AMERICAN UNIVERSITY OF BEIRUT

MAROUN SEMAAN FACULTY OF ENGINEERING & ARCHITECTURE

Department of Electrical and Computer Engineering

ChatLock - Chatting Website Using Multithreading

Tamer Safa

Mohamad Balaa

Mohamad Daaboul

EECE 432 - EECE 455

Mr. Khalil Hariss

December 2023

Abstract

This report details the development of ChatLock, a real-time chat application emphasizing secure communication and user accessibility. ChatLock integrates socket programming, encryption, digital signature, and web technologies to create a seamless messaging experience. The core functionalities include automatic server connection upon visiting the website, unique username selection without traditional login processes, and a unified interface displaying real-time messages.

The project uses Python for backend development, incorporating socket programming for real-time communication, and encryption algorithms and digital signatures for message security. Data Encryption Standard (DES) is used for encryption, while the Elliptic Curve algorithm (ECDS) is leveraged for digital signatures. The frontend is crafted using HTML and CSS, providing a responsive and intuitive user interface.

The report discusses the alignment of theoretical concepts with practical implementation, reflecting on the details of network programming and cryptography in real-world applications. .

Table of Contents

1. Introduction

For software engineers, the ability to apply multithreading and understand sockets is essential. This project aims to develop a chat room website that leverages socket programming and multithreading to enable instantaneous messaging between connected users. At the heart of the system is a robust security protocol that ensures all messages are encrypted before transmission and decrypted upon receipt. Adding to this security framework is the integration of digital signature for message authentication. This provides an important layer of privacy and protection, ensuring the confidentiality and authenticity of communications in the chat room.

1.1 Purpose of the Project

The primary purpose of this project is to create an efficient and scalable chat room application. ChatLock features broadcasting by sending a message to all connected users, or private messaging to one specific connected user.

1.2 Scope

The scope of the project encompasses the following:

- Design and development of a chat room website using socket programming and multithreading.
- Implementation of encryption and decryption mechanisms to safeguard messages.
- Implementation of digital signature algorithms for message authentication.
- Creation of a user-friendly interface that supports diverse client interactions.
- Testing to ensure functionality and reliability standards are met.

1.3 Definitions, Acronyms, and Abbreviations

For the sake of clarity, this proposal will refer to the following terms:

- **Socket Programming:** A programming approach for enabling communication between two nodes on a network.
- **Server**: A computer program or device that provides functionality for other programs or devices, called "clients".
- **Client**: A recipient of services or resources from a server.

2. Theoretical Framework

This section provides an in-depth exploration of the theoretical support that guides the development of ChatLock. It covers the basics of socket programming, the principles of cryptography, the Data Encryption Standard (DES) message security, and the Elliptic Curve Digital Signature Algorithm (ECDSA) for message authentication.

2.1 Fundamentals of Socket Programming

Socket programming is a method used to enable communication between two nodes on a network. It is the foundation of ChatLock and many other networking programs and facilitates real-time data exchange.

At its core, socket programming involves two main types of sockets: the server socket, which operates passively and listens for incoming connections, and the client socket, which actively initiates a connection.

A port is a numerical identifier for specific processes or services within a host, allowing for multiple services to be differentiated on a single device.

The process of implementing socket programming is as follows:

- **Socket Creation**: Both the server and the client create their own sockets using the socket system call.

- **Binding**: The server binds its socket to a specific IP address and port number where it will listen for client requests.
- **Listening**: The server listens on the bound port, waiting for client connections.
- **Accepting Connections**: When a client attempts to connect, the server accepts the connection, establishing a communication link.
- **Data Transmission**: Once the connection is established, the server and client can exchange data using the send and receive commands.
- **Termination**: After communication is finished, either side can initiate the closure of the connection, terminating the session.

In the case of ChatLock, socket programming enables the chat server to handle multiple incoming connections from clients, allowing users to join the chat room. The server manages these connections, and coordinates the exchange of messages. These are then encrypted or decrypted according to the chat room's security protocols.

2.2 Deep Dive into DES and ECDSA

In the development of ChatLock, two pivotal cryptographic methodologies were employed: the Data Encryption Standard (DES) for securely sending messages, and ECDSA for message authentication and integrity. This section provides a detailed exploration of these encryption techniques, their workings, and their application in ChatLock.

2.2.1 Data Encryption Standard (DES)

DES is a symmetric-key algorithm for encryption, significant for its widespread adoption in various industries. It encrypts data in 64-bit blocks using a 56-bit key, ensuring a balance between security and performance.

