Communication & Security Project:

SecureChat-X Documentation

Contents

1. Introduction

The objective of this project is to design and implement a secure chat application that establishes a mutually agreed session key (K_{abc}) among three entities (A, B, and C) using an indirect communication model via a central server (S). The server acts solely as a relay and is never in possession of the session key. Every participant is issued a public key certificate, and all communications are secured by a combination of RSA and AES cryptography.

This document details the complete protocol design, system architecture, cryptographic mechanisms, and implementation details of the application. The content is structured so that even readers without deep prior knowledge of the subject can follow the design decisions and understand the secure communication process. Additional sections discuss security analysis, challenges encountered, and future improvements.

2. System Architecture and Communication Flow

The system consists of four main components:

- **Entities A, B, and C:** These clients perform key establishment and exchange encrypted messages using the shared session key.
- **Chat Server (S):** Acts as a relay that forwards messages between clients without decrypting them.
- **Certificates/Keys:** Each entity (including S) has its own RSA key pair and digital certificate. These ensure authentication and integrity.
- **Certificate Manager and Cryptographic Functions:** Dedicated modules handle creation, storage, verification of certificates, and perform AES/RSA encryption and decryption.

All messages exchanged—including connection initiation, certificate requests/responses, nonce challenge/response exchanges, and encrypted chat messages—are transmitted through Server S. The server's role is strictly as a message forwarder, ensuring end-to-end confidentiality between the communicating entities.

2.1. Communication Model

The application leverages TCP sockets for reliable message delivery. Entities never communicate directly; every message is sent to the server which then relays it to the intended recipients. This architecture simplifies the network topology and allows the server to manage client connections and certificate storage.

3. Protocol Overview

The secure chat protocol comprises several phases:

3.1. Connection Phase

Each client initiates a TCP connection to Server S and immediately sends a `MsgConnection` message containing its certificate. This allows the server to record the client's identity and public key for subsequent operations.

3.2. Certificate Exchange

An initiating client (e.g., Entity A) sends a `MsgCertRequest` to S to obtain the certificates of the other entities (B and C). The server responds with a `MsgCertResponse` that contains the public key certificates needed to encrypt nonce challenges. This exchange is crucial as it ensures that each party can verify the authenticity of the messages from the other parties.

3.3. Nonce-Based Key Establishment

Entity A initiates a challenge-response protocol by generating a random nonce (challenge) for each target entity. These nonces are encrypted using the respective target's public key and sent within a `MsgNonceRequest` message via the server. Each target entity (B and C) decrypts the challenge, re-encrypts it with A's public key to prove correct decryption, and also encrypts its own randomly generated nonce in a `MsgNonceResponse`. Once Entity A collects all nonces (including its own), it combines them—typically using bitwise operations followed by a SHA-256 hash—to derive the shared session key, K_{abc}.

3.4. Secure Chat Messaging

With K_{abc} established, all subsequent chat messages are encrypted using AES in CBC mode. The encrypted messages (denoted as $\{M_x\}_{K_{abc}}$) are then transmitted via Server S, which simply relays the messages without decryption. Each client decrypts incoming messages using the shared session key.

3.5. Expanded Explanation for Diagrams

Several diagrams further illustrate the protocol phases:

3.5.1 High-Level Protocol Diagram

Figure 1 provides a bird's-eye view of the protocol. It clearly delineates the different phases—from connection to key establishment and secure messaging. The diagram emphasizes that the server acts only as a relay and does not have access to the session key, thereby ensuring that end-to-end encryption is maintained.

3.5.2 Detailed Sequence Diagram for Key Establishment

Figure 2 illustrates the precise sequence of messages exchanged during key establishment. This detailed diagram shows:

- The certificate request and response between A and S.
- The nonce challenge generation, encryption with RSA, and subsequent forwarding by the server.
- The nonce response where each target entity confirms reception by re-encrypting the challenge and providing its own nonce.

These steps guarantee that each entity's contribution is included in the final session key, ensuring mutual participation and security.

4. Message Structure and Integrity Verification

Every message in the protocol adheres to a strict structure to ensure consistency and security. The core components are:

- **Certificate:** This contains the sender's identity and public key. The certificate is essential for verifying the digital signature attached to each message.
- **Message Type:** An enumerated field that classifies the message (e.g., connection, certificate request, encrypted text, nonce request/response).
- **Body:** A dictionary structure that holds message-specific data. The contents vary by message type.
- **Digital Signature:** Generated using RSA with SHA-256, the signature covers the certificate and the serialized body. This mechanism ensures the integrity of the message and authenticates the sender.

