IRELE

Mobile and wireless network

Project report

Authors:

Arico Amaury

Mototani Yuta

Tchakounte Loic

Sipakam Cedric

Professor:

JM. Dricot

TA:

Ladner Navid

Academic year:

2024 - 2025

Contents

1 Introduction and concept

1.1 Device

The Wireless and Mobile Network project for the acadmeic year 2025-2026 is built around two ESP32 DevKit boards, electronic sensor and actuators and two laptops running MQTT brokers and visualization tools. The objective of the project is to develop and simulate a concept focused on secure communication in an IoT environment.

Hardware components

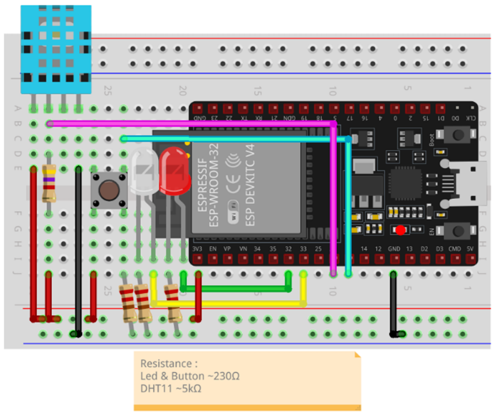


Figure 1.1: ESP32 schematic

The details of the hardwares at disposal are enumerated below:

- **ESP32 boards** The ESP32 is a low-power, energy-efficient microcontroller containing multiples features as
  - 4MB flash memory
  - 520KB RAM
  - Dual-core 32-bit processor
  - Wifi - bluetooth connectivity

  - cryptographic hardware acceleration
  - Low power hardware (PMU, low power core processor)
  - Pheripheral interfaces (GIOPs, DAC, ADC, UART)

- **Sensor and actuators**
  - a DHT11 temperature and humidity sensor,
  - a push button used later for entering the Morse code,
  - two LEDs that provide simple feedback about the system state (Wi-Fi/MQTT connection, data transmission).

- **Computers running the MQTT broker** Personal laptops are used to host the Mosquitto MQTT brokers. Both ESP32 boards connect to this broker over Wi-Fi through a local access point (a smartphone hotspot in our case). The computer can also run additional tools for data visualisation, such as an MQTT dashboard or plotting software.

1.2 Concept

The project simulates a highly sensitive, climate-controlled room, such as a medical storage area, where temperature and humidity must be continuously monitored and regulated. It uses two ESP32 microcontrollers communicating over MQTT through publish-subscribe relation and implementing a secure remote monitoring and control system. Based on the received measurements, the control center can issue control signals, e.g. adjusting an AC unit, to maintain the required environmental conditions.
MQTT protocol is a lightweight internet protocol which relies to message queuing service for communication from machine to machine. It runs over lossless transport layer, in our case TCP. Two types of entities consitute a MQTT communication - the client(s), publishing or receiving messages and the broker, the server collecting the packets and redistributing to the messages to subscribed clients. The MQTT protocol designed around the publish-subscribe communication with small overhead and payload size fits well to IoT devices - devices limited in bandwidth, located in environment with unstable connection and designe for low power consumption.

For the project, one ESP32 acts as a **publisher**, located in the environment to be monitored. It collects data from a DHT11 sensor and transmits them to an MQTT broker. The second ESP32 acts as a **subscriber**. The drawback of MQTT is that its packets are sent in plaintext by default, so additional measures are required to secure our high sensitive communication. Considering that TLS or certificatation authentification are not allowed, the whole idea will be to design a multi-layered security strategy ensuring confidentiality, authentication, and integrity of all exchanged data.

2 Threat Model

2.1 Data flow diagram

In the first chapter, we discussed about the project concept where one ESP32 is used as a data publisher mimicking the data collection of a room we would like the thermal conditions to be controlled via a command center, which is the alias for our second ESP32.
The acquised data are sent from our ESP32 "publisher" to a MQTT server, installed on our machine for the simulation. The second ESP32 is subscribed to the MQTT server, receiving the published data and in charge of monitoring - controlling the room conditions. Those data are displayed on another MQTT server where the data evolution can be plotted with MQTT explorer software.
The data flow is shown on the figure 2.1