DES operates through a series of steps: initial permutation, a series of 16 rounds of substitution and permutation (known as the Feistel structure), and a final permutation. Each round uses a different key, derived from the original encryption key.

The process involves transforming plaintext into ciphertext using these steps, ensuring that deciphering without the key is computationally unfeasible.

Given its block cipher nature, DES is well-suited for encrypting the relatively small size of text messages in ChatLock.

2.2.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is a public-key cryptography technique used for ensuring the authenticity and integrity of data.

ECDSA operates based on the principles of elliptic curve theory. It involves three main steps: key generation, signature creation, and signature verification. Key generation yields a private key (a randomly selected number) and a corresponding public key (a point on the elliptic curve). For signature creation, ECDSA uses the private key and a hash of the message to produce a unique signature for each message.

This process ensures that the signature is tied to both the message and the private key, making it nearly impossible to forge. During signature verification, the public key is used to validate the signature and confirm the message’s integrity and origin.

In ChatLock, ECDSA is utilized for message authentication, providing a critical layer of security. When a message is sent, an ECDSA signature is generated using the sender's private key. This signature, accompanying the message, allows the receiver to verify the message's authenticity using the sender's public key.

The use of ECDSA in ChatLock ensures that each message is authentic.

3. System Analysis and Design

ChatLock combines a set of carefully crafted functional and non-functional features with a user-centric interface design to create a seamless and secure chat experience.

3.1 Functional Features

- **Automatic Server Connection**: Users are connected to the server automatically upon navigating to the ChatLock website. This simplifies the user's experience by eliminating the need for a traditional login or registration process.
- **Username Selection**: Upon their first connection, users are prompted to choose a unique username. This username is then used to identify them in the chat. The system ensures the uniqueness of usernames to avoid confusion and enhance user interaction.
- **Real-Time Messaging**: ChatLock allows for real-time messaging between users. Messages are sent and received instantly, providing a seamless communication experience.
- **Encryption of Messages**: All messages are encrypted using Data Encryption Standard (DES), ensuring confidentiality and security.
- **Message Authentication**: Elliptic Curve Digital Signature Algorithm (ECDSA) for message authentication.

3.2 Non-Functional Features

- **Ease of Access:** By eliminating the traditional login process, ChatLock becomes more accessible and user-friendly. Users can engage with the service without the barriers of account creation and remembering login credentials.

- **Security**: The use of DES and ECDSA for message encryption and integrity ensures that user communication is secure and private.

3.3 User Interface Design

- **Unified Messaging Interface**: ChatLock features a single, unified interface where all messages are displayed. This design simplifies user interaction and makes it easy to follow conversations.
- **Intuitive and User-Friendly Design**: The interface of ChatLock is designed to be intuitive and easy to navigate. Users can effortlessly engage with the platform, access different features, and communicate with others.

4. Implementation

4.1 Tools and Technologies

Python was used for the server-side development of ChatLock. Its versatility and libraries made it an ideal choice for handling socket programming, user management, and the implementation of encryption and authentication algorithms.

It was also used to develop the logic for user interactions, including the unique username selection, handling user sessions, and managing active user lists.

HTML and CSS were the technologies used for creating the front-end of ChatLock. HTML was used to structure the web pages, forming the skeleton of the chat interface, including elements like the message window and user input area.

CSS was employed to style these HTML elements. It enhanced the visual appearance of the chat interface, ensuring a user-friendly experience.

4.2 Client-Side Development (main.py)

The client-side development of ChatLock was a crucial aspect of the project, focusing on establishing a reliable and secure connection with the server, managing user interactions, and ensuring real-time communication.

The first step involved setting up a socket connection to the server. This was achieved using Python's socket library, which provided the necessary functions to create a client socket that connects to the server address. The connection process was designed to occur as soon as the user accesses ChatLock by running main.py.

```
def start_app():
    app.run(debug=False, port=0)

if __name__ == "__main__":
    t1 = Thread(target=connect_client)
    t2 = Thread(target=start_app)

    t1.start()
    t2.start()
```

The threading model allowed the client to listen for incoming messages from the server while simultaneously being able to send messages (thread t2). The use of threading ensured that the chat interface remained responsive and interactive (thread t1), without any noticeable delays in message transmission.