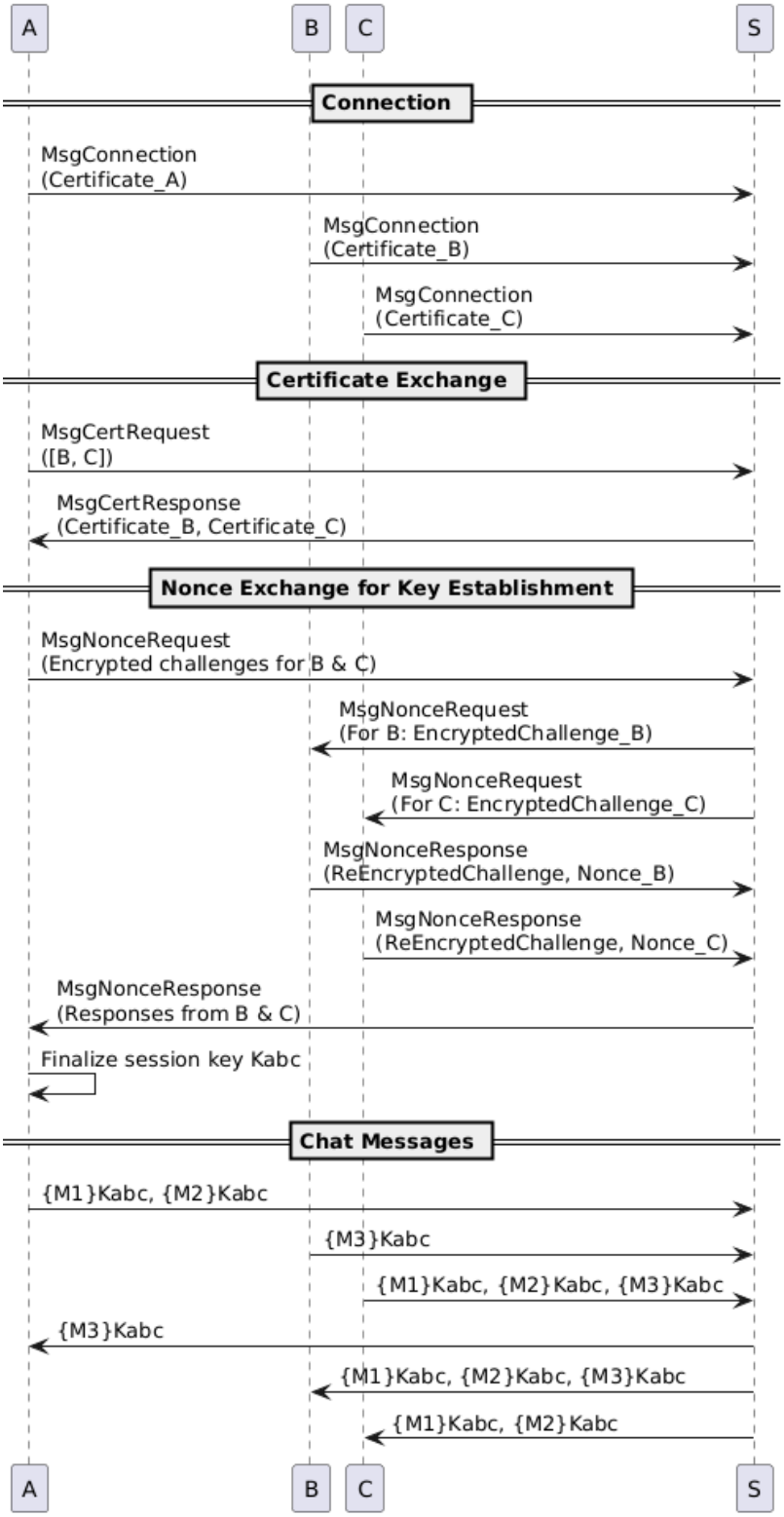

Figure 1: High-Level Protocol Overview Diagram

4.1. Expanded Explanation of Digital Signatures

Digital signatures are created using the sender's private key. The process involves:

