Examination Project

Data Integrity&Security

Pernilla Stalbring Solino-Moreno
YH Akademin- Trollhättan
Advanced Software Developer Embedded Systems
9/6-2025

Table of Contents:

1. Brief summary of the project

This exam project for the Advanced Software Developer – Embedded Systems course implements a secure client-server system to demonstrate data integrity and confidentiality in embedded applications.
The system consists of a Python GUI client (PyQt6) and an ESP32 server, communicating via serial with cryptographic protections:

- HMAC-SHA256 for tamper-proof messaging
- AES-256 for encrypted payloads
- RSA-2048 for secure key exchange

Key objectives:

1. Secure session management (1-minute expiry)
2. Remote control of hardware (relay on pin 32, LED on pin 21)
3. User-friendly logging and session controls

The GUI lets users toggle the relay, read temperature, and view encrypted transactions.
The server uses mbedtls for encryption and authentication, rejecting unauthorized requests.

Methods:
The client begins by exchanging RSA public keys with the server. A shared secret key is used for AES-256 encryption and HMAC-SHA256 to verify message integrity. The system communicates over serial only, with a 60-second session timeout. The GUI is built in PyQt6, and cryptographic operations on the ESP32 are handled using mbedtls.

Results:
The system fulfilled all key goals. It successfully showed how embedded devices can use strong cryptography for protection.
Challenges like AES performance on the ESP32 were resolved through testing and tuning.

This project proves that even low-power devices can support robust encryption and act as secure components in modern IoT systems.

2. Introduction -

Background and Context of the Project

In an age where digital systems are deeply integrated into our everyday lives, safeguarding the flow of information is no longer optional — it's essential.

When the client sends data or controls the hardware, there's a risk that someone could secretly listen or change the information during the transfer.
This project explores how to build a secure communication link between a Python-based client and an ESP32-based embedded server, using serial communication to control physical components like LEDs and relays.

Developed for the Advanced Software Developer – Embedded Systems course at YH Akademin, the project addresses modern security demands by implementing a client-server system with strong cryptographic protection.

The system consists of:

- A Python GUI client (using PyQt6) for user interaction.
- An ESP32 server managing hardware (LED on pin 21, relay on pin 32).
- Secure serial communication using SHA256, HMAC-SHA256, AES-256, and RSA-2048 implemented via mbedtls.

To break that down:

- **SHA256** is a cryptographic hash function that creates a fixed-size fingerprint of data, often used to ensure integrity.
- **HMAC-SHA256** is a way of combining a secret key with a hash (SHA256) to verify that messages haven't been altered.
- **AES-256** is a symmetric encryption algorithm that scrambles the actual data so it can't be read without the correct key.

The project aligns with real-world IoT security needs, where encryption, data integrity, and session management are critical to protecting embedded devices against threats such as replay attacks or tampering.

3. Purpose and Significance

The primary goal is to demonstrate how end-to-end encryption and secure session control can protect embedded systems from common communication threats. Key significance includes:

- **Practical Security**: Uses industry-standard protocols (AES, RSA, HMAC) on resource-constrained hardware like the ESP32.
- **User Control**: Offers a GUI for remote and secure hardware interaction (temperature monitoring, relay control).
- **Educational Value**: Acts as a template for building secure embedded communication systems.

Objectives and Scope

Objectives:

1. Establish a secure session with a 1-minute timeout to prevent idle or stale connections.
2. Encrypt all transactions using AES-256 and authenticate them with HMAC-SHA256.
3. Use RSA-2048 to securely exchange the AES encryption key and initialization vector.
4. Allow the user to interact with hardware securely through a PyQt6 GUI (get temperature, toggle relay).

Scope:

- Only one active session is allowed at a time (server-side).
- Communication is limited to serial; no network or Wi-Fi is involved.
- Physical access to the ESP32 is assumed for initial setup and deployment.

4. System Design

Overall System Architecture

I built a secure client-server system using serial communication between a Python client and an ESP32.
The client is written in Python and uses PyQt6 for the GUI.

The ESP32 server is written in C++ using the Arduino framework.

