Lab Report

Group Members

Adjei Ernest

Volonterio Luca

Fabian Dandy

Hussaini Mohmmad Qasim

ELEC-H423 - Wireless and Mobile Communication

Supervised by Navid Ladner

Université Libre de Bruxelles

Academic Year 2025–2026

Contents

1 Introduction

The Internet of Things (IoT) is becoming very popular in the IT industry, but this technology often has security risks. For our lab project, we wanted to understand these risks by building our own secure IoT system. We used the ESP32 microcontroller because it is affordable and has built-in WiFi, which makes it great for this kind of experiment.
Our main goal was to collect temperature and humidity data from sensors and send it to a server running on our computer. However, we did not just want to send the data; we wanted to make sure it was safe. We focused on implementing security features to ensure the authentication, integrity, and encryption of the data, so that no one can tamper with our messages.

2 System Setup and Activity Flow

Our physical setup, which you can see in Figure 1, consists of two identical nodes connected to a laptop.

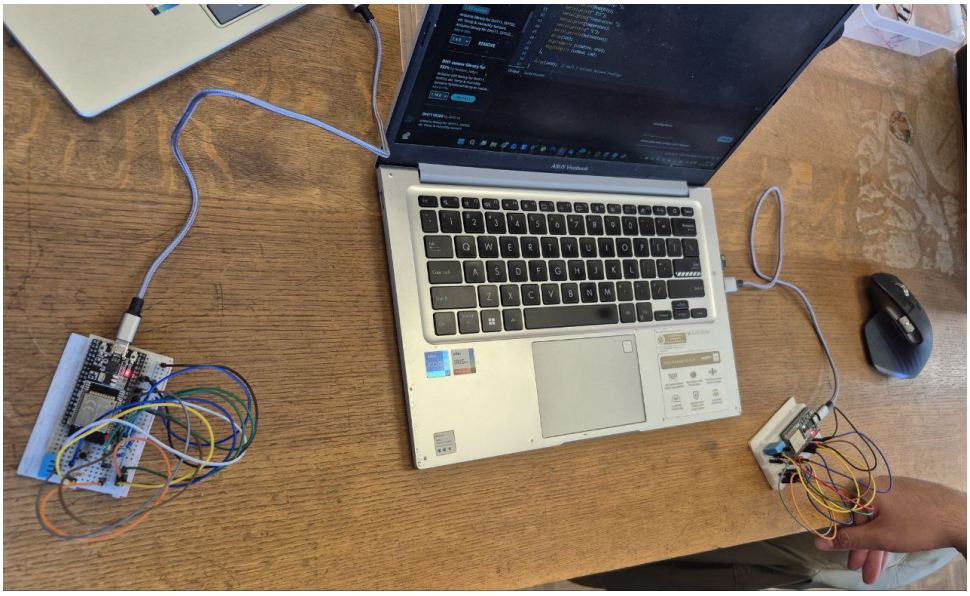

Figure 1: Physical setup of the two ESP32 nodes connected to the laptop.

We used an ASUS laptop as our central station. It powers the boards through USB cables and runs the server that displays the data charts.
For the nodes, we used the ESP32v4 board placed on a breadboard. We connected a blue DHT11 sensor to measure the environment, along with a push button and a red LED. We also added the necessary resistors to wire everything correctly and protect the components.
The extra feature that we implement in the ESP32 is a temperature monitoring with a lock mechanism. This device can be implemented in various usecase such as sensitive-temperature container monitoring, smart fridge and etc. The main functionality of this device are:

1. Use an ESP32 microcontroller to read both temperature and humidity.

2. Send the data to the server using MQTT.

3. View the data on the web dashboard.

4. Receive a command to unlock the lock from the dashboard.

5. The command will only be executed after the button is pressed.

Due to constrain of time that we have, we don't implement the feature using a smart lock. Instead we use the LED to replace the lock, so when the command is sent and executed the white LED will blink indicating the lock is unlocked.