1. Serializing the certificate and the message body into a consistent byte format.

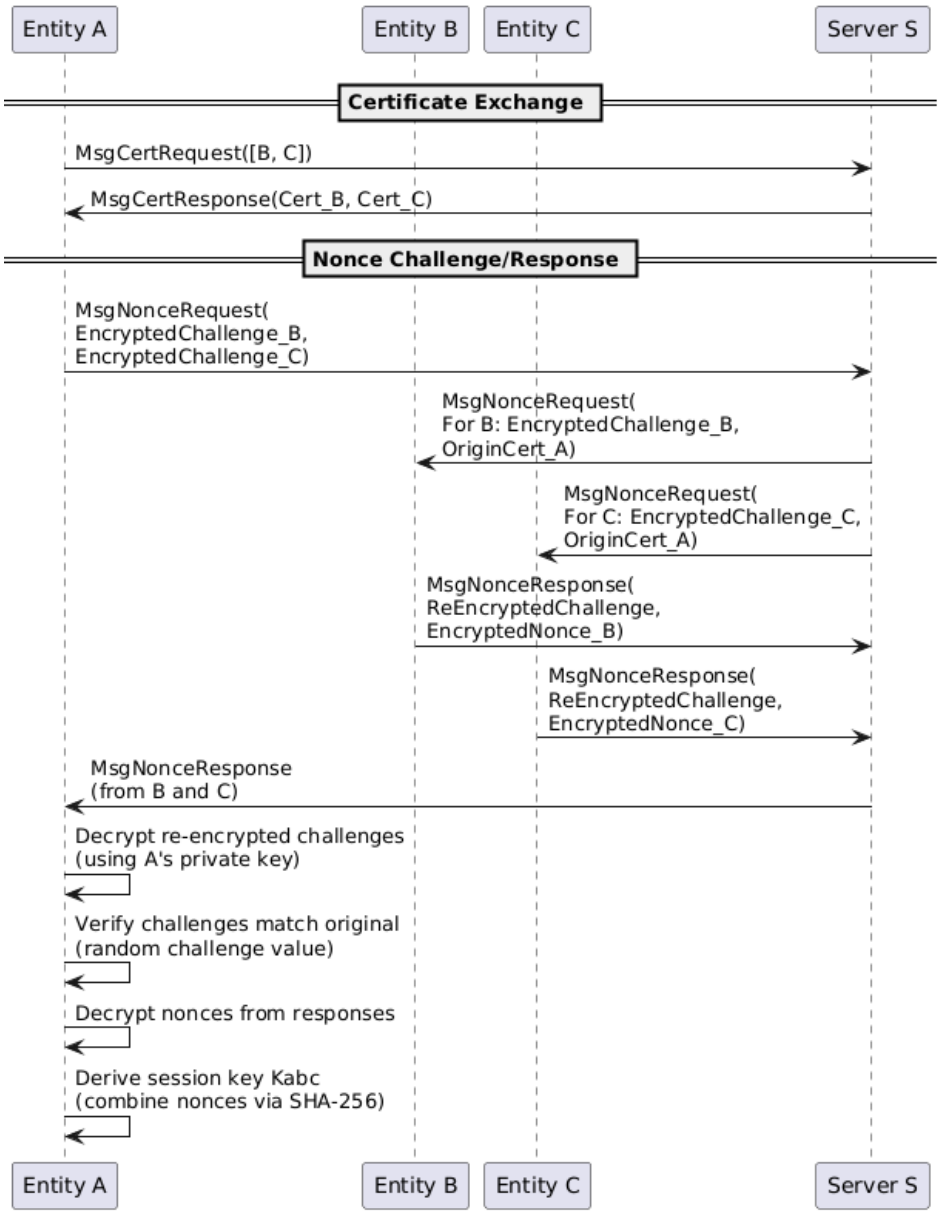

Figure 2: Detailed Sequence Diagram for Key Establishment

2. Concatenating these bytes and generating a hash using SHA-256.

3. Encrypting the hash with the sender's private key.

Upon reception, the receiver uses the sender's public key (obtained from the certificate) to decrypt the signature and compare the resulting hash with its own computation from the message data. If they match, the message is accepted as authentic and unmodified.

4.2. Message Hierarchy Diagram

Figure 3 shows a class diagram of the message types used in the system. This diagram clearly depicts how each specific message (e.g., `MsgConnection`, `MsgEncryptedText`, `MsgCertRequest`, etc.) is derived from a common base class, ensuring that every message carries a certificate, type, body, and signature.

