AES-256 Encryption and Decryption on Raspberry Pi with ECB and CBC Modes

Gliese-514b

Team Members:
Javier Eguia Chaire

Professor:
Dr. Mohamed El-Hadedy Aly

California State Polytechnic University, Pomona Department of Electrical and Computer Engineering, College of Engineering

December 9, 2024

CONTENTS

AES-256 Encryption and Decryption on Raspberry Pi with ECB and CBC Modes

Group I

I. Introduction

The goal of this project was to develop a secure file encryption and decryption system that utilizes a combination of RSA and AES cryptographic algorithms to ensure data confidentiality and secure communication. In today's digital age, the protection of sensitive data during transmission and storage has become a fundamental requirement across various sectors, including finance, healthcare, and governmental organizations. This project aims to demonstrate a practical implementation of a hybrid cryptographic approach to safeguard data from unauthorized access.

The system comprises a client-server architecture where files can be encrypted and decrypted remotely. RSA, an asymmetric cryptographic algorithm, is employed for secure key exchange between the client and server, eliminating the need for direct transmission of sensitive keys over insecure channels. AES, a symmetric cryptographic algorithm, is used for efficient file encryption and decryption due to its high speed and suitability for bulk data.

Furthermore, the system supports switching between two popular AES modes of operation: Cipher Block Chaining (CBC) and Electronic Codebook (ECB). CBC is known for providing robust encryption by ensuring that patterns in the plaintext are not reflected in the ciphertext, making it more secure against statistical attacks. ECB, while faster, encrypts blocks independently, which makes it suitable for scenarios where speed is prioritized over security.

This system's flexibility to toggle between these modes allows users to choose the most appropriate encryption strategy based on their specific use case. Additionally, the system handles text files, ensuring that binary data is processed without loss, and provides seamless feedback to the user regarding encryption and decryption results.

The functionality of this project holds significant potential for applications such as secure file sharing, cloud storage encryption, and real-time data protection in distributed systems. It not only serves as a practical tool but also provides an educational insight into the working principles of modern cryptography. The project emphasizes key aspects such as the secure exchange of cryptographic keys, the use of initialization vectors to enhance security, and the importance of padding in block-based encryption schemes.

By integrating error handling and support for mode switching, this implementation serves as a robust framework for understanding and applying hybrid cryptographic techniques in real-world scenarios. This report elaborates on the system's design, implementation, and potential applications, providing a comprehensive overview of the functionalities and the underlying principles that make this project both secure and versatile.

A. Motivation

In an era where data security is paramount, the need for robust encryption systems to safeguard sensitive files during transmission and storage has never been greater. Cybersecurity threats, such as unauthorized access, data breaches, and eavesdropping, pose significant risks to individuals and organizations alike. This project seeks to address these challenges by implementing a hybrid encryption system that ensures the confidentiality and integrity of sensitive data.

This system combines the strengths of RSA and AES cryptographic algorithms, demonstrating a practical application of hybrid encryption techniques. RSA, an asymmetric cryptographic algorithm, is used to securely exchange keys between the client and server, preventing the exposure of sensitive cryptographic material during transmission. AES, a symmetric cryptographic algorithm, complements RSA by providing efficient and secure file encryption and decryption, suitable for handling large volumes of data.

The project is motivated by the increasing demand for secure communication protocols in various domains, such as secure file sharing, cloud storage, and remote access systems. By incorporating both Cipher Block Chaining (CBC) and Electronic Codebook (ECB) modes of AES encryption, the system provides flexibility in balancing security and performance, catering to a wide range of use cases. CBC offers enhanced security by eliminating patterns in the ciphertext, while ECB provides faster encryption for non-critical applications.

Moreover, this implementation serves as an educational tool for understanding the principles of modern cryptographic systems. It emphasizes the importance of key exchange protocols, the use of initialization vectors to ensure randomness, and the role of padding in block cipher encryption. The system's ability to toggle between different encryption modes further highlights the trade-offs between security and efficiency, allowing users to make informed decisions based on their specific requirements.