3 MQTT-Based System Architecture and Threat Model

3.1 Introduction to MQTT

With the hardware platform and device functionality established, the next design step was to define a communication architecture that enables reliable and secure data exchange between the ESP32 nodes and the backend services. This communication layer is central to the system, as it carries both sensor telemetry and security- critical control information such as authentication challenges and responses. To support these requirements, we adopted the Message Queuing Telemetry Transport (MQTT) protocol as the backbone of the system's communication architecture.

MQTT is a lightweight publish subscribe protocol widely used in IoT environments due to its low bandwidth consumption and minimal computational overhead. Communication is mediated by a central broker that routes messages between publishers and subscribers based on topic subscriptions, enabling loose coupling between system components, as seen in Figure 2 . These properties make MQTT well suited for resource-constrained devices such as the ESP32.

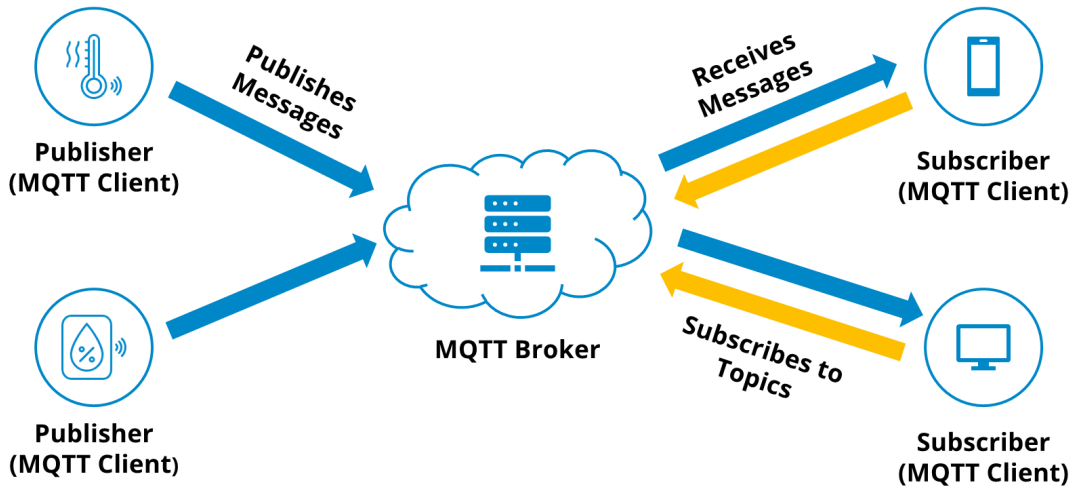

Figure 2: General MQTT publish–subscribe architecture showing publishers, broker, and subscribers.

3.2 Architecture Overview of the Implemented System

The implemented prototype consists of two ESP32 nodes, an MQTT broker, and an application server. Each ESP32 node is equipped with temperature and humidity sensors and a button/LED interface, and participates in bidirectional MQTT communication.

The ESP32 nodes primarily act as publishers by sending encrypted telemetry data (sensor readings and control status) to the MQTT broker. In addition, they subscribe to dedicated control topics used during the authentication phase, enabling a challenge–response exchange with the application server.

The MQTT broker functions as a message router. It forwards messages between ESP32 nodes and the application server without modifying payload contents.

The application server act as the visual representation for the user as well as the user command interface for bidirectional communication. It subscribes to all telemetry and authentication-related topics, verifies message authenticity and integrity, decrypts valid payloads, maintains device state, and visualizes data on a web-based dashboard.

This bidirectional not only allow sending command to the ESP32 but also enable secure device authentication and protected data exchange while keeping the MQTT broker lightweight.

