March 26,2025

Secure Chat Application

Cryptography

Karthikeya Mittapalli – 22CSB0C14

Secure Chat Application

Karthikeya Mittapalli – 22CSB0C14

1. **Introduction:**

In the era of increasing cyber threats, ensuring secure and private communication has become a top priority. Traditional encryption methods, such as RSA and ECC, while secure against classical attacks, are at risk due to advancements in quantum computing. A sufficiently powerful quantum computer could break these cryptographic schemes, leading to decryption of past and future encrypted messages. To ensure long-term security, messaging applications must adopt post-quantum cryptographic techniques that can withstand quantum attacks.

This project implements a Secure Chat Application with End-to-End Encryption (E2EE), combining Elliptic Curve Cryptography (ECC) and Kyber Key Encapsulation Mechanism (Kyber KEM) to provide a hybrid post-quantum key exchange. The system ensures that even in the presence of future quantum computers, past communications remain secure. Additionally, the application enforces message integrity, authentication, and confidentiality using AES-256-GCM encryption, HMAC-SHA256, and ECDSA digital signatures.

Key Issues Addressed:

- **Vulnerability to Quantum Attacks:** ECC and RSA encryption alone cannot provide security against quantum adversaries. This project integrates Kyber KEM to ensure quantum-resistant key exchange.
- **Secure Key Exchange with Forward Secrecy**: Uses ECDH and Kyber512 to derive a shared AES-256 key, ensuring that past messages remain protected even if long-term keys are compromised.
- **Man-in-the-Middle (MITM) Attacks:** Prevents unauthorized interception by ensuring that key exchange and message transmission remain encrypted.
- **Message Integrity and Authentication:** Uses HMAC-SHA256 for integrity verification and ECDSA digital signatures to verify sender authenticity, preventing impersonation and forgery.
- **Real-time Encrypted Communication:** Uses TCP sockets for message exchange while maintaining end-to-end encryption (E2EE).

Importance of the Problem

Most existing secure messaging applications, including WhatsApp and Signal, rely on elliptic curve cryptography (ECC) for key exchange. However, ECC alone is not resistant to quantum attacks. If a sufficiently powerful quantum computer is developed, encrypted conversations could be decrypted, leading to potential security breaches and privacy violations.

Existing Solutions

- Signal Protocol (Used by WhatsApp, Signal): Implements ECDH for key exchange and AES-256 for encryption. While secure today, it lacks resistance against quantum computing threats.
- PGP Encryption: Provides end-to-end encryption but is computationally expensive and not optimized for real-time chat applications.
- Hybrid Cryptography Approaches: Some recent cryptographic frameworks have explored hybrid approaches combining classical and post-quantum cryptography to ensure long-term security.

Key Contributions of This Project

- Hybrid Key Exchange: Combines Elliptic Curve Diffie-Hellman (ECDH) and Kyber (Post-Quantum KEM) to establish a shared secret resistant to both classical and quantum attacks.
- End-to-End Encryption: Uses AES-256-GCM for encrypting messages, ensuring confidentiality.
- Authentication and Integrity: Implements ECDSA for digital signatures and HMAC-SHA256 for message integrity verification.
- Secure Communication Channel: Employs TCP sockets with SSL/TLS to securely transmit messages between clients and servers.