In summary, the motivation behind this project is to address the critical need for secure and efficient encryption systems in a world increasingly reliant on digital communication. By leveraging the complementary strengths of RSA and AES, this system demonstrates a practical solution for protecting sensitive information while providing a foundation for further exploration and development in the field of cryptography.

B. Applications

The functionality of this project can be applied in various fields:

- Secure file transfer in corporate environments.
- Data storage solutions for ensuring the confidentiality of sensitive files.
- Educational tools to demonstrate the fundamentals of cryptography.
- Secure communication systems in military and government applications.

II. System Overview

The system comprises two main components: the **Server** and the **Client**, which interact to securely transmit, encrypt, and decrypt files.

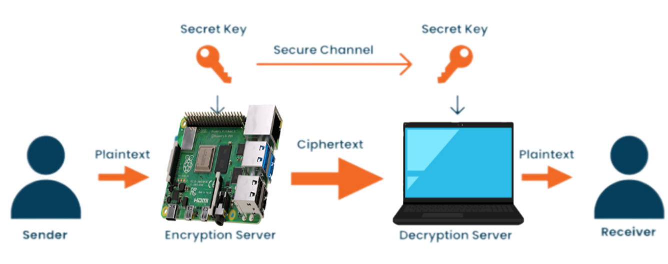

Fig. 1. Project Implementation

A. Server

The server is responsible for securely receiving files from the client, performing the requested encryption or decryption operations, and sending the processed files back. It initializes the cryptographic protocols by generating a private-public RSA key pair and securely exchanging a shared AES key using RSA encryption during the handshake process. Additionally, the server can handle file data in both AES modes: Cipher Block Chaining (CBC) and Electronic Codebook (ECB). The server supports error handling to ensure robustness and reliability during transmission.

B. Client

The client initiates communication with the server and performs file transfer operations. It generates its RSA key pair, which is used during the key exchange phase to securely derive the shared AES key. The client sends files for encryption or decryption, specifying the mode of operation (CBC or ECB). The processed files, whether encrypted or decrypted, are saved locally on the client system for further use.

C. Key Features

This system incorporates several advanced features to ensure functionality and security:

- **Secure Key Exchange:** The RSA key exchange guarantees the secure derivation of a shared secret.
- **AES Mode Switching:** Support for both CBC and ECB modes offers flexibility based on user needs and file sensitivity.
- **Binary-Safe Handling:** The system ensures proper handling of binary files to prevent encoding issues.

- **Error Resilience:** Robust error-handling mechanisms mitigate issues during data transmission and processing.

D. Hybrid Encryption

The system employs a hybrid encryption strategy to achieve the benefits of both asymmetric and symmetric cryptography:

- **RSA:** Used for securely exchanging the shared AES key between the client and the server.
- **AES:** Utilized for high-speed encryption and decryption of the actual file data, ensuring confidentiality and efficiency.

E. Switching Modes (CBC and ECB)

The Advanced Encryption Standard (AES) is employed in two modes:

- **CBC (Cipher Block Chaining):** Each plaintext block is XORed with the previous ciphertext block before encryption, ensuring strong confidentiality. CBC mode is recommended for highly sensitive data as it prevents patterns from being preserved in the ciphertext.
- **ECB (Electronic Codebook):** Each block is encrypted independently, offering faster encryption but weaker security. ECB mode is suitable for non-sensitive data due to its susceptibility to pattern leaks.

The system allows dynamic switching between these modes using the `SWITCH_MODE` command, granting the user flexibility based on specific use cases.

F. Key Derivation

The shared secret is derived using the **HKDF** (HMAC-based Key Derivation Function). HKDF ensures cryptographic security by generating a 32-byte AES key from the random values exchanged during the RSA handshake. This mechanism guarantees that the derived key remains secure even in the presence of partial information leakage.

