Université Libre de Bruxelles

Mobile and wireless networks

ELEC-H423

Project Report

December 2025

DEVOLDER Martin, DEVOLDER Virgile, BERNAGOU Esteban, PIERRET VAlentin

Contents

1 Introduction

Every day, the number of connected devices increases, and with it the need to ensure stable and secure communication between these devices. In a context where data confidentiality and integrity are essential, unsecured communication can easily be intercepted or manipulated.

This project addresses this issue by establishing a secure communication between two ESP32s via an MQTT channel. The goal is to securely transmit data from sensors and display it in real time on a second device and on a server. To achieve this, we have developed a lightweight security protocol adapted to the hardware constraints of microcontrollers.

The project involves several challenges: designing cryptographic and authentication protocols that are sufficiently robust while remaining compatible with the limited capabilities of the ESP32, and ensuring the confidentiality, integrity, and authenticity of the messages exchanged. These challenges guided the technical choices presented in this report.

2 Based Concept of the project

2.1 Main implementation from the labs

2.1.1 Reuse of laboratory equipment

The practical work provided us with several physical components that were directly integrated into our project:

- **Two LEDs** (one red and one white), used to signal important device states.
- **A physical button**, initially used in labs for simple interactions, but reused here to trigger a complete reset of the persistent configuration.
- **A DHT11/DHT22 sensor**, allowing the measurement of temperature or humidity.
- **Basic wiring** necessary for integrating these components.

It is important to note that the OLED screen used in the project was not part of the equipment provided in the labs: it is a personal addition intended to enhance the display of data on each ESP32.

2.1.2 Role of the button and LEDs in our project

In our implementation, the button and LEDs play an essential role in managing the ESP32 configuration:

- Pressing and holding the button triggers a **NVS reset** (non-volatile memory). This operation erases:

  - the locally stored public key,
  - the *client master key*,
  - the Wi-Fi configuration,
  - any other persistent data related to the protocol.

- As soon as the user presses the button, the **red LED lights up** to indicate that the reset procedure is in progress and that the user must hold down the button to confirm it.
- After 5 seconds of continuous pressing, the **white LED lights up**, indicating that the reset is confirmed and will be performed.

This logic ensures safe and predictable handling of the configuration, preventing accidental resets.

2.1.3 Laboratory-based system architecture

Our project is based on two ESP32 boards, each equipped with a DHT sensor. The first measures temperature while the second measures humidity. The two devices exchange their values so that each has a complete view of the environmental data. The OLED screen added to each ESP32 allows the following to be displayed locally:

- the value measured by the local sensor,
- the value received from the other ESP32.

Thus, each board represents a complete information point, combining two measurements from two separate locations.

2.1.4 Hardware integration

The hardware integrated into our project therefore consists of the following components:

- a DHT11/DHT22 sensor for measurement,
- a red LED and a white LED (inherited from practical work),
- a physical button with software debouncing, used to reset the NVS,
- an SSD1306 OLED screen added to the project and connected via an I2C bus.

2.1.5 Functional behaviour

The overall operation of the system, before adding security, is as follows:

1. Each ESP32 initializes its hardware (sensor, LEDs, button, OLED display).
2. The board periodically reads the value of its local sensor.
3. Each board also receives the measurement sent by the other ESP32.
4. The OLED display shows the local value and the remote value, combining the two pieces of information.
5. A "?" appears when no data has been received for more than 10 seconds.
6. Pressing and holding the button triggers the reset procedure: red LED immediately, white LED after 5 seconds, then persistent configuration erased.

2.1.6 Extensions beyond the labs

Our project differs from the original practical work thanks to the following improvements:

- the addition of an OLED screen for real-time display of combined data,
- the use of LEDs and the button to implement a complete NVS reset procedure,
- an architecture with two devices exchanging their respective measurements,
- a communication loss detection logic (timeout),

2.1.7 MQTT Communication

To ensure data exchange between our two ESP32s, we used the MQTT protocol as introduced in the practical exercises. In our implementation, we chose to use Mosquitto as the MQTT server, mainly because of its ease of deployment on Windows and Linux.

However, it is important to note that our project does not depend on Mosquitto in particular. Our entire architecture has been designed to work with **any standard MQTT broker**, as the protocol is based solely on the classic publish and subscribe operations defined in MQTT.