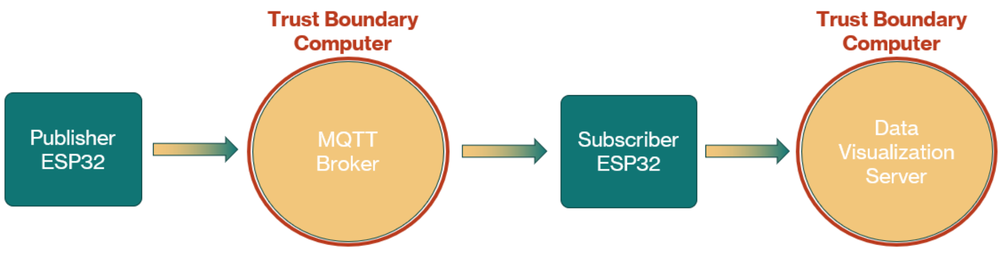


Figure 2.1: Data flow diagram

As remainder, the communication being sensitive, it would need to be robust to any types of attacks and mitigate them by integrating a security strategy considering the constraints on the hardware ressources.

Thus HTTPS / TLS protocol are not well adapted for our communication and our devices (large payload - heavy on CPU usage - multiple handshake - large memory usage). Also key cryptoghraphy containing complex mathematical operation like RSA can be implemented but not suitable for our project (same concerns that TLS). Our security strategy focuses on mitigating the relevant network- and protocol-level threats while remaining compatible with the device capabilities and the required decryption speed. The complete mitigation strategy and the additional security features implemented by our group are presented in the last chapter regarding *Security Mitigation*.

Before discussing about the security strategy, we would need first to identify the security threats our model needs to deal with. The STRIDE model will be used in this perspective in the next section.

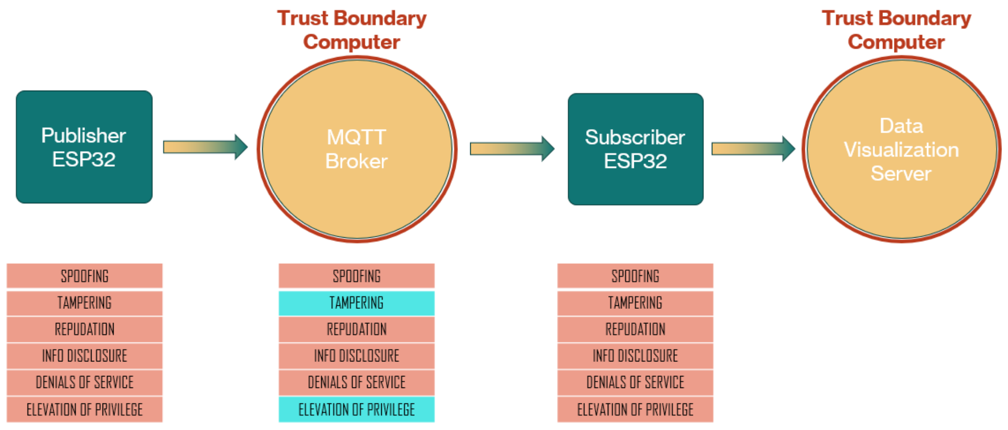


Figure 2.2: STRIDE model verification for project data flow before security mitigations

2.2 STRIDE Threat Model

The STRIDE threat model is an approach used to identify and categorize potential security threats and vulnerabilities in software systems. Developed by Microsoft, the acronym STRIDE represents six main categories of threats. These figures below illustrate all categories, their description and how they are mitigated in the project.