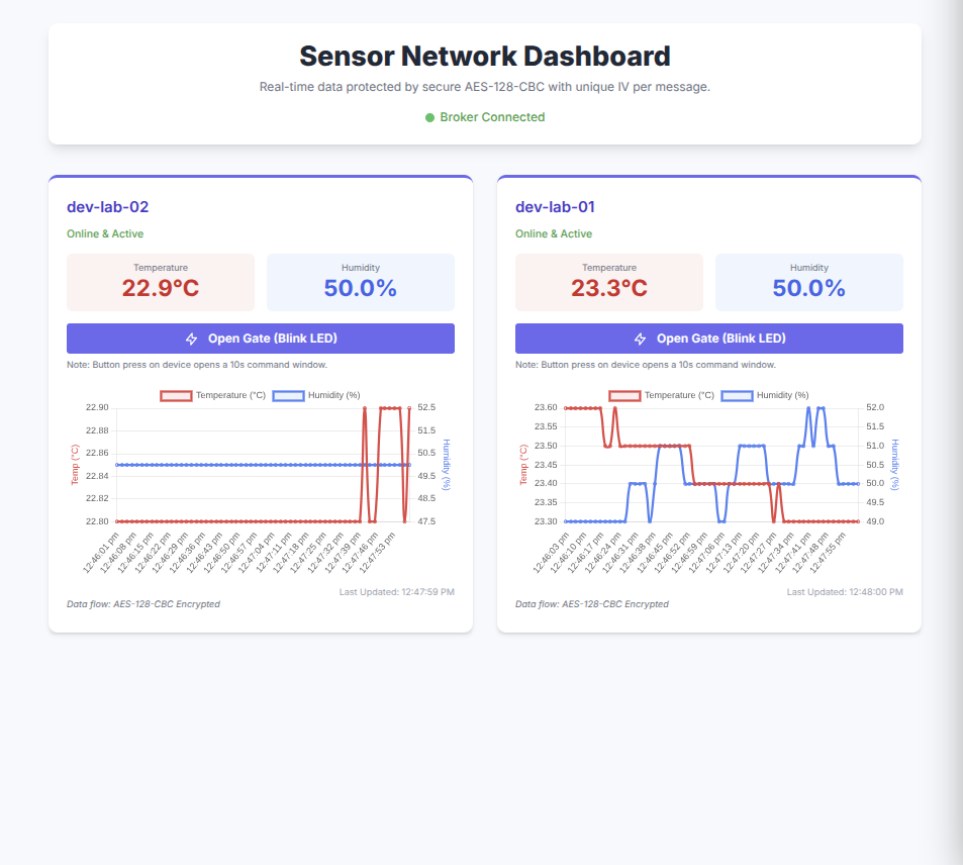

Figure 3: Simple Web-based dashboard to visualize data and send command

3.3 Component Overview

Table 1: Functional overview of system components.

Component	Primary Function
ESP32 Node A (Group 1)	Collects environmental sensor readings (temperature and humidity) , controls LED outputs, and publishes device telemetry to the MQTT broker. This device is also can receive a command to unlock the lock. The command is executed only after the user presses the button, which provides the input (button state) that controls its execution.
ESP32 Node B (Group 2)	Operates as a second, independent sensing node performing the same functions as Node A, enabling parallel data collection and multi-node communication.
MQTT Broker	Central message-routing service that receives published data from ESP32 nodes and distributes it to subscribed clients using the publish subscribe communication model.
Application Server	Subscribes to sensor telemetry from the MQTT broker, processes incoming data, stores historical readings, and visualises system behaviour through charts and dashboards for user consumption. Also have a command interface for user to send a command to ESP32.

3.4 End-to-End Data Flow

1. **Sensor acquisition (ESP32 nodes):** Each ESP32 reads DHT sensor data and button/LED status, then publishes these values to MQTT topics. Each ESP32 also subscribe to a topic from MQTT broker to retrieved data for unlocking a lock.

2. **Message routing (MQTT broker):** The broker receives each published message and forwards it to both the ESP32 and the application server.

3. **Data processing (application server):** The server subscribes to the telemetry topics, stores the messages, and renders visual representations. The server also send a command to the MQTT broker for the ESP32 to fetch.

3.5 Data-Flow Diagram and Trust Boundary

Figure 4 presents the Level 1 data-flow diagram (DFD) of the implemented system, including the trust boundary.

- The **trust boundary** (red dashed line) encloses the ESP32 nodes, which are physically controlled and securely provisioned. These devices store long-term symmetric keys and therefore constitute trusted endpoints.

- The **MQTT broker**, application server, and the network infrastructure lie outside the trust boundary and are treated as untrusted. Consequently, all messages crossing the boundary must be authenticated, integrity-protected, and confidential at the application layer.
- Bidirectional data flows exist across the trust boundary: authentication challenges and commands flow from the application server to the ESP32 nodes, while encrypted telemetry and authentication responses flow in the opposite direction.

