DEVELOPER DOCUMENTATION

Developed by Shobhit Sharma - Student at BME
This project was created as part of a collaborative effort between **Continental Autonomous Mobility Hungary and Budapest University of Technology and Economics (BME).**

Github Repo Link: https: //github.com/bigshowbhit/secure-update-system.git

---

INTRODUCTION:

In the modern automotive industry, secure and reliable software updates are essential to maintain safety, performance, and cybersecurity. This project, **Secure Automotive Software Updates using Cryptography**, demonstrates a proof-of-concept for securely transmitting and applying updates to an embedded system using a combination of RSA and AES encryption.

The system involves two primary components:

- A **laptop** that acts as the update sender.
- An **ESP32 microcontroller** that receives, decrypts, and verifies the update.

The update payload is transmitted via UART and includes:

- An AES key (encrypted with RSA)
- An Initialization Vector (IV)
- An AES-encrypted message along with its digital signature.

By leveraging **hybrid encryption**, the project ensures:

- **Confidentiality** of the update using AES-256.
- **Authenticity** and **Data Integrity** using RSA digital signatures.

This project was a great step in understanding secure embedded communication!

SYSTEM ARCHITECTURE:

The system follows a **hybrid encryption** architecture, combining RSA and AES to ensure secure communication between the Laptop (Host) and the ESP32 (Embedded Device).

Key Components:

- Laptop (Sender)
    - Runs a Python script that:
        - Signs the message using its **RSA private key**.
        - Generates a random **AES-256 key** and IV.
        - Encrypts the message + signature using AES.
        - Encrypts the AES key using the **ESP32's RSA public key**.
        - Sends the combined payload (<enc_key>::<iv>::<enc_payload>) via UART.
- ESP32 (Receiver)
    - Written in C using ESP-IDF and mbedTLS.
    - On receiving the payload:
        - Decrypts the AES key using its **RSA private key**.
        - Decrypts the message using AES-256 and the IV.
        - Verifies the RSA-SHA256 signature using the **Laptop's public key**.
        - Updates its internal message if the signature is valid.

Security Mechanisms:

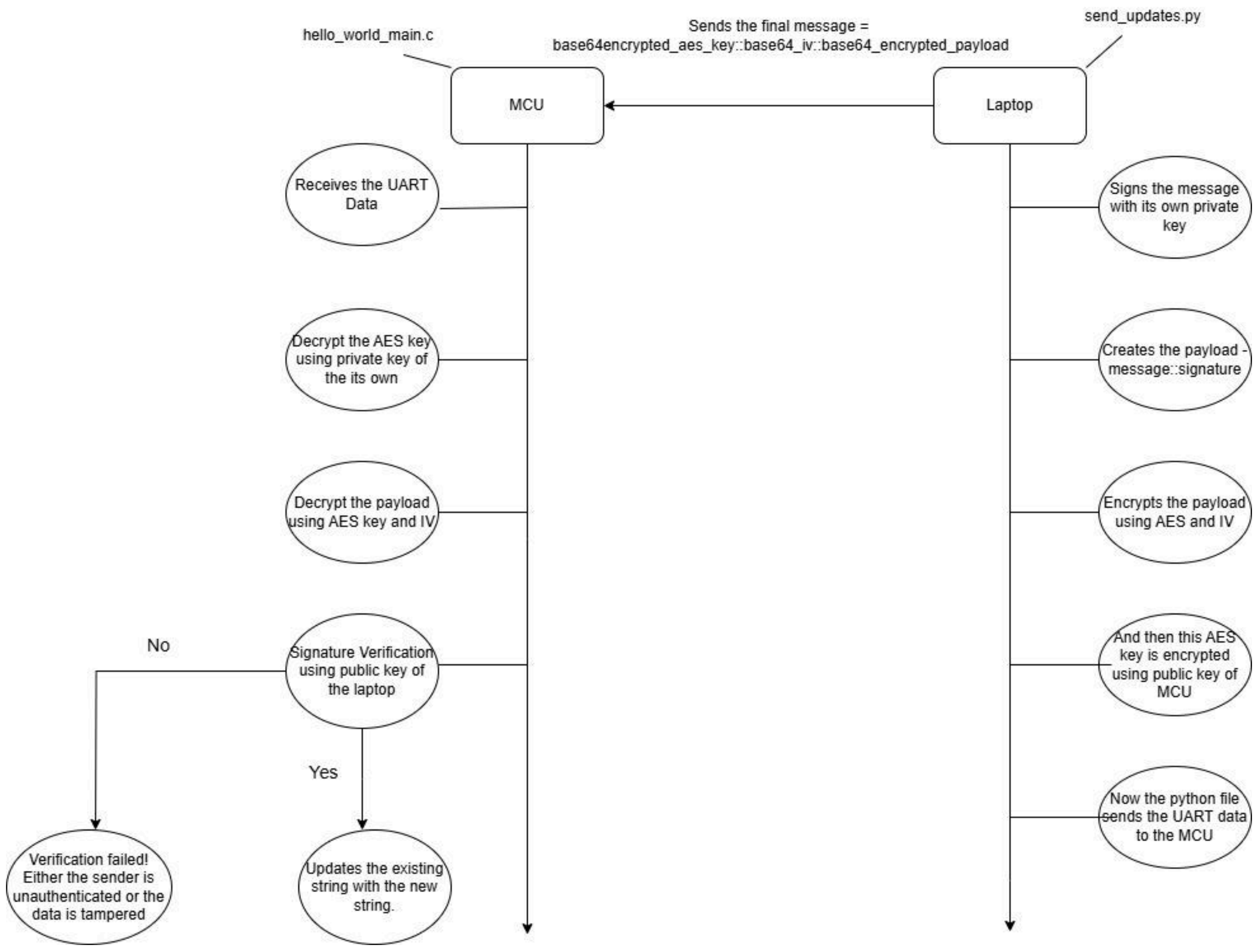
- **Confidentiality**: Ensured via AES encryption of the message.
- **Integrity & Authenticity:** Ensured via digital signature and RSA verification.

