**AES-Secured and SHA-256-Authenticated Real-Time Messaging System**

A Minor Project report submitted

in partial fulfillment of requirement for the award of degree

## **BACHELOR OF TECHNOLOGY**

in

## **SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE**

By

**DEERAJ MONALA 2203A51496**

**PRIYA POOSAALA 2203A51512**

**SRIYA POOSAALA 2203A51513**

**MAHESH SOORA 2203A51159**

**PRANAY ALETTI 2203A51533**

Under the guidance of

Professor, School of CS&AI.



SR University, Ananthsagar,Warangal,Telagnana-506371



**CERTIFICATE**

This is to certify that this project entitled **“SECURE MESSAGING SYSTEM USING AES ENCRYPTION AND SHA-256 AUTHENTICATION**" is the bonafied work carried out by **MAHESH SOORA, PRIYA POOSAALA, SRIYA POOSAALA, DEERAJ MONALA and PRANAY ALETTI** as a Minor Project for the partial fulfillment to award the degree **BACHELOR OF TECHNOLOGY** in **School of Computer Science and Artificial Intelligence** during the academic year 2024-2025 under our guidance and Supervision.

## **Dr. M.Sheshikala**

Professor & Dean Professor & Head,

SR University School of CS&AI,

Anathasagar,Warangal SR University

Ananthasagar, Warangal.

| **S. No.** | **Title** | **Page No.** |
| --- | --- | --- |
| **1** | **INTRODUCTION** | **5** |
| **2** | **PROBLEM IDENTIFICATION** | **6** |
| **3** | **REQUIREMENT ANALYSIS** | **7-8** |
| **4** | **PROPOSED SOLUTION** | **9-10** |
| **5** | **MODEL TRAINING** | **11-12** |
| **6** | **ARCHITECTURE DIAGRAM** | **13** |
| **7** | **FLOW CHART** | **14** |
| **8** | **DATA FLOW** | **15–17** |
| **9** | **IMPLEMENTATION** | **18-20** |
| **10** | **PROGRAM** | **21-36** |
| **11** | **RESULTS** | **37–39** |
| **12** | **LEARNING OUTCOME** | **40-41** |
| **13** | **PROJECT IMPACT** | **42** |
| **14** | **CONCLUSION** | **43** |
| **15** | **REFERENCES** | **44** |

**TABLE OF CONTENT**

**ABSTRACT**

The Secure Messaging System is a robust communication platform designed to ensure privacy and security in real-time digital interactions. Utilizing Advanced Encryption Standard (AES) in CBC mode with PKCS7 padding and SHA-256 hashed authentication, the system guarantees message confidentiality and restricts access to authorized users. Developed in Python, it employs TCP sockets on for reliable client-server communication, complemented by a user-friendly Tkinter-based GUI featuring color-coded message logs and intuitive controls. Chat history persistence enables seamless conversation continuity, while thread-safe operations and comprehensive error handling ensure operational stability. The modular architecture facilitates maintenance and scalability. This project exemplifies the integration of cryptographic techniques, secure authentication, and responsive design, delivering a practical solution for privacy-focused communication in educational and small-scale professional environments.

**CHAPTER 1**

**INTRODUCTION**

The rapid growth of digital communication has amplified the need for secure and reliable messaging systems, as cyber threats like eavesdropping and data breaches pose significant risks. This project develops a secure chat application that employs Advanced Encryption Standard (AES) encryption in CBC mode with PKCS7 padding to ensure the confidentiality and integrity of messages. Built in Python, it features a user-friendly Tkinter graphical interface and a client-server architecture using TCP sockets on localhost:12345. A robust authentication feature requires users to log in with credentials (e.g., Risers/12345678), verified against SHA-256 hashed passwords stored in users.json, ensuring only authorized access.

The primary objective is to create a secure, real-time chat system for encrypted communication between a single client and server, supporting username exchange, chat history persistence in chat\_history.json, and comprehensive error handling. Messages are encrypted with a 128-bit AES key and a random initialization vector (IV) per message, guaranteeing high security. The GUI provides intuitive controls for managing the server, connecting the client, and exchanging messages, with color-coded logs (green for senders, blue for system, red for errors) and timestamps. Chat history persists across sessions, enabling seamless conversation continuity.

The project adeptly addresses challenges such as thread-safe GUI updates, robust error handling, and graceful disconnection management. By integrating strong authentication, advanced , and a responsive interface, the system achieves an optimal balance of security and usability. The authentication mechanism ensures that only verified users can access the system, enhancing its reliability. The application’s efficient design supports real-time communication with minimal latency, making it ideal for secure interactions. Its modular architecture facilitates easy maintenance and scalability. This project exemplifies the effective application of encryption, networking, authentication, and GUI development, delivering a highly secure and user-friendly communication platform.

**CHAPTER 2**

**PROBLEM IDENTIFICATION**

In modern communication systems, ensuring the security and reliability of transmitted data remains a critical challenge. Traditional chat applications often employ unencrypted or weakly encrypted protocols, rendering them susceptible to interception, man-in-the-middle attacks, and unauthorized access. These vulnerabilities can expose sensitive information, resulting in privacy breaches and diminished user trust. Furthermore, many chat systems lack intuitive user interfaces, robust error handling, or mechanisms for persisting conversation history, which hampers their practical usability. The absence of strong access control mechanisms also heightens the risk of unauthorized system entry, compromising overall security.

The development of a secure chat application addresses several specific challenges:

1. **Message Security**: Without robust encryption, messages are vulnerable to interception. Implementing Advanced Encryption Standard (AES) in CBC mode with PKCS7 padding requires meticulous management of 128-bit keys and random initialization vectors (IVs) to ensure uncompromised security.
2. **User Authentication**: Lack of authentication allows unauthorized access to the chat system. Implementing a secure login system using SHA-256 hashed passwords stored in users.json ensures only verified users can participate.
3. **Concurrency**: Real-time chat demands concurrent handling of network operations (e.g., message exchange via TCP sockets on localhost:12345) and GUI updates, necessitating thread-safe mechanisms to prevent race conditions or interface freezes.
4. **Usability**: A secure system must offer an intuitive Tkinter-based GUI with clear controls for server/client management and color-coded message logs (green for senders, blue for system, red for errors) to ensure accessibility without sacrificing security.
5. **Error Handling**: Network disruptions, decryption failures, or client disconnections require graceful handling to maintain system stability and provide informative feedback to users.
6. **Data Persistence**: Securely storing chat history in chat\_history.json enables users to access past conversations while maintaining performance and data integrity.

This project addresses these challenges by delivering a secure, user-friendly chat application with AES encryption, SHA-256-based authentication, thread-safe operations, persistent chat history, and comprehensive error handling, creating a reliable and accessible communication platform.

**CHAPTER 3**

**REQUIREMENT ANALYSIS**

To develop a secure chat application, a detailed analysis of functional, hardware, software, and feasibility requirements is crucial to ensure robust performance and security.