2. Objectives

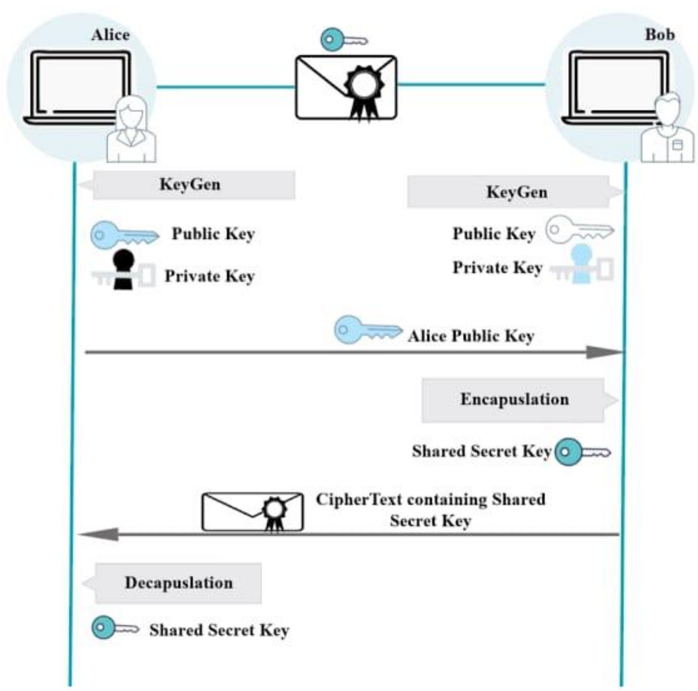
- Implement a hybrid key exchange mechanism using ECDH and Kyber512 to ensure quantum-resistant secure communication.
- Ensure message confidentiality, integrity, and authentication using AES-256-GCM, HMAC-SHA256, and ECDSA.
- Establish a secure and reliable communication channel using TCP sockets with SSL/TLS to prevent eavesdropping and MITM attacks.
- Provide forward secrecy to protect past communications even if long-term keys are compromised.

3. Implementation and Result Analysis

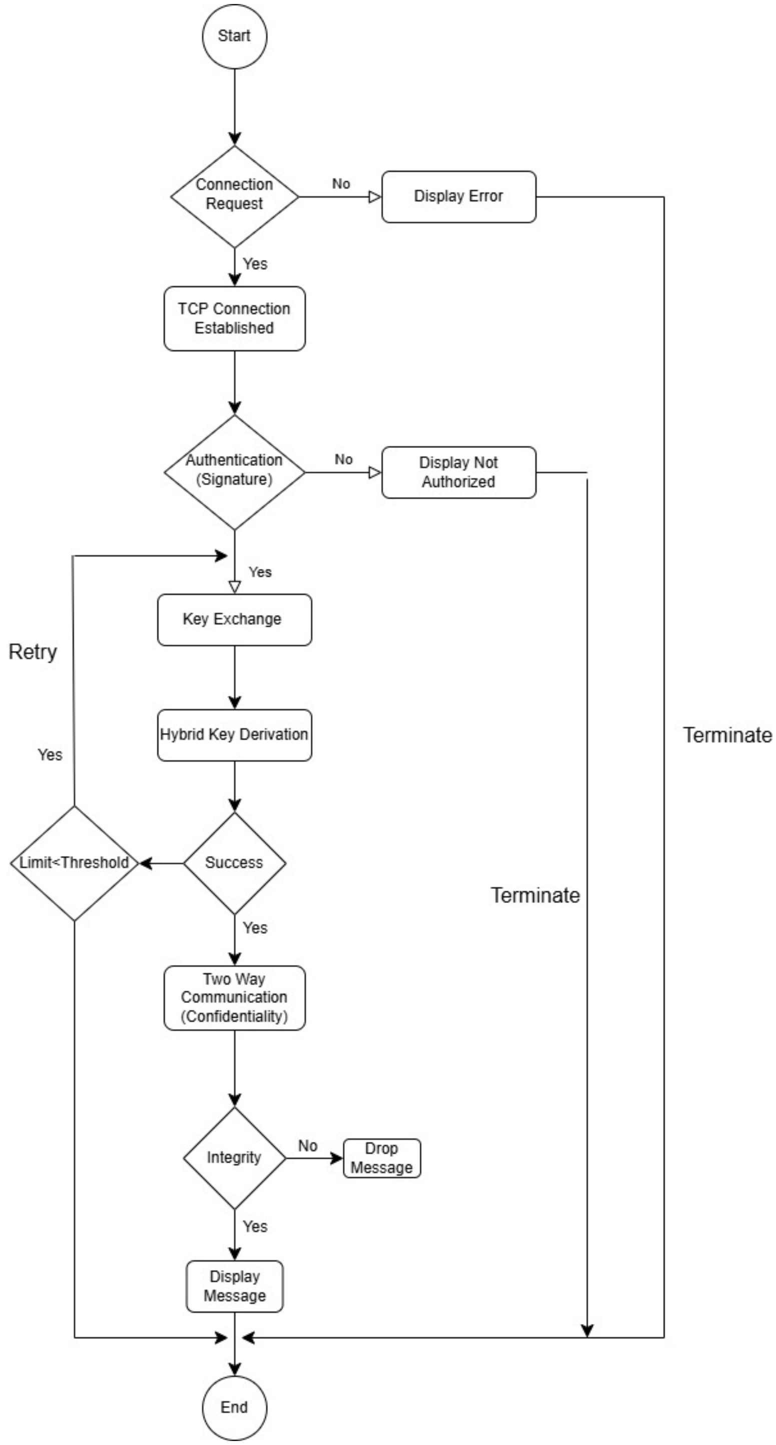
System Architecture (Modules)

