ELEC-H423 - Group 11
Labs project

Dujardin William - Gardeur Nicolas
Meunier Arnaud - Roland Babette

December 2025

Contents

1 Introduction

The objective of this project is to design, implement, and secure a distributed system composed of two ESP32 devices and a custom Python client communicating through an MQTT broker. The system includes both required functionalities, such as LED control and telemetry collection, as well as custom features, most notably an interactive Even/Odd game that introduces additional security challenges.

This report presents the architecture of the implemented features, the threat model applied to the system, and the security mechanisms designed to mitigate the identified risks. The goal is not only to demonstrate functional correctness but also to highlight how security principles can be practically enforced within the constraints of embedded systems and low-resource communication protocols.

2 Base Concept

This section outlines all features implemented in the project, detailing the technologies used and how they were applied. Its purpose is to present the overall design and functionality rather than the security mechanisms, which are covered in later sections. The content is divided into three parts: the required features, our custom features and the project schematics.

2.1 Required Features

- **Led Manipulation**
  This feature uses the two LEDs and the button provided to us. By default, the red LED remains illuminated whenever the ESP is connected to a power source. When the button is pressed, the red LED turns off and the white LED turns on. Once the button is released, the white LED turns off and the red LED turns back on.

- **Telemetry Collection**
  This feature uses the DHT11 sensor provided to us. It collects the room's humidity and temperature in Celsius. After the data is captured, it is displayed in the Arduino IDE's serial monitor and published to the MQTT server under the topic *telemetry/espX* (where X represents the ESP unit number, as we are using two devices, one client per device).

- **Telemetry Display**
  To visualize the collected data in chart form, a third MQTT client (a client different than the esp) is used. This client exists in two versions: an external tool called MQTT Explorer and a custom Python client. MQTT Explorer was initially used when no security mechanisms were in place, as it offers a built-in charting feature and is easy to install and use. The custom Python client is the version currently in use; it was developed to support the implemented security features and to display the data in a chart. In both cases, the data is retrieved by subscribing to the *telemetry/espX* topic.

- **Communication Management**
  - All communications in this project are implemented using the MQTT protocol.
  - The core of the project is the MQTT Server/Broker, which serves as the communication intermediary between the various clients. We selected the recommended broker, Mosquitto, which is hosted on the central computer and listens on port 1883 using IPv4.
  - The ESP devices have two types of connections: serial and Wi-Fi. The serial connection physically links the ESPs to the central computer for displaying debug information on the Arduino IDE Serial Monitor. The Wi-Fi connection places the ESPs on the same network as the central computer, enabling them to send data remotely.
  - The custom Python client relies on its host for the connection. In our case, it is a VM acting as another computer on the same network as the central computer.

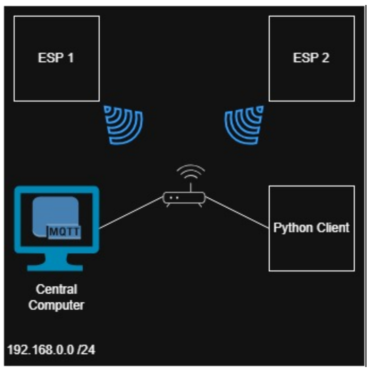

Figure 1: Communication inside the project

2.2 Custom Features

- **Even/Odd game**

  This feature implements a small interactive game using three LEDs (red, yellow, and green) and a button. The game can be started either by pressing the dedicated game button (different from the required-feature button) or by receiving a game invitation from the other ESP device.

  When the game starts, the red LED turns on to indicate that the game is active. The player can then choose between EVEN and ODD by short-pressing the button:

  – Yellow LED ON $\rightarrow$ EVEN

  – Yellow LED OFF $\rightarrow$ ODD

  After making a selection, the player confirms it by performing a long press on the button. The device will then wait for the second ESP's input. Once both players have submitted their choices, the system compares them and displays the result:

  – Green LED blinks $\rightarrow$ both players chose the same option, meaning a win

  – Red LED blinks $\rightarrow$ players chose different options, meaning a loss

  After the result is displayed, the game resets automatically and all LEDs turn off.