In the GUI, I can:

- Start and close a session
- Toggle the relay on pin 32
- Read the ESP32's temperature
- Clear the log

The server handles these requests and controls the hardware:

- Relay (pin 32)
- LED (pin 21)

To make the communication secure, I use:

- **RSA-2048** to send encryption keys
- **AES-256** to encrypt messages
- **HMAC-SHA256** to check that the data is not changed
  I use the **mbedtls** library on the ESP32 and **python-mbedtls** in the Python code.

Design Methodology

I used GitHub to organize the code and keep track of progress. I created a private repository and used a board to plan the work. I made issues for different parts of the system and marked them when they were done.

I followed a modular and step-by-step approach.

First I worked on serial communication between the client and ESP32. Then I added cryptographic features like RSA for key exchange, AES for encryption, and HMAC for message verification. After that, I connected everything to the GUI.

Each part was tested separately before combining them. This approach helped me keep the system structured and i cloud debug.

On the client side, I used object-oriented programming to separate concerns into different classes (for example, communication, session, and GUI logic).

On the server, the structure is more procedural due to the limitations of the ESP32.

5. Key Components and Their Interactions

Client:

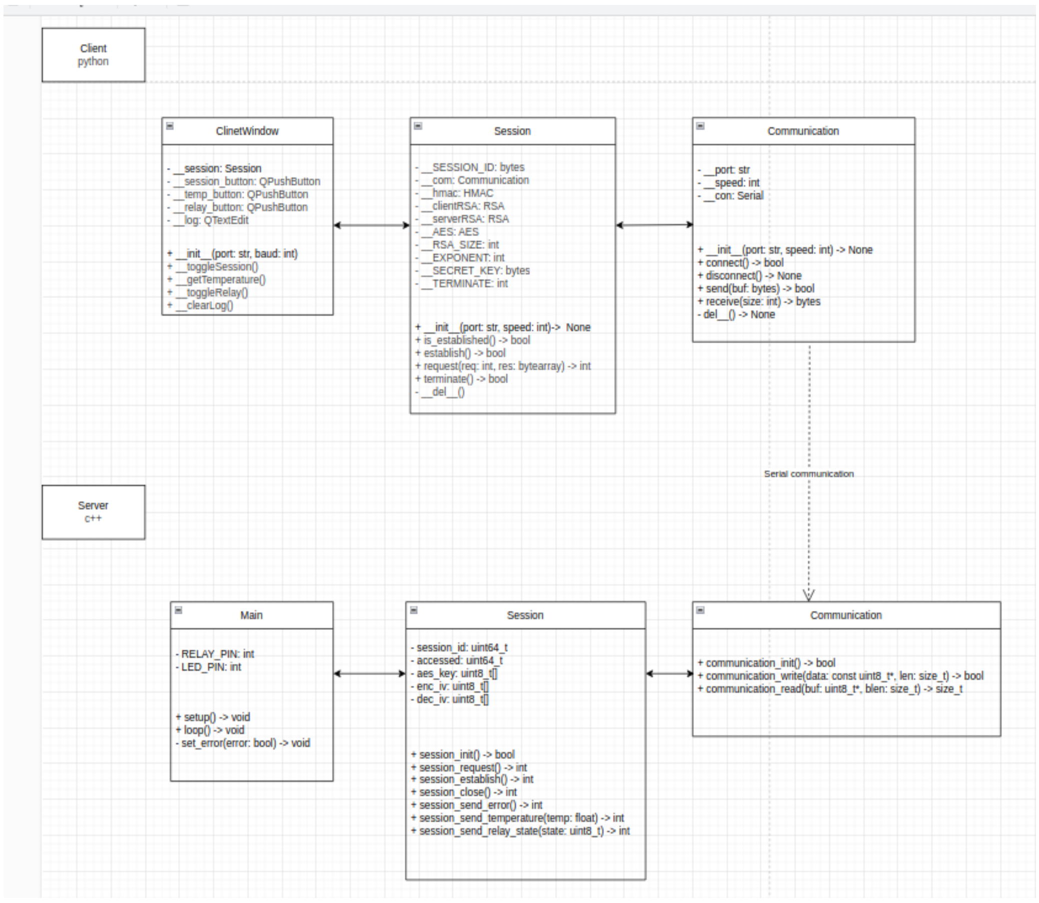
- main.py: GUI with PyQt6
- session.py: handles session, RSA, AES, HMAC
- communication.py: sends and receives data over serial