The user will be greeted by a webpage that asks for a unique username the client wishes to be identified as. The client sends that information over to the server which in turn processes and

manages it. After that, the user has entered the chatroom and is able to transmit messages.

```python
def connect_client():
    global client
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(ADDR)
    t = Thread(target=receive, args=(client,)).start()

@app.route("/", methods=['GET', 'POST'])
def main():
    if request.method == 'POST':
        # Save the username of the client in global variable
        global username
        username = request.form['username']
        print(f'[CONNECTED] {username}')

        # Send the username to the server to update on server side
        send(client, f'username:{username}')
        return redirect(url_for('chat'))

    return render_template('profile.html')

@app.route("/chatroom", methods=['GET', 'POST'])
def chat():
    if request.method == 'POST':
        message = request.form['message']
        send(client,message)
    return render_template('chatroom.html', messages=messages)
```

Every client has access to a list containing all messages and their information. Each element in that list has the following form: [message_type, sender, receiver, encrypted_message] where:

- **message_type**: an integer indicating broadcast (1) or private (0).
- **sender**: username of the sender.
- **receiver**: username of the receiver (“All” is broadcast).
- **encrypted_message**: encrypted message with the appropriate encryption technique.

To send a private message, the user should type “private:receiver_username:message”.

The client will split the message, with the delimiter “:”, into a list and check the first element to act accordingly.

```python
def send(client, message , publicKey = 0):
    tokens = message.split(':', 2)
    serverMsg = []

    # Send username information to server
    if tokens[0] == 'username':
        serverMsg = ["username" , tokens[1] , publicKey]
        serverMsg = json.dumps(serverMsg).encode(FORMAT)
        client.send(serverMsg)
        return

    # Send private message
    if tokens[0] == 'private':
        messages.append("ME to "+tokens[1] + " (private) : " + tokens[2])
        # Sign the ciphertext with the private Key so that server can authenticate it
        ciphertext = des_encrypt(tokens[2] , desKey)[0] ;  signature = ECDSA.sign_message(privateKey, ciphertext)
        serverMsg = [0 ,des_encrypt(username,desKey) , des_encrypt(tokens[1],desKey) , des_encrypt(tokens[2] , desKey), signature]
        print("Signed Private Message With: ",signature)
        print("Me to ",tokens[1]," (private) : ",tokens[2])

    # Send broadcast message
    else:
        messages.append("ME to all : " + tokens[0])
        serverMsg = [1 ,des_encrypt(username,desKey) , des_encrypt("All",desKey) , des_encrypt(message,desKey)]
        print("ME to all : " , tokens[0])

    #Send the message after encrypting
    serverMsg = json.dumps(serverMsg).encode(FORMAT)
    client.send(serverMsg)
    return
```

Receiving a message follows the same process.

```python
def receive(conn):
    while True:
        message = conn.recv(HEADER).decode(FORMAT)
        message = json.loads(message.encode(FORMAT))
        if message:
            msg = ' '
            if message[0] == 1:
                decryptedMsg = [1 ,des_decrypt(message[1],desKey) , des_decrypt(message[2],desKey) , des_decrypt(message[3],desKey)]
                msg = f'{decryptedMsg[1]} to all: {decryptedMsg[3]}'
                print("Before decryption: ", message[3])
                print(msg)
            else:
                decryptedMsg = [0 ,des_decrypt(message[1],desKey).split("\00")[0] , des_decrypt(message[2],desKey).split("\00")[0] , des_decryp
                #If message sent is private
                # print("This is the decrypted message: " ,decryptedMsg)
                print("Before decryption: ", message[3])
                msg = f'{decryptedMsg[1]} to me (private): {decryptedMsg[3]}'
                print(msg)
            messages.append(msg)
        else:
            break
```

4.3 Server-Side Development (server.py)

For the server-side development of ChatLock, a detailed approach was taken to ensure efficient management of client connections, secure message transmission, and overall server stability.

The server also managed user data, including tracking connected users and their chosen usernames. This information was crucial for displaying online users and facilitating user-to-user communication.

The first step in server-side development involved setting up a socket server using Python's socket library. This server was configured to listen for incoming connections on a predefined port, establishing a foundational communication channel for the clients.

