EC5511-ANALOG AND DIGITAL COMMUNICATION LABORATORY

Implementing AES encryption for Secure Data Transmission Using MATLAB and ESP32

Submitted by

KABILESH RAJ T -2022504541

PREMALATHA M -2022504543

ANU PRASANNA R -2022504545

UNNAM PARTHSARATHI -2022504547

BACHELOR OF ENGINEERING

IN

ELECTRONICS AND COMMUNICATION

ENGINEERING

DEPARTMENT OF ELECTRONICS ENGINEERING

MADRAS INSTITUTE OF TECHNOLOGY

ANNA UNIVERSITY: CHENNAI – 600 004

OCTOBER 2024

BONAFIDE CERTIFICATE

Certified that this project report "**Implementing AES Encryption for Secure Data Transmission Using MATLAB and ESP32"** is the Bonafide work of

KABILESH RAJ T	-2022504541
PREMALATHA M	-2022504543
ANU PRASANNA R	-2022504545
UNNAM PARTHSARATHI	-2022504547

who carried out the project under my supervision.

SIGNATURE

Dr. G SUMITHRA

Associate Professor

Dr. S RAM PRABHU

Assistant Professor

Department of Electronics Engineering

Madras Institute of Technology

Anna University Chennai -44.

ACKNOWLEDGEMENT

With profound gratitude and due regards, I sincerely acknowledge with thanks the opportunity provided to us by our respectful, Dean **Dr. RAVICHANDRAN KANDASWAMY,** Madras Institute of Technology, Anna University for providing a good environment and facilities.

We thank our respectful Head of the Electronics Department,

Dr. D MEGANATHAN, Professor , Madras Institute of Technology, Anna University for his encouragement during the project work.

We sincerely express our gratitude to our project guides ,**Dr. G SUMITHRA,** Associate Professor, and **Dr. S RAMPRABHU**, Assistant professor, Department of Electronics Engineering for being an inspiration to us and for valuable guidance throughout the course of this project.

Place: Chennai

Date :23/10/2024

ABSTRACT

This project demonstrates a secure communication system integrating MATLAB and the ESP32 microcontroller. Utilizing AES Encryption, the system ensures data security through encryption and decryption processes, with MATLAB handling encryption and the ESP32 managing wireless data transmission.

Project Overview

1. Development Setup:

- Hardware: ESP32 Development Board, USB cable, and a computer with MATLAB.

- Software: MATLAB for AES encryption/decryption, Arduino IDE for ESP32 programming.

2. Encryption and Decryption:

- MATLAB: Encrypts plaintext messages using AES and transmits the encrypted data via serial communication to the ESP32. Another script decrypts the data upon receipt.

- ESP32: Configured as a Wi-Fi access point to receive and process encrypted messages.

3. System Testing:

- The ESP32 receives encrypted data from MATLAB over serial communication, and the decrypted message is verified for accuracy.

Expected Outcomes:

- Functional System: A prototype demonstrating secure wireless communication with AES encryption.

OBJECTIVE

The main objective of this project is to implement AES encryption for secure data transmission between an ESP32 microcontroller and a MATLAB application. The goal is to ensure that the data exchanged between the two platforms is encrypted, maintaining confidentiality and security during transmission over wireless communication channels such as Wi-Fi or Bluetooth. This is particularly important for sensitive data to prevent unauthorized access and tampering.

Implementing AES Encryption for Secure Data Transmission Using MATLAB and ESP32

AIM: The aim of this project is to develop a secure communication system between an ESP32 microcontroller and MATLAB using AES encryption. The system will ensure that sensitive data transmitted over wireless networks is encrypted, preventing unauthorized access and guaranteeing the integrity and confidentiality of the data during transmission.

Project Flow:

1. **System Setup**:
    - Set up ESP32 and MATLAB.
2. **ESP32 Implementation**:
    - Write code to encrypt data using AES.
    - Transmit encrypted data via Wi-Fi or Bluetooth.
3. **MATLAB Implementation**:
    - Write MATLAB code to decrypt the received data using the same AES key.
4. **Testing & Validation**:
    - Test communication between ESP32 and MATLAB.
    - Ensure successful encryption, transmission, and decryption.
5. **Final Testing & Optimization**:
    - Optimize for performance and reliability.
6. **Documentation**:
    - Document setup, code, and test results.