ESP32 server:

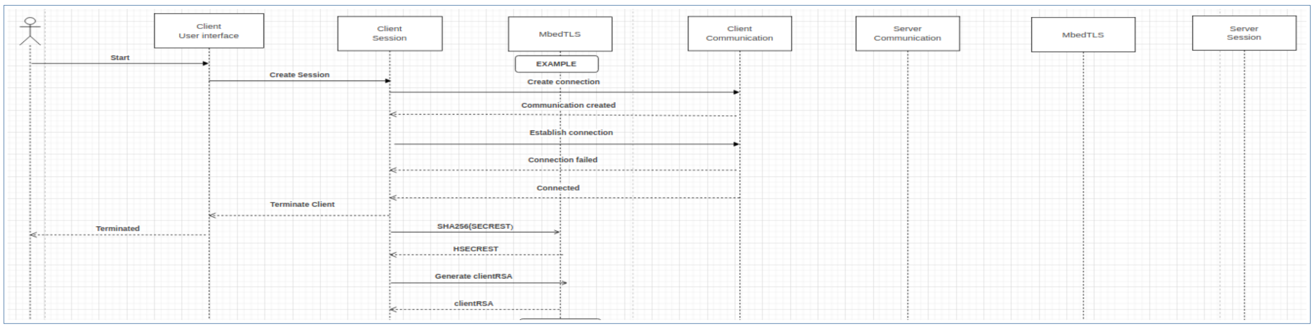
- main.cpp: handles requests and controls relay, LED, and temperature
- session.cpp: checks session, decrypts, and verifies messages
- communication.cpp: handles serial data

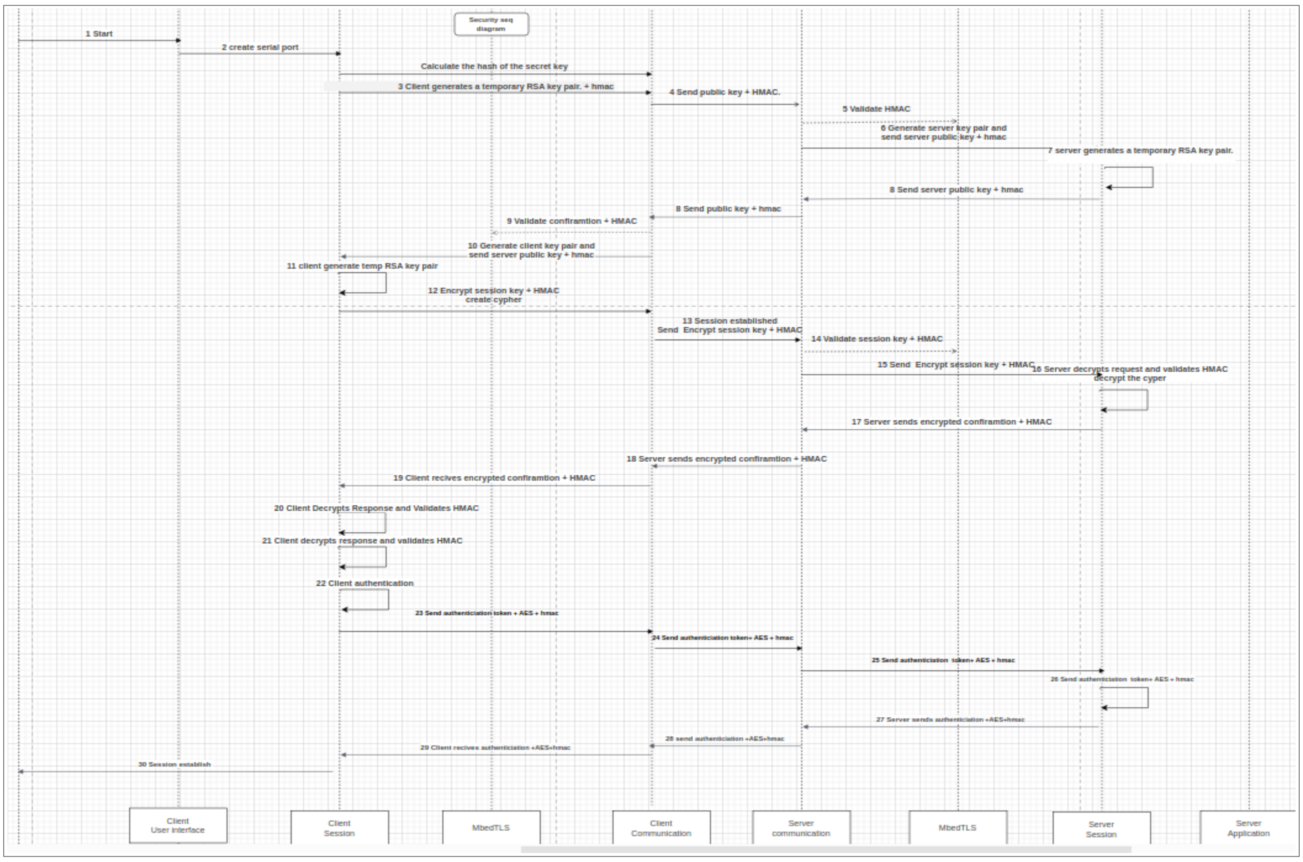
When I press a button in the GUI, like “Get Temperature,” the client sends an encrypted request. The ESP32 decrypts it, reads the temperature, encrypts the result, and sends it back.