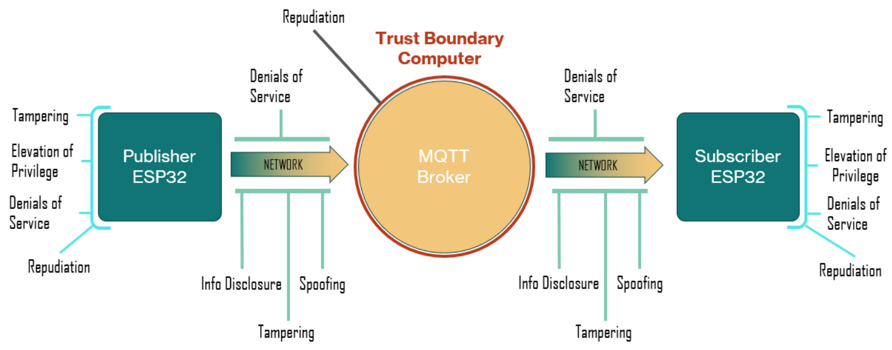


Figure 2.3: STRIDE model and threats localization

STRIDE Category	**Threat Description**	**Attacked Component/Data Flow**	**Security Mitigation (Protocol)**
Spoofing (Authentication)	An attacker impersonates a legitimate ESP32 device or the MQTT Server to inject false data or commands.	ESP32 ↔ Server Communication	**Authentication** using a **Message Authentication Code (MAC)**, such as HMAC-SHA256, to verify the sender's identity with a shared secret key derived from PBKDF2.
Tampering - communication channel (Integrity)	Data (sensor readings) is intercepted and modified while in transit over the network	Sensor Data → Server, Control Commands	**Integrity** enforced by the MAC (HMAC-SHA256). The receiver recalculates and compares the MAC to detect any alteration.
Tampering - hardware (Integrity - Authentication)	Security algorithms or keys saved in the hardware are altered	Clients and brocker hardware.	**Authentification and Integrity** Authentification ensured by the HMAC verification at the receiver - in the contrary the integrity of the message can only be ensured by securing the hardware (assumption).
Information Disclosure (Confidentiality)	An attacker eavesdrops on the network and reads sensitive data (Temperature,Humidity...).	All Communication over the Network	**Encryption** of the MQTT payload using AES-128. An additional layer is provided by Port Knocking to make sniffing and channel monitoring harder.
Denial of Service (DoS)	An attacker floods the ESP32 or MQTT server with excessive traffic or requests, consuming resources.	ESP32 or Server Resources	**Rate-limiting** Controls of the packets rate sending at the publisher and *quiet mode* state at receiver. controls on the MQTT broker (assumption of quiet mode and rejection with HMAC). Lastly, the use of Port Knocking to obscure the communication channel, reducing the window for targeted attacks.
Repudiation (Non-Repudiation)	A legitimate device or the server denies having sent a critical or malicious message.	All Communication	The HMAC proves the message's origin from the authentic sender, and the counter ensures a definitive record of sequential messages, preventing the sender from denying transmission.
Elevation of Privilege (Authorization)	A compromised device gains access to MQTT topics or capabilities it should not have permission for.	ESP32 ↔ Server Communication, MQTT Topic Subscriptions	Strict Authorization controls on the MQTT broker (assumption) - Secured hardwares (assumption) - the requirement to correctly decrypt the secret key via the Morse code sequence before publishing data.

Table 2.1: STRIDE Threat Model and Security Mitigations for ESP32-MQTT Setup

3 Security Mitigation

3.1 Security scheme

The security scheme is designed to protect the IoT communication channel against all attack types explained in the STRIDE section. An application-layer security protocol has been implemented to ensure authentification, data privacy and integrity before it even reaches the network stack. The security protocol from the publisher to the broker can be seen on the figure 3.1 - the security from the broker to the receiver being symmetrical. Each steps of the security strategy will be explained below and the additional security features will be highlited.

