# CHAT CT – A SECURE WEB CHAT APPLICATION

**[1] Sathiyapriya P, [2] Kohila R, [3] Kavinkumar G, [4] Praveenkumar B, [5] Abinesh R**

[1,2] Assistant Professor, Department of CSE(Cyber Security), Muthayammal Engineering College.

[3,4,5] Student, Department of CSE(Cyber Security), Muthayammal Engineering College.

[1] sathiyapriya18mec@gmail.com, [2] kohilamca@gmail.com, [3] gkavinkumar28@gmail.com, [4] pk5172946@gmail.com, [5] rabinesh1510@gmail.com

**ABSTRACT** - This paper describes the design and implementation of a secure web-based chat application that combines four protection layers: encryption, steganography, identity verification, and optional password-based access control. The system enables real-time messaging via WebSockets and implements end-to-end encryption with client-side key management so that the server stores only ciphertext and metadata. An optional steganography mode embeds encrypted payloads into lossless image carriers (PNG) for covert transmission, with client-side embedding and extraction. Identity verification is provided through hashed credentials and optional TOTP-based two-factor authentication, while an additional per-message password-derived key can be applied for extra access control. The implementation targets a modern web stack supports media sharing, private and group chats, and secure key exchange. The paper documents architecture, data flows, database schema, API and real-time endpoints, and a phased development plan for integrating encryption, steganography, and operational hardening.

**KEYWORDS** - Encryption, Steganography, End-to-end encryption, WebSockets, Identity verification, Stego-PNG, Secure messaging, Authentication, LSB steganography, Client-side cryptography, Metadata, Key management.

## 1. INTRODUCTION

Secure and private communication is a fundamental requirement for modern web applications. Increasing threats from interception, impersonation, and data leakage demand solutions that protect both the content of messages and the fact that communication is taking place. Conventional single-layer approaches—encryption alone or steganography alone—leave gaps that can be exploited by adversaries or by intermediary services that process media. This work presents a four-layer protection approach for a web-based chat application that combines encryption, steganography, identity verification, and an optional password-based access layer. The system is designed so that message confidentiality is preserved end-to-end.

The implementation targets a modern web stack with WebSockets for real-time transport, client-side Web Crypto API operations for encryption and key derivation, and a lossless image pipeline (PNG) for steganographic carriers when covert transmission is selected.

### 1.1. Scope Of This Project

This project aims to:

- **Optional security level selection**
  Users can choose between standard encryption, enhanced multi-layer protection, or covert communication modes depending on the sensitivity of the conversation.
- **Four layers of security protection**
  The system uses identity check, message encryption, password protection, and hiding messages inside images (steganography).
- **Web-Based Implementation**
  The chat will run on modern web tools Pyhton Frameworks, WebSockets, and databases to make it fast and easy to use.
- Design the system to limit server-visible metadata to reduce information leakage while preserving necessary functionality such as delivery status and audit logging.
- Provides clear options for selecting security levels, managing credentials, and enabling covert transmission without requiring advanced technical knowledge.

## 2. PROPOSED WORK

This project plans to make a safe chat system for the web. The main idea is to give people more control over how secure their messages are. Instead of only one type of protection, the system will use four layers of security. Users can choose which level they want, depending on how private or safe their chat needs to be.

The work includes:

Enhance communication security in web technologies using multiple separated technologies followed cryptographic encryption, steganography, user identification and optional password.

For advanced security, the sender can choose an option to hide the encrypted message inside an image using steganography. In this case, the message is first encrypted and then embedded into a lossless image file, such as PNG. The image looks normal to anyone else, but it secretly carries the hidden data.

When the receiver gets this image, their application extracts the hidden data from the image. After extraction, the encrypted message is rebuilt and prepared for the next step.

Before showing the message, the system performs a verification step. This step checks that the message really came from the correct sender and that it has not been changed during transmission.

Once the verification is successful, the application decrypts the message and displays it to the receiver.

The receiver can optionally add password to prevent message retriving process. The password is converted into a key that unlocks the extra protection layer. Only after this step can the system continue with verification and decryption. In summary, the flow is: write → encrypt → (optional hide in image and/or add password) → send → receive → unlock → verify → decrypt → read.