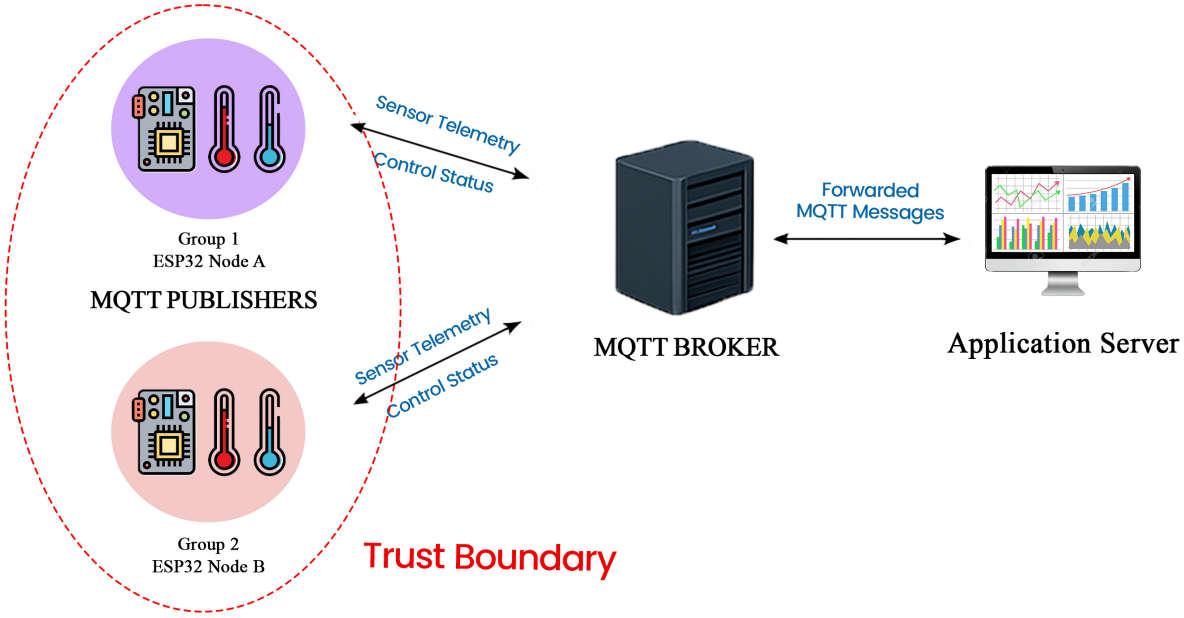

Figure 4: Data-flow diagram of the implemented MQTT-based IoT system. The trust boundary encloses the ESP32 nodes, while the broker and application server are treated as untrusted.

3.6 STRIDE Threat Model

STRIDE applicability by element type

Table 2: Applicability of STRIDE threat categories to system elements.

Element Type	S	T	R	I	D	E
ESP32 Node Processes	✓	✓	✓	✓		✓
Broker & Server Processes	✓	✓			✓	
Data Stores (Server DB, Logs)		✓	✓			
Data Flows (MQTT Messages)		✓		✓		

STRIDE vulnerabilities and mitigations per component

Table 3: STRIDE vulnerabilities and corresponding mitigations.

Component	STRIDE	Vulnerability	Mitigation Implemented
ESP32 Nodes	S, T, R, I, E	Device identity can be spoofed; messages can be replayed or modified; unauthorized command could result in escalation of privilege.	Pre-provisioned symmetric keys; HMAC challenge–response; encrypted NVS; secure boot; monotonic counters for replay protection.
MQTT Broker	S, T, D	Unauthorised publishers may push malicious data; topic flooding attacks may degrade availability.	Broker authentication and ACLs; rate limiting; server-side verification of HMAC and counters.
Application Server	T, R, I	Malicious messages may corrupt data or logic; telemetry databases may be altered.	HMAC verification; counter checks; restricted DB access; protected server-side key storage.
MQTT Data Flows	T, I	Messages may be intercepted, modified, or replayed over the network.	AES-CTR/AES-GCM encryption; HMAC integrity protection; IVs and counters for replay defence.