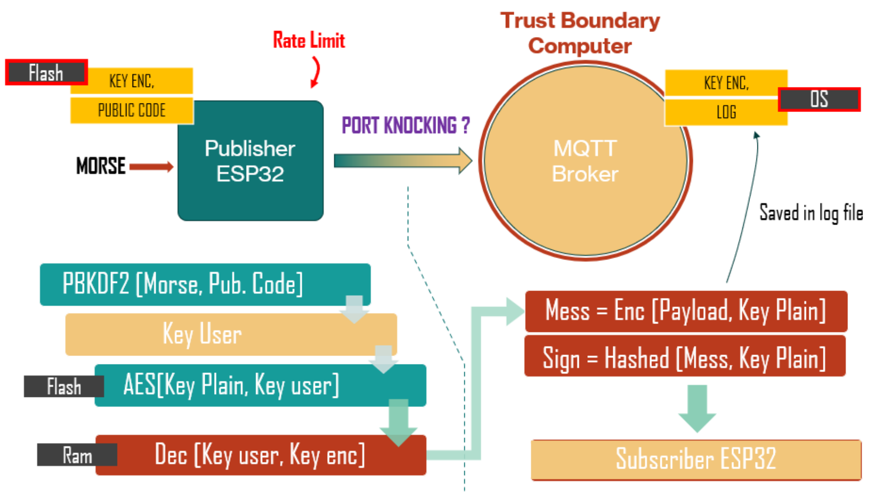


Figure 3.1: Security strategy integration in the project data flow

Key Generation and Management - PBKDF2 with Morse code Feature 1

To avoid storing the master encryption key in plaintext within the flash memory, our group conceptualize a password-based key derivation function. At startup, the user manually inputs a secret password using a push-button interpreted as Morse code. This password is then passed through the PBKDF2 algorithm using HMAC-SHA256 to derive a session key, namely the Key_{user}. The actual encryption key, which is stored in the non-volatile storage (assumed to be secured and considered as trusted area), is then decrypted using this session key. The decrypted key - $\text{Key}_{\text{plain}}$ is stored only in RAM and used for the duration of the session - the RAM memory considered *underwatch* and therefore physically unattackable.

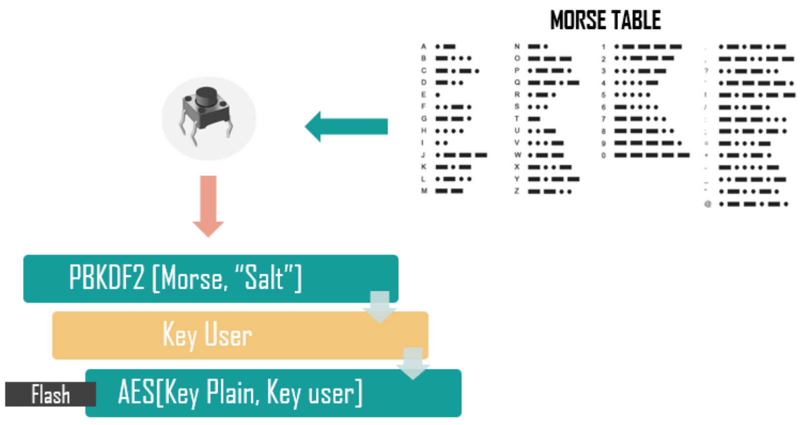


Figure 3.2: Encrypted key generation

Encryption and Confidentiality

The sensor data, including temperature and humidity, is formatted into a JSON structure. This payload is then encrypted using AES-128 in Cipher Block Chaining mode. For each message, a fresh Initialization Vector is randomly generated to ensure that the same data results in different ciphertexts every time. The payload is then encrypted using the plain key - $\text{Key}_{\text{plain}}$ - decrypted during the startup phase.

Authentication verification and Integrity

To ensure the authenticity and integrity of the message, an HMAC-SHA256 signature is generated. The final message payload includes the IV, a sequence counter, and the encrypted data. An HMAC-SHA256 signature is calculated over this entire message using the session key. The message and its corresponding signature are published to separate MQTT topics.

For our project scope, the data flow has been windowed from the first ESP32 (publisher) to the second one (subscriber) for data monitoring. In practice, the subscriber would send back control signals to the first ESP32 in order to adjust the thermal conditions of the room. Thus the data flow is bidirectional. In this perspective, we assume that :

- Each ESP32 has dedicated key (symmetrical) for encryption depending on which device sends packets (ensuring proper authentification). Thus, there would be two pairs of symmetric keys - One pair when the acquisition device sends data - the second pair when the control device sends data back.
- Key are saved at the brocker in protected memory on the host machine and in the secured flash of our ESP32s.
- At each entity, the packet authenticity can be confirmed by the HMAC verification.

Repudiation Protection

A monotonic counter is maintained by the publisher and stored in non-volatile memory to survive reboots. The counter is incremented for every sent message and embedded in the payload. Since the counter is part of the data covered by the HMAC, it cannot be tampered with. The subscriber tracks the last received counter value. Any received message with a counter lower than or equal to the last valid counter is rejected to ensure non-repudiation. Depending on the *quality of service* level selected on the brocker, a packet may be lost **at application layer** without any counter-actions, redelivered multiple times with an acknowledgement system in one direction or ensured to be treated with a bidirectional handshakes. Our strategy ensures that :

- whatever the selected **QoS** level, the client - brocker / client-client communication will not be stalled due to a packet loss.
- all the past packets not delivered in time (before the next packet), containing outdated information about the room condition, will be discarded.