This independence from the broker is essential to our project, particularly because our security layer does not consider the broker to be a trusted entity. Thus, whether the broker is Mosquitto, EMQX, HiveMQ, a cloud service, or a custom implementation, the system works identically, with encryption and authentication being entirely managed at the ESP32 level and Key Management Service (KMS) level via our protocol.

2.2 Application scenario

In a post-apocalyptic world where reliable communications are a thing of the past, two survivors attempt to maintain a vital link between their respective shelters. Sector 7, where they are located, is ravaged by radioactive storms and pockets of corrosive moisture. To maximize their chances of survival, they have improvised an IoT system based on ESP32s powered by solar panels, each equipped with a different environmental sensor:

- **North Shelter** → measures temperature, essential for detecting dangerous heat rises.
- **South Shelter** → measures humidity, a critical factor in preventing corrosion or mould in food reserves.

In order to survive, both survivors need access to both pieces of information. They connected their ESP32s through an MQTT broker found in an abandoned communication station, but they cannot assume it is trustworthy. The network is unstable, possibly monitored, and could even be controlled by a hostile group still active in the area.
To counter this risk, the survivors rely on a more powerful device recovered from an old research bunker: a hardened computer acting as a **Key Management Service (KMS)**. Unlike the ESP32s, this machine has enough resources to host a small web interface. The KMS authenticates both shelters, distributes encryption keys, performs regular key rotation, and continuously records environmental measurements.
A third survivor, stationed in the bunker, uses the KMS dashboard to monitor the situation. From this secure location, he can visualize the evolution of temperature and humidity over time, allowing him to detect long-term trends, anticipate storms, and guide the other two shelters when conditions start to deteriorate. The web interface therefore becomes a strategic observation point rather than a simple technical interface.
Through this mechanism, each survivor permanently receives a secure message displayed on the screen of their ESP32, containing:

- the temperature measured by the North Shelter,
- the humidity measured by the South Shelter,
- or a “?” symbol if communication becomes suspicious or interrupted.

Beyond environmental monitoring, the system also supports emergency signals. By pressing the button of their ESP32 three times rapidly, a survivor can emit a **SOS alert**. This alert is encrypted, transmitted through the compromised MQTT broker, and recognized by the peer device. When received, the red LED of both ESP32s starts flashing, and the OLED screen displays a clear distress message. The KMS dashboard also highlights the alert, enabling the third survivor at the bunker to react immediately and coordinate a rescue if necessary.

Thus, despite a hostile environment, an unreliable network, and makeshift equipment, the survivors now possess a minimal yet secure communication infrastructure capable not only of exchanging critical environmental data, but also of transmitting life-saving distress signals, all under the supervision of a trusted central operator.

