if encryption fails

```

```
  String encHex = "";
  for (size_t i = 0; i < olen; i++) {
    char hex[3];
    sprintf(hex, "%02X", output[i]);
    encHex += hex;
  }

  return encHex;  // Return the hex string of the encrypted data
}

// Function to decrypt data using RSA
String decryptRSA(const String& encHex) {
  if (mbedtls_rsa_check_privkey(&rsa) != 0) {
    Serial.println("RSA private key is not valid.");
  }

  unsigned char encData[1024];  // Buffer to store the encrypted data in binary form
  size_t encIndex = 0;          // Index for filling the encData buffer

  // Convert hexadecimal string back to binary data
  for (size_t i = 0; i < encHex.length(); i += 2) {
    sscanf(encHex.c_str() + i, "%02X", &encData[encIndex++]);  // Parse two hexadecimal characters at a time_elapsed
  }

  unsigned char output[1024];  // Buffer to hold the decrypted data
  size_t olen;                 // Variable to store the length of the decrypted data

  mbedtls_ctr_drbg_context ctr_drbg;  // Context for the CTR_DRBG random number generator
  mbedtls_entropy_context entropy;    // Context for entropy collection
  mbedtls_entropy_init(&entropy);     // Initialize the entropy context
  mbedtls_ctr_drbg_init(&ctr_drbg);   // Initialize the CTR_DRBG context

  uint32_t randomNumber = esp_random();
  char personalization[11];  // Personalization string for the DRBG seeding
  sprintf(personalization, "0x%08X", randomNumber);
  // Seed the CTR_DRBG context with entropy collected plus a personalization string for additional randomness
  mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy, (const unsigned char*)personalization, strlen(personalization));

  // Decrypt the data using the private key
  int ret = mbedtls_rsa_pkcs1_decrypt(&rsa, mbedtls_ctr_drbg_random, &ctr_drbg, MBEDTLS_RSA_PRIVATE, &olen, encData, output, sizeof(output));

  mbedtls_ctr_drbg_free(&ctr_drbg);  // Free the CTR_DRBG context
  mbedtls_entropy_free(&entropy);    // Free the entropy context

  if (ret != 0) {
    Serial.print("Decryption failed with error: ");
    Serial.println(ret);
```

```
    return "";
  }

  if (mbedtls_rsa_check_privkey(&rsa) != 0) {
    Serial.println("RSA private key is not valid.");
  }

  return String((char*)output);  // Convert the decrypted binary data back to a string and return it
}

String exportPublicKey() {
  char buf[626];  // Ensure buffer is large enough for the key
  mbedtls_pk_context pk;
  mbedtls_pk_init(&pk);  // Initialize the PK context

  // Setup the PK context to hold an RSA key
  if (mbedtls_pk_setup(&pk, mbedtls_pk_info_from_type(MBEDTLS_PK_RSA)) != 0) {
    mbedtls_pk_free(&pk);
    return "";
  }

  // Copy the RSA context to the PK context
  mbedtls_rsa_context* rsa_copy = mbedtls_pk_rsa(pk);
  mbedtls_rsa_copy(rsa_copy, &rsa);  // Correctly copy RSA context

  // Check if the public key can be written into buffer
  if (mbedtls_pk_write_pubkey_pem(&pk, (unsigned char*)buf, sizeof(buf)) < 0) {
    mbedtls_pk_free(&pk);
    return "";  // Return empty string on failure
  }

  mbedtls_pk_free(&pk);  // Free the PK context
  return String(buf);    // Return the public key in PEM format
}

// Function to clean up RSA context when no longer needed
void freeRSAKey() {
  volatile char* p = const_cast<char*>(pem_peer.c_str());  // Access the underlying character array of the string
  size_t len = pem_peer.length();                          // Get the length of the string
  while (len--) *p++ = 0;                                  // Overwrite each character with zero
  pem_peer.clear();                                        // Clear the string to remove all content and reduce its size to zero

  secureZeroMemory(&rsa, sizeof(rsa));
  mbedtls_rsa_free(&rsa);  // Free the RSA context and all associated resources
}

void secureZeroMemory(void* ptr, size_t size) {
  volatile uint8_t* p = (volatile uint8_t*)ptr;
```

```
  while (size--) *p++ = 0;
}
```

Listing B.1: Complete Code of Client

B.1 MessageQueue.h

```
#ifndef MESSAGE_QUEUE_H
#define MESSAGE_QUEUE_H

typedef struct measure {
  int temperature;
  int humidity;
  int temperatureAlarm = 25;
  int humidityAlarm = 50;
} measure;

class MessageQueue {
private:
  measure *queueArray;
  int capacity;
  int front;
  int rear;
  int count;

public:
  MessageQueue(int size = 100) {
    capacity = size;
    queueArray = new measure[size];
    front = 0;
    rear = -1;
    count = 0;
  }

  ~MessageQueue() {
    delete[] queueArray;
  }

  void enqueue(measure item) {
    if (!isFull()) {
      rear = (rear + 1) % capacity;
      queueArray[rear] = item;
      count++;
    }
  }

  measure dequeue() {
    measure item;
    if (!isEmpty()) {
      item = queueArray[front];
      front = (front + 1) % capacity;
      count--;
    }
    return item;
  }
```

```

  measure peek() {
    measure item;
    if (!isEmpty()) {
      item = queueArray[front];
    }
    return item;
  }

  int size() {
    return count;
  }

  bool isEmpty() {
    return (count == 0);
  }

  bool isFull() {
    return (count == capacity);
  }
};

#endif
```

Listing B.2: Custom Library for MessageQueue used in Client