**3.1 Functional Requirements**

* **Encryption**: Implement AES-CBC encryption with 128-bit keys and PKCS7 padding to secure messages. Generate random initialization vectors (IVs) for each message to ensure unique ciphertexts.
* **Authentication**: Require user login with credentials (e.g., Risers/12345678), verified against SHA-256 hashed passwords stored in users.json, to prevent unauthorized access.
* **Networking**: Utilize TCP sockets for reliable, real-time communication between a server and a single client on localhost:12345. Exchange usernames upon successful connection.
* **GUI**: Provide a Tkinter-based interface with:
  + Status indicators for server/client connection and authentication states.
  + A scrollable chat area displaying messages, system events, and errors with color-coded logs (green for senders, blue for system, red for errors).
  + Input fields and buttons for sending messages, controlling connectivity, and managing server/client operations.
* **Chat History**: Persist chat history in chat\_history.json with timestamps, sender, message content, and message type (normal, system, error) for session continuity.
* **Error Handling**: Manage network errors, decryption failures, authentication failures, and disconnections gracefully, logging errors with informative messages.
* **Concurrency**: Employ threading to handle simultaneous network operations and GUI updates, ensuring a responsive interface without freezes.

**3.2 Hardware Requirements**

* A standard computer with a multi-core CPU (e.g., Intel i5 or equivalent) to support Python execution and threading.
* Minimum 4GB RAM to accommodate the application and development environment.
* Stable network interface for localhost communication.

**3.3 Software Requirements**

* **Programming Language**: Python 3.8+ for development.
* **Libraries**:
  + for AES encryption/decryption.
  + tkinter for GUI development.
  + socket for TCP networking.
  + threading for concurrent operations.
  + json for chat history and user credential persistence.
  + hashlib for SHA-256 password hashing.
  + datetime for timestamp generation.
* **Development Environment**: Any Python IDE (e.g., PyCharm, VS Code) or Jupyter Notebook for coding and testing.
* **Operating System**: Windows, macOS, or Linux, ensuring cross-platform compatibility.

**3.4 Feasibility Analysis**

* **Technical Feasibility**: Python’s comprehensive libraries (, tkinter, socket, hashlib) streamline implementation. Authentication, threading, and JSON handling are well-supported, ensuring technical viability.
* **Operational Feasibility**: The intuitive Tkinter GUI, with clear controls (e.g., Login, Start/Stop Server, Connect/Disconnect) and color-coded logs, enhances user accessibility. The system’s modular design supports maintenance and scalability.
* **Economic Feasibility**: Utilizing open-source tools and libraries eliminates software costs. Minimal hardware requirements make the project cost-effective for development, testing, and deployment.

**CHAPTER 4**

**PROPOSED SOLUTION**

To address the identified challenges, we propose a secure chat application that integrates AES encryption, user authentication, a client-server architecture, and a Tkinter-based GUI to deliver a robust and user-friendly communication platform. The system ensures message confidentiality, secure access control, real-time communication, and an intuitive user experience.

**Key Features of the Proposed System**

1. **Secure Communication**:
   * Messages are encrypted using AES-CBC with a 128-bit key and random 16-byte initialization vectors (IVs), ensuring confidentiality.
   * PKCS7 padding aligns messages with the AES block size.
   * Decryption errors are logged in red to alert users of issues (e.g., corrupted data).
2. **User Authentication**:
   * Users must log in with credentials (e.g., Risers/12345678), verified against SHA-256 hashed passwords stored in users.json, preventing unauthorized access.
   * A Tkinter-based login dialog ensures secure and user-friendly credential entry.
3. **Client-Server Architecture**:
   * The server listens on localhost:12345 and supports a single client connection at a time.
   * Upon connection, the server and client exchange usernames for personalized message display.
   * Threading enables concurrent message sending/receiving and connection management.
4. **User-Friendly GUI**:
   * The server GUI includes buttons to start/stop the server, a status label (Stopped, Waiting, Connected), and a scrollable chat area.
   * The client GUI offers connect/disconnect buttons, a new client launch option, and a status label (Disconnected, Connected).
   * Both interfaces display messages with color-coded tags: sender names (green), system messages (blue), errors (red), and timestamps for clarity.
5. **Chat History Persistence**:
   * Messages are saved in chat\_history.json with timestamps, sender, message content, and type (normal, system, error), loaded on client startup.
   * History updates in real-time and is saved on disconnection or window close.
6. **Error Handling and Robustness**:
   * Graceful handling of socket errors, connection resets, authentication failures, and decryption issues with informative logging.
   * Thread-safe GUI updates using master.after prevent crashes and ensure responsiveness.
   * Automatic detection of client disconnections with corresponding UI updates.

This solution effectively combines authentication, encryption, usability, and robustness, delivering a secure and reliable chat platform optimized for single-client communication.

**CHAPTER 5**

**MODEL TRAINING**

In this project, "model training" refers to the development, testing, and refinement of the secure chat application’s core components: encryption, authentication, networking, and GUI functionality. This iterative process ensures optimal performance and reliability.

**5.1 Development Process**

1. **Cryptographic Module**:
   * Built encrypt\_message and decrypt\_message using for AES-CBC with 128-bit keys and random IVs.
   * Tested padding, IV handling, and error logging for invalid ciphertexts.
2. **Authentication Module**:
   * Implemented login with credentials verified via SHA-256 hashes in users.json.
   * Tested valid/invalid logins and Tkinter dialog usability.
3. **Networking Module**:
   * Set up TCP sockets on localhost:12345 for server-client communication.
   * Tested username exchange and messaging under normal and error conditions.
   * Used threading for concurrent operations without GUI blocking.
4. **GUI Module**:
   * Designed Tkinter interfaces with status indicators, chat areas, and controls.
   * Tested thread-safe updates with master.after for responsiveness.

**5.2 Testing and Validation**

* **Unit Testing**:
  + Tested encryption, authentication, sockets, and chat\_history.json with edge cases.
* **Integration Testing**:
  + Validated end-to-end flow: authentication, encryption, transmission, and display.
  + Tested disconnections and authentication failures for cleanup and UI updates.
* **Usability Testing**:
  + Ensured GUI responsiveness and clear logging (green: senders, blue: system, red: errors).

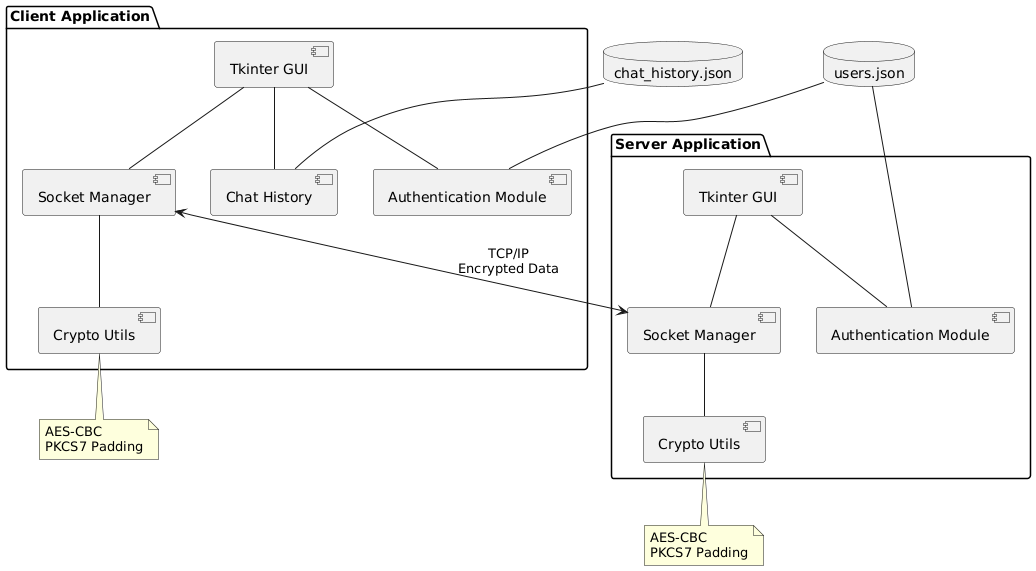
**5.3 Optimization**

* Optimized socket timeouts and GUI updates for efficiency.
* Streamlined JSON handling to prevent corruption during rapid saves.