3 Security Analysis

3.1 Data Flow Diagram

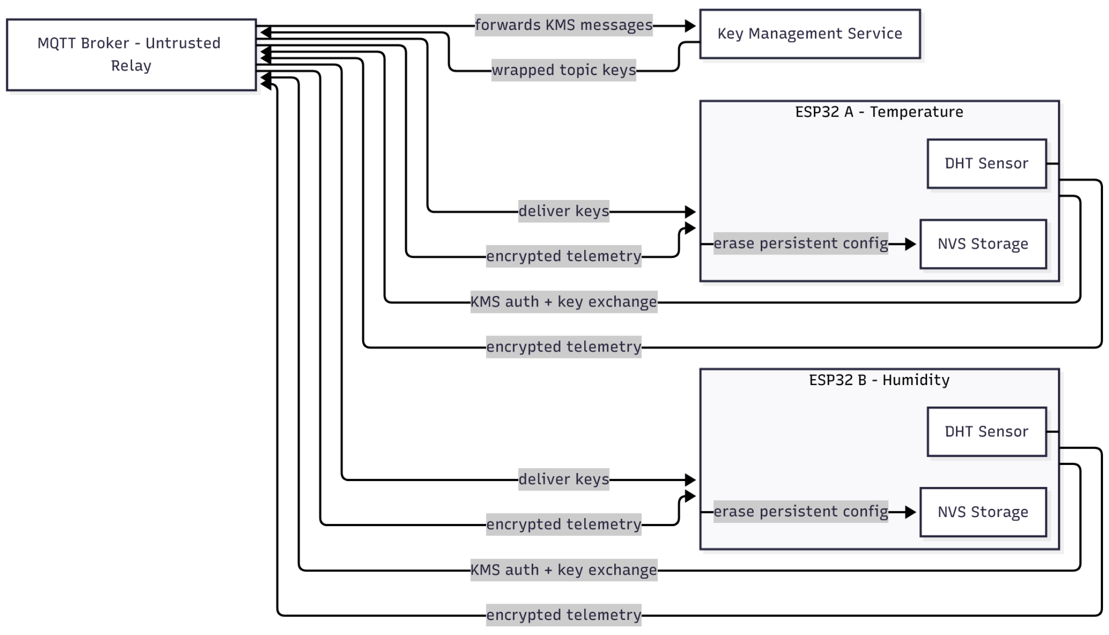

Figure 1: Data flow diagram

The diagram above shows the different entities in our system and the data flows exchanged between them. Each ESP32 periodically reads the value of its local sensor and uses the MQTT broker, considered untrustworthy, to transmit encrypted telemetry to the other device. Before any application exchange, the two ESP32s perform an authentication and key exchange protocol with the Key Management Service (KMS), which distributes topic keys and also ensures their regular rotation in order to limit the impact of any compromise.

The persistent configuration and cryptographic hardware are stored in the NVS, and can be completely reset via the button/LED interface. Each ESP32 then decrypts the received telemetry, updates the remote value, and displays it on its OLED screen. Thus, despite an unreliable broker, the confidentiality, integrity, and authenticity of exchanges are guaranteed end-to-end by our protocol.

3.2 Threat model

In order to identify the risks to which our system is exposed, we adopt the STRIDE threat model. Our analysis is based on the following assumptions:

- The MQTT broker is considered untrustworthy: it can read, modify, delete, or replay messages.
- The Wi-Fi network can be monitored or tampered with by a local attacker.
- The ESP32s may be physically accessible and their NVS memory potentially recoverable after compromise.
- The KMS is the only trusted entity, its private key must remain intact.
- An attacker may attempt to connect to the system by posing as a legitimate ESP32, the KMS must be able to refuse or blacklist a compromised client.

Based on these assumptions, the following threats are identified:

S - Spoofing (identity theft) An attacker could impersonate a legitimate ESP32 in order to publish false data or obtain keys from the KMS. If an ESP is compromised, the system must be able to prevent it from reconnecting in the future (blacklisting).

T - Tampering (data alteration) An attacker or untrustworthy broker can modify MQTT packets in transit, alter sensor values, or inject falsified messages. Telemetry traffic and KMS messages are therefore vulnerable to manipulation.

R - Repudiation (non-repudiation) Without cryptographic mechanisms, a client could deny having sent a measurement, or an attacker could generate messages that cannot be attributed to a real source.

I - Information Disclosure All data transits via an untrusted broker. An attacker can intercept measurements, metadata, or messages exchanged with the KMS if they are not protected.

D - Denial of Service An attacker can block or delay MQTT messages, saturate the broker, or disrupt the Wi-Fi. The system must be able to detect the absence of data, but cannot prevent an attacker from causing unavailability.

E - Elevation of Privilege An attacker could attempt to gain unauthorized access to a topic, interact with the KMS, or hijack messages to elevate their privileges. A compromised ESP could attempt to obtain additional keys or access topics for which it is not intended.

4 Security Implementation

4.1 MQTT Encryption Protocol

This section describes an encryption protocol for MQTT communication between a client and a broker using simple crypto primitives.

4.1.1 Design Goals

The protocol aims to provide the following security properties:

- mutual authentication between producers, subscribers and a Key Management Service (KMS)
- confidentiality of the payload
- integrity and authenticity of the payload
- can interface with any standard MQTT brokers
- the compromise of a single client (aside from the key management server) does not compromise past or future communications (forward and backward secrecy)