COMMUNICATION FLOW DIAGRAM:

The diagram illustrates the secure data exchange between a laptop and the ESP32 using hybrid encryption.

It shows how the message is signed, encrypted, transmitted, and then decrypted and verified by the MCU.

This ensures authenticity, integrity, and confidentiality during automotive software updates.


hello_world_main.c
Sends the final message =
base64encrypted_aes_key::base64_iv::base64_encrypted_payload
send_updates.py
MCU
Laptop
Receives the UART Data
Decrypt the AES key using private key of the its own
Decrypt the payload using AES key and IV
Signature Verification using public key of the laptop
No
Yes
Verification failed! Either the sender is unauthenticated or the data is tampered
Updates the existing string with the new string.
Signs the message with its own private key
Creates the payload - message::signature
Encrypts the payload using AES and IV
And then this AES key is encrypted using public key of MCU
Now the python file sends the UART data to the MCU

COMPONENTS AND TOOLS USED

Hardware:

ESP32 DevKitC-32E — The microcontroller that receives, decrypts, and verifies data.

Software:

- Python 3.12.10 — Used to implement the PC-side script for signing and encrypting messages.
- ESP-IDF — Official Espressif IoT Development Framework used to develop the ESP32 firmware.
- mbedTLS — Lightweight cryptographic library used for RSA, AES, Base64, and SHA256 operations.

Libraries & APIs:

- **ESP32 (C side):**
- mbedtls_pk_* — For RSA key parsing and encryption/decryption
- mbedtls_aes_* — For AES CBC decryption
- mbedtls_base64_* — For Base64 decoding
- mbedtls_sha256_* — For hashing during signature verification

- **Python side:**
- pycryptodome — Used for RSA/AES encryption and signing
- pyserial — Sends encrypted payload over UART to the ESP32
- base64 — Encodes all binary data before transmission

SOURCE CODE STRUCTURE:

