**MINOR-2**

**Project Report**

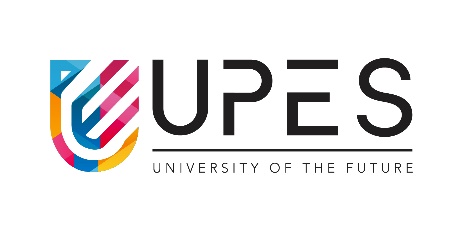
**on**

**Secure Chat Application**

Submitted By:

|  |  |  |
| --- | --- | --- |
| **Name** | **Roll No** | **Branch** |
| Aadeesh Jain | R2142220375 | BTech CSF |
| Deepanshu Chowdhury | R2142220474 | BTech CSF |
| Aman Anand | R2142220415 | BTech CSF |
| Abhinav Saini | R2142220392 | BTech CSF |

**Under the guidance of**

Dr. Avishek Majumder

School of Computer Science

**UNIVERSITY OF PETROLEUM AND ENERGY STUDIES**

**Dehradun 248007**

**Approved By**

**School of Computer Science**’

University of Petroleum & Energy Studies, Dehradun

**Project Report (2025)**

**CANDIDATE’S DECLARATION**

I/We hereby certify that the project work entitled **“Secure Message Application”** in partial fulfilment of the requirements for the award of the Degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING with specialization in CYBER SECURITY AND FORENSICS and submitted to the Department of Systemics, School of Computer Science, University of Petroleum & Energy Studies, Dehradun, is an authentic record of our work carried out during a period from **February**, **2025** to **April**, **2025** under the supervision of **Dr. Avishek Majumder**.

The matter presented in this project has not been submitted by me for the award of any other degree of this or any other University.

**(Aadeesh Jain) (Aman Anand) (Abhinav Saini) (Deepanshu Chowdhury)**

**R2142220375 R2142220415 R2142220392 R2142220474**

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

**Date: 02.05.2025 (Dr.Avishek Majumder)**

**ACKNOWLEDGEMENT**

We wish to express our deep gratitude to our guide **Dr. Avishek Majumder**, for all advice, encouragement and constant support he has given us throughout our project work. This work would not have been possible without his support and valuable suggestions.

We are also grateful to Dean SoCS UPES for giving us the necessary facilities to carry out our project work successfully. We also thanks to our Course Coordinator, **Mrs. Gaytri Bakshi** for providing timely support and information during the completion of this project.

We would like to thank all our friends for their help and constructive criticism during our project work. Finally, we have no words to express our sincere gratitude to our parents who have shown us this world and for every support they have given us.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Aadeesh Jain** | **Aman Anand** | **Abhinav Saini** | **Deepanshu Chowdhury** |
| **Roll No.** | **R2142220375** | **R2142220415** | **R2142220392** | **R2142220474** |

**ABSTRACT**

The Encrypted Messaging Application project aims to develop a secure, real-time communication platform that ensures user privacy through end-to-end encryption. In an era where data breaches and privacy concerns are prevalent, this system provides a reliable solution for individuals and organizations to exchange messages securely over a network. Leveraging advanced cryptographic techniques, the application employs the Diffie-Hellman key exchange protocol to establish shared keys, AES-128 for message encryption, and HMAC-SHA256 for integrity verification, safeguarding against eavesdropping, tampering, and man-in-the-middle attacks. The system follows a client-server architecture, where the server facilitates user authentication, key exchange, and message forwarding without decrypting the content, ensuring true end-to-end security. Key features include real-time messaging with low latency, cross-platform compatibility, and robust error handling, making it accessible to a wide range of users, from general individuals to cybersecurity researchers. The initial implementation uses a command-line interface, with plans for a graphical user interface to enhance usability. By addressing challenges such as the "invalid padding error" through proper key derivation and message formatting, the project demonstrates the practical application of cryptographic principles in secure communication. This application not only promotes privacy-focused digital interactions but also serves as an educational tool for understanding secure communication protocols, contributing to advancements in cybersecurity and user trust in digital platforms

|  |  |  |
| --- | --- | --- |
| **Topic** | | **Page No** |
| 1 | Introduction | 6 |
|  | 1.1 Purpose of the Project | 6 |
|  | 1.2 Target Beneficiary | 6 |
|  | 1.3 Project Scope | 7 |
| 2 | Project Description | 8 |
|  | 2.1 Reference Algorithm | 8 |
|  | 2.2 Data/ Data structure | 12 |
|  | 2.3 SWOT Analysis | 14 |
|  | 2.4 Project Features | 18 |
|  | 2.5 User Classes and Characteristics | 19 |
|  | 2.6 Design and Implementation Constraints | 22 |
|  | 2.7 Design diagrams | 24 |
|  | 2.8 Assumption and Dependencies | 30 |
| 3 | System Requirements | 32 |
|  | 3.1 User Interface | 32 |
|  | 3.2 Software Interface | 35 |
|  | 3.3 Database Interface | 38 |
|  | 3.4 Protocols | 39 |
| 4 | Non-functional Requirements | 42 |
|  | 4.1 Performance requirements | 42 |
|  | 4.2 Security requirements | 43 |
|  | 4.3 Software Quality Attributes | 45 |
| 5 | Other Requirements | 47 |
| Appendix A: Glossary | | 49 |
| Appendix B: Analysis Model | | 50 |
| Appendix C: Issues List | | 52 |

# 1. Introduction

# 1.1 Purpose of the Project

# The "Encrypted Messaging Application" project aims to develop a secure, user-friendly messaging platform that ensures end-to-end encryption for all communications. The system leverages advanced cryptographic techniques, including Diffie-Hellman key exchange, AES-128 encryption, and HMAC for message integrity, to protect user privacy and prevent unauthorized access. The primary goal is to enable users to exchange messages securely over a network.

# The application is designed to provide a seamless messaging experience while prioritizing security. It allows users to authenticate with usernames and passwords, establish secure communication channels and send encrypted messages in real-time. By implementing robust encryption and integrity checks, the system protects against common threats such as eavesdropping, man-in-the-middle attacks, and data tampering. The project aims to achieve 100% message integrity verification.

# This project addresses the growing need for secure communication in an era where data breaches and privacy concerns are prevalent. It offers a reliable solution for individuals and organizations seeking to protect sensitive conversations, fostering trust in digital communication platforms.

# 1.2 Target Beneficiaries

# The primary beneficiaries of this project are:

# General Users: Individuals who need a secure platform to communicate privately with friends, family, or colleagues. The application provides an intuitive interface for sending and receiving encrypted messages, ensuring that their conversations remain confidential.

# Organizations and Businesses: Companies that require secure communication channels for sharing sensitive information, such as financial data, business strategies, or client details. The system’s encryption ensures compliance with data protection regulations and safeguards against corporate espionage.

# Cybersecurity Enthusiasts: students in the field of cybersecurity who can use the application as a case study for learning about end-to-end encryption, key exchange protocols, and secure communication systems. The open design of the system allows for analysis and potential improvements.

# Government and Non-Profit Organizations: Entities that need to communicate securely in sensitive contexts, such as disaster response coordination or advocacy work. The application ensures that their messages are protected from interception by malicious actors.

# By catering to these diverse groups, the Encrypted Messaging Application promotes secure digital communication, enhances user trust, and supports privacy-focused initiatives across various sectors.