Therefore, the counter protocol contibutes on the authentication security layer (when combined with the HMAC) but also prevents replay attacks, ensuring that lost or delayed packets cannot be reused maliciously or providing a misinterpretation of the current thermal situation. It is important to point out that assumption is made here that counters are stored, with the clients keys and the published messages, in a OS-level log file on the brocker side too.

Network Defense via Port Knocking - Feature 2

As an additional layer of obfuscation, the system implements a synchronized port rotation mechanism. Both the publisher and subscriber cycle through a predefined sequence of MQTT broker ports. After a successful transmission, both devices switch to the next port in the sequence, making continuous passive sniffing more difficult for an attacker listening on a single port.

Rate limitation and Quiet mode functionnality - Feature 3

The final security layer, designed to mitigate flooding attacks, relies on rate limiting at the publisher and a quiet mode mechanism at the subscriber. At the publisher side, the setup restricts the sending of packets to one every 20 seconds. At the subscriber side, if 10 consecutive attack attempts are detected, the device temporarily unsubscribes from the broker topics and waits 20 seconds before resubscribing. This approach helps prevent excessive load on the network and ensures system resilience against repeated request floods. Additionnally, the **port knocking** layer provides further protection against denials of services.

3.2 Results

In thi section, we will review the results obtained during the implementation of the thermal regulation concept - previously discussed - with two ESP32 communicating through a MQTT server in a fully securised environment (authentification, confidentiality and integrity).

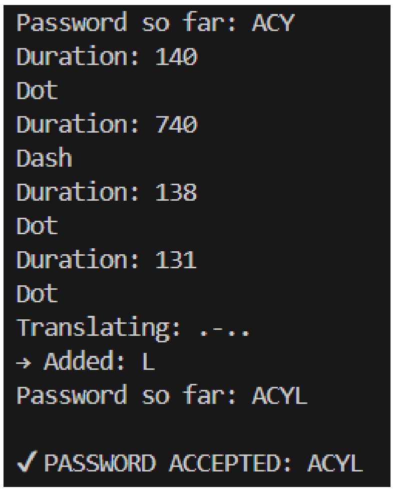
We will proceed step by step by confirming the implementation of :

1. The morse code acquisition with debouncing function for security
2. PBKDF2 key generation and decryption of the encrypted key in the flash memory
3. Encryption of the payload with the decrypted key
4. Decryption of the payload at the subscriber
5. Replay detection verification
6. Port knocking verification

3.2.1 Morse Code

The morse code acquisition passes by the implementation of a look-up table translated unique succession of **dots** and **dashes** to an alphabetic letter. These **dots** and **dashes** being generated depending on the duration when the push button has been pressed. Just to ensure the translation code was correctly implemented, the code compares the morse code encoded and the one we should use to decrypt the key stored in the flash. Of course, this verification is not done in practice as it would mean the morse code is stored in the flash memory which could be an additional break in our security strategy - morse code is supposed to be kept secret and shared verbally.

Figure 3.3 shows the results of the last letter encoding (and the the verification of the morse code).



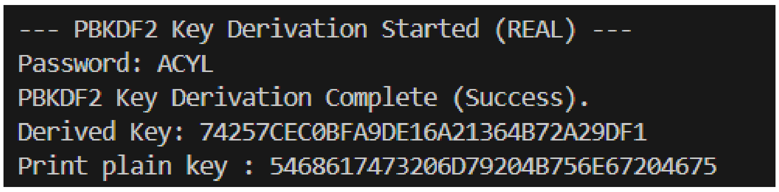
```
Password so far: ACY
Duration: 140
Dot
Duration: 740
Dash
Duration: 138
Dot
Duration: 131
Dot
Translating: .-..
→ Added: L
Password so far: ACYL

✓ PASSWORD ACCEPTED: ACYL
```

Figure 3.3: Morse code implementation Result

3.2.2 PBKDF2 encryption

The following security phase is the decryption of the encrypted key saved in the non-volatile memory through the PBKDF2 encryption and AES decryption. The figure 3.4 shows the following results :