SYSTEM SETUP:

//Library Inclusions

```
#include <WiFi.h>
#include <WiFiUdp.h>
```

Explanation:

WiFi.h: This library is used to connect the ESP32 to a Wi-Fi network. It provides functions to initiate and manage Wi-Fi connections.

WiFiUdp.h: This library provides support for UDP (User Datagram Protocol) communication over Wi-Fi. Since UDP is a connectionless protocol, it is used here to listen for messages that will be sent from MATLAB to the ESP32.

mbedtls/aes.h: This is the header file for AES encryption functionality provided by the **mbedTLS** library. mbedTLS is an open-source cryptographic library that provides security algorithms and protocols such as SSL/TLS, RSA, AES, and more. It is widely used for secure communications in embedded systems.

AES (Advanced Encryption Standard): AES is a symmetric encryption algorithm used to secure data. It can encrypt and decrypt data using a shared key.

These libraries are essential for establishing wireless communication with the ESP32.

ESP32 IMPLEMENTATION:

Arduino IDE Code

```
#include <WiFi.h>
#include <WiFiUdp.h>
#include <mbedtls/aes.h>  // Official ESP32 library for AES

const char* ssid = "Xiaomi 11i";
const char* password = "12345678";

WiFiUDP udp;
WiFiServer server(80);            // Web server running on port 80
unsigned int localUdpPort = 4210;    // Listening port
char incomingPacket[255];          // Buffer for incoming messages
String decryptedMessage = "No message received yet.";  // Default message

// AES key (256 bits) and IV (128 bits)
unsigned char aes_key[32] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                             0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
                             0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
                             0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F}; // 32 bytes
(256-bit key)

unsigned char iv[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F}; // 16 bytes IV
```

```
void setup() {
  Serial.begin(115200);
  delay(1000);

  // Connect to Wi-Fi
  Serial.print("Connecting to WiFi");
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Start listening for UDP messages
  udp.begin(localUdpPort);
  Serial.printf("Listening on UDP port %d\n", localUdpPort);

  // Start the web server
  server.begin();
}
```

```
void loop() {
  // Check for incoming UDP messages
  int packetSize = udp.parsePacket();
  if (packetSize) {
    int len = udp.read(incomingPacket, 255);
    if (len > 0) {
      incomingPacket[len] = 0;  // Null-terminate the received message
    }

    // Reset the IV before each decryption (as it's CBC mode)
    unsigned char iv_copy[16];
    memcpy(iv_copy, iv, 16);  // Copy the original IV

    // Decrypt the message using AES-256 in CBC mode
    mbedtls_aes_context aes;
    mbedtls_aes_init(&aes);
    mbedtls_aes_setkey_dec(&aes, aes_key, 256);

    unsigned char decrypted[255];  // Buffer for decrypted message

    mbedtls_aes_crypt_cbc(&aes, MBEDTLS_AES_DECRYPT, len, iv_copy,
(unsigned char*)incomingPacket, decrypted);

    // Remove padding (assuming PKCS#7 padding)
```

```
  int padding_len = decrypted[len - 1];  // Last byte contains padding length
  if (padding_len > 16) {
    padding_len = 0;  // In case of incorrect padding, prevent removal
  }
  int decrypted_len = len - padding_len;

  decrypted[decrypted_len] = 0;  // Null-terminate decrypted string

  // Store the decrypted message
  decryptedMessage = String((char*)decrypted);
  Serial.printf("Decrypted message: %s\n", decryptedMessage.c_str());

  mbedtls_aes_free(&aes);
}
// Handle web client requests
WiFiClient client = server.available();
if (client) {
  Serial.println("New client connected.");
  String currentLine = "";
  while (client.connected()) {
    if (client.available()) {
      char c = client.read();
      if (c == '\n') {
        // If the current line is blank, it means the request is complete
        if (currentLine.length() == 0) {
```

```
          // Send a HTTP response with the decrypted message
          client.println("HTTP/1.1 200 OK");
          client.println("Content-type:text/html");
          client.println();
          client.print("<html><head>");
          client.print("</head><body>");
          client.print("<h1 style=\"text-align:center;\">DECRYPTED UDP
MESSAGE</h1>");
          client.print("<h2 style=\"text-align:center;\">");
          client.print(decryptedMessage);  // Display decrypted message
          client.print("</h2>");
          client.println("</body></html>");
          break;
        } else {
          currentLine = "";
        }
      } else if (c != '\r') {
        currentLine += c;
      }
    }
  }
  client.stop();
  Serial.println("Client disconnected.");
 }
}
```