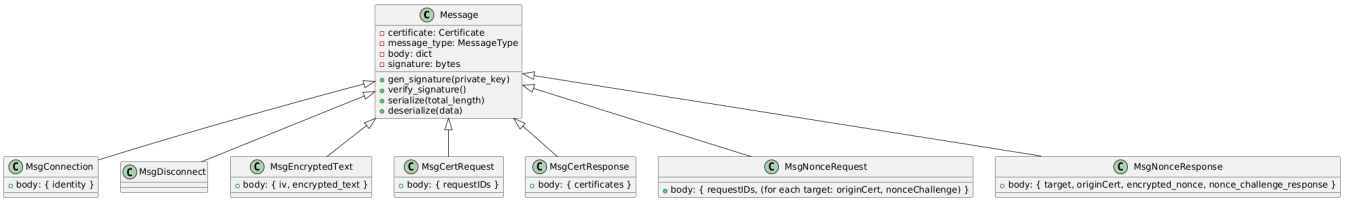

Figure 3: Message Structure Diagram

4.3. Authentication and Integrity Verification Flow

Figure 4 summarizes the process:

1. **Signature Generation:** At the sender's side, the signature is generated over the certificate and message body.
2. **Transmission:** The signed message is serialized and sent over the network.
3. **Verification:** Upon receipt, the receiver verifies the signature using the sender's public key.

This process is vital to prevent tampering and spoofing in the communication channel.

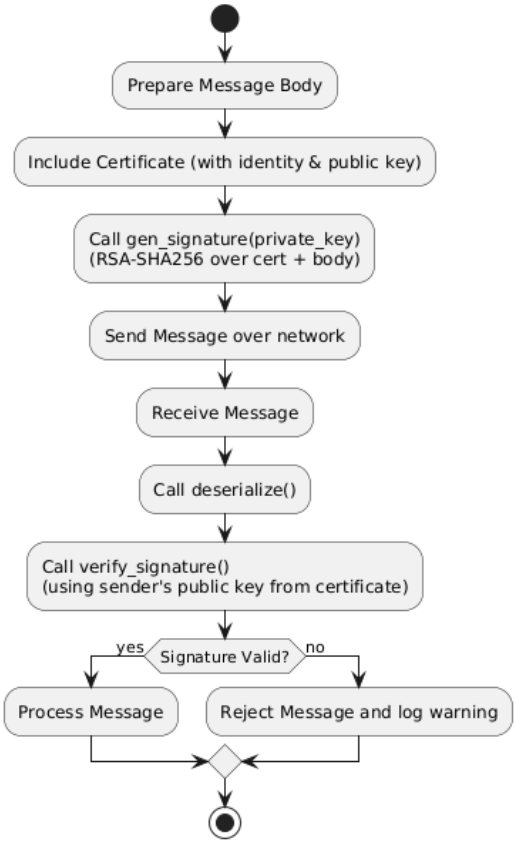

Figure 4: Authentication and Integrity Verification Diagram

5. Implementation Details

The secure chat application is implemented in Python and organized into multiple modules to ensure modularity and clarity.

5.1. Directory Structure

The project's directory structure is as follows:

```
SecureChatApp
├── Certs
│   ├── Certificate_A.cer
│   ├── Certificate_B.cer
│   ├── Certificate_C.cer
│   └── Certificate_S.cer
├── ChatClient.py
├── ChatServer.py
├── Core_Functions
│   ├── CertificateManager.py
│   ├── EncryptionManager.py
│   ├── Entity.py
│   └── MessageProtocol.py
├── GenerateCertificates.py
└── Keys
    ├── PrivateKey_A.pem
    ├── PrivateKey_B.pem
    ├── PrivateKey_C.pem
    └── PrivateKey_S.pem
```

5.2. Key Establishment and Nonce Exchange

The session key is derived using a nonce-based challenge-response mechanism. The key steps include:

- Requesting certificates for the target entities.
- Generating a random nonce and encrypting it with the public keys of B and C.
- Sending a `MsgNonceRequest` and receiving corresponding `MsgNonceResponse` messages.
- Combining all nonces (using bitwise OR and hashing with SHA-256) to create the final session key.