1. Morse code encoded with the push button
2. Derived 16-bytes key generated by the PBKDF2 algorithm (Morse, "Salt", HMAC 256, 1000 iterations)
3. Encrypted 32-bytes key saved in the flash - $\text{Key}_{\text{enc}} = \text{AES}(\text{IV code}, \text{Key}_{\text{plain}})$
4. Decryption of the key - $\text{Key}_{\text{plain}} = \text{Dec}(\text{IV code}, \text{Key}_{\text{enc}})$
5. $\text{Key}_{\text{plain}} = 0x54, 0x68, 0x61, 0x74, 0x73, 0x20, 0x6d, 0x79, 0x20, 0x4b, 0x75, 0x6e, 0x67, 0x20, 0x46, 0x75$ - **Thats my Kung Fu** in 16 ASCII



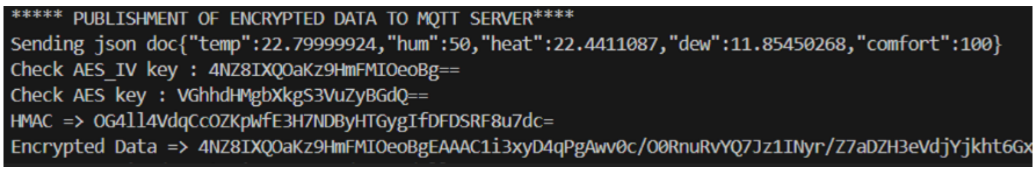
```
--- PBKDF2 Key Derivation Started (REAL) ---
Password: ACYL
PBKDF2 Key Derivation Complete (Success).
Derived Key: 74257CEC0BFA9DE16A21364B72A29DF1
Print plain key : 5468617473206D79204B756E67204675
```

Figure 3.4: PBKDRF2 encryption verification

3.2.3 Encrypted data publishment

Once the plain key ($\text{Key}_{\text{plain}}$) has been generated, the thermal data can be encrypted with AES-128 bits encryption. The figure 3.5 shows the details fo the encryption functionality and the results of the encrypted payload.

1. Thermal conditions gathered in a java script object notation (readable text to send structured data)
2. IV and plain key shown for transparency in **Arduino String**
3. Hmac and cypher payload generation with AES-128 bits encryption.
4. Publishment of the data on the MQTT server after client connexion on MQTT server on dedicated port.



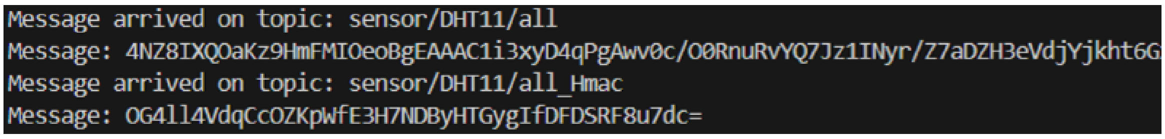
```
***** PUBLISHMENT OF ENCRYPTED DATA TO MQTT SERVER****
Sending json doc{"temp":22.79999924,"hum":50,"heat":22.4411087,"dew":11.85450268,"comfort":100}
Check AES_IV key : 4NZ8IXQOaKz9HmFMIOeoBg==
Check AES key : VGhhdHMgbXkgS3VuZyBGdQ==
HMAC => OG4ll4VdqCcOZKpWfE3H7NDByHTGygIfDFDSRF8u7dc=
Encrypted Data => 4NZ8IXQOaKz9HmFMIOeoBgEAAAC1i3xyD4qPgAwv0c/O0RnuRvYQ7Jz1INyr/Z7aDZH3eVdjYjkht6Gx
```

Figure 3.5: Encrypted data publishment on MQTT server

3.2.4 Decryption at the subscriber

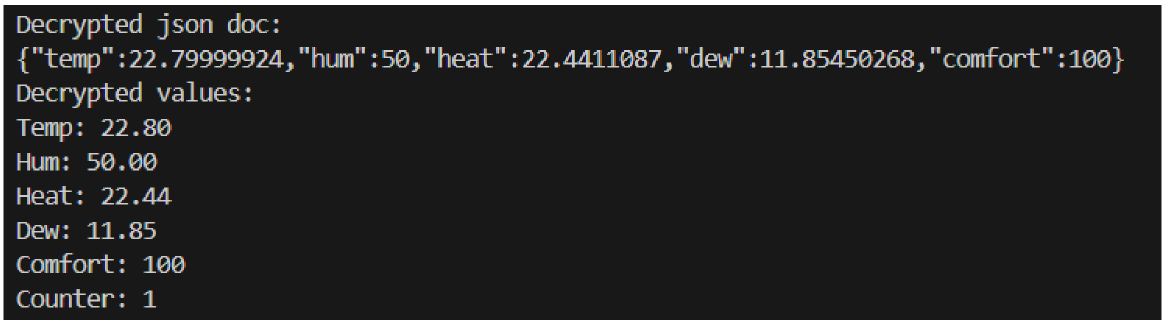
The second ESP32 subscribed to the same MQTT server - listening to the same port as the publisher - receives the cypher message containing HMAC signature and encrypted payload. From the inverse encryption process, the cypher is decrypted and the result details are shown in the figure 3.6 and 3.7.