4 Security Implementation

4.1 Security Protocol for Confidentiality, Integrity, and Authentication

To address the main STRIDE threats on MQTT messages and the ESP32 nodes, we add a small symmetric-key security protocol on top of plain MQTT. Each ESP32 is given a random device identifier ($DevID$) and a secret key (K_auth) during provisioning. Both values are stored on the board and on the application server. $DevID$ is then used as the device identity in the MQTT topics (for example, $dev/\{DevID\}/telemetry$) instead of the MAC address, and K_auth is the shared secret that we use to authenticate the board, protect message integrity, and encrypt the payload to ensure confidentiality.

Device authentication is done with a simple challenge–response exchange over dedicated control topics. After the ESP32 connects to the broker with low-privilege MQTT credentials, the server sends a random challenge (N_{srv}) on a topic such as $dev/\{DevID\}/auth\_req$. The ESP32 replies on $dev/\{DevID\}/auth\_resp$ with its own nonce (N_{dev}) and a hash-based message authentication code (HMAC) computed over ($DevID, N_{\text{srv}}, N_{\text{dev}}$) using K_auth. The server recomputes the HMAC and only accepts the device as authenticated if the values match and N_{srv} is the current, unused challenge. In practice, this means an attacker cannot just replay an old

authentication message to pretend to be a valid node, because the challenge is fresh every time.

After a device is authenticated, every telemetry message includes a session counter and an HMAC to guarantee integrity and detect replays. Each time the ESP32 publishes sensor data, it increments a counter ($sess\_ctr$), encrypts the payload P to form ciphertext `ct`, and sends a structure of the form $\{dev\_id, ctr, ct, hmac\}$.

Confidentiality is ensured by encrypting the payload P on the ESP32 using AES-128 in Cipher Block Chaining (CBC) mode with PKCS#7 padding. The 16-byte AES key (K_aes) is derived directly from the first 16 bytes of the 32-byte K_auth secret. Crucially, a new, random 16-byte Initialization Vector (`IV`) is generated for every message. The resulting ciphertext (`ct`) is a single hex string composed of the concatenated `IV` followed by the encrypted data:

$$ct = Hex(\texttt{IV} \,\|\, \texttt{Ciphertext})$$

The `IV` length is 16 bytes (32 hex characters), ensuring it is unique and non-repeating for each message.

The HMAC for telemetry (device → server) messages is computed as

$$HMAC_{\text{telemetry}} = HMAC(K\_auth, DevID \,\|\, sess\_ctr \,\|\, ct)$$

On the receiving side, the JavaScript client (connected via WebSockets) keeps track of the last accepted counter per device ($last\_ctr$). For each new message it checks that $sess\_ctr > last\_ctr$ and verifies the HMAC. Messages with a wrong HMAC or a counter that does not increase are dropped. If the HMAC is valid, the client extracts the `IV` from the beginning of ct and decrypts the remainder of the string using K_aes to recover the cleartext readings.

4.1.1 Mechanism for Commands (Server → Device)

The protocol is bidirectional and uses a similar counter and HMAC mechanism for commands sent from the server to the device, but with key differences to preserve device availability and minimize attack surface:

1. **Command Window:** The ESP32 node **unsubscribes** from the $/cmd$ topic by default. The command topic is only subscribed to (the "window is opened") for a short, predetermined period (10 seconds) immediately after a button press. This prevents command messages from being accepted at any time.

2. **Separate Counter ($last\_server\_ctr$):** The command message uses a separate counter, $last\_server\_ctr$, which tracks the last command received **by the device**. The server increments its own counter ($server\_ctr$) for each command sent.

3. **Command Validation and Encryption:** The server encrypts the command payload using AES-128-CBC with a dynamically generated `IV`, resulting in ct_{cmd}. The command

message structure sent is $\{dev\_id, ctr, ct_{\text{cmd}}, hmac_{\text{cmd}}\}$, where the HMAC is calculated as:

$$HMAC_{\text{cmd}} = HMAC(K\_auth, DevID \parallel ctr \parallel ct_{\text{cmd}})$$

4. **Device Checks:** When the device receives a command, it performs three mandatory checks:
   - **Window Check:** Is the command window currently open?
   - **Replay Check:** Is the received ctr strictly greater than the stored $last\_server\_ctr$?
   - **Integrity Check:** Does the $HMAC_{\text{cmd}}$ verify correctly using K_auth?