4.1.2 Overview

In order to achieve the above properties, we propose to implement a specific "client", which is responsible for authenticating other clients (both providers and subscribers) and for distributing a shared secret key to encrypt/decrypt the payload. We also propose a way to rotate these keys periodically.
The diagram below shows the components involved in the protocol.
Each MQTT producer can encrypt payloads using topic-specific keys obtained from the Key Management Service (KMS). The MQTT broker acts as a message relay, forwarding encrypted messages to the appropriate subscribers. Each MQTT subscriber retrieves the necessary topic keys from the KMS to decrypt the received payloads.

```
+--------------------+                          +--------------------+
|   MQTT Producer    |                          |   MQTT Producer    |
|         #1         |                          |         #2         |
|        (esp)       |                          |        (esp)       |
+----------+---------+                          +----------+---------+
           |                                               |
           | encrypted                           encrypted |
           | payloads                            payloads  |
           |                                               |
           v                                               v
+--------------------------------------------------------------------+
|                            MQTT Broker                             |
|                          (Message Relay)                           |
+----------+--------------------------------------+------------------+
           |                                      |
           | encrypted messages                   | encrypted messages
           |                                      |
           v                                      v
+------------------------------+    +----------------------------+
|    Key Management Service    |    |      MQTT Subscriber       |
| (Producer/Subscriber Client)|    |      (server or esp)       |
|                              |    |                            |
|- Authenticates other clients|    |- Receives encrypted msgs   |
|- Distributes topic keys      |    |- Decrypts payloads         |
|- Manages key rotation        |    |- Consumes application data|
+------------------------------+    +----------------------------+
```

4.2 Key management and key establishment protocols

4.2.1 Keys

key	**description**
KMS_auth_pubkey and KMS_auth_privkey	The Key Management Service (KMS) has an RSA or EC key pair (no certificate) known from all other clients. That key is used to prove authenticity of the KMS to other clients, and to sign messages sent by the KMS (like topic key distribution messages).
KMS_master_key	A symmetric key (256 bits) known only from the KMS, used to derive per-topic client HMAC keys.

4.2.2 Client master key derivation

A client is provisioned with a key derived from the KMS_master_key.

$$\text{CLIENT\_master\_key} = \text{HKDF}(\text{IKM} = \text{KMS\_master\_key}, \text{salt} = "[CLIENT\_ID]", \\ \text{info} = "[CLIENT\_ID]", \text{length} = 32\,\text{bytes})$$

At client installation time, the client receives:

- the CLIENT_master_key derived as described above
- the KMS_auth_pubkey

4.2.3 Key Management Service normal operation

Once started, the KMS listens to topics `[TOPIC]/+/kms/#`, ready to receive authentication requests from clients wanting to connect.
When a client connects to the broker, it subscribes to a topic `[TOPIC]/[CLIENT_ID]/kms/#` to receive all topics from the KMS under that client ID.
Then it derives a TOPIC_key for the topic it wants to publish/subscribe to.

$$\text{TOPIC\_key} = \text{HKDF}(\text{CLIENT\_master\_key}, \text{salt} = "Topicname", \text{info} = "Topicname", \\ \text{length} = 64\,\text{bytes})$$

$$\text{TOPIC\_auth\_key} = \text{first 32 bytes}, \\ \text{TOPIC\_key\_encryption\_key} = \text{last 32 bytes}$$

The client then publishes an authentication request to the KMS on topic `[TOPIC]/[CLIENT_ID]/kms/auth`, with the payload containing:

- a challenge for authenticating KMS (random nonce of 32 bytes)

The KMS receives the authentication request, and responds by publishing to topic `[TOPIC]/[CLIENT_ID]/kms/clientauth` a message containing:

- the challenge received from the client
- a signature of the challenge using KMS_auth_privkey
- a nonce (32 bytes random)

The signing is performed by the KMS with an RSA-2048 key using the PSS-SHA256 schema, ensuring robust server authentication.

The client receives the authentication response, verifies the signature using KMS_auth_pubkey, and checks that the challenge matches. If valid, the client considers the KMS authenticated.

At this point, the client uses the TOPIC_auth_key to HMAC the received nonce, and sends it back to the KMS on topic `[TOPIC]/[CLIENT_ID]/kms/clientverify`, with the payload containing:

- the HMACed nonce

Upon reception of the HMACed nonce, the KMS derives the same topic key and verifies the message. If valid, the KMS considers the client authenticated for that topic.
If it is the first time a client authenticates for that topic, the KMS generates a fresh random AES256 key and maintains it. It also creates an index for that fresh TOPIC_key.
The TOPIC_key is wrapped using the TOPIC_key_encryption_key, and sent to the client on topic `[TOPIC]/[CLIENT_ID]/kms/key`.
The client unwraps (decrypts) the TOPIC_key and stores it.