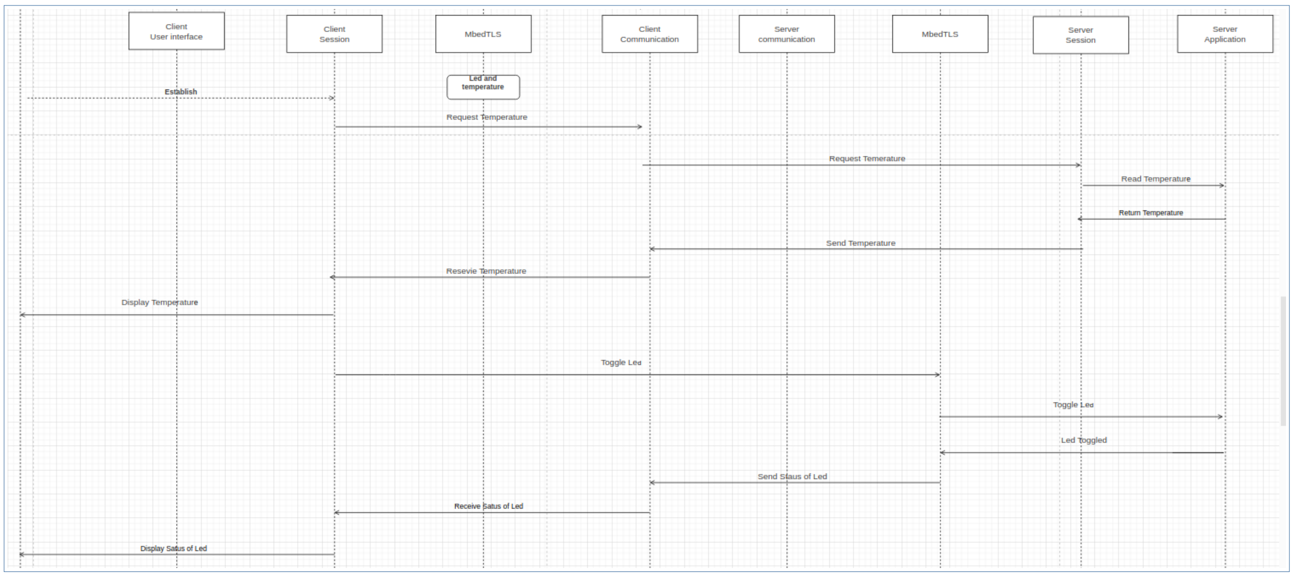
Uml diagrams



6. Sequence Diagram




Security seq diagram
1 Start
2 create serial port
Calculate the hash of the secret key
3 Client generates a temporary RSA key pair. + hmac
4 Send public key + HMAC.
5 Validate HMAC
6 Generate server key pair and send server public key + hmac
7 server generates a temporary RSA key pair.
8 Send server public key + hmac
8 Send public key + hmac
9 Validate confiramtion + HMAC
10 Generate client key pair and send server public key + hmac
11 client generate temp RSA key pair
12 Encrypt session key + HMAC create cypher
13 Session established Send Encrypt session key + HMAC
14 Validate session key + HMAC
15 Send Encrypt session key + HMAC
16 Server decrypts request and validates HMAC decrypt the cyper
17 Server sends encrypted confiramtion + HMAC
18 Server sends encrypted confiramtion + HMAC
19 Client recives encrypted confiramtion + HMAC
20 Client Decrypts Response and Validates HMAC
21 Client decrypts response and validates HMAC
22 Client authentication
23 Send authenticiation token + AES + hmac
24 Send authenticiation token+ AES + hmac
25 Send authenticiation token+ AES + hmac
26 Send authenticiation token+ AES + hmac
27 Server sends authenticiation +AES+hmac
28 send authenticiation +AES+hmac
29 Client recives authenticiation +AES+hmac
30 Session establish
Client
User interface
Client
Session
MbedTLS
Client
Communication
Server
communication
MbedTLS
Server
Session
Server
Application



7. Implementation -

Development environment and tools used

I used **Visual Studio Code** with the **PlatformIO extension** for the ESP32 server-side C++ code.

On the client side, I used **Python** with the **PyQt6** library to build the GUI.

Version control and collaboration were handled using **GitHub**, where I created a private repository and used the **GitHub Projects board** to plan, track, and complete tasks.

I also used **mbedtls** for implementing cryptographic functions like RSA, AES, and HMAC on the server.

Key algorithms and data structures

The project implements key cryptographic algorithms for securing communication:

- **RSA-2048** is used for public-key encryption to exchange keys securely.