  Although the game by itself is relatively simple, its main purpose is to serve as a practical scenario to implement a commitment scheme, preventing players from cheating by changing their choice after seeing the opponent's selection which is a more interesting functionality to implement. As this is a security-related topic, additional technical details will be explained in the following section.

2.3 Project schematics

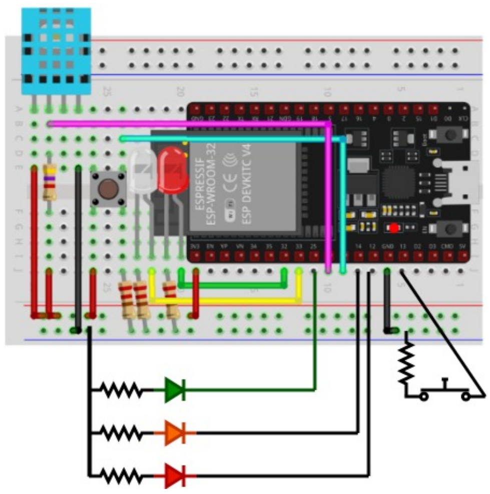

Figure 2: Project schematics

The required feature uses the provided schematics without modification.
For the custom feature, we added three LEDs and one additional button. These components share the same ground as the rest of the circuit. The following GPIO pins are used for input/output:

- GPIO 13: Button
- GPIO 12: Red LED
- GPIO 14: Yellow LED
- GPIO 25: Green LED

3 Security Analysis

3.1 Data Flow Diagram

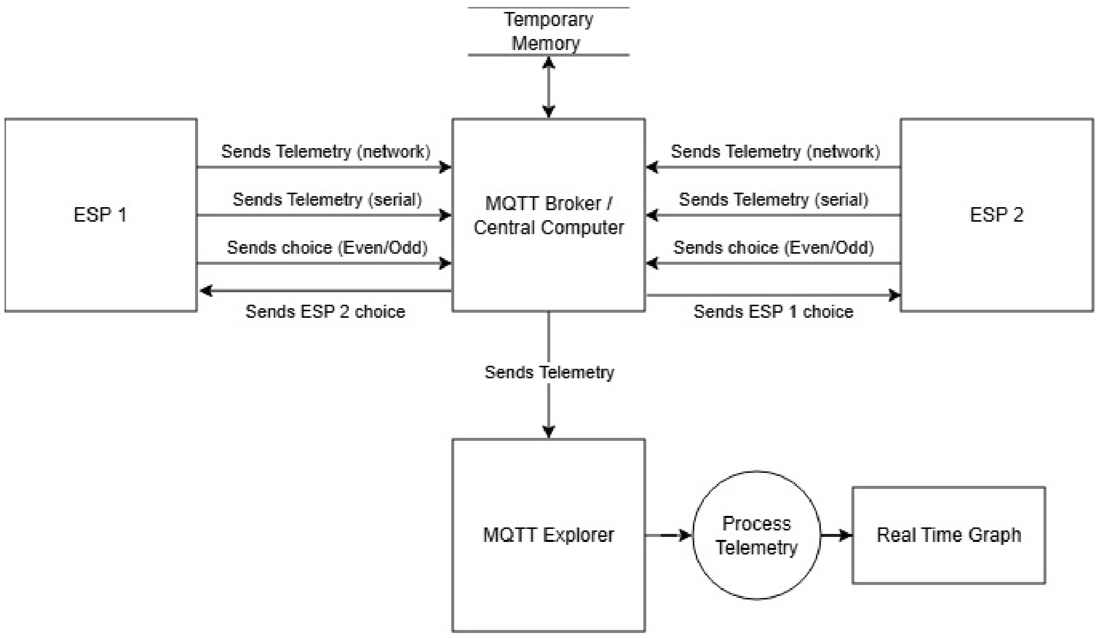

Figure 3: Data Flow Diagram of the project without security

3.2 Assumptions

- Physical security is out of scope. We assume that only trusted and authorized personnel has access to the legitimate devices.
- Network is not fully trusted.
- MQTT topic names are not critical information. As explained later, topic cannot be hidden efficiently but the scheme used when handling data prevent content leak. In our case, the attacker know the "what" but not the data content (ie. Knows telemetry but not the value of it).