Only if all three checks pass is the $last\_server\_ctr$ updated, and the payload decrypted and executed (e.g., blinking the LED). This way, any modification, injection, or replay of MQTT packets is caught at the application layer, even though the MQTT transport itself is not secure.

Overall, this protocol strengthens the laboratory setup without requiring TLS on the broker. Confidentiality is provided by mandatory payload encryption with AES-128-CBC using dynamically generated `IV`s, integrity and source authenticity are enforced by HMACs keyed with K_auth, and replay attacks are blocked by session counters and fresh nonces during authentication and command transmission. At the same time, availability is preserved: the broker still behaves like a simple message router, and potentially malicious or corrupted traffic is filtered at the edges (ESP32 and application server) before it can affect stored data or the visualisations shown to users.

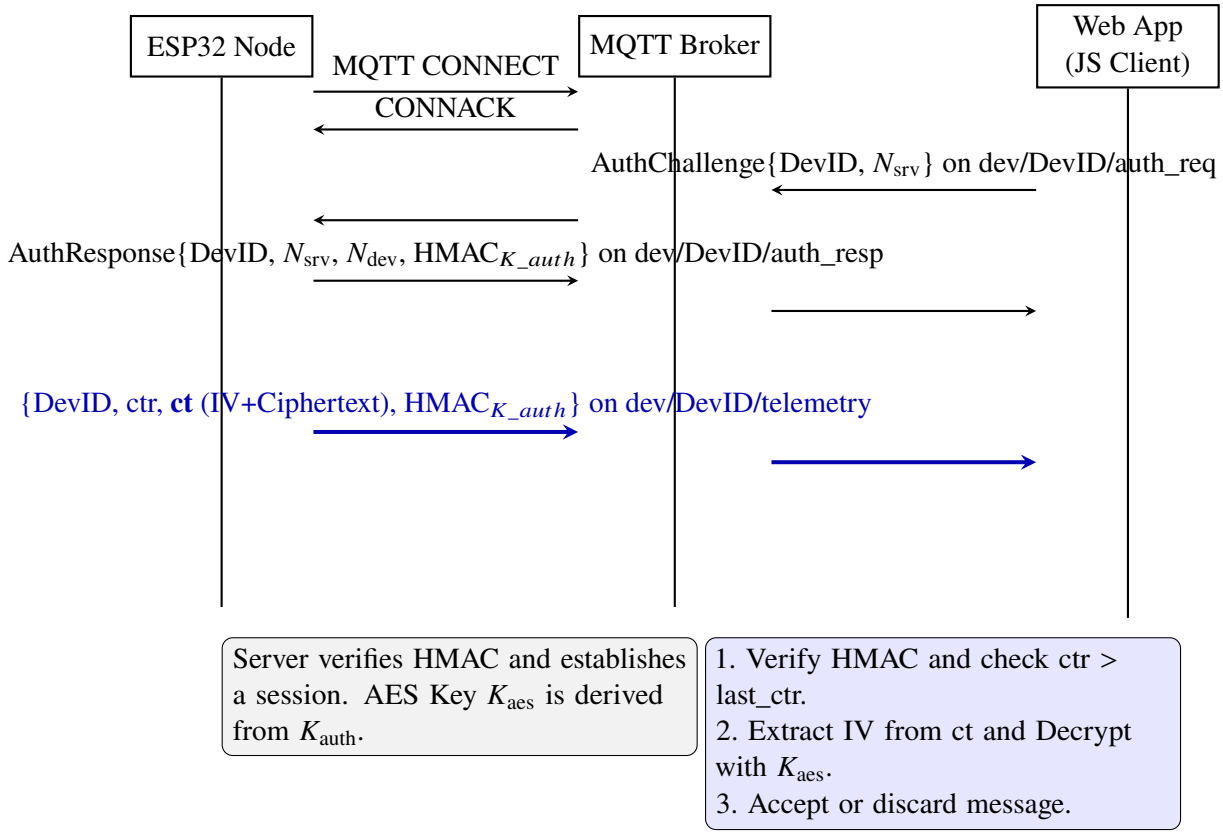

Figure 5: Application-layer authentication, integrity, and confidentiality protocol for **Telemetry** (Device → Server).