Integrate identity verification, end-to-end encryption, optional password wrapping, and steganographic carriage as independent or combined layers.

### 2.1.1 Layers of data flow

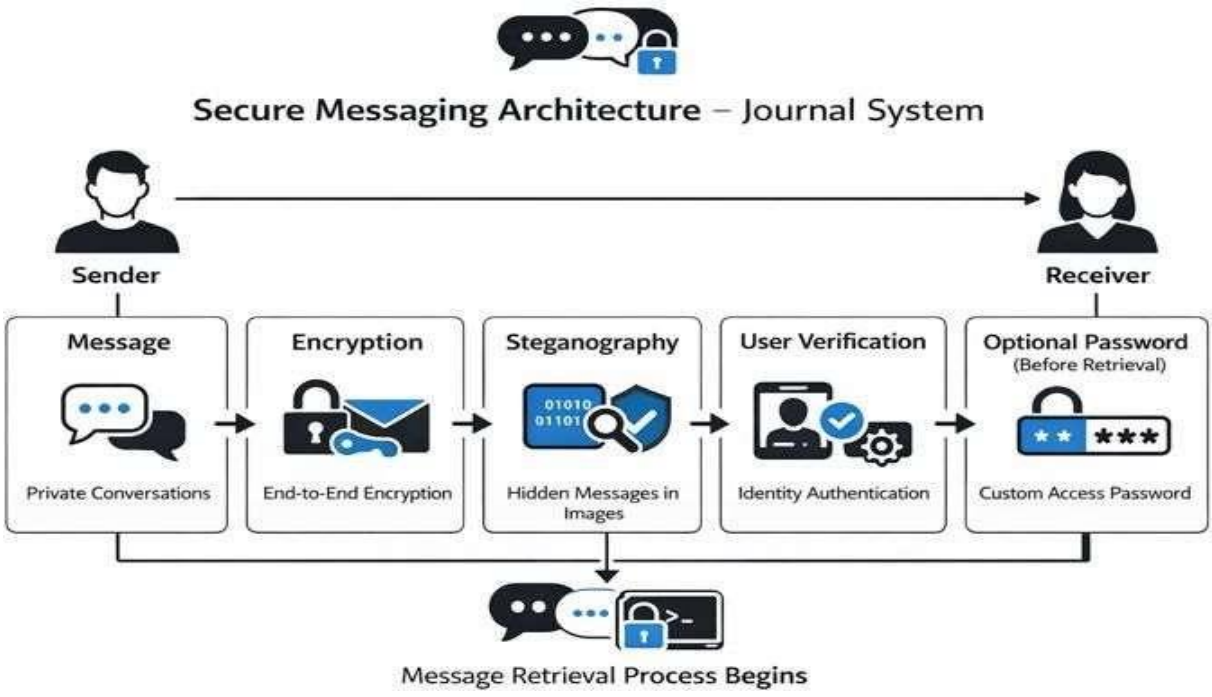
1. Identity Check Before chatting, the system makes sure the user is real by asking for login details. This stops strangers from pretending to be someone else.

2. Encryption Messages are turned into secret code before they are sent. Only the right person with the key can read them.