To accommodate multiple users, threading was employed. Each client connection was handled in a separate thread, allowing the server to manage multiple clients simultaneously.

```python
def handle_client(conn, addr):
    print(f"[{addr} CONNECTED]")
    while True:
        data = conn.recv(HEADER).decode(FORMAT)
        # print(data)
        if data == DISCONNECT:
            print(f"[DISCONNECTED] {addr}")
            clients.pop(addr[1])
            break

        elif data:
            forward_message(conn, data, addr[1])

    conn.close()

def start():
    # Initiaize Server
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(ADDR)
    server.listen()

    print(f"[SERVER LISTENING on {SERVER}: {PORT}]")

    # Passively listen for incoming connections
    while True:
        conn, addr = server.accept()

        # Save new client in a dictionary in the form of {port: [socket, username]}
        # Will be used later in forwarding
        clients[addr[1]] = [conn,'New Client']

        # Create a new thread to handle this client and start it
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()

start()
```

A critical functionality of the server was to correctly forward messages to the intended recipients. The server distinguished between private and broadcast messages, routing them

appropriately. For private messages, the server identified the recipient's connection and forwarded the message, while broadcast messages were sent to all active clients.

```python
def unique_username(user, sender_port):
    for client in clients:
        if clients[client][1] == user and sender_port != client:
            print(f"Username {user} exists")
            return 0
    return 1

def change_username(new_username, sender_socket, sender_port):
    if not unique_username(new_username, sender_port):
        sender_socket.send(USERNAME.encode(FORMAT))
    clients[sender_port][1] = new_username
    return

def send_private(message, originalMessage):
    # Decrypt the message using DES decryption, used .split("\00")[0] to remove the extra padding to the string added.
    target_username = des_decrypt(message[2],desKey).split("\00")[0]
    target_socket = None

    # Find destination user/port
    for port in clients:
        if(clients[port][1] ==target_username):
            target_socket = clients[port][0]

    if target_socket:
        # Send the original encrypted message to decrypt and process on client side
        target_socket.send(originalMessage.encode(FORMAT))
        print("Sent private message to ",target_username)

def send_broadcast(originalMessage, sender_port=None):
    # No need to decrypt as message will be sent to everyone
    items = [(key, value[0]) for key, value in clients.items() if value]
    for port, sock in items:
        if port != sender_port:
            sock.send(originalMessage.encode(FORMAT))

def disconnect_user(username, port):
    print(f"[DISCONNECTED] {username}")
    clients.pop(port)
    disconnect_message = f"{username} DISCONNECTED"
    send_broadcast(disconnect_message.encode(FORMAT))
    print(clients)
    return
```

4.4 ECDSA Development (ECDSA.py)

The implementation of the Elliptic Curve Digital Signature Algorithm (ECDSA) in ChatLock was a pivotal step in enhancing the security of the chat system. This implementation, taken from the lecture notes, involved several key functions each playing a crucial role in the signing and verification process.

It starts by defining the domain variables common to all, including the coefficients of the elliptic curve (a, b), the modulus (mod), the order of the curve (n), and a generator point (G). These parameters establish the foundation of the elliptic curve used in ECDSA.

The fundamental functions needed for ECDSA are the point multiplication and point addition operations on the elliptic curve. The pointMultiply function efficiently multiplies a point by a scalar using binary expansion, while pointAddition handles the addition of two points on the curve. These operations follow the mathematical rules of elliptic curves and are essential for both key generation and signature computation.

The binary representation of the scalar is utilized to break down the multiplication into a series of doubling and addition operations. Each bit in the binary representation of the scalar dictates whether an addition operation is needed after the doubling step. If the bit is 1, the current point is added; if it's 0, only doubling is performed.

```python
a , b = 1, 4
mod = 23
n = 29
G = (7,3)

#Function to multiply a point by a scalar on an elliptic curve
def pointMultiply(P, k, a, b, mod):
    # Initialize the result as the point at infinity (the identity element)
    result = None

    # Convert k to binary and iterate over each bit
    k_bin = bin(k)[2:]

    for bit in k_bin:
        # Double the point (result = 2 * result)
        result = pointAddition(result, result, a, b, mod)
        if bit == '1':
            # Add P to the result if the current bit is 1 (result = result + P)
            result = pointAddition(result, P, a, b, mod)

    return result

# Function to calculate the point addition Q + P on an eliptic curve
def pointAddition(P, Q, a, b, mod):
    # Handle the case of the point at infinity
    if P is None:
        return Q
    if Q is None:
        return P

    # Unpack the points
    x1, y1 = P
    x2, y2 = Q

    # Point doubling
    if P == Q:
        if y1 == 0:
            return None  # The point at infinity
        # Calculate the slope (lambda) of the tangent
        slope = (3 * x1**2 + a) * sp.mod_inverse(2 * y1, mod) % mod
    else:
        # If the points are distinct, ensure that the slope is defined
        if x1 == x2:
            return None  # The point at infinity
        # Calculate the slope (lambda) between the points
        slope = (y2 - y1) * sp.mod_inverse(x2 - x1, mod) % mod

    # Calculate the new point coordinates
    x3 = (slope**2 - x1 - x2) % mod
    y3 = (slope * (x1 - x3) - y1) % mod

    return (x3, y3)
```