4.2 Securely store credential inside ESP32

The security protocol we created heavily relies on a pre-shared key inside the ESP32. However, storing important credentials such as the pre-shared key or Wi-Fi password cannot be as simple as hard coding the key directly, since static analysis is possible if an attacker gains access to the ESP32. To securely store credentials in the ESP32, we are going to use the Non-Volatile Storage (NVS) library with encryption. Furthermore, to ensure that only authorized firmware can be flashed to the ESP32, we will enable secure boot, allowing only signed firmware to be installed.

4.2.1 Non-Volatile Storage (NVS)

Non-Volatile Storage is a storage library available on the ESP32 that stores key-value pairs. This storage saves data in the device's flash memory, maintaining data persistence through restarts or power loss. The library also supports multiple data types such as strings, integers, and binary blobs, making it ideal for storing various types of credentials in this case.
Since the data is stored in flash, as opposed to volatile memory, it only needs to be provisioned once. To further strengthen credential storage, flash encryption can be enabled. Flash encryption on the ESP32 generates a unique AES-256 key during the first boot and stores it in efuse, which is protected from software access by default. This key is used by the NVS encryption feature to encrypt all data stored in flash memory. With this feature enabled, credentials cannot be

accessed through static analysis.
For the implementation, the device need to be flash by two different Firmware:

1. Provisioning Firmware that have all the credential such as Keys and Wifi password.

2. Application Firmware of the intended feature.

By doing this steps it will make sure that the key cannot be read by using static analysis attack as the latest program that was uploaded to the devices was working program that read the keys from the NVS memory. The next step that we need to implement are the secure boot to make sure the program cannot be changed to retrieve the key.

4.2.2 Secure boot

Secure boot is a security measure that is built inside ESP32. The idea of secure boot is that ESP32 on each boot will check the firmware installed in them by checking its digital signature. By checking the digital signature, ESP32 will only run firmware that is known by it. It's important that The ESP32 stores a cryptographic hash of the signing key in its efuse memory on first boot, making this trust decision permanent and hardware-enforced. This technique will further hardened the ESP32 to prevent a more direct attack on the devices such as injecting their own firmware and trying to read the credentials.

4.2.3 Steps to Implement Secure Boot and Flash Encryption

Enable Secure Boot and Flash Encryption via Arduino ESP32 toolchain
⇓
Generate Secure Boot signing key
⇓
Develop provisioning firmware to flash credentials into encrypted NVS
⇓
Execute provisioning firmware to initialise NVS encryption key
⇓
Generate flash encryption key and firmware digital signature
⇓
Flash final application firmware and securely access keys from NVS

Figure 6: Implementation steps for Secure Boot and Flash Encryption on ESP32

5 Conclusion

In conclusion, we successfully achieved our main goal of building a secure IoT system using the ESP32 microcontroller. We set up two nodes to collect temperature and humidity data and sent

this information to a local server using the MQTT protocol.
The most important part of our project was the custom security protocol we built. Since standard MQTT is not secure enough on its own, we added a layer of protection. We used a shared secret key and a challenge-response method to authenticate devices, which prevents unauthorized access. To stop replay attacks, where a hacker resends an old message, we implemented a counter that increases with every message. We also encrypted the data payload so that even if someone intercepts the message, they cannot read it.
Beyond the communication protocol, we also secured the physical device. We used the ESP32's Non-Volatile Storage (NVS) with Flash Encryption to safely store our passwords and keys. We also enabled Secure Boot to ensure that only our signed firmware can run on the board.
There was one limitation in our final result. Due to time constraints, we could not implement the full smart lock mechanism we originally planned. Instead, we used a red LED to simulate the lock status. Despite this change, the core system works well and proves that affordable hardware like the ESP32 can be made secure against common threats.