3.3 Threat Model

STRIDE Category	Threat Description	Impact / Consequence
Spoofing	Attacker impersonates the ESP32 devices or Python MQTT client.	Fake data is injected into the system. Unauthorized clients receive sensor data. Fake clients send fake choices
Tampering	Modification of MQTT messages during transit (telemetry or choice).	Corrupted or falsified telemetry and choice. Legitimate clients modify his choice to win.
Repudiation	No reliable proof that a specific ESP32 sent or did not send certain data.	Fake data is injected. Client can deny their choice or accuse a modification.
Information Disclosure	Data are in plaintext due to the lack of TLS	Exposure of sensor data and MQTT credentials. Unauthorized access to topics. Client can know what choice will result in a win.
Denial of Service	Flooding the broker.	System becomes unavailable. Loss of telemetry data. Game is unplayable.
Elevation of Privilege	None	None

Table 1: STRIDE Threat Model for the project

4 Security Implementation

4.1 MQTT Broker

Very little can be done to improve security in this area. Without TLS, any form of client authentication is inherently insecure, as credentials are transmitted in plaintext. This allows an attacker to intercept authentication data and impersonate a legitimate client from the perspective of the MQTT broker. Consequently, additional security mechanisms, such as restricting topic read/write access, can be bypassed with minimal effort. While one could argue that limiting authorized topics for all clients might provide some protection, this is largely ineffective as long as legitimate topics cannot be fully controlled.
Furthermore, the system remains vulnerable to denial-of-service attacks via topic flooding, and no practical mitigation exists within the scope of this project. Despite this vulnerability, the likelihood of an attacker exploiting it is considered low, and therefore the overall risk for this use case is minimal.

4.2 Key management

Our security implementation makes use of both asymmetric and symmetric cryptography, managed as follows:

- **RSA Keys**
  We use RSA key pairs to ensure **confidentiality**, **authenticity**, and **integrity**. The specific usage of these keys is described in subsequent sections. The key pairs were generated in advance using OpenSSL and hardcoded into each device that requires them (the two ESP32s and the custom Python client). This approach was chosen because we have full control over the devices and the number of devices is limited.

- **AES Keys**
  Since the payloads are relatively large, encrypting them directly with RSA is not practical. Therefore, we use one-time AES symmetric keys along with an initialization vector (IV). These keys and IVs are generated randomly using a dedicated random generator to maximize security. Each AES key is used only for the encryption and decryption of a single payload and is never stored.

4.3 Telemetry

While we cannot prevent attackers from reading or writing to our telemetry topics, it is possible to ensure that the content of the messages remains unreadable and that only messages sent by legitimate users are

accepted. This can be achieved by using cryptographic keys (both asymmetric and symmetric) and the associated mechanisms.
In the case of the telemetry data, the flow is as follows:

1. Generate a string containing the telemetry data.
2. Hash this string and sign it using the private key of ESP that generated the data.
3. Concatenate the telemetry string and the signature into a single payload.
4. Generate a one-time-use AES symmetric key.
5. Encrypt the payload with the symmetric key (the payload is too large to be encrypted directly with the public key).
6. Encrypt the AES key with the recipient's public key, in this case, the custom Pyhton client.
7. Concatenate the two ciphertexts into a single message and encode it in Base64.
8. Send the resulting message via MQTT.

This approach provides several guarantees:

- An illegitimate listener cannot determine the content of the message, ensuring **confidentiality**.
- Only the intended recipient, who can decrypt the AES key, is able to access the payload, further ensuring **confidentiality**.
- Decrypting the payload allows the recipient to verify the **authenticity** of the sender via the digital signature.
- The signature ensures that the telemetry data has not been tampered with, preserving **integrity** by comparing the hashs.
- Any failure in decryption or signature verification indicates that the message is either illegitimate or has been tampered with and should be discarded.