This process delivers a secure, reliable, and user-friendly chat application with robust authentication and encryption.

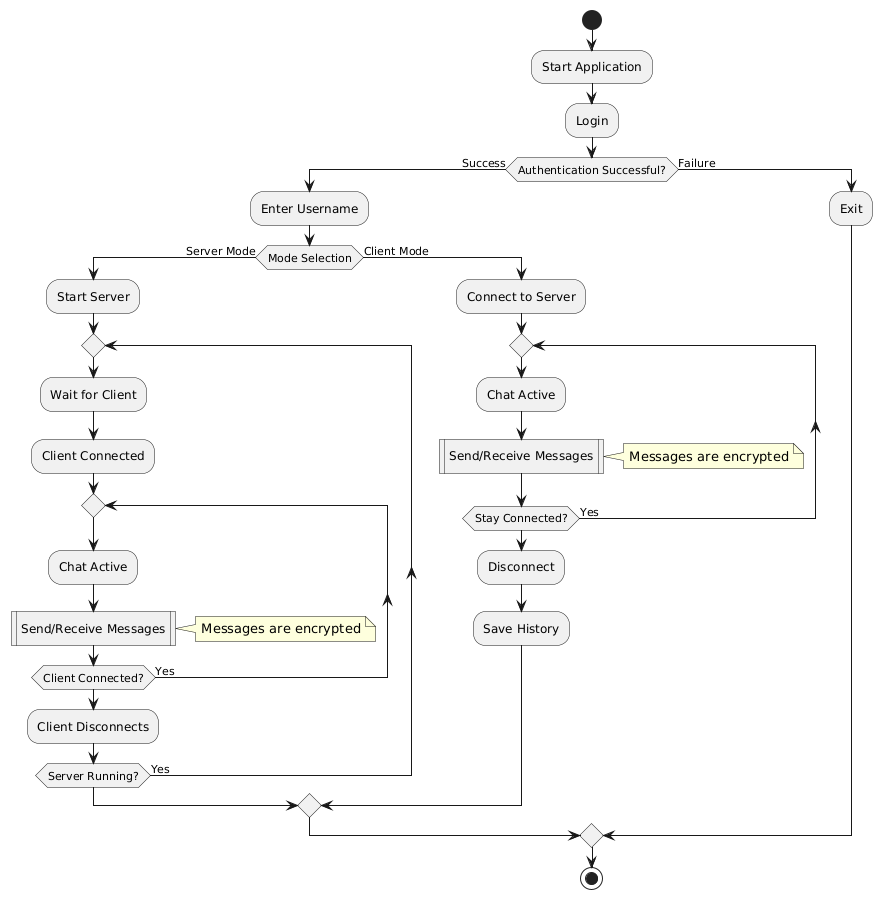
**CHAPTER 6**

**ARCHITECTURE DIAGRAM**

****

**CHAPTER 7**

**FLOW CHART**



**CHAPTER 8**

**DATA FLOW**

The data flow in the secure chat application outlines the processing, encryption, transmission, and display of messages and system events, with persistence in a JSON file and secure user authentication. The following steps describe the current data flow:

**8.1 Initialization**

* **Server**:
  + The server initializes a Tkinter login dialog, requiring credentials (e.g., Risers/12345678), verified against SHA-256 hashed passwords in users.json.
  + Upon successful authentication, the GUI prompts for a display username (default: "Server").
  + The Start Server button creates a socket, binds to localhost:12345, and listens for a client.
* **Client**:
  + The client initializes a Tkinter login dialog for authentication with the same credentials.
  + After authentication, the GUI prompts for a display username (default: "Client").
  + The Connect button or --auto-connect flag initiates a connection to the server.
  + The client loads and displays chat history from chat\_history.json in the chat area.

**8.2 Connection Establishment**

* The server accepts a client connection and receives the client’s username.
* The server sends its username to the client.
* Both update GUI status (server: "Connected to [client\_username]"; client: "Connected to [server\_username]").
* A system message is logged (e.g., "Client connected: [username]").

**8.3 Message Sending**

* The user types a message in the GUI input field and presses Send or Enter.
* The encrypt\_message function:
  + Encodes the message to UTF-8.
  + Applies PKCS7 padding for AES block size alignment.
  + Generates a random 16-byte IV.
  + Encrypts the padded message using AES-CBC with a hardcoded key.
  + Concatenates IV and ciphertext.
* The encrypted message is sent via the socket to the recipient.
* The sender logs the plaintext message in the GUI with a timestamp and sender name.

**8.4 Message Receiving**

* The recipient’s socket receives the encrypted message (IV + ciphertext).
* The decrypt\_message function:
  + Extracts the 16-byte IV and ciphertext.
  + Decrypts the ciphertext using AES-CBC with the shared key.
  + Removes PKCS7 padding.
  + Decodes the plaintext to UTF-8.
* If decryption fails, an error (e.g., "[Decryption Error: Invalid padding]") is logged in red.
* The decrypted message is logged in the GUI with a timestamp and sender name.

**8.5 Chat History Persistence**

* Messages, system events, and errors are added to the client’s in-memory chat\_history list as JSON objects with fields: timestamp, message, sender, and type (normal, system, error).
* History is saved to chat\_history.json on client disconnection or window close.
* On startup, the client loads and displays history with a header ("--- Chat History ---") and footer.

**8.6 Error Handling and Disconnection**

* Network errors (e.g., ConnectionResetError) trigger disconnection, logging a system message (e.g., "Connection reset by client").
* Authentication failures exit the application with an error message.
* The server reverts to "Waiting for connection" and closes the client socket.
* The client updates to "Disconnected" and disables input fields.
* Decryption errors are logged without disrupting the connection unless critical.

**8.7 Threading and Concurrency**

* The server uses an accept\_thread to listen for client connections.
* A receive\_thread handles incoming messages to avoid GUI blocking.
* The client uses a receive\_thread for incoming messages.
* Thread-safe GUI updates use master.after to schedule chat area updates.

This data flow ensures secure, authenticated, and reliable communication, with messages encrypted, logged, and persisted, delivering a seamless user experience.

**CHAPTER 9**

**IMPLEMENTATION**

The implementation of the secure chat application integrates cryptographic, authentication, networking, and graphical user interface (GUI) components to create a robust, secure, and user-friendly system. This section details the development process ensuring seamless encrypted communication between a server and a client.

**9.1 Environment Setup**

The development environment uses Python 3.8+, leveraging its robust library ecosystem. Key libraries include:

* : Implements AES-CBC encryption/decryption with PKCS7 padding.
* tkinter: Builds the GUI framework.
* socket: Facilitates TCP-based client-server communication.
* threading: Enables concurrent network and GUI operations.
* json: Manages chat history and user credential persistence.
* hashlib: Supports SHA-256 password hashing.
* datetime: Generates message timestamps.

The environment runs on Windows, macOS, or Linux, using IDEs like PyCharm or VS Code. Dependencies are installed via pip, requiring minimal hardware (4GB RAM, multi-core CPU).