G. File Handling and Data Flow

The overall system data flow can be summarized as follows:

1) The client connects to the server and initiates the RSA-based handshake for secure key exchange.
2) The client sends a file along with the command specifying the operation (ENCRYPT/DECRYPT) and the desired AES mode (CBC/ECB).
3) The server processes the file by applying the requested encryption or decryption using the derived shared key.
4) The processed file is sent back to the client, which saves it locally for verification or further use.

This architecture provides a seamless and secure pipeline for handling sensitive files, ensuring both confidentiality and usability.

III. Fundamental Components of the Code

A. Hybrid Encryption

The system uses hybrid encryption:

- **RSA:** Asymmetric encryption is used for securely exchanging a shared secret between the client and the server.
- **AES:** Symmetric encryption is used for efficiently encrypting and decrypting the files.

B. Switching Modes (CBC and ECB)

AES supports different modes of operation:

- **CBC (Cipher Block Chaining):** Ensures confidentiality by XORing each plaintext block with the previous ciphertext block.
- **ECB (Electronic Codebook):** Encrypts each block independently. While faster, it is less secure due to patterns in data being preserved.

The system allows the user to switch between these modes dynamically using the `SWITCH_MODE` command.

C. Key Derivation

The shared secret is derived using HKDF (HMAC-based Key Derivation Function), which ensures cryptographic security by generating a 32-byte key from the random values exchanged during the RSA handshake.

IV. Implementation Details

A. Server Code

The server code is responsible for receiving files, encrypting or decrypting them, and sending the results back to the client. Below is a snippet illustrating RSA and AES setup:

```python
import os
import socket
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric.padding import OAEP, MGF1
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.serialization import load_pem_public_key, Encoding, PublicFormat

# Generate RSA keys
server_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
server_public_key = server_key.public_key()

# Serialize the public key
server_public_pem = server_public_key.public_bytes(
    encoding=Encoding.PEM,
    format=PublicFormat.SubjectPublicKeyInfo
)

# Set up the server
HOST = "0.0.0.0"
PORT = 5000
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))
server_socket.listen(1)

print(f"[Server] Listening on {HOST}:{PORT}")
conn, addr = server_socket.accept()
print(f"[Server] Connection established with {addr}")

# Send the server's public key to the client
conn.sendall(server_public_pem)

# Receive the client's public key
client_public_pem = conn.recv(4096)
client_public_key = load_pem_public_key(client_public_pem)

# Perform Diffie-Hellman-like exchange using RSA
server_random = os.urandom(32)
encrypted_client_random = conn.recv(256)
client_random = server_key.decrypt(
    encrypted_client_random,
    OAEP(mgf=MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
)

# Encrypt and send the server random
encrypted_server_random = client_public_key.encrypt(
    server_random,
    OAEP(mgf=MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
)
conn.sendall(encrypted_server_random)

# Derive a shared secret
shared_secret = HKDF(
    algorithm=hashes.SHA256(),
    length=32,
    salt=None,
    info=b'shared secret'
).derive(server_random + client_random)

# AES Setup
iv = os.urandom(16)
conn.sendall(iv)

# Default mode: CBC
current_mode = modes.CBC(iv)
cipher = Cipher(algorithms.AES(shared_secret), current_mode)
encryptor = cipher.encryptor()
decryptor = cipher.decryptor()

try:
    while True:
        # Receive command
        command = conn.recv(16).decode("utf-8").strip().upper()
        if command == "EXIT":
            print("[Server] Exiting.")
            break
        elif command == "SWITCH_MODE":
            # Switch between CBC and ECB
            if isinstance(current_mode, modes.CBC):
                current_mode = modes.ECB()
                print("[Server] Switched to ECB mode.")
            else:
                current_mode = modes.CBC(iv)
                print("[Server] Switched to CBC mode.")
            # Update cipher, encryptor, and decryptor
            cipher = Cipher(algorithms.AES(shared_secret), current_mode)
            encryptor = cipher.encryptor()
            decryptor = cipher.decryptor()
        elif command in ["ENCRYPT", "DECRYPT"]:
            # Receive file size and data
            file_size = int.from_bytes(conn.recv(4), 'big')
            data = conn.recv(file_size)

            if command == "ENCRYPT":
                processed_data = encryptor.update(data) + encryptor.finalize()
                print("[Server] Data encrypted.")
            elif command == "DECRYPT":
                processed_data = decryptor.update(data) + decryptor.finalize()
                print("[Server] Data decrypted.")