- **AES-256 (CBC mode)** is used to encrypt data during an active session.
- **HMAC-SHA256** ensures data integrity by verifying that the message was not altered.

A **session ID** is generated during handshake and used to validate the client's identity. This session expires after one minute of inactivity.

Code structure and organization

The project is split into two main parts:

- **Client (Python)**: Contains ClientWindow, Session, and Communication classes, each with clear responsibilities for GUI, session handling, and serial communication.
- **Server (C++)**: Divided into main.cpp, session.cpp/h, and communication.cpp/h. The session module handles key exchange, encryption, and session control. communication handles serial I/O.

This modular design made the system easier to test and debug.

Technical challenges and solutions

This project was challenging overall, especially because it involved encryption, serial communication, and hardware control. One problem was making sure the communication between the client and server worked correctly. At first, some messages got lost or arrived out of order. I solved this by clearing buffers and adding better control of the session state. And support from you.

The client side was also difficult because it handled GUI actions, serial messages, and session security at the same time. I tested each part step by step to make sure things worked before combining them

8. Testing and Evaluation -

Testing Methodology

I tested the system continuously while developing.
Each time I made a change, I rebuilt the project using make or PlatformIO, depending on whether I worked on the server or client side.
The system was tested directly on the ESP32 microcontroller, since errors wouldn't always show clearly in the terminal.
In the early stages, I used the LED on pin 21 to help debug — for example, to confirm if the server was running or if an error had occurred.

Most testing was done by running the full project and interacting with it through the GUI.

I clicked buttons to start or close the session, request temperature, and toggle the relay. This let me check if the system responded correctly to each command and if the encryption and session logic worked as expected.