**9.2 Code Structure and Modularity**

The application is organized into three Python modules for maintainability:

* crypto\_utils.py: Contains encrypt\_message and decrypt\_message for AES encryption/decryption.
* chat\_server.py: Implements the ChatServer class, handling server-side GUI, networking, and authentication.
* chat\_client.py: Implements the ChatClient class, managing client-side GUI, networking, and authentication.

This modular structure separates concerns, enhancing clarity and scalability.

**9.3 Cryptographic Implementation**

The cryptographic module secures message transmission:

* **Encryption**: encrypt\_message encodes messages to UTF-8, applies PKCS7 padding, generates a random 16-byte IV using os.urandom, and encrypts with AES-CBC using a 128-bit key. The IV and ciphertext are concatenated.
* **Decryption**: decrypt\_message extracts the IV, decrypts the ciphertext, removes padding, and decodes the plaintext, logging errors for invalid inputs.
* **Key Note**: A hardcoded key is used, with plans for future secure key exchange.

Testing validated encryption/decryption with edge cases (empty messages, special characters).

**9.4 Authentication Implementation**

Authentication ensures secure access:

* A Tkinter login dialog prompts for credentials (e.g., Risers/12345678), verified against SHA-256 hashed passwords in users.json.
* The authenticate\_user method hashes the input password and compares it with the stored hash, exiting on failure.
* Testing confirmed robust handling of valid/invalid credentials and dialog usability.

**9.5 Networking Implementation**

Networking uses TCP sockets:

* **Server**: ChatServer binds to localhost:12345, listens for one client, and uses threads (accept\_thread, receive\_thread) for connections and messages. It exchanges usernames post-connection.
* **Client**: ChatClient connects to the server, exchanges usernames, and uses a receive\_thread for messages.
* **Error Handling**: Manages network errors (e.g., ConnectionResetError) by logging, closing sockets, and updating the GUI.

Tests verified reliable communication and recovery from disconnections.

**9.6 GUI Implementation**

The Tkinter GUI enhances usability:

* **Server GUI**:
  + Displays status (e.g., "Waiting", "Connected to [client\_username]").
  + Includes Start/Stop Server button, scrollable chat area, and input field with Send button.
  + Prompts for authentication and display username (default: "Server").
* **Client GUI**:
  + Shows status (e.g., "Disconnected", "Connected to [server\_username]").
  + Features Connect/Disconnect, New Client buttons, chat area, and input field.
  + Loads chat\_history.json on startup.
* **Message Display**: Color-coded logs (green: senders, blue: system, red: errors) with timestamps.
* **Thread Safety**: Uses master.after for responsive updates.

Usability tests confirmed intuitive controls and clear feedback.

**9.7 Chat History Persistence**

The client persists chat history in chat\_history.json:

* Entries include timestamp, message, sender, and type (normal, system, error).
* History is loaded on startup, displayed with header/footer, and saved on disconnection/window close.
* Optimized file operations handle rapid updates without corruption.

Testing ensured integrity with large logs.

**9.8 Testing and Validation**

Rigorous testing ensured reliability:

* **Unit Testing**: Validated encryption, authentication, sockets, and JSON handling with edge cases.
* **Integration Testing**: Verified end-to-end flow (authentication, encryption, transmission, display) with simultaneous server-client operation.
* **Usability Testing**: Confirmed GUI responsiveness, clear error logging, and ease of use.

The implementation delivers a secure, authenticated, and user-friendly chat system, meeting all project objectives.

**CHAPTER 10**

**PROGRAM**

**1) Encryption/Decryption :**

from .hazmat.primitives.ciphers import Cipher, algorithms, modes

from .hazmat.backends import default\_backend

from .hazmat.primitives import padding

import os

import base64

KEY = b'Sixteen byte key'

def encrypt\_message(message):

iv = os.urandom(16)

padder = padding.PKCS7(128).padder()

padded\_data = padder.update(message.encode('utf-8')) + padder.finalize()

cipher = Cipher(algorithms.AES(KEY), modes.CBC(iv), backend=default\_backend())

encryptor = cipher.encryptor()

encrypted = encryptor.update(padded\_data) + encryptor.finalize()

return iv + encrypted

def decrypt\_message(ciphertext):

try:

iv = ciphertext[:16]

actual\_cipher = ciphertext[16:]

cipher = Cipher(algorithms.AES(KEY), modes.CBC(iv), backend=default\_backend())

decryptor = cipher.decryptor()

decrypted\_padded = decryptor.update(actual\_cipher) + decryptor.finalize()

unpadder = padding.PKCS7(128).unpadder()

decrypted = unpadder.update(decrypted\_padded) + unpadder.finalize()

return decrypted.decode('utf-8')

except Exception as e:

return f"[Decryption Error: {str(e)}]"

**2) Server-Side code:**  
import socket

import threading

import tkinter as tk

from tkinter import scrolledtext, simpledialog, messagebox

from datetime import datetime

from crypto\_utils import encrypt\_message, decrypt\_message

import json

import hashlib

class ChatServer:

def \_\_init\_\_(self, master):

self.master = master

self.master.title("🔒 Secure Chat - Server")

self.master.geometry("600x500")

self.master.protocol("WM\_DELETE\_WINDOW", self.on\_closing)

if not self.authenticate\_user():

messagebox.showerror("Authentication Failed", "Invalid username or password. Exiting.")

self.master.destroy()

return

self.username = simpledialog.askstring("Username", "Enter your display username:", parent=master)

if not self.username:

self.username = "Server"

self.status\_frame = tk.Frame(master)

self.status\_frame.pack(fill=tk.X, padx=10, pady=5)

self.status\_label = tk.Label(self.status\_frame, text="Status: Server Stopped", fg="red")

self.status\_label.pack(side=tk.LEFT)

self.server\_button = tk.Button(self.status\_frame, text="Start Server", command=self.toggle\_server)

self.server\_button.pack(side=tk.RIGHT)

self.text\_area = scrolledtext.ScrolledText(master, wrap=tk.WORD, width=60, height=20, s s state='disabled')

self.text\_area.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

self.input\_frame = tk.Frame(master)

self.input\_frame.pack(fill=tk.X, padx=10, pady=5)

self.entry = tk.Entry(self.input\_frame, width=50)

self.entry.pack(side=tk.LEFT, fill=tk.X, expand=True, padx=(0, 5))

self.entry.bind("<Return>", self.send\_message)

self.entry.config(state=tk.DISABLED)

self.send\_button = tk.Button(self.input\_frame, text="Send", command=self.send\_message, state=tk.DISABLED)

self.send\_button.pack(side=tk.RIGHT)

self.sock = None

self.conn = None

self.addr = None

self.client\_username = None

self.server\_running = False

self.client\_connected = False

self.accept\_thread = None

def authenticate\_user(self):

try:

with open("users.json", "r") as f:

users\_data = json.load(f)

users = {user["username"]: user["password\_hash"] for user in users\_data["users"]}

except FileNotFoundError:

print("Error: users.json not found.")