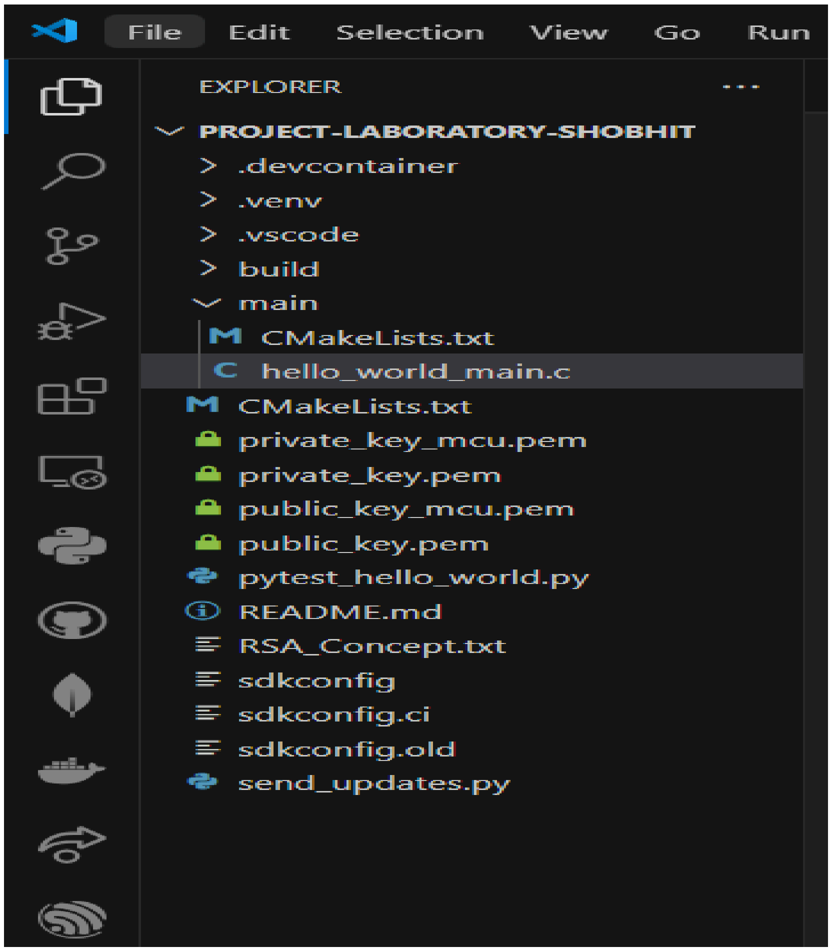
send_updates.py (Python Script – Laptop Side):

- Generates 2 RSA key pairs (one for the laptop, one for the MCU)
- Uses the laptop's private key to sign the message
- Encrypts the message using AES-256 with a random key and IV
- Encrypts the AES key using the MCU's public RSA key
- Sends the final base64-encoded payload over UART in the format:
  b64encrypted_aes_key::b64iv::b64encrypted_payload

hello_world_main.c (ESP32 Firmware):

- Contains embedded public and private RSA keys generated on the laptop
- Receives the payload over UART
- Decrypts the AES key using its private RSA key
- Decrypts the payload using the decrypted AES key and IV
- Verifies the signature using the laptop's public key
- If valid, updates the internal string with the received message

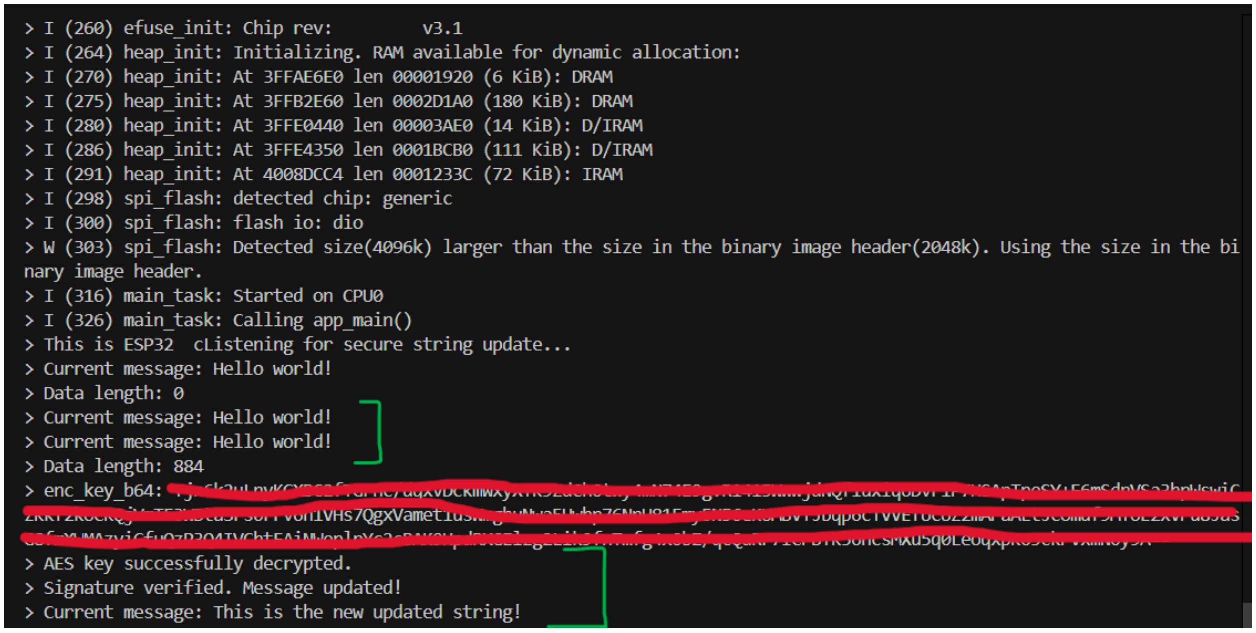
Note: Two RSA key pairs are generated during development on the laptop. The MCU's key pair and the laptop's public key are embedded into the ESP32 firmware to support secure hybrid encryption and signature verification.



```
PROJECT-LABORATORY-SHOBHIT
  > .devcontainer
  > .venv
  > .vscode
  > build
  v main
      CMakeLists.txt
      hello_world_main.c
    CMakeLists.txt
    private_key_mcu.pem
    private_key.pem
    public_key_mcu.pem
    public_key.pem
    pytest_hello_world.py
    README.md
    RSA_Concept.txt
    sdkconfig
    sdkconfig.ci
    sdkconfig.old
    send_updates.py
```