Test Cases and Results

- **Session Management:**
    - Starting a session worked when clicking "Establish Session."
    - After 1 minute without interaction, the session expired correctly and the GUI disabled buttons.
    - Clicking "Close Session" properly terminated the session.
- **Temperature Reading (internal ESP32 sensor):**
    - When requested, the ESP32 returned temperature values.
    - The GUI showed an error message if the data was invalid or if the session was expired.
- **Relay Toggle (Pin 32):**
    - The relay toggled between ON and OFF with each button click.
    - The new state was correctly shown in the log area.
    - If the session was expired, the request failed as expected.
- **Cryptographic Protection:**
    - RSA key exchange, AES encryption, and HMAC worked together without breaking communication.
    - All messages were verified and decrypted correctly on both sides.

Performance Evaluation

The system ran smoothly overall. Session handling was reliable, and commands were executed securely. There was a small delay when using cryptographic functions on the ESP32, but this did not interfere with functionality. The project met all expected security and control goals.

9. Results and Discussion -

Analysis of Project Outcomes

Now when I’m done and look at the results of my project, I’m pleased with how the system turned out.

I got the client and server to talk securely over serial, and I can control the relay and get temperature from the ESP32 using the GUI.

The session handling works with timeouts, and the encryption with RSA, AES, and HMAC gives the protection I aimed for.

Comparison with Initial Objectives

All my main goals were completed:

- A session is only valid for one minute.
- Messages are encrypted and authenticated.
- The GUI can control hardware and log messages.
- Only one session at a time, and the server checks the session ID.

Strengths and Limitations

Strengths:

The biggest win is that I managed to use real cryptography on a small embedded device. The communication is stable, and the code is organized so I could test each part as I built it.
I also fixed issues based on feedback — like removing unnecessary mbedtls headers from session.h and making sure a new session can be established after one expires.

Limitations:

One problem is that the ESP32 sometimes struggles with the crypto speed, and error messages could be more detailed. The GUI is simple and could show more about what’s happening in the background. There's still a short delay before a new session can be started after expiration, but it works.

Potential Improvements

In the future I could:

- Improve the feedback in the GUI, especially for session status and errors.
- Try other encryption methods that are faster on the ESP32.
- Add more features like logs or controls for other pins or sensors.

10. Conclusion -

Summary of key findings

After completing the project, I was satisfied with the results.
I managed to build a system where the client can securely communicate with the ESP32 using encryption and session control.
The data is protected with RSA, AES, and HMAC, and the session handling works well.
The system responds as expected and rejects invalid or expired requests, which shows that the security features are working.
During development, I also fixed issues pointed out in feedback — for example, I removed the unnecessary mbedtls headers from session.h. I also made sure that a new session can be established after expiration. There is a short delay before it can be re-established, but it now works as intended.

Implications of the project

This project shows that even small embedded devices like the ESP32 can be part of secure communication systems.
I learned how important encryption and structure are when building something that handles data and hardware.
It also gave me a better understanding of how to divide the project into smaller parts and test each one.

Future work and recommendation

If I continue working on this, I would like to add support for Wi-Fi or other network communication, not just serial.
It would also be interesting to add more hardware functions and maybe some logging on the server side.
Better error messages and maybe support for more than one user could also make the system more complete.

References :

- ARM Limited. (n.d.). *mbed TLS (formerly PolarSSL)*. ARM. Available at: https://github.com/Mbed-TLS/mbedtls [Accessed 9 June 2025].
- Riverbank Computing Ltd. (n.d.). *PyQt6 Reference Guide*. Available at: https://www.riverbankcomputing.com/static/Docs/PyQt6/ [Accessed 9 June 2025].
- Python Software Foundation. (n.d.). *hmac — Keyed-Hashing for Message Authentication*. Available at: https://docs.python.org/3/library/hmac.html [Accessed 9 June 2025].
- Python Software Foundation. (n.d.). *PyCryptodome Documentation*. Available at: https://www.pycryptodome.org/src/public_key/rsa [Accessed 9 June 2025].
- Python Software Foundation. (n.d.). *PyCryptodome Documentation - AES Module*. Available at: https://www.pycryptodome.org/src/cipher/aes [Accessed 9 June 2025].