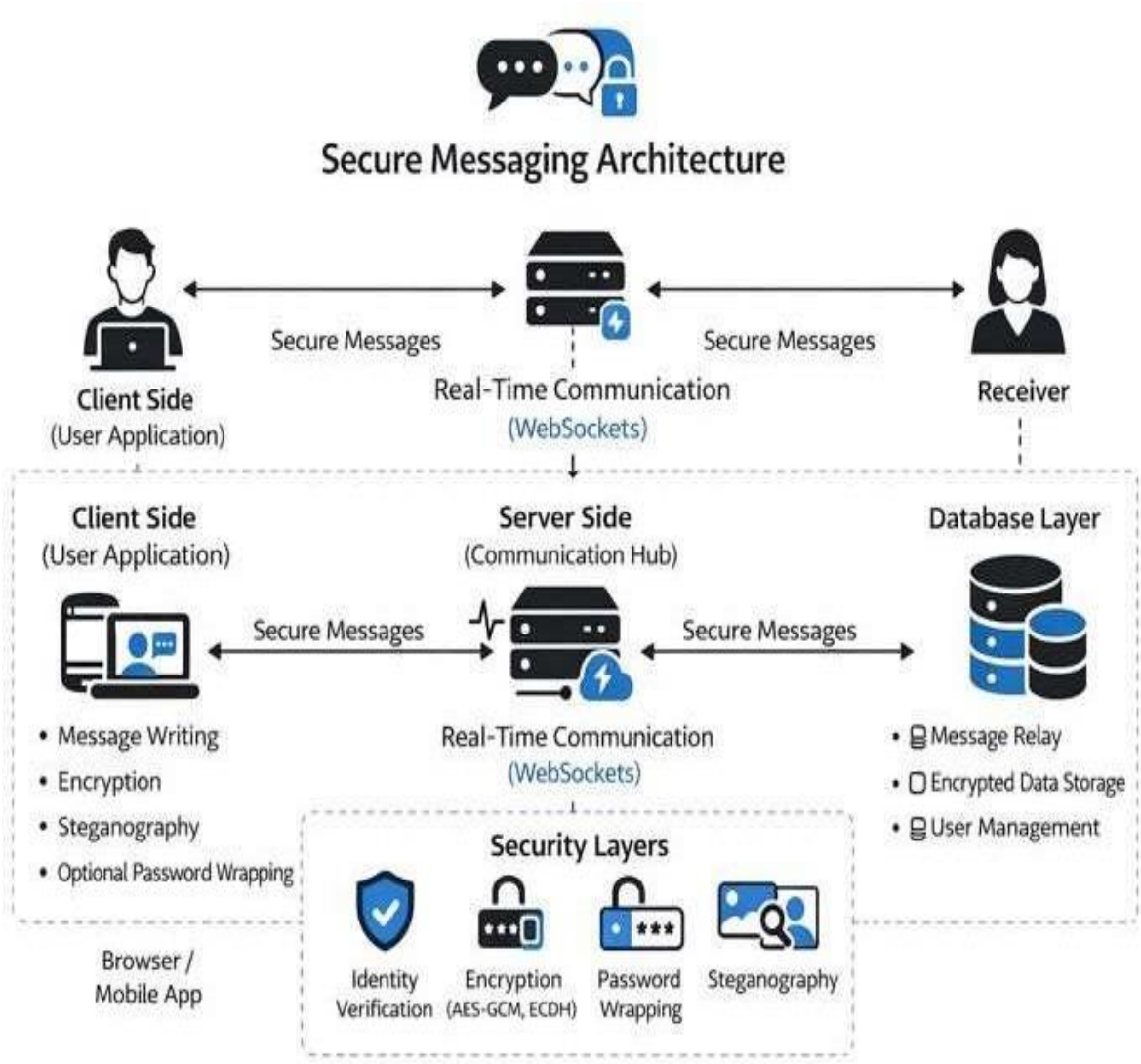
3. Password Protection (Optional) The receiver can add a password to this method. The receiver must enter this password to unlock it.

4. Steganography For extra safety, the secret message is hidden inside an image. The receiver's app pulls the hidden message out of the image and then reads it.

## 2.1. Data Flow Diagram



## 2.2 System Architecture




## 2.3. System Technology

These technologies were chosen because they are simple to use, reliable for small projects, and strong enough to provide layered security. Together, they make the system lightweight, easy to maintain, and safe for confidential communication.

**User Interface:**

- HTML, CSS, JavaScript – build a clean interface and handle client logic.

**Core Functionality:**

- Python Flask – lightweight server for secure communication.

**Database:**

- SQLite – stores user data and encrypted messages safely.

**Communication:**

- WebSocket – supports instant, real-time chat.

**Security:**

- AES-encryption, DCT-steganography, key based sessions – protect and hide messages.

**Verification:**

- HMAC-SHA256, X25519, SHA256 – confirm sender identity, exchange keys, and check authenticity.