A code snippet from `ChatClient.py` is provided below to illustrate these functions:

```python
def obtain_session_key(self):
    cert_req_msg = MsgCertRequest(self.entities)
    cert_req_msg.certificate = self.certificate
    cert_req_msg.gen_signature(self.private_key)
    self.client_socket.send(cert_req_msg.serialize(MESSAGE_BUFFER_SIZE))

def _send_nonce_request(self, targets: list):
    self.nonce_challenge = randint(1000000, 9999999)
    challenge_bytes = self.nonce_challenge.to_bytes(4, 'little')
    challenge_map = {}
    for target in targets:
        pub_key = self.public_keys[target]
        challenge_map[target] = encrypt(challenge_bytes, pub_key)
    nr_msg = MsgNonceRequest(targets, self.certificate, challenge_map)
```

```
    nr_msg.certificate = self.certificate
    nr_msg.gen_signature(self.private_key)
    self.client_socket.send(nr_msg.serialize(MESSAGE_BUFFER_SIZE))

def _finalize_session_key(self):
    combined_value = 0
    for ent, nonce in self.nonces.items():
        combined_value |= int.from_bytes(nonce, 'big')
    hashed = sha256(combined_value.to_bytes(16, 'big')).digest()
    self.session_key = hashed[:16]
```

Listing 1: Nonce Request and Session Key Derivation in ChatClient

5.3. Server Message Forwarding

The server's role is to forward messages without decrypting them. The following code snippet from `ChatServer.py` demonstrates this functionality:

```
def _handle_encrypted_text(self, message):
    sender_id = message.certificate.identity
    for cid, fileno in self.connection_ids.items():
        if cid != sender_id:
            conn = self.connections[fileno]
            conn.send(message.serialize())
```

Listing 2: Message Forwarding in ChatServer

5.4. Certificate and Key Generation

The `GenerateCertificates.py` script is responsible for creating RSA key pairs and certificates for all entities. This script uses the RSA library to generate keys and then creates a certificate by signing the public key along with the entity identity. An excerpt is shown below:

```
cert_s = Certificate(identity='S')
pub_key_s, priv_key_s = newkeys(2048)
cert_s.public_key = pub_key_s
cert_s.gen_signature(priv_key_s)
with open("Keys/PrivateKey_S.pem", "wb") as f:
    f.write(priv_key_s.save_pkcs1('PEM'))
cert_s.save("Certs/Certificate_S.cer")
```

Listing 3: Certificate Generation in GenerateCertificates.py

5.5. Additional Implementation Considerations

The code is designed with modularity in mind. Each module focuses on a specific task:

- **Entity Module:** Provides a common framework for both clients and the server.
- **Certificate Manager:** Handles loading, saving, and signing of certificates.
- **Encryption Manager:** Manages all cryptographic operations including AES encryption, RSA encryption, and digital envelope creation.
- **Message Protocol:** Defines the structure and types of messages exchanged within the system.

This clear separation of concerns not only improves maintainability but also allows for easy future enhancements.

6. Diagrams and Data Flow

This section elaborates on the diagrams used throughout the document and explains their significance in detail.

6.1. Data Flow Diagram (DFD)

Figure 5 illustrates the overall data flow within the application:

- **User Input:** A user types a plaintext message.
- **Encryption Process:** The plaintext is encrypted using AES with the shared session key K_{abc}.
- **Message Relay:** The encrypted message is sent to Server S, which then relays it to the other connected clients.
- **Decryption:** The receiving clients decrypt the message using K_{abc} and display it to the user.

This diagram helps illustrate how data moves securely from one client to another without exposing sensitive information.