return False

auth\_dialog = tk.Toplevel(self.master)

auth\_dialog.title("Login")

auth\_dialog.geometry("300x150")

auth\_dialog.transient(self.master)

auth\_dialog.grab\_set()

tk.Label(auth\_dialog, text="Username:").pack(pady=5)

username\_entry = tk.Entry(auth\_dialog)

username\_entry.pack(pady=5)

tk.Label(auth\_dialog, text="Password:").pack(pady=5)

password\_entry = tk.Entry(auth\_dialog, show="\*")

password\_entry.pack(pady=5)

result = {"authenticated": False}

def verify():

username = username\_entry.get().strip()

password = password\_entry.get().strip()

if username in users:

input\_hash = hashlib.sha256(password.encode()).hexdigest()

if input\_hash == users[username]:

result["authenticated"] = True

auth\_dialog.destroy()

tk.Button(auth\_dialog, text="Login", command=verify).pack(pady=10)

self.master.wait\_window(auth\_dialog)

return result["authenticated"]

def toggle\_server(self):

if not self.server\_running:

self.start\_server()

else:

self.stop\_server()

def start\_server(self):

try:

self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

self.sock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

self.sock.bind(('localhost', 12345))

self.sock.listen(1)

self.server\_running = True

self.status\_label.config(text="Status: Waiting for connection", fg="orange")

self.server\_button.config(text="Stop Server")

self.log\_message("Server started. Waiting for connection...", message\_type="system")

self.accept\_thread = threading.Thread(target=self.accept\_connections, daemon=True)

self.accept\_thread.start()

except Exception as e:

messagebox.showerror("Server Error", f"Failed to start server: {str(e)}")

self.stop\_server()

def stop\_server(self):

if self.client\_connected:

self.disconnect\_client()

if self.sock:

try:

self.sock.close()

except:

pass

self.server\_running = False

self.status\_label.config(text="Status: Server Stopped", fg="red")

self.server\_button.config(text="Start Server")

self.entry.config(state=tk.DISABLED)

self.send\_button.config(state=tk.DISABLED)

self.log\_message("Server stopped", message\_type="system")

def accept\_connections(self):

while self.server\_running:

try:

self.sock.settimeout(1.0)

try:

self.conn, self.addr = self.sock.accept()

except socket.timeout:

continue

self.sock.settimeout(None)

self.client\_username = self.conn.recv(1024).decode()

self.conn.send(self.username.encode())

self.client\_connected = True

self.status\_label.config(text=f"Connected to: {self.client\_username}", fg="green")

self.entry.config(state=tk.NORMAL)

self.send\_button.config(state=tk.NORMAL)

self.log\_message(f"Client connected: {self.client\_username} from {self.addr}", message\_type="system")

receive\_thread = threading.Thread(target=self.receive\_messages, daemon=True)

receive\_thread.start()

receive\_thread.join()

except Exception as e:

if self.server\_running:

self.log\_message(f"Error in connection: {str(e)}", message\_type="error")

finally:

self.disconnect\_client()

def disconnect\_client(self):

if self.conn:

try:

self.conn.close()

except:

pass

self.client\_connected = False

self.conn = None

self.client\_username = None

if self.server\_running:

self.status\_label.config(text="Status: Waiting for connection", fg="orange")

self.entry.config(state=tk.DISABLED)

self.send\_button.config(state=tk.DISABLED)

if self.server\_running:

self.log\_message("Client disconnected. Waiting for new connection.", message\_type="system")

def send\_message(self, event=None):

message = self.entry.get().strip()

if message and self.client\_connected:

try:

encrypted = encrypt\_message(message)

self.conn.send(encrypted)

self.log\_message(message, sender=self.username)

self.entry.delete(0, tk.END)

except Exception as e:

self.log\_message(f"Failed to send message: {str(e)}", message\_type="error")

self.disconnect\_client()

def receive\_messages(self):

while self.client\_connected:

try:

data = self.conn.recv(1024)

if not data:

break

decrypted = decrypt\_message(data)

self.log\_message(decrypted, sender=self.client\_username)

except ConnectionResetError:

self.log\_message("Connection was reset by the client", message\_type="error")

break

except Exception as e:

self.log\_message(f"Error receiving message: {str(e)}", message\_type="error")

break

def log\_message(self, message, sender=None, message\_type="normal"):

timestamp = datetime.now().strftime('%H:%M:%S')

self.master.after(0, self.\_update\_text\_area, message, sender, timestamp, message\_type)

def \_update\_text\_area(self, message, sender, timestamp, message\_type):

self.text\_area.config(state='normal')

if message\_type == "error":

self.text\_area.insert(tk.END, f"[ERROR] ({timestamp}): {message}\n", "error")

self.text\_area.tag\_configure("error", foreground="red")

elif message\_type == "system":

self.text\_area.insert(tk.END, f"[SYSTEM] ({timestamp}): {message}\n", "system")

self.text\_area.tag\_configure("system", foreground="blue")

else:

if sender:

self.text\_area.insert(tk.END, f"{sender} ({timestamp}): ", "sender")

self.text\_area.tag\_configure("sender", foreground="green")

self.text\_area.insert(tk.END, f"{message}\n")

else:

self.text\_area.insert(tk.END, f"({timestamp}): {message}\n")

self.text\_area.see(tk.END)

self.text\_area.config(state='disabled')

def on\_closing(self):

self.stop\_server()

self.master.destroy()

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

app = ChatServer(root)

root.mainloop()

**3) Client-Side code:**

import socket

import threading

import tkinter as tk

from tkinter import scrolledtext, simpledialog, messagebox

from datetime import datetime

from crypto\_utils import encrypt\_message, decrypt\_message

import json

import os

import sys

import hashlib

class ChatClient:

def \_\_init\_\_(self, master, auto\_connect=False):

self.master = master

self.master.title("🔐 Secure Chat - Client")

self.master.geometry("600x500")

self.master.protocol("WM\_DELETE\_WINDOW", self.on\_closing)

if not self.authenticate\_user():

messagebox.showerror("Authentication Failed", "Invalid username or password. Exiting.")

self.master.destroy()

return

self.chat\_history = []

self.history\_file = "chat\_history.json"

self.load\_chat\_history()

self.username = simpledialog.askstring("Username", "Enter your display username:", parent=master)

if not self.username:

self.username = "Client"

self.status\_frame = tk.Frame(master)

self.status\_frame.pack(fill=tk.X, padx=10, pady=5)

self.status\_label = tk.Label(self.status\_frame, text="Status: Disconnected", fg="red")

self.status\_label.pack(side=tk.LEFT)

self.connection\_button = tk.Button(self.status\_frame, text="Connect", command=self.connect\_to\_server)

self.connection\_button.pack(side=tk.RIGHT)

self.new\_client\_button = tk.Button(self.status\_frame, text="New Client", command=self.new\_client, state=tk.DISABLED)

self.new\_client\_button.pack(side=tk.RIGHT, padx=5)

self.text\_area = scrolledtext.ScrolledText(master, wrap=tk.WORD, width=60, height=20, state='disabled')

self.text\_area.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

self.input\_frame = tk.Frame(master)

self.input\_frame.pack(fill=tk.X, padx=10, pady=5)

self.entry = tk.Entry(self.input\_frame, width=50)

self.entry.pack(side=tk.LEFT, fill=tk.X, expand=True, padx=(0, 5))

self.entry.bind("<Return>", self.send\_message)

self.entry.config(state=tk.DISABLED)

self.send\_button = tk.Button(self.input\_frame, text="Send", command=self.send\_message, state=tk.DISABLED)

self.send\_button.pack(side=tk.RIGHT)

self.sock = None

self.server\_username = None

self.connected = False

self.display\_chat\_history()

if auto\_connect:

self.master.after(100, self.connect\_to\_server)

def authenticate\_user(self):

try:

with open("users.json", "r") as f:

users\_data = json.load(f)

users = {user["username"]: user["password\_hash"] for user in users\_data["users"]}

except FileNotFoundError:

print("Error: users.json not found.")