4.2.4 Key Rotation Mechanism

To strengthen long-term security, the KMS performs a **key rotation every minute**. A fresh AES-256 topic key is generated, wrapped with the client-specific *TOPIC_key_encryption_key*, and sent to each ESP32 on the key MQTT topic. Clients verify, decrypt, and install the new key transparently.
Each rotated key is associated with an **epoch number**. For synchronization purposes, ESP32 devices temporarily keep the previous epoch's key, allowing them to accept slightly delayed messages while still rejecting older or replayed data. This mechanism ensures smooth transitions despite network latency.
Regular rotation provides:

- **Forward secrecy**: a compromised key cannot decrypt future data.
- **Backward secrecy**: past messages remain protected even if a device is compromised.
- **Reduced attack window**: key usefulness is strictly limited in time.

This lightweight rotation strategy significantly improves the resilience of the system against persistent or long-term attacks.

4.2.5 Key Resynchronization Mechanism

If an ESP32 cannot decrypt a received message, for example because it missed a key rotation or rejected several invalid packets, it sends a resynchronization request to the KMS on the topic `[TOPIC]/[CLIENT_ID]/kms/request_key`.
Upon receiving this request, the KMS simply:

- identifies the latest valid epoch
- wraps the corresponding key using the client's *TOPIC_key_encryption_key*
- sends it back on the usual `.../kms/key` topic

The ESP32 installs the recovered key and immediately resumes normal operation. Only the current epoch and the previous one are ever redistributed, ensuring both robust synchronization and strict security. Combined with the rotation mechanism, this provides strong resilience against desynchronization in unstable environments.

4.2.6 Persistent Counter and Anti-Replay Protection

Each ESP32 maintains a **monotonic counter**, stored in the persistent NVS. Before encrypting a message, the device loads the counter, increments it, uses it in the AES-GCM AAD, and stores the new value. Because the counter survives reboots, **IV reuse is impossible**, which is essential for GCM security.
On reception, each ESP32 rejects any message whose counter is lower than the last accepted one, providing strong **anti-replay protection**. This mechanism ensures message freshness and prevents an attacker, or the untrusted broker, from replaying old ciphertexts.

4.3 Authentication scheme

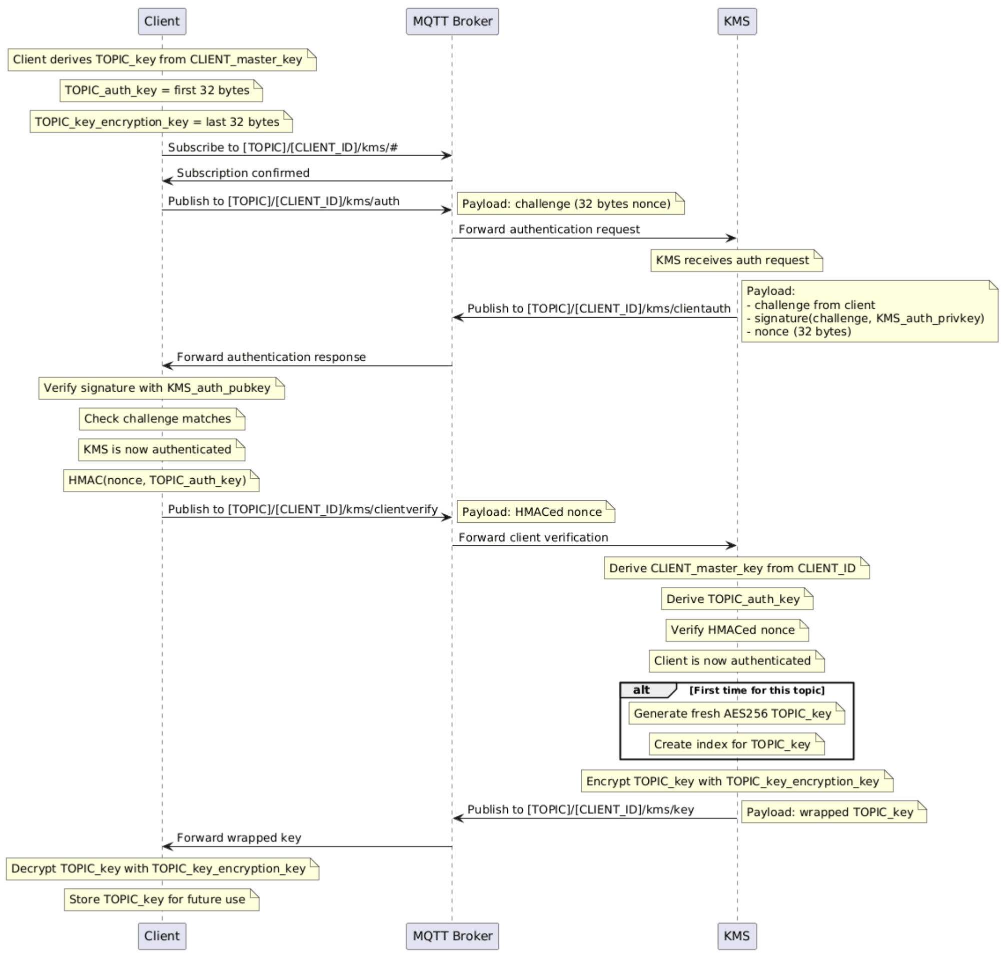
Client
MQTT Broker
KMS
Client derives TOPIC_key from CLIENT_master_key
TOPIC_auth_key = first 32 bytes
TOPIC_key_encryption_key = last 32 bytes
Subscribe to [TOPIC]/[CLIENT_ID]/kms/#
Subscription confirmed
Publish to [TOPIC]/[CLIENT_ID]/kms/auth
Payload: challenge (32 bytes nonce)
Forward authentication request
KMS receives auth request
Publish to [TOPIC]/[CLIENT_ID]/kms/clientauth
Payload:
- challenge from client
- signature(challenge, KMS_auth_privkey)
- nonce (32 bytes)
Forward authentication response
Verify signature with KMS_auth_pubkey
Check challenge matches
KMS is now authenticated
HMAC(nonce, TOPIC_auth_key)
Publish to [TOPIC]/[CLIENT_ID]/kms/clientverify
Payload: HMACed nonce
Forward client verification
Derive CLIENT_master_key from CLIENT_ID
Derive TOPIC_auth_key
Verify HMACed nonce
Client is now authenticated
alt
[First time for this topic]
Generate fresh AES256 TOPIC_key
Create index for TOPIC_key
Encrypt TOPIC_key with TOPIC_key_encryption_key
Publish to [TOPIC]/[CLIENT_ID]/kms/key
Payload: wrapped TOPIC_key
Forward wrapped key
Decrypt TOPIC_key with TOPIC_key_encryption_key
Store TOPIC_key for future use
Client
MQTT Broker
KMS