Module	Purpose
Connection Management	Handles client connections using TCP sockets. Manages authentication and session security.
Key Exchange & Hybrid Key Derivation	Implements hybrid key exchange using **ECDH + Kyber512**, ensuring quantum resistance and forward secrecy. Derives AES-256 encryption keys using HKDF.
Message Encryption & Decryption	Encrypts messages before sending and decrypts incoming messages using **AES-256-GCM** for confidentiality and HMAC-SHA256 for integrity verification.
Digital Signature & Authentication	Uses **ECDSA** to sign authentication nonces for verifying client identity. Ensures secure client authentication and prevents forgery.

Flow Diagram



Project Work Flow Chart



Algorithm:

Key Encapsulation

KEM.KeyGen()

$z \leftarrow B^{32}$

$(pk, s) = \text{PKE.KeyGen}()$

$sk = (s||pk||H(pk)||z)$

return (pk, sk)

$\xrightarrow{pk}$

KEM.Enc(pk)

$m_0 \leftarrow B^{32}$

$m = H(m_0)$

$(\bar{K}, r) = G(m||H(pk))$

$c = \text{PKE.Enc}(pk, m; r)$

$K = \text{KDF}(\bar{K}, H(c))$

return (c, K)

$\xleftarrow{c}$

KEM.Dec(c, $sk = (s||pk||H(pk)||z)$)

$m' = \text{PKE.Dec}(c, s)$

$(\bar{K}', r') = G(m'||H(pk))$

$c' = \text{PKE.Enc}(pk, m'; r')$

if $c = c'$

then return $K = \text{KDF}(\bar{K}', H(c))$

else return $K = \text{KDF}(z, H(c))$

AES-GCM Encryption

NIST Special Publication 800-38D

Algorithm 4: GCM-AE$_K$ (IV, P, A)

Prerequisites:
approved block cipher CIPH with a 128-bit block size;
key K;
definitions of supported input-output lengths;
supported tag length t associated with the key.

Input:
initialization vector IV (whose length is supported);
plaintext P (whose length is supported);
additional authenticated data A (whose length is supported).

Output:
ciphertext C;
authentication tag T.

Steps:
1. Let $H = \text{CIPH}_K(0^{128})$.
2. Define a block, J_0, as follows:
   If len(IV)=96, then let $J_0 = IV \| 0^{31} \| 1$.
   If len(IV) $\neq$ 96, then let $s = 128 \lceil \text{len}(IV)/128 \rceil - \text{len}(IV)$, and let
   $J_0 = \text{GHASH}_H(IV \| 0^{s+64} \| [\text{len}(IV)]_{64})$.
3. Let $C = \text{GCTR}_K(\text{inc}_{32}(J_0), P)$.
4. Let $u = 128 \cdot \lceil \text{len}(C)/128 \rceil - \text{len}(C)$ and let $v = 128 \cdot \lceil \text{len}(A)/128 \rceil - \text{len}(A)$.
5. Define a block, S, as follows:
   $S = \text{GHASH}_H(A \| 0^v \| C \| 0^u \| [\text{len}(A)]_{64} \| [\text{len}(C)]_{64})$.
6. Let $T = \text{MSB}_t(\text{GCTR}_K(J_0, S))$.
7. Return (C, T).