return False

auth\_dialog = tk.Toplevel(self.master)

auth\_dialog.title("Login")

auth\_dialog.geometry("300x150")

auth\_dialog.transient(self.master)

auth\_dialog.grab\_set()

tk.Label(auth\_dialog, text="Username:").pack(pady=5)

username\_entry = tk.Entry(auth\_dialog)

username\_entry.pack(pady=5)

tk.Label(auth\_dialog, text="Password:").pack(pady=5)

password\_entry = tk.Entry(auth\_dialog, show="\*")

password\_entry.pack(pady=5)

result = {"authenticated": False}

def verify():

username = username\_entry.get().strip()

password = password\_entry.get().strip()

if username in users:

input\_hash = hashlib.sha256(password.encode()).hexdigest()

if input\_hash == users[username]:

result["authenticated"] = True

auth\_dialog.destroy()

tk.Button(auth\_dialog, text="Login", command=verify).pack(pady=10)

self.master.wait\_window(auth\_dialog)

return result["authenticated"]

def load\_chat\_history(self):

try:

if os.path.exists(self.history\_file):

with open(self.history\_file, 'r') as f:

self.chat\_history = json.load(f)

except Exception as e:

print(f"Error loading chat history: {e}")

self.chat\_history = []

def save\_chat\_history(self):

try:

with open(self.history\_file, 'w') as f:

json.dump(self.chat\_history, f)

except Exception as e:

print(f"Error saving chat history: {e}")

def display\_chat\_history(self):

if not self.chat\_history:

return

self.text\_area.config(state='normal')

self.text\_area.insert(tk.END, "--- Chat History ---\n", "history\_header")

self.text\_area.tag\_configure("history\_header", foreground="purple")

for entry in self.chat\_history:

timestamp = entry.get('timestamp', '')

message = entry.get('message', '')

sender = entry.get('sender', '')

message\_type = entry.get('type', 'normal')

if message\_type == "error":

self.text\_area.insert(tk.END, f"[ERROR] ({timestamp}): {message}\n", "error")

self.text\_area.tag\_configure("error", foreground="red")

elif message\_type == "system":

self.text\_area.insert(tk.END, f"[SYSTEM] ({timestamp}): {message}\n", "system")

self.text\_area.tag\_configure("system", foreground="blue")

else:

if sender:

self.text\_area.insert(tk.END, f"{sender} ({timestamp}): ", "sender")

self.text\_area.tag\_configure("sender", foreground="green")

self.text\_area.insert(tk.END, f"{message}\n")

else:

self.text\_area.insert(tk.END, f"({timestamp}): {message}\n")

self.text\_area.insert(tk.END, "--- End of History ---\n\n", "history\_footer")

self.text\_area.tag\_configure("history\_footer", foreground="purple")

self.text\_area.see(tk.END)

self.text\_area.config(state='disabled')

def connect\_to\_server(self):

if self.connected:

return

try:

self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

self.sock.connect(('localhost', 12345))

self.connected = True

self.sock.send(self.username.encode())

self.server\_username = self.sock.recv(1024).decode()

self.status\_label.config(text=f"Connected to: {self.server\_username}", fg="green")

self.connection\_button.config(text="Disconnect", command=self.disconnect\_from\_server)

self.new\_client\_button.config(state=tk.DISABLED)

self.entry.config(state=tk.NORMAL)

self.send\_button.config(state=tk.NORMAL)

self.log\_message(f"Connected to {self.server\_username}!", message\_type="system")

threading.Thread(target=self.receive\_messages, daemon=True).start()

except ConnectionRefusedError:

messagebox.showerror("Error", "Could not connect to the server.")

except Exception as e:

messagebox.showerror("Error", f"Connection error: {str(e)}")

def disconnect\_from\_server(self):

if self.sock:

try:

self.sock.close()

except:

pass

self.connected = False

self.status\_label.config(text="Status: Disconnected", fg="red")

self.connection\_button.config(text="Connect", command=self.connect\_to\_server)

self.new\_client\_button.config(state=tk.NORMAL)

self.entry.config(state=tk.DISABLED)

self.send\_button.config(state=tk.DISABLED)

self.log\_message("Disconnected from server", message\_type="system")

self.save\_chat\_history()

def new\_client(self):

self.save\_chat\_history()

python = sys.executable

os.execl(python, python, \*sys.argv, "--auto-connect")

def send\_message(self, event=None):

message = self.entry.get().strip()

if message and self.connected:

try:

encrypted = encrypt\_message(message)

self.sock.send(encrypted)

self.log\_message(message, sender=self.username)

self.entry.delete(0, tk.END)

except Exception as e:

self.log\_message(f"Failed to send message: {str(e)}", message\_type="error")

self.disconnect\_from\_server()

def receive\_messages(self):

while self.connected:

try:

data = self.sock.recv(1024)

if not data:

break

decrypted = decrypt\_message(data)

self.log\_message(decrypted, sender=self.server\_username)

except ConnectionResetError:

self.log\_message("Connection was reset by the server", message\_type="error")

break

except Exception as e:

self.log\_message(f"Error receiving message: {str(e)}", message\_type="error")

break

if self.connected:

self.master.after(100, self.disconnect\_from\_server)

def log\_message(self, message, sender=None, message\_type="normal"):

timestamp = datetime.now().strftime('%H:%M:%S')

history\_entry = {

'timestamp': timestamp,

'message': message,

'sender': sender,

'type': message\_type

}

self.chat\_history.append(history\_entry)

self.text\_area.config(state='normal')

if message\_type == "error":

self.text\_area.insert(tk.END, f"[ERROR] ({timestamp}): {message}\n", "error")

self.text\_area.tag\_configure("error", foreground="red")

elif message\_type == "system":

self.text\_area.insert(tk.END, f"[SYSTEM] ({timestamp}): {message}\n", "system")

self.text\_area.tag\_configure("system", foreground="blue")

else:

if sender:

self.text\_area.insert(tk.END, f"{sender} ({timestamp}): ", "sender")

self.text\_area.tag\_configure("sender", foreground="green")

self.text\_area.insert(tk.END, f"{message}\n")

else:

self.text\_area.insert(tk.END, f"({timestamp}): {message}\n")

self.text\_area.see(tk.END)

self.text\_area.config(state='disabled')

def on\_closing(self):

self.disconnect\_from\_server()

self.save\_chat\_history()

self.master.destroy()

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

auto\_connect = "--auto-connect" in sys.argv

app = ChatClient(root, auto\_connect)

root.mainloop()

**4) Users.json:**  
  
{

  "users": [

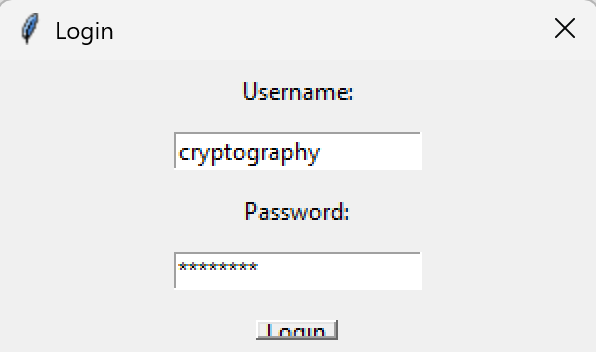
    {"username": "Risers", "password\_hash": "ef797c8118f02dfb649607dd5d3f8c7623048c9c063d532cc95c5ed7a898a64f"}