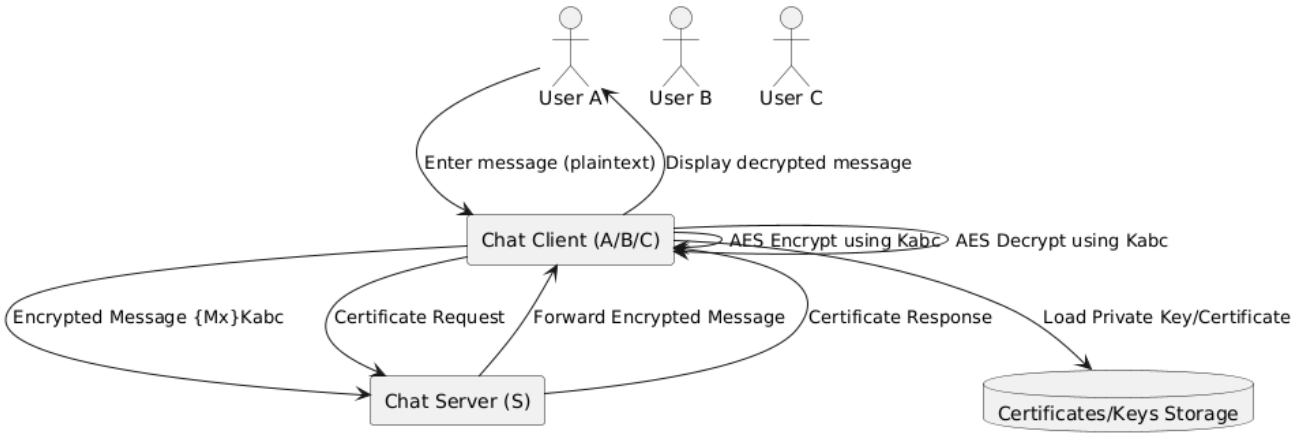

Figure 5: Data Flow Diagram (DFD)

6.2. Deployment Diagram

Figure 6 depicts the physical deployment of the system components:

- **Client Machines:** Separate machines or terminals running the chat client, each with locally stored keys and certificates.
- **Server Machine:** A dedicated machine running the chat server, responsible for message relay and certificate storage.
- **Network Connections:** Communication is established over TCP sockets ensuring reliable message delivery.

This diagram is critical in understanding how the application is deployed in a networked environment and how the responsibilities are divided.

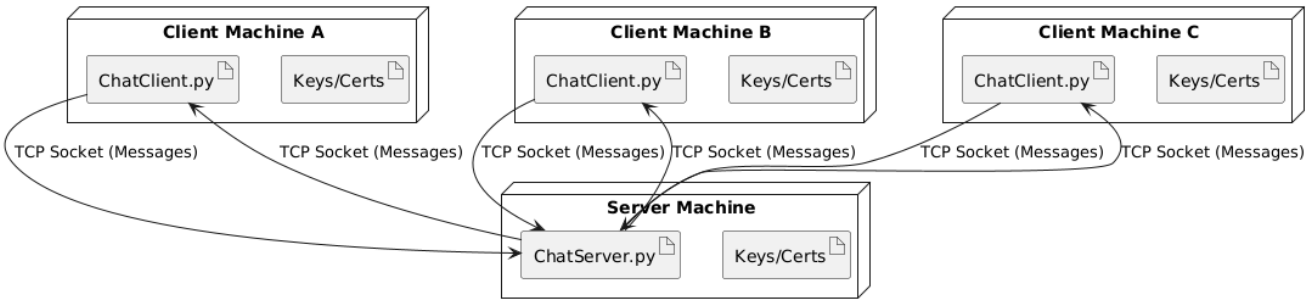

Figure 6: Deployment Diagram

6.3. State Diagram for ChatClient

Figure 7 shows the lifecycle of a ChatClient:

1. **Initialization:** The client loads its certificate and private key.
2. **Connecting:** Establishes a TCP connection with Server S and sends a connection message.
3. **Certificate Exchange:** Requests and receives certificates necessary for encrypting nonce challenges.
4. **Key Establishment:** Engages in the nonce-based challenge-response protocol to derive K_{abc}.
5. **Chatting:** Operates in active communication mode where messages are encrypted and decrypted using K_{abc}.
6. **Disconnecting:** Gracefully terminates the connection upon user request.

The state diagram clearly represents these transitions and shows how the client moves between different operational states.

6.4. Detailed Cryptographic Operations Diagram

Figure 8 focuses on the cryptographic processes:

- **AES Encryption/Decryption:** The plaintext is padded (PKCS#7) and encrypted using AES in CBC mode, ensuring confidentiality.
- **RSA Operations:** RSA is used for encrypting nonce challenges and wrapping the AES key into a digital envelope.
- **Digital Envelope:** This mechanism ensures that only the intended recipient, who possesses the correct private key, can decrypt the AES key.

The diagram provides a visual representation of how these cryptographic operations are chained together to secure data transmission.

6.5. UML Class Diagram

Figure 9 is the UML diagram for the main classes in the codebase:

- **Entity Class:** Serves as the abstract base for both ChatClient and ChatServer, handling private key loading and message parsing.
- **Certificate Class:** Manages the creation, signing, loading, and saving of certificates.