TESTING AND RESULTS:

- The ESP32 firmware was tested by sending multiple encrypted payloads via serial from the laptop.
- Initially, the ESP32 displays the default string: "Hello world!".
- Upon receiving a valid, signed, and encrypted payload, the string is successfully updated to the new verified message.
- If the payload is tampered or unsigned, the ESP32 rejects the update, ensuring authenticity and integrity.
- Logs on both sides (Python terminal and ESP32 serial monitor) confirm correct key decryption, payload processing, and signature verification.



```
> I (260) efuse_init: Chip rev:         v3.1
> I (264) heap_init: Initializing. RAM available for dynamic allocation:
> I (270) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
> I (275) heap_init: At 3FFB2E60 len 0002D1A0 (180 KiB): DRAM
> I (280) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
> I (286) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
> I (291) heap_init: At 4008DCC4 len 0001233C (72 KiB): IRAM
> I (298) spi_flash: detected chip: generic
> I (300) spi_flash: flash io: dio
> W (303) spi_flash: Detected size(4096k) larger than the size in the binary image header(2048k). Using the size in the bi
nary image header.
> I (316) main_task: Started on CPU0
> I (326) main_task: Calling app_main()
> This is ESP32  cListening for secure string update...
> Current message: Hello world!
> Data length: 0
> Current message: Hello world!
> Current message: Hello world!
> Data length: 884
> enc_key_b64: [redacted]
> AES key successfully decrypted.
> Signature verified. Message updated!
> Current message: This is the new updated string!
```

CHALLENGES FACED:

- **Stack Overflow:** Using large static buffers (like 512 bytes) for decryption led to stack overflow on ESP32; resolved by switching to dynamic malloc() allocations.

- **Base64 Decoding Issues:** Ensuring accurate buffer sizing and handling null-terminators was critical when decoding input from the serial stream.

- **mbedTLS API Usage:** Some APIs (like mbedtls_pk_parse_key and mbedtls_pk_decrypt) required non-null f_rng even if randomness wasn't needed for decryption, causing confusion during early stages.

- **Algorithm Selection:**

  RSA Considered First: RSA offers strong security and is ideal for public-key cryptography but is inefficient for large data due to size and speed limitations.

  AES Considered: AES is fast and well-suited for encrypting bulk data but requires a secure method to share the symmetric key.

  Why Hybrid Encryption?

  - Combines the strengths of both: RSA securely encrypts the AES key, while AES efficiently encrypts the payload.

- Ensures both **confidentiality** (AES) and **secure key exchange & authenticity** (RSA + SHA256 signature).

Final Approach: AES used for message encryption; RSA used to encrypt AES key and to verify sender authenticity using digital signatures.

FUTURE WORK:

- **Full Firmware Updates:**
  Extend the current string-based update mechanism to perform complete firmware updates. The ESP32 will securely verify and apply new firmware binaries using the same cryptographic pipeline.

- **Over-The-Air (OTA) Support:**
  Transition from UART-based updates to secure OTA mechanisms. This includes handling data transmission over Wi-Fi or Bluetooth, increasing usability and real-world deployment potential.

- **Improved Key Management:**
  Currently, both RSA key pairs are generated on the laptop side and embedded into the firmware. Future improvements could include secure on-device key generation or provisioning via a secure channel.

CONCLUSION:

This project successfully implemented a secure communication system between a laptop and an ESP32 microcontroller using hybrid encryption, combining RSA and AES. The system ensured both data confidentiality and authenticity by encrypting the AES key with RSA and verifying the payload using RSA-SHA256 signatures. Through this work, I gained hands-on experience in applying cryptographic algorithms, managing secure UART communication, and designing firmware capable of handling encrypted payloads on a resource-constrained device. Overall, the project bridged theoretical cryptographic knowledge with practical application in the context of secure automotive software updates.