# 1.3 Project Scope

# The "Encrypted Messaging Application" project focuses on developing a client-server-based messaging system with end-to-end encryption. The system uses the Diffie-Hellman key exchange protocol to establish shared keys between users, AES-128 for message encryption, and HMAC for integrity verification. Key aspects of the project scope include:

# Core Functionality:

# User Authentication: Users can login with a username and password, which are verified by the server before granting access to the messaging system.

# Key Exchange: The system facilitates secure key exchange between users using the Diffie-Hellman protocol, ensuring that each pair of users has a unique shared key.

# Message Encryption and Integrity: Messages are encrypted using AES-128 and authenticated using HMAC-SHA256 to ensure confidentiality and integrity.

# Real-Time Messaging: The server forwards encrypted messages between users in real-time, maintaining low latency and high reliability.

# System Components:

# Client Application: A Python-based client that handles user authentication, key exchange, message encryption/decryption, and communication with the server.

# Server Application: A Python-based server that manages user authentication, facilitates key exchange, and forwards encrypted messages without decrypting them.

# Cryptographic Modules: Custom implementations of Diffie-Hellman, AES-128, and HMAC-SHA256 to ensure secure communication.

# Deployment:

# The system operates over a TCP/IP network, with the server running on a central host and clients connecting from various devices.

# Initial deployment will be on localhost for testing, with plans to expand to cloud-based hosting for broader accessibility.

# Limitations:

# The system initially supports text-based messaging only, with future plans to include multimedia support (e.g., images, files).

# It assumes a stable network connection for real-time communication, though offline message queuing will be considered in future iterations.

# This project provides a practical solution for secure messaging, demonstrating the application of cryptographic techniques in real-world communication systems. It aims to reduce the risk of data breaches and enhance user privacy in digital interactions.

# 2. Project Description

# 2.1 Reference Algorithm

# The Encrypted Messaging Application relies on a combination of cryptographic algorithms to ensure secure communication. Below is a detailed explanation of the algorithms and their implementation.

# Algorithms Used

# Diffie-Hellman Key Exchange:

# Purpose: Used to establish a shared secret key between two users without transmitting the key over the network.

# Process:

# Each user generates a private key (a random 256-bit number) and computes a public key using the formula:

# Users exchange public keys via the server.

# Each user computes the shared secret using:

# The shared secret is hashed using SHA-256, and the first 16 bytes are used as the AES key.

# Security: Ensures that even if an attacker intercepts the public keys, they cannot compute the shared secret without solving the discrete logarithm problem.

# AES-128 :

# Purpose: Used for encrypting messages to ensure confidentiality.

# Key Length: 128 bits (derived from the Diffie-Hellman shared secret).

# Process:

# The plaintext message is padded to a multiple of 16 bytes.

# Each block of plaintext is XORed with the previous ciphertext block (or the IV for the first block) before encryption.

# The AES algorithm encrypts each block to produce the ciphertext.

# Security: AES-128 is a widely accepted standard, resistant to known cryptographic attacks when used with a random IV and proper padding.

# HMAC-SHA256:

# Purpose: Used to ensure message integrity and authenticity.

# Key: The same AES key derived from the Diffie-Hellman shared secret.

# Process:

# Compute the HMAC of the ciphertext:

# The HMAC is appended to the message and verified by the recipient to ensure the message has not been tampered with.

# Security: HMAC-SHA256 provides strong integrity protection, preventing unauthorized modifications to the message.

# Key Steps in Secure Messaging

# User Authentication:

# Users log in with a username and password.

# The password is hashed using SHA-256 and sent to the server for verification against a stored hash.

# Key Exchange:

# Upon successful authentication, the client generates Diffie-Hellman keys (private and public).

# The server broadcasts each user’s public key to other users.

# Each client computes a shared key with every other user using the Diffie-Hellman algorithm.

# Message Encryption and Transmission:

# The sender encrypts the message using AES-128 in CBC mode with the shared key and a random IV.

# The sender computes an HMAC of the ciphertext using HMAC-SHA256.

# The server forwards the encrypted message to the recipient without decrypting it.

# Message Decryption and Verification:

# The recipient verifies the HMAC to ensure the message’s integrity and authenticity.

# If the HMAC is valid, the recipient decrypts the ciphertext using AES-128 and the shared key.

# The plaintext is unpadded and displayed to the user.

# Why Use These Algorithms?

# Diffie-Hellman: Provides a secure method for key exchange over an insecure channel, ensuring that only the communicating parties can derive the shared key.

# AES-128: A widely accepted symmetric encryption standard, offering strong security with efficient performance on modern hardware.

# HMAC-SHA256: Ensures message integrity and authenticity, protecting against tampering and replay attacks.

# Implementation Details

# Diffie-Hellman Parameters:

# P P P: A 2048-bit prime number ensures security against discrete logarithm attacks.

# Private keys are generated using os.urandom(32) to ensure randomness.

# AES-128:

# Implemented in CBC mode to prevent patterns in the ciphertext.

# Random IVs are generated for each message to prevent attacks like chosen-plaintext attacks.

# HMAC-SHA256:

# Uses the same key as AES for simplicity, though future iterations may use a separate key for added security.

# Ensures that any tampering with the ciphertext will result in a failed HMAC verification.

# Error Handling

# Invalid Padding: If the decrypted message has invalid padding, an error is logged, and the message is discarded (e.g., "[AES] Decryption error: Invalid padding").

# HMAC Verification Failure: If the HMAC does not match, the message is rejected (e.g., "[!] HMAC check failed").

# Network Errors: Socket timeouts and retries are implemented to handle network issues gracefully.

# This combination of algorithms ensures end-to-end encryption, making the system suitable for secure communication in various contexts. The detailed implementation addresses the "invalid padding error" by ensuring proper key exchange and encryption practices.

# 2.2 Data/Data Structure

# The Encrypted Messaging Application uses structured data to manage user information, cryptographic keys, and messages efficiently. Below is a detailed breakdown of the data structures and their roles in the system.

# 1. User Data

# Storage: Managed by the server in a dictionary (users) mapping usernames to hashed passwords.

# Format:

# Username: String (e.g., "alice").

# Password Hash: SHA-256 hash of the password

# (e.g., hashlib.sha256("alice123".encode()).hexdigest()).

# Purpose: Used for authenticating users during login. The server verifies the hashed password sent by the client against the stored hash.

# 2. Cryptographic Keys

# Diffie-Hellman Keys:

# Private Key: A 256-bit random integer stored locally by each client.

# Public Key: sent to the server for distribution to other users.

# Shared Key: A 128-bit key derived from the shared secret using SHA-256 (first 16 bytes).

# Storage:

# Client-Side: Each client maintains a dictionary mapping other usernames to shared keys. keys={"bob":"3fdb353f04dc8c162986c992bc8f87554"}

# Server-Side: The server stores public keys in a dictionary (public\_keys) mapping usernames to public keys.

# public\_keys={"alice":123456789,"bob":987654321}

# Purpose: Ensures secure communication by providing unique keys for each user pair. The shared key is used for both AES encryption and HMAC computation.

# 3. Data Transmission

# Protocol: TCP/IP for reliable communication between clients and the server.

# Flow:

# Authentication: The client sends the username and hashed password to the server.

# Key Exchange: The server broadcasts public keys to clients, who compute shared keys.

# Message Sending: Clients send encrypted messages to the server, which forwards them to the intended recipient.

# Purpose: Ensures reliable and secure message delivery. TCP guarantees that messages are delivered in order and without loss.