1024-QAM (Quadrature Amplitude Modulation)

1024-QAM is a high-order modulation scheme that represents data by encoding 10 bits per symbol (since 2^10 = 1024

In QAM, both the amplitude and phase of the carrier wave are varied to represent different symbols. With 1024-QAM, there are 1024 different symbol combinations, meaning that more data can be transmitted with each symbol.

Advantage: Higher data rates can be achieved because more information is packed into each transmitted symbol. For example, 1024-QAM can provide a 25% increase in data rate over 256-QAM.

Trade-off: Higher-order QAM (like 1024-QAM) requires a cleaner signal with a better signal-to-noise ratio (SNR), which means it's most effective at short distances or in low-interference environments.

OFDMA (Orthogonal Frequency-Division Multiple Access)

OFDMA is an advanced version of OFDM (Orthogonal Frequency-Division Multiplexing). While OFDM divides a channel into multiple subcarriers, OFDMA takes it a step further by assigning different subcarriers to different devices, allowing simultaneous communication with multiple users.

This enhances spectral efficiency because each user can transmit on a subset of subcarriers rather than occupying the entire bandwidth.

Advantage: This allows multiple devices to use the same channel more efficiently, reducing congestion and latency, especially in environments with a high density of devices (like offices, stadiums, or homes with many IoT devices).

The Combination of 1024-QAM + OFDMA:

1024-QAM increases the data rate by encoding more bits per symbol.

OFDMA improves the efficiency and capacity of the network by allowing multiple users to share the same channel, reducing collisions and optimizing bandwidth usage.

Together, this combination in 802.11ax (Wi-Fi 6) provides:

Higher throughput.

Better handling of many devices (like smartphones, laptops, IoT devices) simultaneously.

Improved performance in dense environments, like apartment buildings or crowded public spaces.

This technology is one of the key reasons Wi-Fi 6 is much more efficient and faster compared to previous generations like Wi-Fi 5 (802.11ac).

Explanation of ESP32 Code:

WiFi Setup:

- **#include <WiFi.h>**: This library allows the ESP32 to connect to Wi-Fi networks.
- **const char* ssid = "Xiaomi 11i";**: This is the name (SSID) of the Wi-Fi network the ESP32 will connect to.
- **const char* password = "12345678";**: This is the password for the Wi-Fi network.

The ESP32 tries to connect to the specified Wi-Fi network, and once connected, it prints the assigned IP address.

2. UDP Setup:

- **#include <WiFiUdp.h>**: This library enables the use of the **UDP protocol** to send and receive messages over the network.
- **WiFiUDP udp;**: This creates a UDP object to handle communication.
- **unsigned int localUdpPort = 4210;**: This is the local UDP port number where the ESP32 will listen for incoming messages.

3. AES Encryption Setup:

- **#include <mbedtls/aes.h>**: This is the AES library from **mbedTLS**, which is built into the ESP32 framework. It provides functionality for AES encryption and decryption.
- **unsigned char aes_key[32]**: This is a 256-bit (32 bytes) AES key that will be used to decrypt the incoming messages. In AES-256 encryption, the key length is 256 bits.
- **unsigned char iv[16]**: This is a 128-bit **Initialization Vector (IV)**, required in CBC mode. The IV adds randomness to the encryption process and ensures that the same plaintext will encrypt to different ciphertext each time.