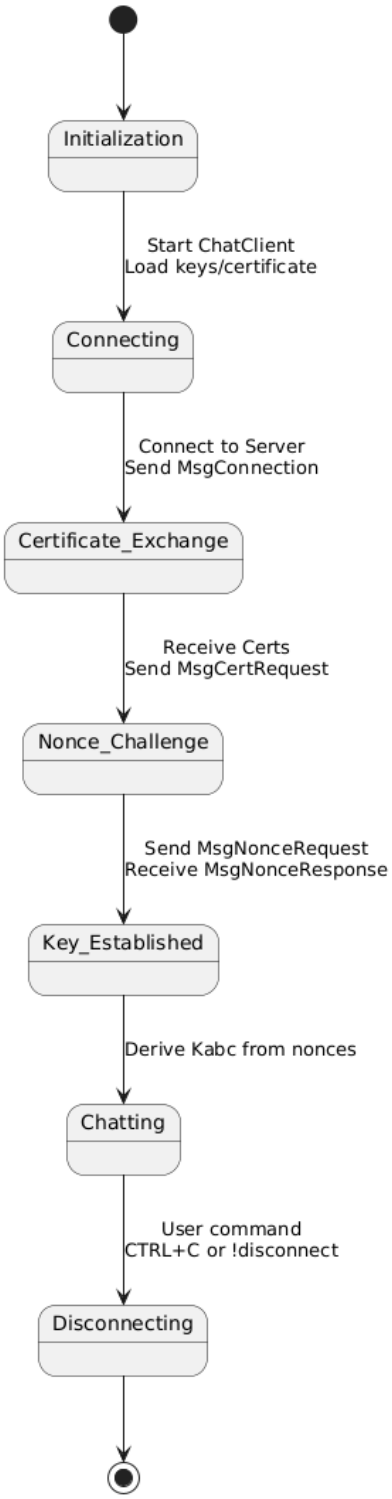

Figure 7: State Diagram for ChatClient

- **EncryptionManager:** Provides the functions for AES and RSA encryption/decryption.
- **Message Hierarchy:** Outlines how different message types inherit from a common Message class and incorporate digital signatures.

This diagram illustrates the relationships and dependencies between the core modules and supports un-