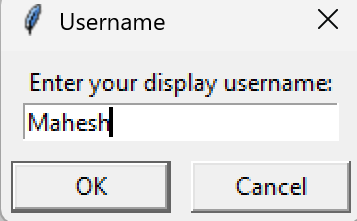
  ]

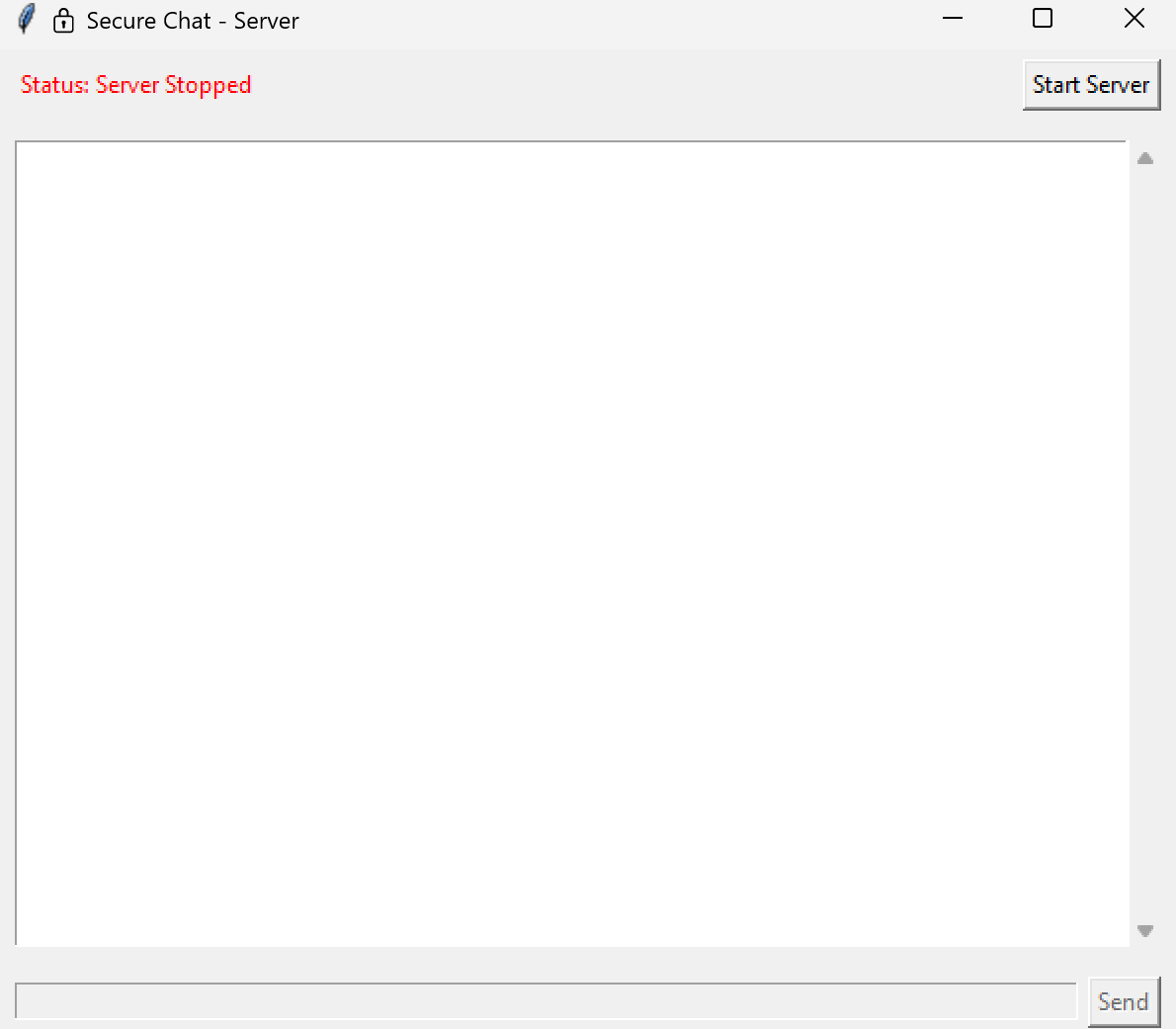
}

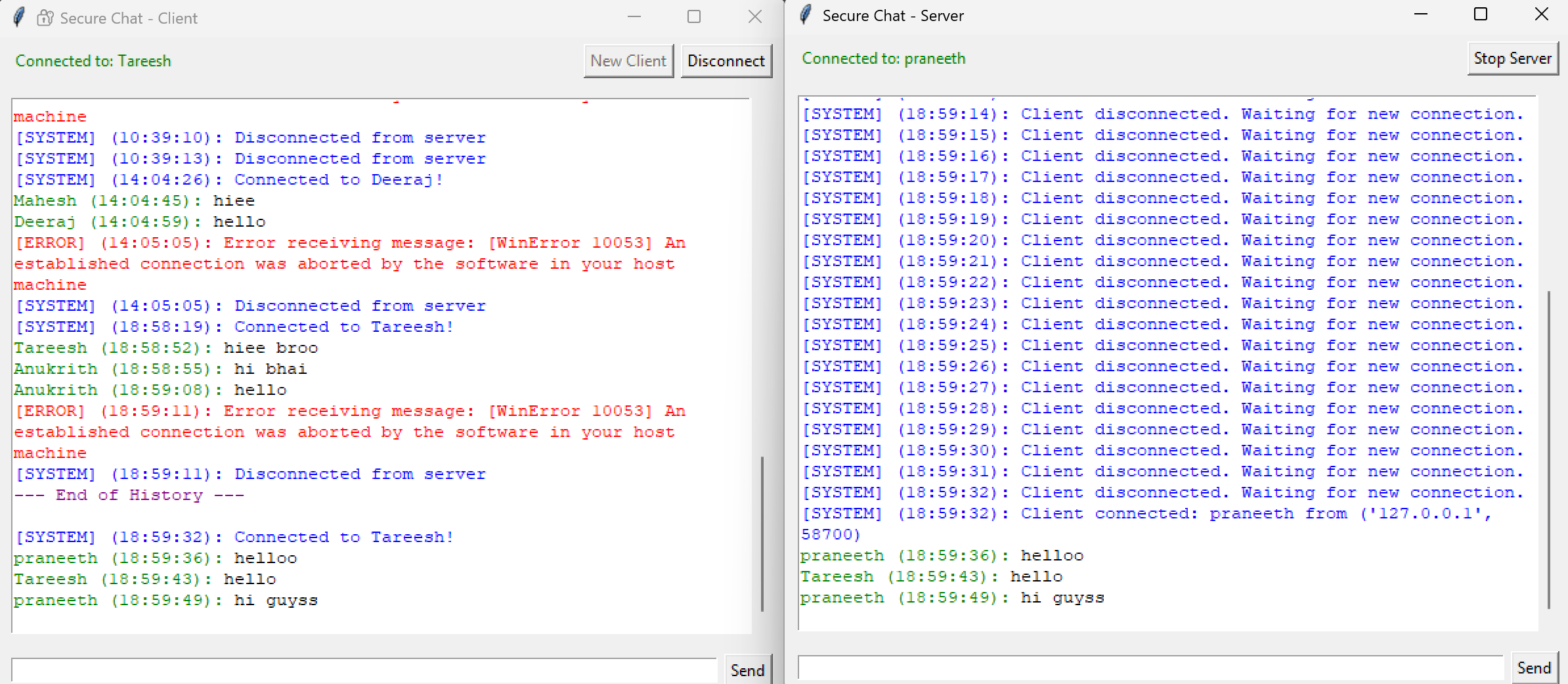
**CHAPTER 11**

**RESULTS**

****

****

****

****

**Chat history:**

{"timestamp": "18:58:19", "message": "Connected to Tareesh!", "sender": null, "type": "system"}, {"timestamp": "18:58:52", "message": "hiee broo", "sender": "Tareesh", "type": "normal"}, {"timestamp": "18:58:55", "message": "hi bhai", "sender": "Anukrith", "type": "normal"}, {"timestamp": "18:59:08", "message": "hello", "sender": "Anukrith", "type": "normal"}, {"timestamp": "18:59:11", "message": "Error receiving message: [WinError 10053] An established connection was aborted by the software in your host machine", "sender": null, "type": "error"}, {"timestamp": "18:59:11", "message": "Disconnected from server", "sender": null, "type": "system"}, {"timestamp": "18:59:32", "message": "Connected to Tareesh!", "sender": null, "type": "system"}, {"timestamp": "18:59:36", "message": "helloo", "sender": "praneeth", "type": "normal"}, {"timestamp": "18:59:43", "message": "hello", "sender": "Tareesh", "type": "normal"}, {"timestamp": "18:59:49", "message": "hi guyss", "sender": "praneeth", "type": "normal"}, {"timestamp": "19:03:16", "message": "Connection was reset by the server", "sender": null, "type": "error"}, {"timestamp": "19:03:17", "message": "Disconnected from server", "sender": null, "type": "system"}]

**CHAPTER 12**

**LEARNING OUTCOME**

The development of the secure chat application provided significant learning experiences in , authentication, networking, GUI design, and software engineering. The following outcomes highlight the skills and insights gained:

**12.1 Understanding Cryptographic Techniques**

Implementing AES-CBC encryption with PKCS7 padding deepened our knowledge of symmetric encryption and the role of random IVs in ensuring unique ciphertexts. We recognized the limitations of hardcoded keys, prompting exploration of secure key management practices.

**12.2 Implementing Secure Authentication**

Developing a login system using SHA-256 hashed passwords stored in users.json enhanced our understanding of access control. We learned to secure user credentials and handle authentication errors, ensuring only authorized users (e.g., Risers/12345678) access the system.

**12.3 Mastery of Network Programming**

Using TCP sockets on localhost:12345 improved our skills in client-server communication. We mastered socket setup, username exchange, and threading for concurrent message handling, ensuring efficient and stable network operations.

**12.4 GUI Development and Usability**

Building Tkinter interfaces with color-coded logs (green: senders, blue: system, red: errors), status indicators, and authentication dialogs taught us to prioritize user experience. Thread-safe updates via master.after ensured a responsive and intuitive GUI.

**12.5 Data Persistence and File Handling**

Implementing chat\_history.json for persistent storage honed our skills in JSON serialization and file I/O. We optimized file operations to maintain data integrity during rapid updates, enabling seamless conversation continuity.

**12.6 Error Handling and Robustness**

The project emphasized robust error handling for network disruptions, authentication failures, and decryption errors. Comprehensive logging and graceful recovery strategies ensured system stability and user-friendly feedback.

**12.7 Problem-Solving and Critical Thinking**

Addressing challenges like authentication integration, thread synchronization, and GUI responsiveness sharpened our problem-solving skills. Iterative testing and refinement balanced security, functionality, and usability.

**12.8 Software Engineering Practices**

The modular structure (crypto\_utils.py, chat\_server.py, chat\_client.py) and rigorous testing (unit, integration, usability) reinforced best practices. We learned to design maintainable systems and document processes effectively.

These outcomes equipped us with practical skills in secure system design, preparing us for advanced projects in communication and software development.