4. Listening for UDP Packets:

- In the loop() function, the ESP32 constantly listens for incoming UDP packets:
    - **int packetSize = udp.parsePacket();**: This checks if a UDP packet is received. If so, it returns the size of the packet.
    - **udp.read(incomingPacket, 255);**: This reads the received packet into the incomingPacket buffer.
    - The received packet is displayed in hexadecimal format for debugging using Serial.printf.

5. Decrypting the UDP Packet (AES-256 in CBC Mode):

- **AES Context**:
    - **mbedtls_aes_context aes;**: Creates a context to hold the AES decryption information.

- **mbedtls_aes_init(&aes);**: Initializes the AES context.
- **mbedtls_aes_setkey_dec(&aes, aes_key, 256);**: Configures the AES context with the 256-bit key for decryption.

- **Decryption**:
  - **mbedtls_aes_crypt_cbc(&aes, MBEDTLS_AES_DECRYPT, len, iv_copy, (unsigned char*)incomingPacket, decrypted);**: Decrypts the incoming packet using AES-256 in CBC mode.
    - iv_copy: A copy of the IV is passed to the function because CBC mode modifies the IV during decryption.
    - incomingPacket: The encrypted data (ciphertext).
    - decrypted: Buffer to store the decrypted data (plaintext).
- **Removing PKCS#7 Padding**:
  - AES-256 in CBC mode uses padding (commonly **PKCS#7**) to ensure that the plaintext is a multiple of the block size (16 bytes).
  - **int padding_len = decrypted[len - 1];**: The last byte of the decrypted message indicates the padding length.
  - **int decrypted_len = len - padding_len;**: This computes the length of the actual plaintext by subtracting the padding length.
- **Handling HTTP Requests**

  Client Connection:
  - The server checks for new client connections (server.available()).
  - If a client connects, it reads the incoming HTTP request line by line and processes it.

  HTTP Response:
  - Once the entire HTTP request is received (indicated by a blank line), the ESP32 responds with a status code 200 OK.
  - The response contains an HTML page that displays the decrypted UDP message stored in decryptedMessage.

Client Disconnection:

- After sending the response, the client connection is closed using client.stop().

- **Final Output**
  - The program listens for encrypted UDP messages, decrypts them using AES-256 CBC, and serves the decrypted message on a webpage via an HTTP server. This is useful for secure communication systems, where encrypted messages can be sent to the ESP32, decrypted, and then accessed through a web interface.

MATLAB IMPLEMENTATION:

```
% MATLAB code to send AES-256 encrypted messages to ESP32 using Java

import java.net.DatagramSocket
import java.net.DatagramPacket
import java.net.InetAddress

% Define the ESP32 IP address and port
ipAddress = '192.168.39.141';
port = 4210;

% Define the message you want to send
message = input('Enter the message : ', 's');

% AES-256 key (256-bit)
key = uint8([0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, ...
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, ...
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, ...
        0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F]);  % 256-bit key

% Initialization vector (IV) for AES (16 bytes)
```

```
iv = uint8([0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, ...
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F]);  % 16-byte
IV

% Convert message to uint8
messageBytes = uint8(message);

% Pad message to multiple of 16 bytes (AES block size)
padLength = 16 - mod(length(messageBytes), 16);
messagePadded = [messageBytes, uint8(repmat(padLength, 1,
padLength))];

% Encrypt the message using AES-256 in CBC mode
cipher = javax.crypto.Cipher.getInstance('AES/CBC/NoPadding');
keySpec = javax.crypto.spec.SecretKeySpec(key, 'AES');
ivSpec = javax.crypto.spec.IvParameterSpec(iv);
cipher.init(javax.crypto.Cipher.ENCRYPT_MODE, keySpec, ivSpec);
encryptedMessage = cipher.doFinal(messagePadded);

% Create a UDP socket using java.net.DatagramSocket
udpSocket = DatagramSocket();

% Convert IP address to Java InetAddress
```

```
inetAddress = InetAddress.getByName(ipAddress);


% Create a UDP packet with the encrypted message
packet = DatagramPacket(encryptedMessage,
length(encryptedMessage), inetAddress, port);


% Send the packet via UDP
udpSocket.send(packet);


disp('Encrypted message sent.');
disp(encryptedMessage);


% Close the socket after use
udpSocket.close();
```