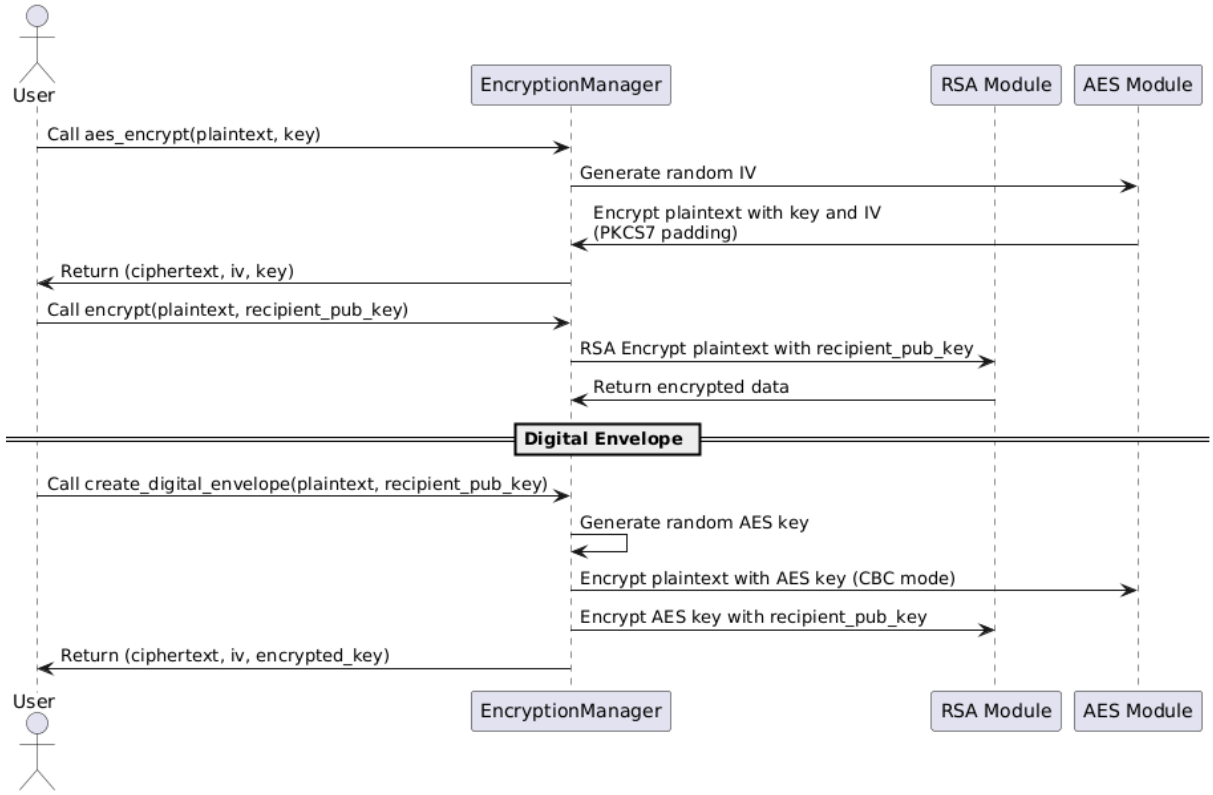

Figure 8: Detailed Cryptographic Operation Diagram

derstanding the overall software architecture.

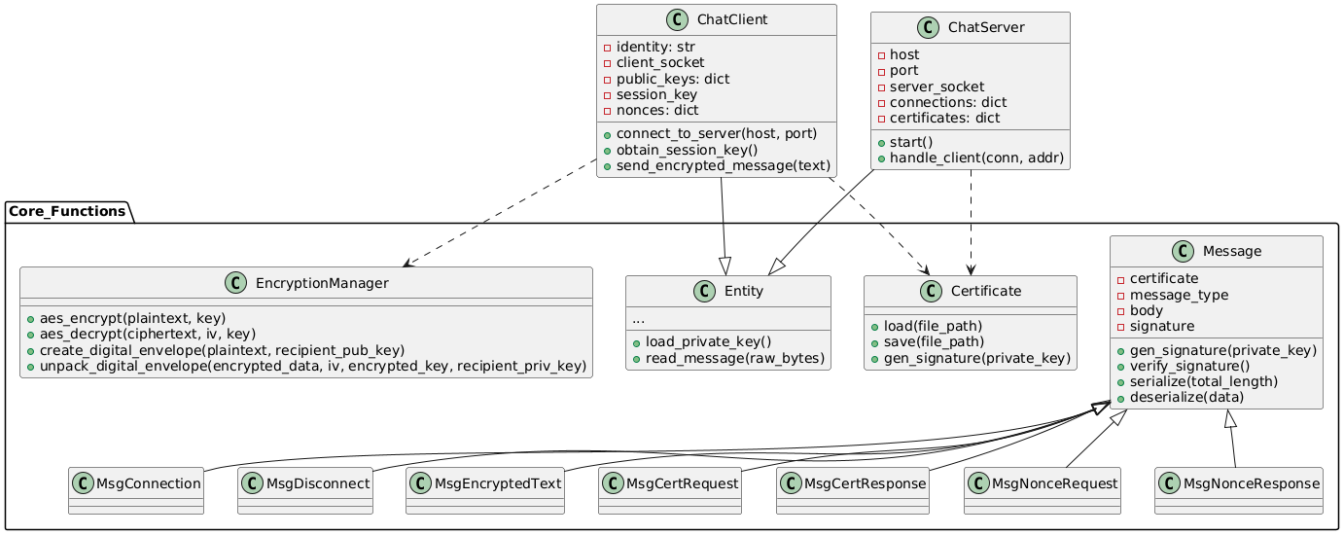

Figure 9: UML Class Diagram

7. Conclusion

This document has provided an in-depth explanation of the Secure Chat Application, covering every phase from initial connection and certificate exchange to nonce-based key establishment and secure message relaying. By employing RSA for key wrapping and AES for message encryption, the system guarantees confidentiality, integrity, and authentication in all communications.

The modular design and clear separation of functions facilitate easy maintenance and future enhancements. Overall, this project demonstrates a practical application of cryptographic principles in securing real-time communication over untrusted networks.

8. References

- Python `rsa` library documentation.
- `pycryptodomex` documentation for AES encryption.
- Python socket programming documentation.
- General literature on public key infrastructure (PKI) and digital signatures.
- General literature on nonce-based key establishment protocols.
- ChatGPT - for refactoring and boilerplate code.
- LaTeX - for document preparation.