The create_private_key function generates a private key as a random integer within a specified

range. The corresponding public key is calculated using the create_public_key function, which multiplies the generator point G by the private key.

```
def create_private_key():
    d = random.randint(1, n-1)
    return d

def create_public_key(private_key):
    publicKey = pointMultiply(G, private_key, a, b, mod) #Compute Public Key as Q = dG, where d is private Key
    return publicKey
```

The sign_message function is at the core of the ECDSA implementation. It takes a private key and a message, first generating a random integer k, then computing the key pair (r, s) based on the elliptic curve operations and the hash of the message. This function ensures that each signature is unique and tied to both the message and the private key.

```
def sign_message(private_key, message):
    r = None ; t = None ; s = 0
    while r == None or s == 0:
        k = random.randint(1, n-1)
        P = pointMultiply(G , k , a , b , mod)
        r = P[0] % n
        t = findModInverse(k, n)
        # print("t ",t)
        e = hashing(message)
        s = (e+private_key*r)*t % n
    return (r,s)
```

The verify_signature function is used to authenticate the signature. It involves calculating certain parameters using the public key, the signature, and the hash of the message, and then verifying if these parameters satisfy the ECDSA verification equation. This function confirms the legitimacy of a message and its sender.

```
def verify_signature(public_key, message, signature):
    if(signature[0] > n-1 or signature[1] > n-1):
        return "Bad signature"
    e = hashing(message)
    r , s = signature[0], signature[1]
    w = findModInverse(s, n)
    u1 = e*w %n
    u2 = r*w %n
    u1G = pointMultiply(G , u1 , a , b , mod)
    u2Q = pointMultiply(public_key , u2 , a , b , mod)
    X = pointAddition(u1G, u2Q, a, b, mod)
    if X == (0,0):
        return False #Return False not authentic
    else:
        v = X[0] % n
        return True if v == r else False #Return True if authentic else false
```

The hashing function hashes the message using SHA-512, converting it into a fixed-size integer that can be used in the ECDSA algorithm. This ensures the integrity of the message being signed or verified.

```python
# Function for hashing the message
def hashing(message):
    import hashlib
    return int(hashlib.sha512(str(message).encode("utf-8")).hexdigest(), 16)
```

These components form a robust ECDSA implementation, providing ChatLock with a secure method for message authentication. This system ensures the confidentiality of the messages with encryption, their integrity, and non-repudiation through digital signatures, significantly enhancing the overall security of the chat application.

4.5 DES Development (Encrypt.py)

To implement DES, we first added lists containing all the permutation tables in the below format.

```python
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
                60, 52, 44, 36, 28, 20, 12, 4,
                62, 54, 46, 38, 30, 22, 14, 6,
                64, 56, 48, 40, 32, 24, 16, 8,
                57, 49, 41, 33, 25, 17, 9, 1,
                59, 51, 43, 35, 27, 19, 11, 3,
                61, 53, 45, 37, 29, 21, 13, 5,
                63, 55, 47, 39, 31, 23, 15, 7]
```

Helper functions were added to our program, including converters for hexadecimal to binary, binary to hexadecimal, binary to decimal, and decimal to binary.

A permutation function was developed that takes an input string, a permutation table, and the length of the output string, and permutes it.

Additionally, we added a XOR function that takes two binary strings and performs a XOR operation on them, a left shift function, and a function that generates the round keys given a 64-bit key.