AES-GCM Decryption

Algorithm 5: GCM-ADK(IV,C,A,T)

Prerequisites:
approved block cipher CIPH with a 128-bit block size;
key K;
definitions of supported input-output lengths;
supported tag length t associated with the key.

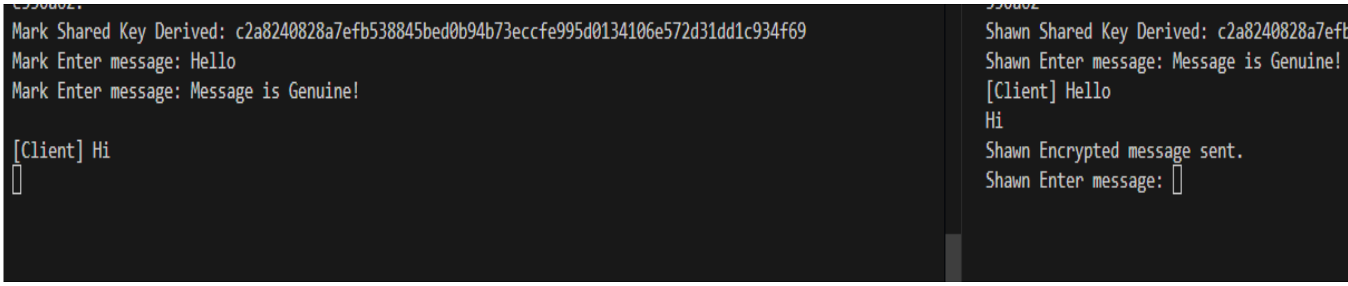
Input:
initialization vector IV;
ciphertext C;
additional authenticated data A;
authentication tag T.

Output:
plaintext P or indication of inauthenticity *FAIL*.

Steps:
1. If the bit lengths of IV, A or C are not supported, or if len(T) $\neq t$, then return FAIL.
2. Let $H = \text{CIPH}_K(0^{128})$.
3. Define a block, J_0, as follows:
   If len(IV)=96, then $J_0 = IV \| 0^{31} \| 1$.
   If len(IV) $\neq$ 96, then let $s = 128 \lceil \text{len}(IV)/128 \rceil - \text{len}(IV)$, and
   $J_0 = \text{GHASH}_H(IV \| 0^{s+64} \| [\text{len}(IV)]_{64})$.
4. Let $P = \text{GCTR}_K(\text{inc}_{32}(J_0), C)$.
5. Let $u = 128 \cdot \lceil \text{len}(C)/128 \rceil - \text{len}(C)$ and let $v = 128 \cdot \lceil \text{len}(A)/128 \rceil - \text{len}(A)$.
6. Define a block, S, as follows:
   $S = \text{GHASH}_H(A \| 0^v \| C \| 0^u \| [\text{len}(A)]_{64} \| [\text{len}(C)]_{64})$
7. Let $T' = \text{MSB}_t(\text{GCTR}_K(J_0, S))$.
8. If $T = T'$, then return P; else return *FAIL*.

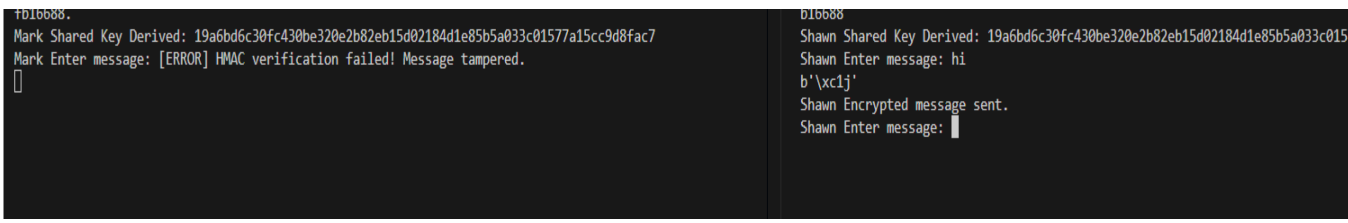
Result Analysis

Communication (Confidentiality)



Authentication

Integrity



- The Cryptographic Services are Properly Achieved
- Quantum Security: The hybrid key exchange (ECDH + Kyber) ensures resistance against quantum decryption.
- Low Latency Messaging: The use of TCP sockets ensures reliable real-time communication.
- Efficient Encryption & Decryption: AES-256-GCM provides fast encryption while maintaining data integrity.