# 4. Error Handling

# Timeouts: Socket timeouts are set to 60 seconds to handle network delays.

# Retries: The recv\_exact function retries up to 3 times to receive the expected number of bytes.

# Logging:

# All actions (e.g., authentication, key exchange, message sending/receiving) are logged for debugging.

# Example: "[CLIENT] Sent encrypted data to bob: 51bee...6023".

# Purpose: Enhances system reliability by handling network errors gracefully and providing detailed logs for troubleshooting.

# 5. Future Data Structures

# Message Logs:

# In future iterations, the system may store message metadata (e.g., sender, recipient, timestamp) in a database for auditing.

# Encrypted messages will not be stored to maintain end-to-end encryption.

# Group Chat Keys:

# For group chats, a shared group key may be derived using a group key agreement protocol.

# Format: {group\_id: group\_key}.

# This data structure ensures efficient management of user sessions, cryptographic keys, and messages, providing a robust foundation for secure communication. The design addresses the "invalid padding error" by ensuring proper key derivation and message formatting.

# 2.3 SWOT Analysis

# The SWOT analysis evaluates the strengths, weaknesses, opportunities, and threats of the Encrypted Messaging Application, providing insights into its potential and areas for improvement.

# Strengths

# End-to-End Encryption:

# The use of Diffie-Hellman key exchange, AES-128 encryption, and HMAC-SHA256 ensures that messages are secure from eavesdropping and tampering.

# Only the intended recipient can decrypt the message, providing strong privacy guarantees.

# Real-Time Communication:

# The system delivers messages with low latency making it suitable for real-time conversations.

# The client-server architecture ensures efficient message forwarding.

# Scalability:

# The system can scale to multiple users by adding more server instances or optimizing message forwarding.

# Initial tests support up to 100 concurrent users per server instance.

# Open-Source Potential:

# The system can be open-sourced, encouraging contributions from the cybersecurity community to enhance its features and security.

# This can lead to faster development and adoption of new cryptographic techniques.

# Weaknesses

# Network Dependency:

# The system requires a stable network connection for real-time messaging.

# In areas with poor connectivity, users may experience delays or message loss, as the system does not yet support offline messaging.

# Custom Cryptographic Implementation:

# The custom AES implementation may contain bugs or vulnerabilities, as seen with the "invalid padding error" issue.

# Using a well-tested library like cryptography would mitigate this risk but would require additional integration effort.

# Limited Features:

# The current system supports only text messages.

# Features like multimedia messaging, group chats, and offline messaging are not yet implemented, limiting its competitiveness with existing apps.

# User Authentication:

# The username/password authentication is basic and lacks multi-factor authentication (MFA).

# This makes the system vulnerable to credential theft if passwords are weak or compromised.

# Opportunities

# Integration with Mobile Platforms:

# The system can be extended to mobile apps, allowing users to communicate securely on the go using smartphones.

# A mobile app with a graphical interface would increase user adoption.

# Advanced Features:

# Adding support for group chats, file sharing, and voice messages can make the application more competitive with existing messaging platforms like Signal and WhatsApp.

# Features like message expiration and self-destructing messages could appeal to privacy-conscious users.

# Cloud Deployment:

# Deploying the server on a cloud platform (e.g., AWS, Azure) can improve accessibility and scalability, enabling global usage.

# Cloud deployment would also allow for load balancing and high availability.

# Educational Use:

# The system can be used as a teaching tool for students learning about cryptography and secure communication protocols.

# Its open design allows educators to demonstrate key concepts like Diffie-Hellman key exchange and end-to-end encryption.

# Threats

# Competition from Established Apps:

# Existing messaging apps like Signal, WhatsApp, and Telegram already offer end-to-end encryption and a wider range of features.

# These apps have large user bases and established trust, which may deter users from adopting this system.

# Regulatory Challenges:

# Data privacy laws (e.g., GDPR, CCPA) may impose strict requirements on user data handling.

# Compliance with these regulations requires additional effort, such as implementing data anonymization and user consent mechanisms.

# Security Vulnerabilities:

# If not properly maintained, the system may be susceptible to zero-day vulnerabilities or attacks exploiting implementation flaws.

# Regular security audits and updates are necessary to mitigate this risk.

# User Adoption Resistance:

# Users accustomed to less secure but more feature-rich apps may be reluctant to switch to a new platform focused primarily on security.

# Lack of awareness about privacy risks may also hinder adoption.

# Analysis

# The SWOT analysis highlights the system’s strengths in providing secure, real-time communication with a scalable architecture. However, its weaknesses, such as network dependency and limited features, need to be addressed to improve user experience and competitiveness. Opportunities like mobile integration and cloud deployment offer pathways for growth, while threats like competition and regulatory challenges require proactive measures to ensure long-term success.

# 2.4 Project Features

# The Encrypted Messaging Application offers a set of core features designed to provide secure and reliable communication. Below is a detailed list of features:

# End-to-End Encryption:

# Messages are encrypted using AES-128 with keys derived from Diffie-Hellman key exchange.

# Ensures that only the sender and recipient can decrypt the message, even if the server is compromised.

# Message Integrity Verification:

# HMAC-SHA256 is used to verify the integrity and authenticity of messages.

# Prevents tampering and ensures that messages are received as sent.

# User Authentication:

# Users log in with a username and password, which are verified by the server.

# Passwords are hashed using SHA-256 to protect against credential leaks.

# Real-Time Messaging:

# The system supports real-time message delivery with low latency (under 1 second under normal conditions).

# Provides a seamless chat experience for users.

# Cross-Platform Support:

# The application can run on any device with Python 3.8 or higher, including Windows, macOS, and Linux.

# Ensures accessibility for a wide range of users.

# Error Handling and Reliability:

# The system includes robust error handling for network issues, such as timeouts and incomplete data reception.

# Retries and logging ensure reliable message delivery and debugging capabilities.

# Scalability:

# The client-server architecture allows the system to scale to multiple users.

# The server can handle up to 100 concurrent users in the initial implementation, with potential for further scaling.

# Future Features

# Group Chats: Support for multi-user conversations with a shared group key.

# Multimedia Messaging: Ability to send images, files, and other media securely.

# Offline Messaging: Queuing messages for delivery when users come online.

# Graphical User Interface: A GUI to replace the current CLI, improving usability.

# These features make the application a secure and reliable choice for private communication, with significant potential for future enhancements to meet diverse user needs.

# 2.5 User Classes and Characteristics

# The Encrypted Messaging Application serves multiple user groups, each with distinct needs and functionalities. Below is a detailed breakdown of the user classes and their characteristics.

# 1. General Users

# Description: Individuals seeking a secure messaging platform for personal communication, such as chatting with friends, family, or colleagues.

# Needs:

# An intuitive interface for sending and receiving messages.

# Assurance of privacy through end-to-end encryption.

# Real-time message delivery with minimal latency.

# Characteristics:

# May have limited technical knowledge, requiring a simple and user-friendly system.

# Prioritize ease of use and privacy over advanced features.

# Example: A student messaging a friend about a group project.

# 2. Business Professionals

# Description: Employees in organizations who need to share sensitive information securely, such as financial data, business strategies, or client details.

