Project Report: Secure Chat App with End-to-End Encryption (E2EE)

1. Introduction

The primary objective of this project was to design and implement a real-time communication application that guarantees **End-to-End Encryption (E2EE)** for all chat messages. E2EE is a cryptographic method ensuring that messages are encrypted on the sender's device and can only be decrypted by the intended recipient, thereby preventing the server or any third parties from reading the content.

The security model employs a **hybrid cryptographic approach**:

- **Asymmetric Encryption (RSA):** Used for the secure exchange of the smaller, faster symmetric session key.
- **Symmetric Encryption (AES/Fernet):** Used for the fast, bulk encryption/decryption of the actual chat content.

2. Abstract

The Secure Chat App is a full-stack, real-time web application built using **Python (Flask)** and **JavaScript**. The **Flask-SocketIO** framework manages the real-time communication channel. Crucially, all sensitive security operations—including key generation, message encryption, and decryption—are executed **client-side** in the browser. The server's role is strictly limited to user registration, public key storage, and relaying the encrypted cipher-text, which validates the core principle of E2EE.

3. Tools & Technologies

Category	Tool/Library	Purpose
Backend Framework	Python 3, Flask	Provides the web server structure and routing.
Real-time Comms	Flask-SocketIO	Manages bi-directional, persistent WebSocket connections for message relay.

Category	Tool/Library	Purpose
Server Crypto	cryptography (Python)	Used to generate and manage RSA public/private key objects on the server.
Client Crypto	JavaScript Forge	Handles all browser-side encryption (AES-CFB) and decryption using the session key.
Environment	Virtual Environment (venv), Git Bash (MINGW64)	Used to isolate project dependencies and execute commands.

4. Project Implementation Phases

Phase 1: Cryptography Utility (crypto_utils.py)

This phase built the foundation for cryptographic operations:

- **RSA Key Generation:** Implemented the generate_rsa_key_pair function to create and serialize 2048-bit RSA keys.
- **Symmetric Utilities:** Implemented functions for generating the fast AES session key (generate_aes_key) and for message encryption/decryption using Fernet/AES.

Phase 2: Server Architecture (app.py)

This phase established the server's non-cryptographic, management role:

- **User Registration:** Implemented the register event to generate and store each user's RSA **public key** in the USERS dictionary.
- **Secure Relay:** The encrypted_message handler was designed to transmit the raw, encrypted payload across the SocketIO room without any attempt at decryption, guaranteeing the security of the communication.

Phase 3: Client Implementation (templates/index.html)

This phase built the user interface and handled all security operations in the browser:

- **Client-Side Key Management:** JavaScript logic was written to generate and store the transient AES session key locally when a user joins a room.

- **E2EE Workflow:** The client performs two essential tasks: **(1) Encrypts** the message using the local AES key before transmission, and **(2) Decrypts** the received cipher-text using the local AES key before displaying the readable plain text.

5. Test Script: E2EE Communication Verification

The following test sequence confirms the successful implementation of E2EE:

Step	Action	Expected Output (Server Terminal)	Expected Output (Client Browser)	E2EE Result
1.	Alice $\to$ Join Room "Secure"	Alice joined room Secure	Alice sees: [You] has entered the Secure room.	Success
2.	Bob $\to$ Join Room "Secure"	Bob joined room Secure	Bob sees: [You] has entered the Secure room.	Success
3.	Alice $\to$ Send: "Test message"	Relayed encrypted message in room Secure from Alice	Bob's browser instantly displays: **[Alice] Test message**	**VERIFIED:** Server sees cipher-text, client decrypts plain text.
4.	Bob $\to$ Send: "Reply received"	Relayed encrypted message in room Secure from Bob	Alice's browser instantly displays: **[Bob] Reply received**	**VERIFIED:** Server sees cipher-text, client decrypts plain text.

6. Conclusion

The Secure Chat App successfully implements the required E2EE architecture. By localizing all encryption and decryption operations to the client's browser, the application prevents the server from accessing the plain text messages or the secret AES keys. This design confirms that the project fulfils the core security objective, resulting in a private, secure, and fully functional communication platform.