4. **Conclusion**

   This project successfully implements a secure, end-to-end encrypted chat application that is resistant to both classical and quantum attacks. By leveraging hybrid cryptography (ECDH + Kyber), AES-256-GCM, and ECDSA, the application ensures confidentiality, integrity, and authentication of messages. This approach significantly enhances security while maintaining efficient performance, making it a robust solution for future-proof messaging applications.

5. **Learning Outcomes**
   - Understood Elliptic Curve Cryptography (ECC) and its role in secure key exchange.
   - Implemented post-quantum cryptography (Kyber-512 KEM) for hybrid key exchange.
   - Developed a secure messaging system using AES-256-GCM for encryption.
   - Learned message authentication and integrity verification techniques (ECDSA, HMAC-SHA256).
   - Designed a TCP socket-based secure communication framework.

   Future Improvements

   - Key Session Management (Security Enhancement)
   - Secure Private Key Storage
   - Encrypted Chat History Storage (Feature Improvement)

6. **Source Code**
   1. **Shared Secret Derivation Module**

```python
    def hybrid_key_exchange(self, peer_ecdh_public_bytes,server_kyber_pub):
        peer_ecc_public_key = self.deserialize_public_key(peer_ecdh_public_bytes)
        # ECDH key agreement
        ecdh_shared_secret = self.derive_ecdh_shared_secret(peer_ecc_public_key)
        # Kyber key encapsulation
        ciphertext, kyber_shared_secret = self.encapsulate_kyber_secret(server_kyber_pub)
        # Combine secrets using HKDF
        hybrid_shared_secret = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b"AES-GCM Secure Key",
            backend=default_backend()
        ).derive(kyber_shared_secret + ecdh_shared_secret)

        key_material = HKDF(
            algorithm=hashes.SHA256(),
            length=48,  # 32 bytes for AES-256, 16 bytes for HMAC-SHA256 key
            salt=None,
            info=b"AES-HMAC Key Separation",
        ).derive(hybrid_shared_secret)

        aes_shared_key = key_material[:32] # First 32 bytes for AES-256
        hmac_shared_key = key_material[32:] # Last 16 bytes for HMAC

        return aes_shared_key,hmac_shared_key, ciphertext
```

2. Hybrid Key Exchange & Secure Key Encapsulation

```python
# Step: Secure Hybrid Key Exchange & Shared Key Derivation
server_ecdh_pub = server_key_exchange.get_ecdh_public_bytes()
server_kyber_pub, server_kyber_secret = server_key_exchange.kyber.keygen()

length = struct.unpack(">I", client_socket.recv(4))[0]
client_ecdh_pub = recv_exact(client_socket, length)
print(f"{server_id} Received Client's ECDH Public Key: {client_ecdh_pub.hex()}")

client_socket.sendall(struct.pack(">I", len(server_ecdh_pub)) + server_ecdh_pub)
client_socket.sendall(struct.pack(">I", len(server_kyber_pub)) + server_kyber_pub)
print(f"{server_id} Sent ECC and Kyber Public Keys.")

length = struct.unpack(">I", client_socket.recv(4))[0]
kyber_ciphertext = recv_exact(client_socket, length)
print(f"{server_id} Received Kyber Ciphertext (length={len(kyber_ciphertext)}).")

aes_shared_key,hmac_shared_key = server_key_exchange.hybrid_key_decapsulation(kyber_ciphertext, server_kyber_secret, client_ecdh_pub)
print(f"{server_id} Shared Key Derived: {aes_shared_key.hex()}")

encryption = MessageEncryption(aes_shared_key)

# Start Secure Communication Threads
threading.Thread(target=receive_messages, args=(client_socket, encryption,hmac_shared_key)).start()
threading.Thread(target=send_messages, args=(client_socket, encryption,hmac_shared_key)).start()
```