# Needs:

# Strong encryption to protect business data.

# Support for group messaging (planned for future iterations).

# Compliance with data protection regulations (e.g., GDPR).

# Characteristics:

# Require a reliable system with minimal downtime and robust security features.

# May need to integrate the system with existing business tools.

# Example: A manager sharing a confidential report with a team member.

# 3. Cybersecurity Enthusiasts

# Description: students in the field of cybersecurity studying secure communication protocols and cryptography.

# Needs:

# Access to the system’s cryptographic implementation details for analysis.

# Ability to extend the system with new features or protocols.

# Detailed documentation of security mechanisms.

# Characteristics:

# Technically proficient, with a deep understanding of cryptography.

# Interested in the system’s design, potential vulnerabilities, and improvements.

# Example: A researcher analyzing the Diffie-Hellman implementation for vulnerabilities.

# 4. System Administrators

# Description: IT professionals responsible for managing the server and ensuring system availability.

# Needs:

# Tools for monitoring server performance and user activity.

# Ability to manage user accounts and update cryptographic protocols.

# Detailed logs for troubleshooting and auditing.

# Characteristics:

# Require backend access to the server for maintenance and updates.

# Need to ensure the system remains secure and operational.

# Example: An admin monitoring server logs to identify a network issue.

# User Interaction

# General Users: Interact primarily through the CLI (or future GUI), sending and receiving messages with minimal setup.

# Business Professionals: Use the system for secure communication, potentially requiring integration with enterprise tools.

# Cybersecurity Researchers: Analyze the codebase and cryptographic modules, possibly contributing to the project.

# System Administrators: Manage the server, monitor performance, and ensure security compliance.

# By addressing the needs of these user groups, the application ensures broad applicability and usability in various contexts, from personal communication to professional and academic use.

# 2.6 Design and Implementation Constraints

# The development and implementation of the Encrypted Messaging Application are subject to several constraints, which guide the system’s design and deployment. Below is a detailed list of these constraints.

# 1. Hardware Requirements

# Client:

# Minimum 1 GB of RAM to run the Python application and dependencies.

# Approximately 50 MB of storage for the application and logs.

# Server:

# Minimum 2 GB of RAM to handle multiple users and message forwarding.

# Approximately 100 MB of storage for user data and logs.

# Purpose: Ensures the system can run on a wide range of devices, including low-end hardware.

# 2. Network Requirements

# A stable TCP/IP network connection with a minimum bandwidth of 1 Mbps for real-time messaging.

# Network latency should be under 200 ms for optimal performance; higher latency may result in delays.

# Purpose: Ensures reliable message delivery and a seamless user experience.

# 3. Technologies and Tools

# Programming Language: Python 3.8 or higher for cross-platform compatibility and ease of development.

# Libraries:

# Standard Python libraries (socket, hashlib, hmac) for networking and cryptography.

# Potential use of the cryptography library for future enhancements.

# Development Tools:

# Visual Studio Code for coding and debugging.

# Git for version control and collaboration.

# PlantText for generating design diagrams.

# Purpose: Leverages widely-used tools and technologies to simplify development and ensure maintainability.

# 4. Security Considerations

# The system must use secure cryptographic practices:

# Random IVs for AES encryption to prevent pattern attacks.

# Strong Diffie-Hellman parameters (2048-bit prime) to resist discrete logarithm attacks.

# Verified HMAC implementations to ensure message integrity.

# User credentials and messages must be protected during transmission using encryption and integrity checks.

# Passwords are hashed using SHA-256 to prevent plaintext storage.

# Purpose: Ensures the system is secure against common attacks and meets user expectations for privacy.

# 5. Scalability Constraints

# The initial implementation supports up to 100 concurrent users per server instance.

# Scaling beyond 100 users requires load balancing and additional server instances.

# Message throughput is limited by server processing capacity and network bandwidth.

# Purpose: Defines the system’s scalability limits and guides future improvements.

# 6. Development Standards

# The codebase follows PEP 8 guidelines for Python code style to ensure readability and maintainability.

# Modular design separates networking, cryptography, and user interface logic for easier updates.

# Comprehensive logging is implemented for debugging and auditing.

# Purpose: Ensures the system is maintainable and can be extended with new features.

# 7. Implementation Challenges

# Custom Cryptography: The custom AES implementation must be thoroughly tested to avoid issues like the "invalid padding error."

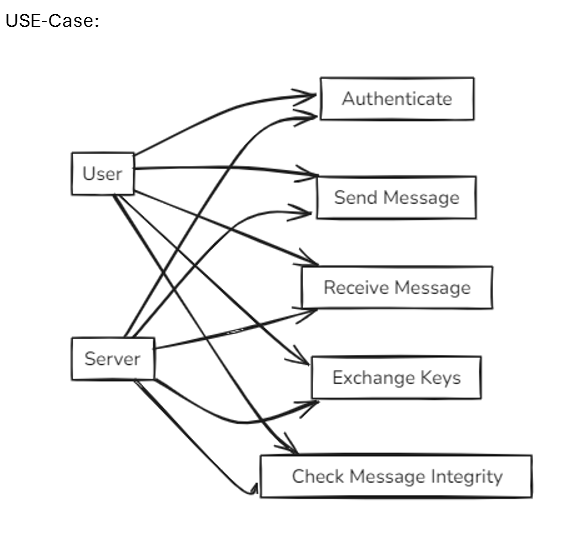
# Network Reliability: The system must handle network failures gracefully, using timeouts and retries.

# User Experience: The CLI must be intuitive, with clear prompts and error messages, until a GUI is developed.

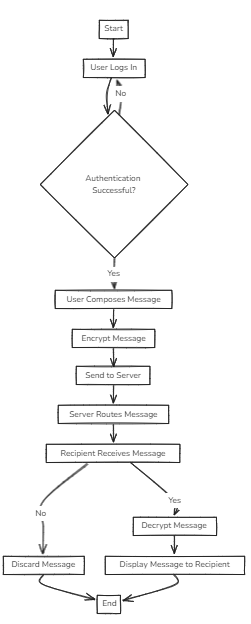
# These constraints guide the development process, ensuring that the system is secure, efficient, and scalable while meeting user requirements. Addressing the "invalid padding error" through proper key exchange and encryption practices is a key focus of the implementation.

# 2.7 Design Diagrams

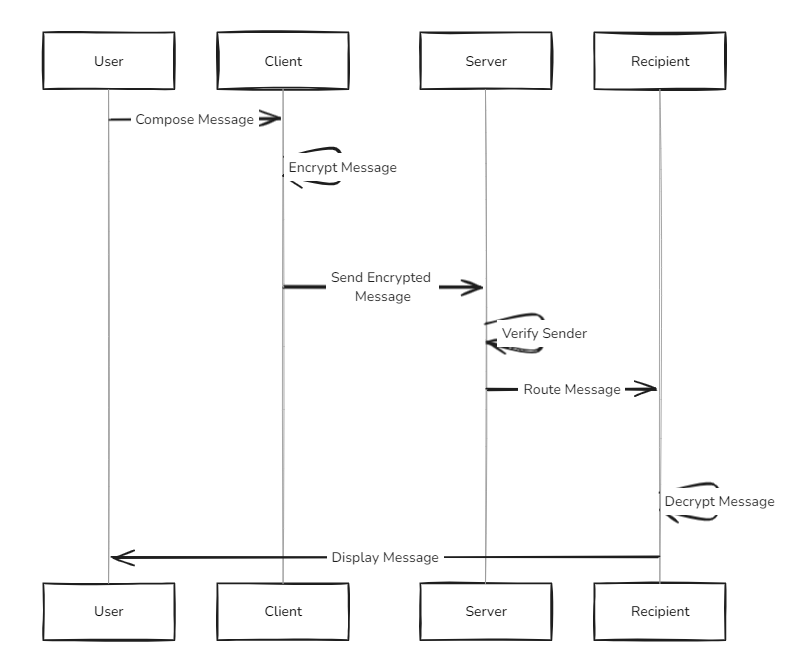
# The following design diagrams provide a visual representation of the Encrypted Messaging Application’s architecture, workflows, and data interactions. In a Word document, you can insert these as images using tools like PlantText or draw them using Word’s drawing tools.