```

```python
            # Send processed data back
            conn.sendall(len(processed_data).to_bytes(4, 'big'))
            conn.sendall(processed_data)
            print(f"[Server] {command}ed data sent back.")
        else:
            print(f"[Server] Invalid command received: {command}")

except Exception as e:
    print(f"[Server Error] {e}")

finally:
    conn.close()
    server_socket.close()
```

B. Client Code

The client code is responsible for sending files to the server and receiving the processed files. Below is a snippet showing how files are sent:

```python
import os
import socket
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric.padding import OAEP, MGF1
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.serialization import Encoding, PublicFormat, load_pem_public_key

# Generate RSA keys
client_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
client_public_key = client_key.public_key()

# Serialize the public key
client_public_pem = client_public_key.public_bytes(
    encoding=Encoding.PEM,
    format=PublicFormat.SubjectPublicKeyInfo
)

# Connect to the server
HOST = "192.168.50.219"  # Replace with server's IP
PORT = 5000
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((HOST, PORT))

# Receive server's public key
server_public_pem = client_socket.recv(4096)
server_public_key = load_pem_public_key(server_public_pem)

# Send client's public key
client_socket.sendall(client_public_pem)

# Perform Diffie-Hellman-like exchange using RSA
client_random = os.urandom(32)
encrypted_client_random = server_public_key.encrypt(
    client_random,
    OAEP(mgf=MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
)
client_socket.sendall(encrypted_client_random)

encrypted_server_random = client_socket.recv(256)
server_random = client_key.decrypt(
    encrypted_server_random,
    OAEP(mgf=MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
)

# Derive a shared secret
shared_secret = HKDF(
    algorithm=hashes.SHA256(),
    length=32,
    salt=None,
    info=b'shared secret'
).derive(client_random + server_random)

# AES Setup
iv = client_socket.recv(16)
mode = modes.CBC(iv)  # Default mode: CBC

try:
    while True:
        command = input("Enter command (ENCRYPT, DECRYPT, SWITCH_MODE, EXIT): ").strip().upper()
        if command == "EXIT":
            client_socket.sendall(command.encode().ljust(16, b' '))
            break
        elif command == "SWITCH_MODE":
            client_socket.sendall(command.encode().ljust(16, b' '))
            print("[Client] Requested to switch mode.")
        elif command in ["ENCRYPT", "DECRYPT"]:
            client_socket.sendall(command.encode().ljust(16, b' '))

            # Get file path
            file_path = input("Enter file path: ").strip()
            if not os.path.exists(file_path):
                print("[Client] File does not exist.")
                continue

            with open(file_path, "rb") as file:
                file_data = file.read()

```

```
            # Send file size and data
            client_socket.sendall(len(file_data).to_bytes(4, 'big'))
            client_socket.sendall(file_data)
            print(f"[Client] File sent for {command.lower()}ion.")

            # Receive processed file
            file_size = int.from_bytes(client_socket.recv(4), 'big')
            processed_data = client_socket.recv(file_size)

            # Save the processed file
            output_file = f"{command.lower()}ed_output.txt"
            with open(output_file, "wb") as output:
                output.write(processed_data)
            print(f"[Client] Processed file saved as '{output_file}'.")
        else:
            print("[Client] Invalid command.")

except Exception as e:
    print(f"[Client Error] {e}")

finally:
    client_socket.close()
```

V. Testing and Results

A. Testing Scenarios

The system was tested under the following scenarios:

1) Encrypting a text file using CBC mode and decrypting it back to its original form.
2) Switching to ECB mode and repeating the encryption and decryption process.
3) Handling binary-safe files to ensure the system supports non-text files.