**CHAPTER 13**

**PROJECT IMPACT**

The secure chat application significantly enhances secure communication for personal messaging, small-scale professional collaboration, and educational purposes. By integrating AES-CBC encryption and SHA-256-based authentication, it ensures message confidentiality and restricts access to authorized users. The intuitive Tkinter GUI, color-coded message logs, and persistent chat history in chat\_history.json make it accessible to non-technical users, while robust error handling guarantees reliability.

The project’s impact includes:

* **Enhanced Privacy**: Encrypted messages and secure authentication protect sensitive data, building user trust in digital communication.
* **Educational Value**: The system demonstrates , authentication, networking, and GUI design, serving as a valuable learning tool for students and developers.
* **Operational Reliability**: Thread-safe operations and comprehensive error handling ensure stable performance in real-world scenarios.
* **Cost-Effectiveness**: Developed with open-source tools, the application is ideal for low-budget deployments in small organizations or academic settings.

The modular design and robust foundation make this application a reliable and secure communication solution, with strong potential for adoption in privacy-focused environments.

**CHAPTER 14**

**CONCLUSION**

The secure chat application successfully integrates AES-CBC encryption, SHA-256-based authentication, client-server networking, and a Tkinter-based GUI to deliver a highly secure and user-friendly communication platform. AES encryption with PKCS7 padding and random IVs ensures message confidentiality, while authentication (e.g., Risers/12345678) restricts access to authorized users via users.json. TCP sockets on localhost:12345 facilitate reliable real-time messaging, and the intuitive GUI provides clear controls, status indicators, and color-coded logs (green: senders, blue: system, red: errors) for enhanced usability. Chat history persistence in chat\_history.json enables seamless conversation continuity, and robust error handling maintains system stability across various scenarios.

The project adeptly addressed challenges such as secure data transmission, user authentication, concurrency, and usability through a modular design (crypto\_utils.py, chat\_server.py, chat\_client.py) and rigorous testing. Despite the limitation of a hardcoded encryption key, the system establishes a robust foundation for advanced security enhancements. The development process fostered deep expertise in , authentication, networking, GUI design, and software engineering, equipping the team for future secure system development.

By prioritizing robust security and user-centric design, this project contributes significantly to safe digital communication, with practical applications in education, small-scale collaboration, and privacy-focused environments. It exemplifies the effective synergy of encryption, authentication, and usability in modern communication systems.

**CHAPTER 15**

**REFERENCES**

1. Stallings, W. (2017).  *and Network Security: Principles and Practice* (7th ed.). Pearson.
   * Provides a comprehensive overview of cryptographic techniques, including AES-CBC and padding schemes, guiding the project’s encryption module.
2. Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks* (5th ed.). Pearson.
   * Covers TCP socket programming and client-server architectures, informing the networking implementation on localhost:12345.
3. Lutz, M. (2013). *Learning Python* (5th ed.). O’Reilly Media.
   * A key resource for Python programming, including Tkinter for GUI development, threading for concurrency, and hashlib for SHA-256 authentication.
4. McKinney, W. (2017). *Python for Data Analysis* (2nd ed.). O’Reilly Media.
   * Discusses JSON handling and file I/O, supporting the implementation of chat\_history.json and users.json persistence.
5. Python Software Foundation. (2025). *Python Documentation: socket, threading, tkinter, json, hashlib modules*. Retrieved from https://docs.python.org/3/.
   * Official documentation for Python libraries used in encryption, authentication, networking, and GUI components.
6. Schneier, B. (1996). *Applied : Protocols, Algorithms, and Source Code in C* (2nd ed.). Wiley.
   * Offers practical insights into , influencing the design of the AES encryption and authentication mechanisms.