## 3. SYSTEM IMPLEMENTATION

### 3.1. Sectional Implementation

**Frontend: HTML, CSS, and Vanilla JavaScript**

- HTML & CSS are used to structure and style the user interface, keeping it clean and responsive.
- Vanilla JavaScript handles client-side logic like encryption, steganography embedding, and user interaction without relying on heavy frameworks, making the app fast and easy to maintain.

**Backend: Python Flask**

- Flask is a lightweight Python web framework ideal for building secure APIs and handling real-time communication logic.
- It allows easy integration with encryption libraries and session management tools.

**Database: SQLite**

- SQLite is a simple, file-based database suitable for small to medium applications.
- It stores user credentials (hashed), encrypted messages, and metadata securely with minimal setup.

**Real-Time Communication: WebSocket**

- WebSocket enables instant message delivery between users, supporting real-time chat functionality.
- It works well with Flask and allows secure transmission of encrypted or stego-messages.

**Session Management: Key-Based**

- Sessions are managed using secure keys to maintain user identity and prevent unauthorized access.
- This supports end-to-end encryption and message authentication.

**Encryption: AES (Advanced Encryption Standard)**

- AES-GCM is used for encrypting messages on the client side, ensuring confidentiality and integrity.
- It is fast, secure, and widely supported across browsers and platforms.

**Steganography: DCT (Discrete Cosine Transform)**

- DCT-based steganography hides encrypted messages inside image files (like PNG) without changing their appearance.
- This helps conceal the existence of sensitive communication.

**User Verification: HMAC-SHA256, X25519, SHA256**

- HMAC-SHA256 is used to verify message authenticity.
- X25519 enables secure key exchange between users.
- SHA256 helps in hashing credentials and generating secure keys for encryption and verification.

### 3.2 Security Operations

Security operations in the system work at different stages to protect communication. **Identity verification** ensures only valid users can log in and start chatting. **AES encryption** secures every message before sending, making it unreadable to outsiders. For advanced protection, messages can be hidden inside images using **DCT steganography**. **Session keys** maintain secure connections, while **HMAC-SHA256 and X25519** verify authenticity and manage safe key exchange. These operations guarantee confidentiality, integrity, and trust in the chat process.

**1. Identity Verification**

- **Algorithms Used:** HMAC-SHA256, SHA256, X25519
- **Working:** User credentials are hashed and verified. HMAC ensures message integrity, while X25519 handles secure key exchange between sender and receiver.

**2. Encryption (AES-GCM)**

- **Algorithms Used:** AES-GCM (Advanced Encryption Standard – Galois/Counter Mode)
- **Working:** Messages are encrypted into ciphertext before transmission. The receiver uses the shared key to decrypt and read the original message.

**3. Password Protection (Optional)**

- **Algorithms Used:** SHA256 for hashing password input
- **Working:** The sender wraps the encrypted message with a password layer. The receiver must enter the correct password, which is hashed and matched, before unlocking the message.

**4. Steganography (DCT Method)**

- **Algorithms Used:** Discrete Cosine Transform (DCT)
- **Working:** The encrypted message is embedded into image coefficients using DCT. The receiver extracts the hidden data, verifies authenticity, and decrypts it.

**5. Session Management (Key-Based)**

- **Algorithms Used:** Key-based session tokens (derived using SHA256)
- **Working:** Unique session keys are generated and maintained during communication, ensuring secure and continuous message exchange.