Fig. 2. Server Implementation

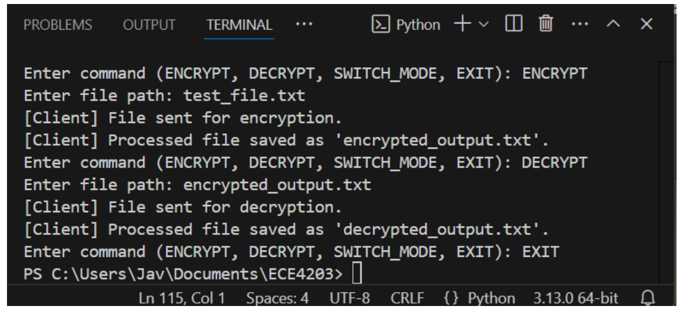

Fig. 3. Client Implementation

B. Results

The system successfully encrypted and decrypted text files, demonstrating secure and efficient file handling. The dynamic mode switching functionality worked as expected, toggling between CBC and ECB modes seamlessly.

VI. Conclusion and Future Work

A. Conclusion

This project successfully implemented a secure file encryption and decryption system using hybrid cryptography, showcasing the practical integration of RSA and AES algorithms for robust data protection. By leveraging RSA's capability for secure key exchange and AES's efficiency in file encryption and decryption, the system provides a comprehensive solution for safeguarding sensitive information during transmission and storage.

The dual-mode implementation of AES, incorporating both Cipher Block Chaining (CBC) and Electronic Codebook (ECB), highlights the flexibility and adaptability of the system. While CBC mode enhances security by introducing randomness to prevent recognizable patterns in the ciphertext, ECB mode offers speed and simplicity for non-critical applications. This capability makes the system suitable for a wide range of scenarios, from secure file-sharing platforms to cloud storage services.

The project also emphasized key cryptographic principles, including secure key exchange protocols, the use of initialization vectors, and block cipher padding mechanisms. By implementing these features, the system ensures that files remain protected against common cryptographic attacks, such as replay attacks and plaintext-pattern analysis.

Furthermore, the project serves as an educational framework for understanding the strengths and trade-offs associated with hybrid cryptography. It provides valuable insights into how cryptographic algorithms can be combined to achieve security objectives, balancing efficiency, and protection. This system demonstrates the practical relevance of cryptography in modern applications, addressing real-world concerns such as secure communication and data privacy.

In conclusion, the successful development and implementation of this system demonstrate the effectiveness of hybrid cryptography in addressing critical security needs. By combining theoretical knowledge with practical application, the project not only underscores the importance of secure file encryption but also lays the groundwork for further advancements in cryptographic systems.

B. Future Work

Potential improvements to the system include:

- Adding support for additional AES modes, such as GCM (Galois/Counter Mode).
- Implementing file integrity verification using HMAC.
- Extending support to large file sizes using streaming encryption.
- Creating a graphical user interface (GUI) for user-friendly interaction.

REFERENCES

[1] *Advanced Encryption Standard*, TutorialsPoint. Available: https://www.tutorialspoint.com/cryptography/advanced_encryption_standard.htm.

[2] B. K. Jena, “AES Encryption: Secure Data with Advanced Encryption Standard,” Simplilearn, Jul. 16, 2024. Available: https://www.simplilearn.com/tutorials/cryptography-tutorial/aes-encryption.

[3] V. Shukla, “AES Encryption in Rust,” *Backend Engineer*, Aug. 15, 2024. Available: https://backendengineer.io/aes-encryption-rust.

[4] Thanatos, “Final Demo [Video],” YouTube, Dec. 9, 2024. Available: https://www.youtube.com/watch?v=kEI_3Fc0BYo.