Explanation of MATLAB Code:

- **Import Java Libraries**: The code imports Java classes for UDP networking (DatagramSocket, DatagramPacket, InetAddress).
- **Define ESP32 Connection**: It sets the ESP32's IP address and port (4210), which should match the settings in the ESP32 code.
- **Prepare the Message**: The plaintext message ('The code is working') is defined, and an AES-256 encryption key (256 bits) and initialization vector (IV, 128 bits) are specified.
- **Pad the Message**: The message is padded to ensure its length is a multiple of 16 bytes, which is required for AES encryption.

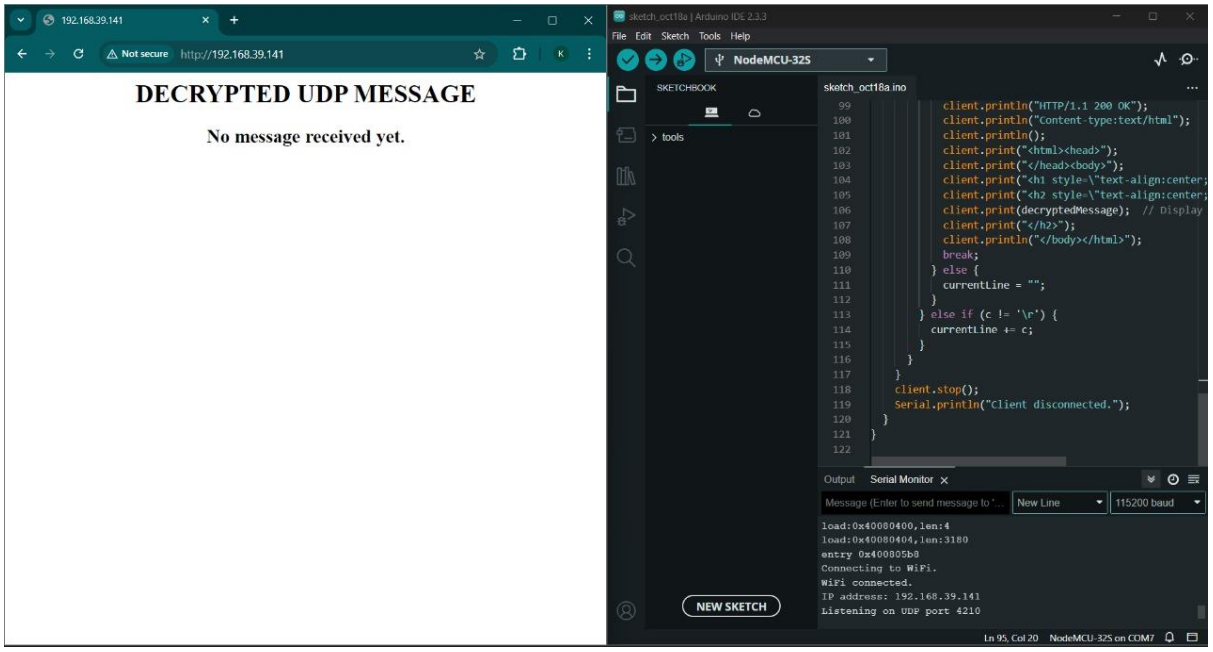
- **Encrypt the Message**: The message is encrypted using AES-256 in CBC mode. The encryption key and IV are provided to the cipher.
- **Create UDP Socket:** A UDP socket is created, and the IP address of the ESP32 is converted to a Java InetAddress.
- **Send the Encrypted Message**: A UDP packet containing the encrypted message is created and sent to the ESP32.
- **Close the Socket:** Finally, the UDP socket is closed to free resources.