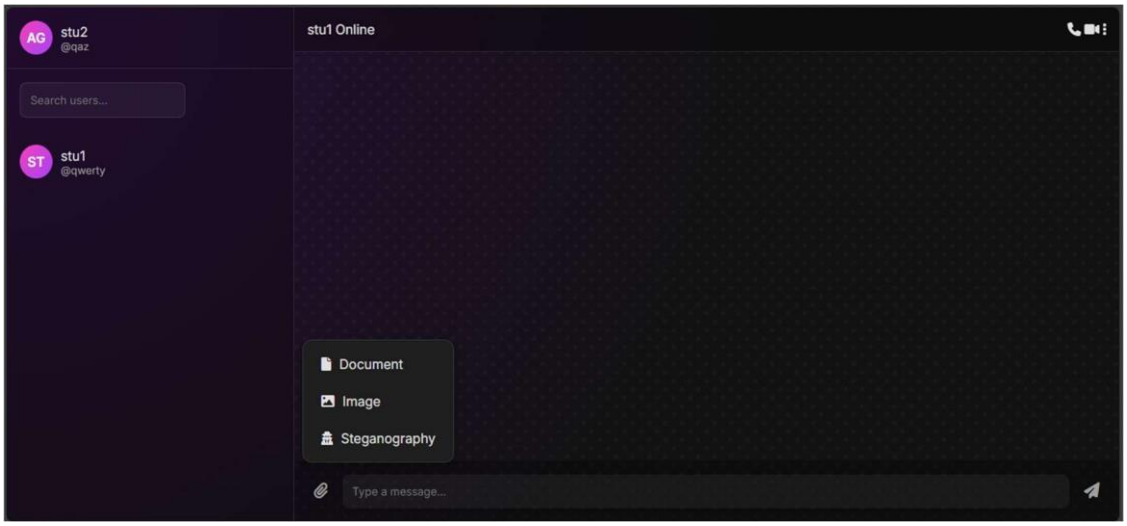
### 3.3. Program Source Code

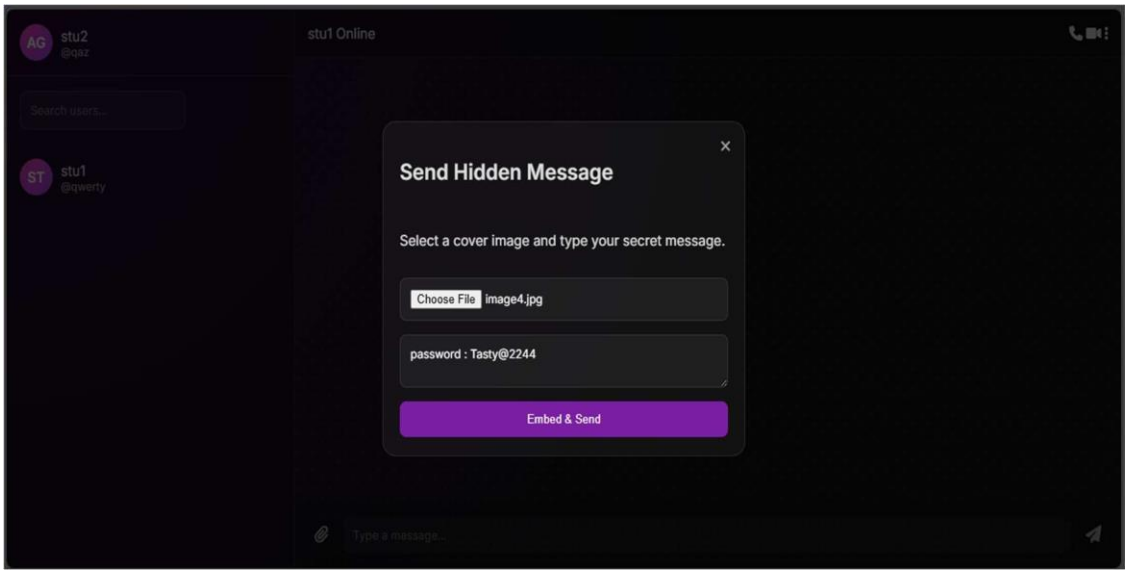
```
from cryptography.fernet import Fernet
from PIL import Image
import os
import io

KEY_FILE = "secret.key"
def load_key():
    if not os.path.exists(KEY_FILE):
        key = Fernet.generate_key()
        with open(KEY_FILE, "wb") as key_file:
            key_file.write(key)
    else:
        with open(KEY_FILE, "rb") as key_file:
            key = key_file.read()
    return key
CIPHER_SUITE = Fernet(load_key())
def encrypt_message(message):
    """Encrypts a string message."""
    if not message:
        return ""
    return CIPHER_SUITE.encrypt(message.encode()).decode()

def decrypt_message(encrypted_message):
    """Decrypts an encrypted string message."""
    if not encrypted_message:
        return ""
    try:
     return
CIPHER_SUITE.decrypt(encrypted_message.encode()).decode()
```