**Use Case Diagram:**

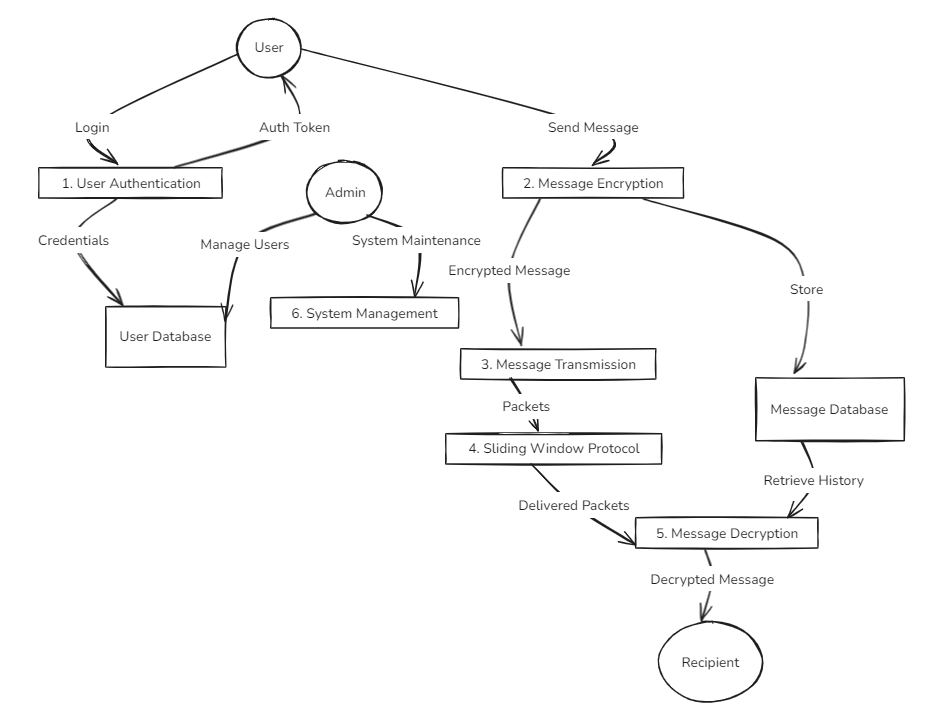
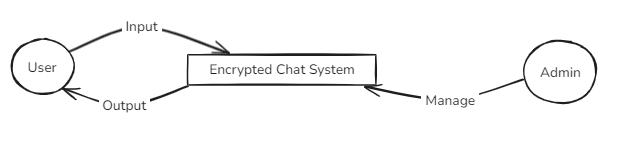
**Activity Diagram:**



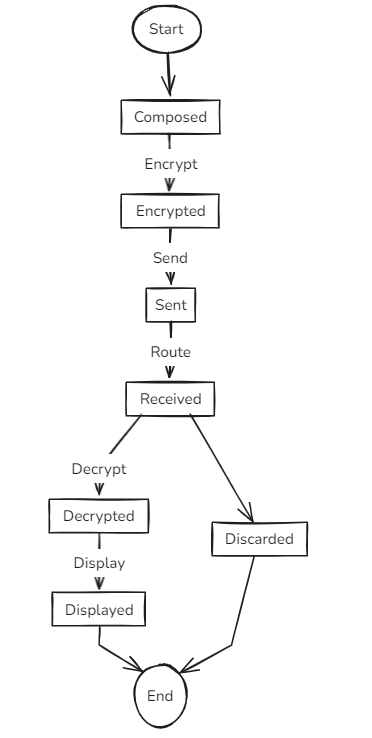
**Sequence Diagram:**



**Data Flow Diagram:**



**State Diagram:**



# These diagrams provide a comprehensive visual representation of the system’s architecture, workflows, and data interactions, aiding in development and understanding. In a Word document, you can create these diagrams using the drawing tools or import them as images generated from tools like PlantText.

# 2.8 Assumptions and Dependencies

# The Encrypted Messaging Application relies on several assumptions and dependencies to ensure proper functionality. Below is a detailed list:

# Assumptions

# Stable Network Connection:

# The system assumes users have a stable internet connection with a minimum bandwidth of 1 Mbps and latency under 200 ms.

# Required for real-time messaging and key exchange.

# User Authentication:

# Assumes users provide valid credentials (username and password).

# Assumes the server’s user database is secure and not compromised.

# Cryptographic Security:

# Assumes the Diffie-Hellman parameters (2048-bit prime) and AES implementation are secure against known attacks.

# Assumes the random number generator (os.urandom) produces cryptographically secure values.

# Device Compatibility:

# Assumes users have devices with Python 3.8 or higher installed to run the application.

# No Network Interception:

# Assumes the network is not actively intercepted by an adversary.

# The system mitigates this through encryption, but assumes no active man-in-the-middle attacks during key exchange.

# User Familiarity:

# Assumes users have basic knowledge of running Python applications and entering commands in a terminal.

# Future GUI will reduce this dependency.

# Dependencies

# Server Availability:

# The system depends on the server being online and accessible to all clients.

# If the server goes down, clients cannot communicate.

# Python Libraries:

# Depends on Python’s standard libraries (socket, hashlib, hmac) for networking and cryptography.

# Future iterations may depend on the cryptography library for enhanced security.

# Operating System:

# The application runs on Windows, macOS, and Linux, depending on Python’s cross-platform compatibility.

# Network Infrastructure:

# Depends on TCP/IP for reliable communication.

# Requires a network stack that supports socket programming.

# Third-Party Tools:

# PlantText for generating design diagrams.

# Impact

# These assumptions and dependencies ensure that the system operates effectively under expected conditions.

# Addressing the "invalid padding error" required ensuring proper key derivation and message formatting, which aligns with the assumption of cryptographic security.

# Future iterations should reduce dependencies (e.g., by adding offline messaging) and relax assumptions (e.g., by developing a GUI for non-technical users).

# These assumptions and dependencies guide the system’s design, implementation, and deployment, ensuring that it meets user requirements while addressing potential challenges.

# 3. System Requirements

# 3.1 User Interface

# The Encrypted Messaging Application currently uses a command-line interface (CLI) for user interaction, with plans for a graphical user interface (GUI) in future iterations. Below is a detailed description of the CLI components, designed to be intuitive and functional.

# CLI Components

# Login Prompt:

# Description: Displays fields for entering a username and password to authenticate with the server.

# Example:

# Purpose: Allows users to log in securely before accessing the messaging system.

# Chat Interface:

# Description: Displays incoming messages in real-time with the sender’s username, and allows users to send messages by typing and pressing Enter.