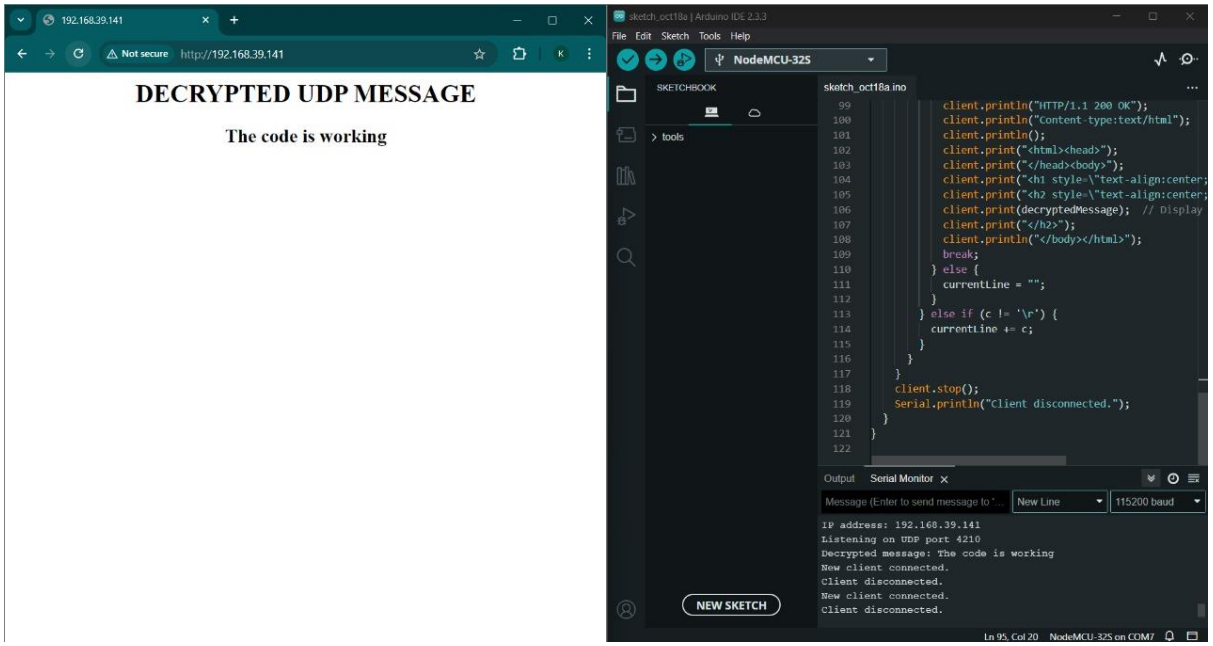
OUTPUT:

Encrypted message

```
>> Project
Enter the message : The code is working
Encrypted message sent.
   -114
    119
   -104
    -20
    120
     75
     98
    112
     67
    -11
     27
     74
   -127
      0
     -6
     42
    -17
    -86
    -21
   -123
     50
    -44
     50
   -106
    -93
     36
    -63
     95
     24
     60
     -1
   -128
```

Decrypted message





APPLICATIONS OF THE PROJECT:

Satellite Communication:

- **Data Encryption**: Secure communication between satellites and ground stations, protecting sensitive data related to national security and scientific research.
- **Telemetry and Control**: Ensure the integrity and confidentiality of telemetry data sent from satellites to control centers.

Military Communications:

- **Encrypted Messaging**: Secure communication channels for military personnel, ensuring that sensitive operational data and commands are transmitted securely.

Drone Operations:

- **Secure Control and Data Transmission**: Ensure that communication between drones and operators is secure, particularly during surveillance and combat missions.

Cybersecurity for Defence Systems:

- **Secure Network Protocols**: Implement robust encryption methods to protect defence networks from cyber threats and attacks.

CONCLUSION:

The project successfully implemented secure data transmission using encryption methods alongside the ESP32 and MATLAB. It enhances security by providing robust protection against unauthorized access, making it suitable for high-security applications. By integrating the ESP32's IoT capabilities with MATLAB's data processing, the project demonstrates an effective approach to handling real-time secure data transmission. Future improvements can include transitioning to more advanced encryption methods and implementing additional security measures to further strengthen the system. Overall, this project contributes to the ongoing need for secure communication technologies, highlighting their critical importance in an interconnected world, especially for applications that require high confidentiality and integrity.