3. Message Encryption & Decryption Module

```python
class MessageEncryption:
    """Handles AES-256-GCM encryption and decryption of messages."""

    def __init__(self, shared_key):
        self.shared_key = shared_key

    def encrypt(self, plaintext):
        """Encrypts a message using AES-256-GCM."""
        iv = os.urandom(12)
        cipher = Cipher(algorithms.AES(self.shared_key), modes.GCM(iv))
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(plaintext) + encryptor.finalize()
        return iv, encryptor.tag, ciphertext

    def decrypt(self, iv, tag, ciphertext):
        """Decrypts a message using AES-256-GCM."""
        cipher = Cipher(algorithms.AES(self.shared_key), modes.GCM(iv, tag))
        decryptor = cipher.decryptor()
        return decryptor.update(ciphertext) + decryptor.finalize()
```

4. Digital Signature Module

```python
class DigitalSignature:
    """Handles digital signature generation and verification using ECDSA."""

    def sign_message(self, message, private_key):
        """Signs a message using the provided ECDSA private key."""
        return private_key.sign(message, ec.ECDSA(hashes.SHA256()))

    def verify_signature(self, message, signature, public_key):
        """Verifies an ECDSA signature using the provided public key."""
        try:
            public_key.verify(signature, message, ec.ECDSA(hashes.SHA256()))
            return True
        except:
            return False

    @staticmethod
    def load_private_key(filename):
        """Loads a private key from a PEM file."""
        with open(filename, "rb") as key_file:
            return serialization.load_pem_private_key(
                key_file.read(),
                password=None  # Assuming no password encryption
            )

    @staticmethod
    def load_public_key(filename):
        """Loads a public key from a PEM file."""
        with open(filename, "rb") as key_file:
            return serialization.load_pem_public_key(key_file.read())
```

5. Authentication Module

```python
def verify_connection(client_socket):
    # Client Authentication
    # Step: Receive Client ID
    length_data = client_socket.recv(4)  # Receive 4-byte length
    if not length_data:
        print(f"{server_id} Error: No data received for client ID length.")
        client_socket.close()
        return

    length = struct.unpack(">I", length_data)[0]
    client_id = recv_exact(client_socket, length).decode()
    print(f"{server_id} Received Client ID: {client_id} (Length: {length})")

    # Step: Retrieve Stored Client ECC Public Key
    try:
        client_public_key = load_public_key(client_id)
    except FileNotFoundError:
        print(f"{server_id}] No registered public key found for client ID: {client_id}. Rejecting connection.")
        client_socket.sendall(struct.pack(">I", 0))  # Send rejection signal
        client_socket.close()
        return

    # Step: Receive Nonce & Signature from Client
    length = struct.unpack(">I", client_socket.recv(4))[0]
    data = recv_exact(client_socket, length)

    nonce_length = 32  # We know nonce is always 32 bytes
    nonce = data[:nonce_length]
    client_signature = data[nonce_length:]  # Remaining bytes are the signature

    # Step: Verify Client's Signature on Nonce
    if not server_signature.verify_signature(nonce, client_signature, client_public_key):
        print(f"{server_id} Client authentication failed.")
        client_socket.sendall(struct.pack(">I",0))
        client_socket.close()
        return False
    else:
        client_socket.sendall(struct.pack(">I",4))
        print(f"{server_id} Client authentication successful.")

```

7. References

1. Kyber KEM: Post-Quantum Cryptography: https://medium.com/@hwupathum/crystals-kyber-the-key-to-post-quantum-encryption-3154b305e7bd
2. NIST PQC Standardization Process: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography
3. AES256-GCM: https://medium.com/@pierrephilip/aes256-gcm-key-rotation-in-c-2be80c03cac2#:~:text=It's%20a%20symmetric%20key%20algorithm,to%20a%2032%20byte%20requirement.