Figure 2: Auth process

4.4 Message encryption and publication

Once the key has been received from the KMS, a client (publisher or subscriber) can use it to encrypt/decrypt messages for that topic.

4.4.1 Encryption Steps

1. Generate a random IV (initialization vector) of 12 bytes.
2. Use an internal counter (starting at 0) for that topic, incremented for each message published.
3. Construct the AAD (additional authenticated data) as follows:
   `AAD = counter || topic_name`
4. Derive the AES key:
   `AES_key = HKDF(IKM=TOPIC_key, salt="IV||counter", info="topic_name", length=32 bytes)`
5. Encrypt using AES-256-GCM with AAD and IV.
6. Construct the final message payload:
   `payload = IV || counter || ciphertext || GCM_tag`

Parameters:

- AES key size: 256 bits (32 bytes)
- GCM IV size: 12 bytes
- GCM tag size: 16 bytes
- Counter size: 4 bytes

4.4.2 Decryption Steps

1. Parse the received payload:
   - IV (first 12 bytes)
   - counter (next 4 bytes)
   - ciphertext (next N bytes)
   - GCM_tag (last 16 bytes)
2. Construct AAD:
   `AAD = counter || topic_name`
3. Derive AES key:
   `AES_key = HKDF(IKM=TOPIC_key, salt="IV||counter", info="topic_name", length=32 bytes)`
4. Decrypt using AES-256-GCM with AAD and IV.
5. If the tag is valid, the plaintext is returned.

4.5 Mitigation of identified threats

This section presents the countermeasures implemented to address the threats identified in the STRIDE analysis. Our security protocol is based on centralized authentication via KMS, end-to-end encryption and rigorous key management.

S - Spoofing Mutual authentication between each ESP32 and the KMS prevents any unauthorized actor from obtaining a valid key. The KMS also maintains a list of unauthorized clients and can blacklist a compromised ESP, preventing any future reconnection with its identifier.

T - Tampering All application messages are encrypted and authenticated via AES-GCM. Any message altered by the broker or an attacker is automatically rejected during integrity tag verification. The use of AES-GCM ensures that the broker cannot alter the message content.