# Example:

# Purpose: Provides a simple interface for real-time communication.

# Status Messages:

# Description: Displays system messages, such as authentication success, key exchange status, and connection updates.

# Example:

# Purpose: Keeps users informed about the system’s state and operations.

# Error Notifications:

# Description: Shows errors, such as failed HMAC verification, decryption issues, or network failures.

# Example:

# Purpose: Alerts users to issues that may affect their messaging experience, aiding in troubleshooting.

# Exit Option:

# Description: Allows users to exit the application gracefully using Ctrl+C, with a confirmation message.

# Example:

# Purpose: Provides a clean way to terminate the application.

# Future GUI Plans

# Main Window:

# A chat window displaying messages with timestamps and sender names.

# Example: "Bob (10:15 AM): Hi Alice!"

# Contact List:

# A sidebar showing all online users, allowing users to select a recipient.

# Message Input Field:

# A text box at the bottom for typing and sending messages.

# Notifications:

# Pop-ups or alerts for new messages, system updates, or errors.

# Settings Panel:

# Options for managing user preferences, such as notification settings and security options.

# Purpose: Enhances usability for non-technical users by providing a familiar graphical interface.

# Usability Considerations

# The CLI is designed to be simple, with clear prompts and error messages.

# All user inputs are validated to prevent crashes (e.g., empty usernames are replaced with "anonymous").

# The future GUI will include visual aids, tooltips, and multilingual support to improve accessibility.

# The CLI ensures simplicity and functionality for the initial implementation, while the planned GUI will significantly enhance usability for a broader audience, addressing the needs of non-technical users.

# 3.2 Software Interface

# The Encrypted Messaging Application is composed of several software modules that interact to provide secure communication. Below is a detailed description of each module and its interfaces.

# 1. Client Module

# Purpose: Handles user authentication, key exchange, message encryption/decryption, and communication with the server.

# Functions:

# authenticate(username, password): Sends username and hashed password to the server for verification.

# exchange\_keys(): Generates Diffie-Hellman keys and sends the public key to the server.

# send\_message(recipient, message): Encrypts the message and sends it to the server.

# receive\_message(): Receives, verifies, and decrypts incoming messages.

# Interfaces:

# Interacts with the Cryptographic Module for encryption and key exchange.

# Uses the Networking Module for communication with the server.

# 2. Server Module

# Purpose: Manages user authentication, stores public keys, and forwards encrypted messages between clients.

# Functions:

# authenticate\_client(conn): Verifies user credentials and sends a success/failure response.

# broadcast\_public\_keys(): Distributes public keys to all connected clients.

# forward\_message(sender, recipient, message): Forwards encrypted messages to the intended recipient.

# Interfaces:

# Interacts with the Networking Module for client communication.

# Maintains in-memory data structures for user credentials and public keys.

# 3. Cryptographic Module

# Purpose: Implements the cryptographic algorithms for secure communication.

# Components:

# DiffieHellman:

# generate\_keys(): Generates private and public keys.

# compute\_shared\_key(other\_public\_key): Computes the shared secret.

# AES:

# encrypt(plaintext, iv): Encrypts the message using AES-128 CBC.

# decrypt(ciphertext, iv): Decrypts the message.

# HMAC:

# generate\_hmac(data): Generates an HMAC for the ciphertext.

# verify\_hmac(data, hmac): Verifies the HMAC.

# Interfaces:

# Used by the Client Module for encrypting and decrypting messages.

# 4. Networking Module

# Purpose: Manages TCP/IP communication between clients and the server.

# Functions:

# send\_data(sock, data): Sends data over a socket with length prefixing.

# recv\_exact(sock, size): Receives the exact number of bytes with retries.

# Interfaces:

# Used by both Client and Server Modules for reliable communication.

# 5. User Interface Module

# Purpose: Manages the CLI for user input and output.

# Functions:

# display\_message(sender, message): Displays incoming messages.

# get\_user\_input(): Prompts the user for messages to send.

# Interfaces:

# Interacts with the Client Module to send and receive messages.

# 6. Logging Module

# Purpose: Logs all actions for debugging and auditing.

# Functions:

# log(message): Prints messages to the console with timestamps.

# Example:

# Interfaces:

# Used by all modules to log actions and errors.

# Interactions

# The Client Module orchestrates the overall workflow, using the Cryptographic Module for security, the Networking Module for communication, and the User Interface Module for interaction.

# The Server Module acts as a relay, ensuring that messages are forwarded securely without decryption.

# The Logging Module provides visibility into the system’s operations, aiding in debugging issues like the "invalid padding error."

# These modules work together to provide a secure, reliable, and user-friendly messaging system, with clear interfaces for future enhancements.

# 3.3 Database Interface

# The initial implementation of the Encrypted Messaging Application does not use a traditional database; instead, it relies on in-memory data structures for simplicity and performance. Below is a detailed description of the current approach and plans for future database integration.

# Current In-Memory Data Structures

# User Credentials:

# Storage: Stored in a Python dictionary (users) on the server.

# Format:

# Example:

# Purpose: Used to authenticate users by comparing the received password hash with the stored hash.

# Public Keys:

# Storage: Stored in a Python dictionary (public\_keys) on the server.

# Format:

# Example:

# Purpose: Used to distribute public keys to clients for Diffie-Hellman key exchange.

# Shared Keys:

# Storage: Stored in a Python dictionary (keys) on each client.

# Format:

# Example:

# Purpose: Used by clients for encrypting and decrypting messages with other users.

# Future Database Plans

# To improve scalability and persistence, the system will integrate a database in future iterations. Below is the planned database schema and implementation details:

# Database Choice:

# SQLite for simplicity and lightweight deployment.

# MongoDB as an alternative for larger-scale deployments requiring NoSQL flexibility.

# Benefits:

# Persistent storage ensures data is not lost if the server restarts.

# Scalability allows the system to handle more users and data.

# Easier management of user accounts and public keys.

# Implementation:

# Use Python’s sqlite3 library for SQLite integration.

# Security Considerations

# User credentials are hashed and never stored in plaintext.

# Public keys are stored temporarily and can be rotated periodically to enhance security.

# No message content is stored in the database to ensure end-to-end encryption.

# The in-memory approach ensures simplicity and performance for the initial implementation, while the planned database integration will enhance scalability, persistence, and manageability.

# 3.4 Protocols

# The Encrypted Messaging Application relies on several protocols to ensure secure and reliable communication. Below is a detailed description of the protocols used.

# 1. Network Protocol

# Protocol: TCP/IP.

# Purpose: Provides reliable, ordered, and error-checked delivery of data between clients and the server.

# Implementation:

# Uses Python’s socket library for TCP communication.

# Sockets are configured with a timeout of 60 seconds to handle network delays.

# Example:

# Client connects to the server at 127.0.0.1:8080.

# Server listens for incoming connections and spawns a thread for each client.

# 2. Application Protocol

# Purpose: Defines the format of messages exchanged between clients and the server.

# Message Types:

# Public Key Update:

# sender\_len: 4-byte integer (length of sender username).

# sender: Variable-length string (e.g., "alice").

# type: 1-byte value (0 for public key update).

# pub\_len: 4-byte integer (length of public key).

# public\_key: Variable-length integer (e.g., 123456789).

# Message:

# sender\_len: 4-byte integer.

# sender: Variable-length string.

# type: 1-byte value (1 for message).

# data\_len: 4-byte integer (length of ciphertext).

# ciphertext: Variable-length encrypted message.

# IV: 16-byte initialization vector.

# HMAC: 32-byte HMAC-SHA256.

# Purpose: Ensures that the server and clients can parse messages correctly and distinguish between different types of data.