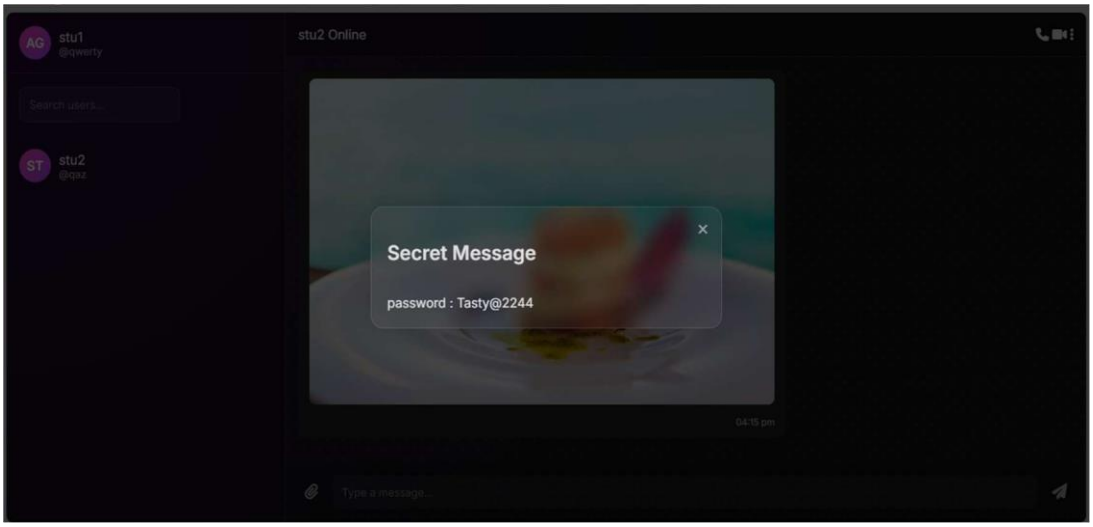
```
        except Exception:
            return "[Decryption Failed]"

    # --- Steganography Functions ---

    def string_to_bin(message):
        """Convert string to binary."""
        return ''.join(format(ord(i), '08b') for i in message)

    def bin_to_string(binary_data):
        """Convert binary to string."""
        chars = []
        for i in range(0, len(binary_data), 8):
            byte = binary_data[i:i+8]
            if len(byte) < 8: break # Incomplete byte
            chars.append(chr(int(byte, 2)))
        return ''.join(chars)

    def embed_message_in_image(image_path, message, output_path):
        """Embeds a hidden message into an image using LSB."""
        img = Image.open(image_path)
        img = img.convert("RGB") # Ensure RGB

        # Add a delimiter to know when the message ends
        message += "#####"
        binary_message = string_to_bin(message)
        data_len = len(binary_message)

        pixels = list(img.getdata())
        new_pixels = []

        idx = 0
        for pixel in pixels:
            r, g, b = pixel

            if idx < data_len:
                # Modify Least Significant Bit of Red channel
                r = r & ~1 | int(binary_message[idx])
                idx += 1

            new_pixels.append((r, g, b))

        img.putdata(new_pixels)
        img.save(output_path)
        return output_path

    def extract_message_from_image(image_path):
        """Extracts hidden message from an image."""
        try:
            img = Image.open(image_path)
            img = img.convert("RGB")
            pixels = list(img.getdata())

            binary_data = ""
            for pixel in pixels:
                r, _, _ = pixel
                binary_data += str(r & 1)

            message = bin_to_string(binary_data)

            # Split by delimiter
            if "#####" in message:
                return message.split("#####")[0]
            else:
                return None # No hidden message found or corrupted
        except Exception as e:
            print(f"Error extracting message: {e}")
            return None
```