1. Encrypted message and HMAC signature sent to the subscriber
2. Payload decrypted with the plain key (decrypted at the subscriber with the same secret morse code)
3. Display of the counter countering replay attacks.



```
Message arrived on topic: sensor/DHT11/all
Message: 4NZ8IXQOaKz9HmFMIOeoBgEAAAC1i3xyD4qPgAwv0c/O0RnuRvYQ7Jz1INyr/Z7aDZH3eVdjYjkht6G
Message arrived on topic: sensor/DHT11/all_Hmac
Message: OG4ll4VdqCcOZKpWfE3H7NDByHTGygIfDFDSRF8u7dc=
```

Figure 3.6: Subscription to MQTT server topics



```
Decrypted json doc:
{"temp":22.79999924,"hum":50,"heat":22.4411087,"dew":11.85450268,"comfort":100}
Decrypted values:
Temp: 22.80
Hum: 50.00
Heat: 22.44
Dew: 11.85
Comfort: 100
Counter: 1
```

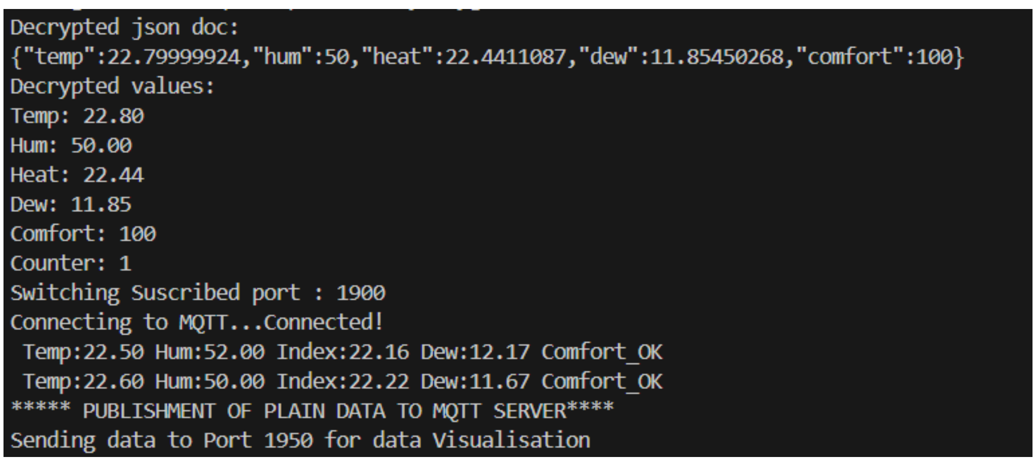
Figure 3.7: Decryption verification

3.2.5 Port Knocking

To complete the results section, the terminal shows the results got during data sending experiment from the subscriber side.

One can observe that the port is switching once the packet has been received - **Switching Subscribed port : 1900**. This port switch simulates a synchronised port switching between the publisher - MQTT server and the subscriber.

Figure 3.8



```
Decrypted json doc:
{"temp":22.79999924,"hum":50,"heat":22.4411087,"dew":11.85450268,"comfort":100}
Decrypted values:
Temp: 22.80
Hum: 50.00
Heat: 22.44
Dew: 11.85
Comfort: 100
Counter: 1
Switching Suscribed port : 1900
Connecting to MQTT...Connected!
 Temp:22.50 Hum:52.00 Index:22.16 Dew:12.17 Comfort_OK
 Temp:22.60 Hum:50.00 Index:22.22 Dew:11.67 Comfort_OK
***** PUBLISHMENT OF PLAIN DATA TO MQTT SERVER****
Sending data to Port 1950 for data Visualisation
```

Figure 3.8: Port switching for sniffing security improvement

3.3 Discussion

In this section, we discuss the security assumptions and the remaining limitations of our implementation.