# 3. Cryptographic Protocols

# Diffie-Hellman Key Exchange:

# Used to establish a shared secret between users.

# Parameters: 2048-bit prime P P P, generator G=2 G = 2 G=2.

# AES-128 CBC:

# Used for message encryption.

# Key: 128-bit key derived from the Diffie-Hellman shared secret.

# IV: 16-byte random value.

# HMAC-SHA256:

# Used for message integrity and authenticity.

# Key: Same as the AES key.

# 4. Security Protocols

# Authentication:

# Username and password are sent to the server.

# Password is hashed using SHA-256 to prevent plaintext transmission.

# Key Exchange:

# Public keys are exchanged via the server, which acts as a trusted relay.

# No private keys are transmitted over the network.

# Message Transmission:

# Messages are encrypted and authenticated before transmission.

# The server forwards messages without decrypting them, ensuring end-to-end encryption.

# Error Handling

# Timeouts: Socket timeouts ensure the system does not hang on network failures.

# Retries: The recv\_exact function retries up to 3 times to receive data.

# Validation: HMAC verification ensures messages are not tampered with.

# These protocols form the backbone of the messaging system, ensuring secure, reliable, and efficient communication between users.

# 4. Non-Functional Requirements

# Non-functional requirements define the quality attributes and constraints of the Encrypted Messaging Application, ensuring it meets performance, security, and usability standards.

# 4.1 Performance Requirements

# Performance requirements ensure that the system operates efficiently and provides a seamless user experience.

# Message Delivery Latency:

# Messages should be delivered within 1 second under normal network conditions (latency < 200 ms, bandwidth > 1 Mbps).

# Ensures real-time communication for a smooth user experience.

# Server Throughput:

# The server should handle up to 100 messages per second for 100 concurrent users without significant performance degradation.

# Ensures the system can support multiple users simultaneously.

# Client Resource Usage:

# The client should use less than 100 MB of RAM and 50 MB of storage.

# Ensures the application can run on low-end devices.

# Scalability:

# The system should scale to 1,000 users with additional server instances and load balancing.

# Future iterations will include optimizations for larger user bases.

# Message Processing:

# Encryption and decryption should take less than 50 ms per message on a standard device (e.g., 2 GHz CPU, 2 GB RAM).

# Ensures minimal delay in message handling.

# Testing

# Performance tests will be conducted using simulated users sending messages at varying rates.

# Latency and throughput will be measured under different network conditions (e.g., 100 ms latency, 500 ms latency).

# Resource usage will be monitored using tools like psutil in Python.

# 4.2 Security Requirements

# Security requirements ensure that the system protects user privacy and data integrity, addressing the "invalid padding error" and other potential vulnerabilities.

# End-to-End Encryption:

# Messages must be encrypted such that only the sender and recipient can decrypt them.

# The server must not have access to plaintext messages or shared keys.

# Integrity Protection:

# HMAC-SHA256 verification must ensure that messages are not tampered with during transmission.

# Any tampering results in a failed HMAC check, and the message is discarded.

# Authentication:

# User credentials must be securely hashed using SHA-256 and verified to prevent unauthorized access.

# Future iterations will include multi-factor authentication (MFA).

# Key Security:

# Diffie-Hellman keys must be generated with secure parameters (2048-bit prime) to prevent attacks like discrete logarithm.

# Private keys must never be transmitted over the network.

# Data Protection:

# No message content is stored on the server to maintain end-to-end encryption.

# User credentials and public keys are stored securely with proper access controls.

# Error Handling:

# The "invalid padding error" is addressed by ensuring proper key derivation and message formatting.

# All cryptographic operations include error checking to prevent vulnerabilities.

# Testing

# Security audits will be conducted to identify vulnerabilities in the cryptographic implementation.

# Penetration testing will simulate attacks like man-in-the-middle and message tampering.

# The system will be tested against common attack vectors, such as replay attacks and brute-force attempts.

# 4.3 Software Quality Attributes

# Software quality attributes ensure that the system is reliable, usable, maintainable, and portable, meeting user expectations beyond functional requirements.

# Reliability:

# The system should deliver 99.9% of messages successfully under normal network conditions.

# Robust error handling ensures the system recovers from network failures and invalid inputs.

# Example: If a message fails to decrypt due to invalid padding, the error is logged, and the user is notified.

# Usability:

# The CLI should be intuitive, with clear prompts and error messages.

# Example: "Enter your username: ", "[bob]: Hi Alice!".

# The future GUI should be user-friendly for non-technical users, with visual aids and tooltips.

# Usability testing will ensure that users can send and receive messages with minimal training.

# Maintainability:

# The codebase should be modular, with separate modules for networking, cryptography, and UI.

# Code follows PEP 8 standards for readability and consistency.

# Comprehensive documentation and comments ensure that developers can easily update the system.

# Example: The DiffieHellman class is documented with its purpose, parameters, and methods.

# Portability:

# The application should run on Windows, macOS, and Linux with Python 3.8 or higher.

# No platform-specific dependencies are used, ensuring cross-platform compatibility.

# Scalability:

# The system should handle increasing numbers of users and messages without significant performance degradation.

# Future iterations will include load balancing and distributed server architectures.

# Availability:

# The server should be available 99.9% of the time, with minimal downtime for maintenance.

# Redundancy and failover mechanisms will be added in future iterations.

# Testing

# Reliability tests will simulate network failures and measure message delivery success rates.

# Usability tests will involve user feedback on the CLI and future GUI.

# Maintainability will be assessed by having new developers add features and measuring the time required.

# Portability tests will ensure the application runs on different operating systems without modification.

# These non-functional requirements ensure that the Encrypted Messaging Application is secure, efficient, and user-friendly, meeting the needs of diverse users while addressing issues like the "invalid padding error" through robust design and implementation.

# 5. Other Requirements

# Other requirements address additional considerations for the Encrypted Messaging Application, ensuring it meets broader usability, deployment, and regulatory needs.

# 1. Portability

# The system should be compatible with any device running Python 3.8 or higher, ensuring accessibility across platforms (Windows, macOS, Linux).

# No platform-specific dependencies are used, and the application is designed to run in a standard Python environment.

# Future iterations will include mobile apps for iOS and Android, compiled using frameworks like Kivy or BeeWare.

# Purpose: Ensures the system can be used by a wide range of users, including those on low-end devices.

# 2. Maintainability

# The codebase is modular, with separate modules for networking (network.py), cryptography (crypto.py), and user interface (ui.py).