4.4 Even/Odd game

In this game, there are several security challenges that need to be addressed:

- Ensuring the **integrity**, **authenticity**, and **confidentiality** of the data.
- Preventing players from changing their choice once it has been made.
- Preventing players from learning the choice of the other player before the reveal.

At this stage, the first issue can be addressed by following the same data flow described in Section 4.3 Telemetry and using the public key of the other ESP32 device. Doing so will provide security guarantees for every data exchange required for the Even/Odd game.

To address the second and third issues, we implement a **commitment scheme**. The idea is that each player sends their choice to the other in a way that is unreadable and impossible to deduce, effectively obfuscating the choice. Once this exchange is complete, each player sends the key required to deobfuscate the initial message. This allows both players to verify the choices and determine the outcome of the game (win or lose) without any possibility of cheating, resolving the issues we have.

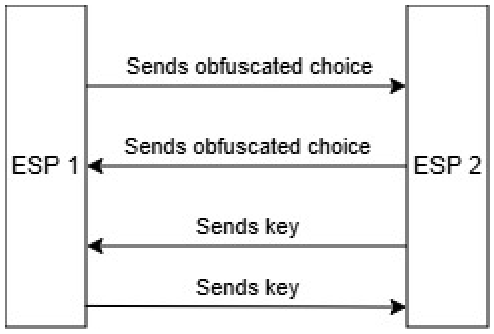

Figure 4: Commitment scheme illustration

In practice, the flow of the commitment scheme in our project is as follows:

1. Player 1 makes a choice.
2. The choice is concatenated with a random salt, which acts as the "key" mentioned previously. The salt is generated using a dedicated random generator and is sufficiently large to prevent brute-force attempts to deduce the choice in advance.
3. The concatenated value is then hashed to create the **commitment**, which is sent to the other player. The salt is stored locally for later verification.
4. Upon receiving the commitment, Player 2 also makes a choice and follows the same procedure as Player 1.
5. Once both commitments have been exchanged, each player sends their plaintext choice concatenated with their local salt.
6. Upon receiving the choice and salt from the other player, each participant hashes the concatenated value and compares it with the previously received commitment. If the hashes match, it confirms that no cheating occurred. If they do not match, it indicates that the other party altered their choice mid-way, signifying an attempted cheat.

5 Threat and Mitigations Summary

Threat	**Mitigation Implemented**
Impersonation of ESP32 or Python MQTT client (Spoofing)	Mitigated using **RSA key pairs**. Only legitimate devices can sign valid data; forged messages fail verification.
Modification of MQTT messages during transit (Tampering)	Fixed using **digital signatures**. Any modification results in signature verification failure and the message is discarded.
Disclosure of telemetry or game choices due to lack of TLS (Information Disclosure)	Protected using **AES encryption** for each payload. Only the intended recipient can decrypt the AES key (RSA-encrypted).
Players changing their Even/Odd choice after seeing the opponent's (Cheating)	Mitigated with a **commitment scheme** (hash(choice + salt)). The salt + choice sent later must match the commitment.
Repudiation (denying having sent a message)	Digital signatures provide **non-repudiation**: a sender cannot deny a signed telemetry packet or commitment.
Denial-of-Service through topic flooding (DoS)	**Not fully preventable**. MQTT ACLs reduce exposure, but without TLS attackers can still impersonate clients. Considered low-risk for this project.

Table 2: Summary of threats and corresponding mitigation measures

6 Conclusion

The project successfully demonstrates how a simple MQTT-based architecture can be reinforced with strong security guarantees, even in the absence of TLS. By combining RSA key pairs, one-time AES encryption, and digital signatures, we ensure the confidentiality, authenticity, and integrity of all data exchanges. Additionally, the implementation of a commitment scheme provides cheating resistance for the Even/Odd game, addressing fairness and trust between the two ESP32 devices.

While certain limitations remain, most notably the inability to fully mitigate denial-of-service attacks or secure client authentication at the broker level without TLS, the adopted measures significantly reduce the overall attack surface and provide robust protection against realistic threats within the project's scope.