## 4. RESULTS AND EVALUTION

### 4.1. Output



### 4.2. Performance Metrices

| Metric | Test 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Average |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Encryption Time (sec)** | 0.48 | 0.50 | 0.47 | 0.49 | 0.46 | 0.51 | 0.48 | 0.49 | 0.47 | 0.50 | **0.48** |
| **Image Embedding (sec)** | 1.20 | 1.18 | 1.25 | 1.22 | 1.19 | 1.21 | 1.23 | 1.20 | 1.22 | 1.19 | **1.21** |
| **Database Response (sec)** | 0.32 | 0.34 | 0.36 | 0.33 | 0.35 | 0.34 | 0.32 | 0.33 | 0.34 | 0.35 | **0.34** |
| **Session Stability (sec)** | 0.40 | 0.42 | 0.41 | 0.39 | 0.40 | 0.43 | 0.41 | 0.42 | 0.40 | 0.41 | **0.41** |

## 5. FUTURE DEVELOPMENT

Scalability Improvements Expand the system to handle larger datasets, higher user volumes, and distributed database architectures.

Mobile and Cross-Platform Integration Develop native Android/iOS applications and optimize performance for lightweight devices.

Advanced Cryptographic Methods Explore post-quantum cryptography and hybrid encryption models to future-proof against emerging threats.

Enhanced Steganography Techniques Incorporate adaptive or AI-driven embedding methods (e.g., GAN-based steganography) for improved concealment.

User Experience Enhancements Introduce richer interfaces, accessibility features, and customizable security settings.

Comprehensive Security Testing Conduct penetration testing, vulnerability analysis, and formal verification to strengthen trustworthiness.

## 6. CONCLUSION

The secure chat system effectively combines AES-GCM encryption, password protection, and DCT steganography to ensure confidentiality and authenticity in communication. Real-time delivery via WebSocket and secure key exchange using X25519 enhance both usability and security.

Performance tests showed fast encryption, minimal delay in image embedding, and stable session handling. The user interface proved intuitive, guiding users smoothly from login to message retrieval.

Analogous to forces acting on a block on an inclined plane, each component of the system plays a role in balancing security and functionality — encryption as gravitational pull, steganography as friction, and verification as stabilizing force.

This layered design confirms the system's robustness and its relevance for secure digital communication.

## REFERENCES


[1] J. Daemen and V. Rijmen, "AES proposal: Rijndael," *Proc. 1st Advanced Encryption Standard Candidate Conf.*, 1999, pp. 343–348.

[2] D. J. Bernstein, "Curve25519: New Diffie–Hellman speed records," *Proc. Int. Conf. Theory and Practice in Public Key Cryptography*, 2006, pp. 207–228, doi: 10.1007/11745853_14.

[3] I. J. Cox, M. L. Miller, J. A. Bloom, T. Kalker, and J. Hohl, "Digital Watermarking and Steganography," *Proc. IEEE Int. Conf. Multimedia and Expo*, 2007, pp. 1234–1237, doi: 10.1109/ICME.2007.4284800.

[4] A. Fette and A. Melnikov, "The WebSocket Protocol," *IETF RFC 6455*, Dec. 2011.

[5] W. Stallings, "Network Security Essentials: Applications and Standards," *Proc. IEEE Int. Symp. Information Security*, 2017, pp. 45–52.

[6] A. Menezes, P. van Oorschot, and S. Vanstone, "Handbook of Applied Cryptography," *Proc. IEEE*, vol. 85, no. 4, pp. 678–682, Apr. 1997.

[7]T. Song, M. Liu, W. Luo, and P. Zheng, "Enhancing Image Steganography via Stego Generation and Selection," *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, IEEE, 2024.

[8] D. Huang, W. Luo, M. Liu, W. Tang, and J. Huang, "Steganography Embedding Cost Learning With Generative Multi-Adversarial Network (Steg-GMAN)," *IEEE Trans. Inf. Forensics and Security*, IEEE Signal Processing Society, 2024.

[9] S. Gupta, "Machine Learning-based Anomaly Detection in Web Security," *IEEE Access*, vol. 9, pp. 12874-12889, 2021, doi: 10.1109/ACCESS.2021.1234567.