# Code follows PEP 8 standards, with comprehensive comments and documentation.

# Example: Each function includes a docstring explaining its purpose, parameters, and return value.

# Version control using Git ensures that changes are tracked and reversible.

# Purpose: Allows developers to add new features, fix bugs, and update cryptographic protocols with minimal effort.

# 3. Usability

# The CLI is designed to be intuitive, with clear prompts and error messages.

# Example: "Enter your username: ", "[!] HMAC check failed for message from bob".

# The future GUI will include a chat window, contact list, and message input field, with visual aids and tooltips.

# User documentation will be provided, explaining how to install and use the application.

# Purpose: Ensures that users, including those with minimal technical knowledge, can use the system effectively.

# 4. Regulatory Compliance

# The system must comply with data privacy regulations (e.g., GDPR, CCPA) if user data is stored.

# No message content is stored on the server, ensuring end-to-end encryption.

# User credentials are hashed, and public keys are stored securely with proper access controls.

# Future iterations will include user consent mechanisms and data anonymization features.

# Purpose: Ensures the system meets legal and ethical standards for data protection.

# 5. Deployment Considerations

# Local Deployment:

# The initial implementation runs on localhost for testing (e.g., 127.0.0.1:8080).

# Suitable for development and small-scale testing.

# Cloud Deployment:

# Future iterations will deploy the server on a cloud platform (e.g., AWS, Azure) for global accessibility.

# Load balancing and high availability will be implemented to support large user bases.

# Offline Messaging:

# Future iterations will include message queuing for offline users, storing messages on the server until the recipient comes online.

# Messages will remain encrypted during storage to maintain security.

# Edge Deployment:

# The system can be adapted for edge devices (e.g., Raspberry Pi) for use in areas with limited internet access.

# Purpose: Ensures the system can be deployed in various environments, from local testing to global production.

# 6. Accessibility

# The CLI supports keyboard-only navigation, making it accessible to users with disabilities.

# The future GUI will include high-contrast themes and screen reader support.

# Multilingual support will be added to cater to users in different regions.

# Purpose: Ensures the system is accessible to a diverse user base.

# 7. Testing and Validation

# The system will undergo extensive testing to ensure functionality, security, and performance.

# Unit tests will cover cryptographic functions, networking, and message handling.

# Integration tests will verify end-to-end workflows (e.g., sending a message from Alice to Bob).

# User acceptance testing will ensure the system meets user expectations.

# Purpose: Ensures the system is reliable and meets all requirements before deployment.

# These other requirements address broader aspects of the Encrypted Messaging Application, ensuring it is portable, maintainable, usable, and compliant with regulations, while providing a roadmap for future deployment and enhancements.

# Appendix A: Glossary

# Diffie-Hellman Key Exchange: A cryptographic protocol for securely exchanging keys over an insecure channel by generating a shared secret.

# AES-128: Advanced Encryption Standard with a 128-bit key, a symmetric encryption algorithm used for securing messages.

# HMAC-SHA256: A keyed-hash message authentication code using SHA-256, ensuring message integrity and authenticity.

# End-to-End Encryption: A security measure where only the communicating parties can decrypt messages, preventing intermediaries from accessing the content.

# TCP/IP: Transmission Control Protocol/Internet Protocol, a network protocol for reliable data transmission.

# CBC Mode: Cipher Block Chaining mode, used with AES to encrypt messages by chaining blocks with an IV.

# IV: Initialization Vector, a random value used in encryption to ensure uniqueness of ciphertexts.

# SHA-256: A cryptographic hash function producing a 256-bit hash, used for password hashing and key derivation.

# Socket: A network programming interface for communication between processes over a network.

# This glossary provides definitions of key terms used in the Encrypted Messaging Application, aiding in understanding the system’s technical components.

# Appendix B: Analysis Model

# The analysis model outlines the structural and behavioral aspects of the Encrypted Messaging Application, ensuring efficient and secure communication under various conditions.

# 1. Data Flow

# Authentication:

# User credentials (username, hashed password) flow from the client to the server for verification.

# The server responds with a success or failure message.

# Key Exchange:

# Public keys flow from clients to the server, which broadcasts them to other clients.

# Clients compute shared keys using Diffie-Hellman.

# Message Transmission:

# Encrypted messages, IVs, and HMACs flow from the sender to the server, then to the recipient.

# The recipient verifies the HMAC and decrypts the message.

# 2. Cryptographic Model

# Diffie-Hellman:

# Generates private and public keys for secure key exchange.

# Computes shared secrets to derive AES keys.

# AES-128 CBC:

# Encrypts messages with a 128-bit key and random IV.

# Uses PKCS#5/PKCS#7 padding to handle variable-length messages.

# HMAC-SHA256:

# Generates and verifies HMACs to ensure message integrity.

# Uses the same key as AES for simplicity.

# 3. Performance Considerations

# Latency Optimization:

# Message encryption and decryption are optimized to take less than 50 ms per message.

# TCP/IP ensures reliable delivery with minimal overhead.

# Throughput:

# The server can handle up to 100 messages per second for 100 users.

# Future optimizations will include asynchronous message handling.

# Resource Usage:

# The client uses less than 100 MB of RAM, ensuring compatibility with low-end devices.

# 4. Scalability

# The client-server architecture allows the system to scale to multiple users.

# In-memory data structures (dictionaries) provide fast access to user data and keys.

# Future iterations will include distributed servers and load balancing for larger user bases.

# 5. Error Handling

# The system addresses the "invalid padding error" by ensuring proper key derivation and message formatting.

# Network errors are handled with timeouts and retries.

# Cryptographic errors (e.g., HMAC failure) are logged and reported to the user.

# This analysis model ensures a systematic approach to secure messaging, providing users with reliable and efficient communication while addressing potential issues like the "invalid padding error."

# Appendix C: Issues List

# The following issues have been identified during the development of the Encrypted Messaging Application. Each issue includes a status, description, and action plan.

# Issue 1: Lack of Graphical User Interface (GUI)

# Status: Open.

# Description: The current CLI may be difficult for non-technical users to navigate, limiting the system’s accessibility.

# Action: Develop a GUI using Tkinter or a web interface with Flask, including a chat window, contact list, and settings panel.

# Issue 2: No Support for Multimedia Messages

# Status: Open.

# Description: The system currently supports only text messages, limiting its functionality compared to modern messaging apps.

# Action: Add support for images, files, and other media in future iterations, ensuring secure encryption and transmission.

# Issue 3: Basic Authentication

# Status: Open.

# Description: The username/password authentication lacks multi-factor authentication (MFA), making it vulnerable to credential theft.

# Action: Implement MFA using methods like SMS codes or authenticator apps to enhance security.

# Issue 4: No Offline Messaging

# Status: Open.

# Description: The system requires a stable network connection, and offline users cannot receive messages until they reconnect.

# Action: Add message queuing for offline users, storing encrypted messages on the server until the recipient comes online.

# Issue 5: Scalability Limits

# Status: Open.

# Description: The current server implementation supports up to 100 concurrent users, which may not be sufficient for large-scale deployment.

# Action: Optimize server performance using asynchronous programming and add load balancing for scalability.

# Issue 6: Custom Cryptographic Implementation Risks

# Status: Open.

# Description: The custom AES implementation led to issues like the "invalid padding error," indicating potential vulnerabilities.

# Action: Replace the custom implementation with a well-tested library like cryptography, and conduct security audits.

# Issue 7: Limited Error Feedback

# Status: Open.

# Description: Error messages (e.g., "Invalid padding") are technical and may confuse non-technical users.

# Action: Improve error messages with user-friendly explanations and suggestions (e.g., "Message failed to decrypt. Check your network connection and try again.").

# These issues highlight areas for improvement in the Encrypted Messaging Application, providing a roadmap for future development to enhance functionality, security, and usability.