R - Repudiation Each client derives unique keys from its *CLIENT_master_key*. Messages authenticated via GCM are therefore necessarily attributable to the legitimate sender, which prevents the generation of untraceable messages.

I - Information Disclosure The content of MQTT messages, including exchanges with the KMS, is protected by end-to-end encryption. The broker, considered untrustworthy, never sees the data in clear text, even when acting as a mandatory relay.

D - Denial of Service Although the protocol cannot prevent an attacker from disrupting the network or the broker, each ESP32 detects the prolonged absence of messages and reports the loss of communication by displaying a "?". This allows the system to identify denial of service situations without compromising data security.

E - Elevation of Privilege The strict separation between KMS keys, *CLIENT_master_keys* and *TOPIC_keys* limits the impact of a local compromise. Regular key rotation by the KMS prevents the prolonged use of a potentially exposed key, ensuring forward/backward secrecy.

5 Feature

5.1 OLED display of secure data

In the initial configuration, the measurements exchanged between the two ESP32s were only available in the form of MQTT messages or via debugging tools. To make the system autonomous and directly usable by the survivors in our scenario, we added an **OLED** screen to each ESP32. This screen continuously displays:

- the local measurement (temperature for the North shelter, humidity for the South shelter),
- the remote measurement received from the other ESP32,
- a `?` symbol when communication becomes suspicious or no valid message has been received for a certain period of time.

This feature transforms the ESP32s into true autonomous terminals, allowing survivors to have an immediate and secure view of their environment without relying on additional equipment. The degraded behaviour displaying `?` also plays an essential role in the event of attacks, network outages, or malicious manipulation of the MQTT broker.

5.2 Emergency alert system (SOS)

A second feature has been added to respond to critical situations: the transmission of a **secure SOS signal**. When a survivor quickly presses the button on their ESP32 three times, an alert message is sent on the telemetry channel. This message is encrypted and authenticated like any other system data, ensuring that an adversary cannot alter it or spoof its origin.
Upon receiving a valid SOS, both ESP32s adopt specific behaviour:

- the **red LED flashes** to immediately attract attention,
- the **OLED screen displays an explicit alert message**,
- the signal is also sent to the **KMS**, whose web dashboard highlights the emergency status.

This feature greatly enhances the system: it no longer simply transmits environmental data, but becomes a real tool for coordination and safety for survivors. The SOS, treated as a priority and tamper-proof message, alert other shelters as well as the operator located in the bunker where the KMS resides.

6 Conclusion

The objective of this project was to design a reliable communication system between two ESP32s exchanging environmental information in a context where the network cannot be considered secure. In the simplest version, the MQTT broker could be seen as a trusted entity, capable of aggregating and directly displaying the transmitted data. However, such a model has a structural weakness: if the broker is compromised, observed, or manipulated, all the exchanged measurements can be intercepted, modified, or falsified.

We therefore chose to reverse this founding assumption by considering the broker as **untrustworthy**. This change leads to a complete reconfiguration of the architecture: the broker now plays only a simple transport role, unable to read or alter data thanks to the authenticated encryption applied by the ESP32s. The trust logic is moved to a dedicated entity: a Key Management Service (KMS), hosted on a more powerful computer and capable of performing critical functions such as client authentication, distribution of topic-derived keys, and regular rotation of these keys.

This same KMS also becomes the central observation point for the network. It replaces the role initially assigned to the broker by hosting a web interface that allows users to view the temperature and humidity data received from the two shelters, handling only data that has been validated and decrypted via a secure channel. This shift in responsibility is the cornerstone of our architecture: sensitive data only passes between trusted entities.

The added features enhance the consistency and usefulness of the system. The OLED screen integrated into each ESP32 allows instant display of local measurements, received data, or a communication loss symbol. The emergency alert system (SOS), triggered by three quick presses of the button, demonstrates the protocol's ability to securely transmit critical messages. The alert is simultaneously displayed on both ESP32s and highlighted on the KMS dashboard, allowing the operator present at the bunker to respond quickly.

By adopting an architecture where trust is strictly limited to the KMS and where all communications are protected end-to-end, we have demonstrated that it is possible to achieve a robust IoT system even under extreme conditions. This approach, focused on security by design, provides a solid foundation for expanding the network, integrating new nodes, or adding other types of alerts while maintaining a strong guarantee of confidentiality, integrity, and authenticity of the data exchanged.