Hardware trust assumptions

In our design, we "assume" that the MQTT broker runs on a trusted computer and that the ESP32 flash memory is protected at the hardware level. Under this assumption, the attackers can observe and modify the wireless traffic, but they cannot directly read or overwrite the keys stored in flash or the files of the MQTT server.

If this assumption is violated (for instance, if an attacker obtains physical access to the board and can dump the flash contents), our scheme would not be sufficient: the long–term AES key and the PBKDF2 parameters could be extracted, and all traffic could be decrypted offline.

Port knocking and synchronisation

As an additional defence in depth, we implemented a form of *port knocking* on the ESP32 side. Instead of always sending encrypted MQTT traffic to a single fixed port, the publisher and subscriber cycle through a predefined list of ports (1883 → 1900 → 1925 → 1960 → ...). After each encrypted payload is sent and acknowledged, both ESP32 devices update an internal index and reconnect to the broker using the next port in the sequence. This makes passive sniffing slightly more difficult, because an attacker cannot simply capture all traffic by listening on a single port.

In a *strict* port-knocking design, the MQTT broker would enforce the same sequence: the server would initially keep all application ports closed and only open the next port in the list after a correct "knock" on the current one. Such behaviour requires the synchronisation between the client and the server. If the broker does not follow the same sequence, an attacker who listens on all open ports can still observe every packet. A time-based synchronisation with ports sequence shared in amont through the first encrypted packets would enhance the port knocking security. It is worth mentionning that such security implementation may become complex for scalability (clock synchronisation - port list sharing - dedicated port connection for new client connection).

In our implementation the Mosquitto broker is configured to listen on all ports of the sequence at the same time, without enforcing any ordering as it would requires a modification of the server code source. As a consequence, our port-switching mechanism ,implemented as it is, is a naive/theoretical protection against sniffing. A possible improvement for future work would be to move the port-knocking logic to the server side with a clock-based port sequence switch so that only the correct sequence of ports, **at the right timing**, is accepted and all other connection attempts are rejected.

Side–channel leakage

The implemented counter-measures mainly address network-level threats such as spoofing, tampering and replay. On a small micro-controller there are additional side-channel attacks: an attacker could measure execution time, count low-level instructions or observe power consumption in order to recover information about the AES key.

Protecting against these attacks is outside the scope of this work but one should be aware that a multi-layered protection such as implemented in this projet would not able to prevent this kind of low-hardware level attacks. In practice, defending against side channels would require constant-time cryptographic implementations and specific hardware counter-measures, which are not provided in our prototype.

Security level of AES-128

The payload is encrypted with AES in CBC mode with a 128-bit key. From a purely cryptographic point of view, AES-128 is still considered secure for most applications and no practical attack is known that breaks the algorithm with a realistic amount of resources.

However, for highly sensitive data and long-term confidentiality, standards often recommend larger keys (e.g. AES-256) and additional protections such as key rotation. In our project the main limitations are therefore not the theoretical strength of AES-128 itself but the way keys are stored, derived and handled on low-cost IoT hardware.

4 Conclusion

This project successfully developed and implemented a functional and highly secure Internet of Things (IoT) setup using two ESP32 microcontrollers communicating via an MQTT broker. We met all core requirements, including reading sensor data (temperature and humidity), enabling push-button control, and facilitating data exchange with a centralized server. On top of the core requirements we add some features while implementing the setup: room comfort estimation, morse password to start the communication, counters to mark each payload and an external server to publish the data.
We successfully achieved the required security properties and we added some other contributions:

- Confidentiality: Achieved by encrypting the entire data payload using AES.
- Integrity and Authentication: Ensured through the use of an HMAC signature generated from a secret key and verified by the recipient.
- Our contribution to secure key storage involved deriving the key through PBKDF2 using a secret Morse Code input from the push button. This added a crucial layer of physical security and user-based authentication for the initial key decryption.

Future Work

- Implementing true hardware-level security, such as leveraging the ESP32's Flash Encryption features to ensure that stored key cannot be read at all.
- Implement Port knocking with the MQTT server such that during the transmission of one packet the MQTT reads only the designated port.