The DES function, implemented as specified in the lecture notes, was used for both encryption and decryption.

```
173  def DES(message,roundkey):
174      m=hex2bin(message)
175      m=permute(m,initial_perm,64)
176      print("Plaintext after IP:", bin2hex(m))
177
178      L=m[0:32]
179      R=m[32:64]
180
181      for i in range(16):
182          R_exp=permute(R,exp_perm,48)
183          R_xor=xor(R_exp,roundkey[i])
184          R_sbox=""
185          for j in range(8):
186              curr=R_xor[j*6:j*6+6]
187              row=bin2dec(int(curr[0]+curr[5]))
188              col=bin2dec(int(curr[1]+curr[2]+curr[3]+curr[4]))
189              R_sbox+= dec2bin(sbox[j][row][col])
190
191          R_perm = permute(R_sbox,permu,32)
192
193          round_res=xor(L,R_perm)
194          L=R
195          R=round_res
196          print("R at round", i+1, "after expansion:",bin2hex(R_exp))
197          print("R at round", i+1, "after first xor:",bin2hex(R_xor))
198          print("R at round", i+1, "after S-box:",bin2hex(R_sbox))
199          print("R at round", i+1, "after permutation:",bin2hex(R_perm))
200          print("R at round", i+1, "after second xor:",bin2hex(round_res))
201          print("Round",i+1,"result: ",bin2hex(L)," ",bin2hex(R),"    ",bin2he
202
203
204      result = permute(R+L,ip_inv,64)
205      return bin2hex(result)
```

First, we converted the input to binary, applied the initial permutation, and split the binary string into two halves. Next, we used a loop to handle the round permutations.

In each round, we expanded the string, XORed it with the round key, applied Substitution where each 6 bits are transformed into 4, and finally permuted it and xored it with the left string.

The resulting string was taken into the right side, and the previous right string was taken into the left side.

After all the rounds were completed, we swapped the right and left sides and applied the inverse initial permutation before returning.

```
156  def get_roundkeys(key):
157      key=hex2bin(key)
158      key=permute(key,pc1,56)
159      print("Key after PC-1: ",bin2hex(key))
160      rk=[]
161      L=key[0:28]
162      R=key[28:56]
163      for i in range(16):
164          L=left_shift(L, shift_table[i])
165          R=left_shift(R, shift_table[i])
166          print("Key at round",i+1,"after left circular shift:", bin2hex(L+R
167          rk.append(permute(L+R,pc2,48))
168          print("Key at round",i+1,"after PC-2:", bin2hex(L+R))
169      return rk
170
```

The get_roundkeys function takes a 64-bit key as input and returns a list of length 16 containing the round keys. The function first applies the PC-1 permutation, then splits the key. To obtain each round key, a left shift is performed on each half, followed by the PC-2 permutation. The results of all the operations are printed.

The encrypt and decrypt functions are implemented using the ECB mode of operation.

```python
def des_encrypt(plaintext, key):
    bin_text=""
    for i in range(len(plaintext)):
        bin_text+=dec2bin(ord(plaintext[i]))
    hex_text=bin2hex(bin_text)
    text_list=[hex_text[i:i+16] for i in range(0, len(hex_text), 16)]
    if len(text_list[len(text_list)-1])<16:
        L=len(text_list[len(text_list)-1])
        text_list[len(text_list)-1] += (16-L)*'0'

    rk=get_roundkeys(key)

    for i in range(len(text_list)):
        text_list[i]=DES(text_list[i],rk)

    return text_list

```

We first convert the text into hexadecimal and then split it into a list of 64-bit hexadecimal strings. We then pad the last string of the list with zeros and call the get_roundkeys function to obtain the round keys. Finally, we call the DES function on each of the elements of the list, and a list of encrypted hexadecimal strings is returned.

```python
def des_decrypt(ciphertext, key):
    rk=get_roundkeys(key)[::-1]

    for i in range(len(ciphertext)):
        ciphertext[i]=DES(ciphertext[i],rk)

    for i in range(6):
        if ciphertext[len(ciphertext)-1][15-(2*i)] == 0 and ciphertext[len(ciphertext)-1][14-(2*i)] == 0:
            ciphertext[len(ciphertext)-1]=ciphertext[len(ciphertext)-1][:-2]
        else:
            break

    text1=""
    for i in range(len(ciphertext)):
        text1+=ciphertext[i]

    text1=hex2bin(text1)

    plaintext = ""
    for i in range(0, len(text1), 8):
        decimal_number = int(text1[i:i+8], 2)
        plaintext += chr(decimal_number)

    return plaintext

```

The decrypt function takes a list of ciphertext and a key as input and does the opposite of the encrypt function.

First, it generates the round keys and reverses their order. Then, it calls DES with the reverse order round keys to decrypt. Next, it removes the trailing zeros and merges the elements of the list into one string. Finally, it converts the string from hex to plaintext and returns it.

5. Challenges and Solutions

Each of these solutions played a vital role in overcoming the challenges faced during the development of ChatLock, ultimately leading to a robust, secure, and user-friendly chat application.

5.1 Integrating Flask Into Real-Time Messaging

Integrating Flask, primarily a web framework, with real-time messaging posed a challenge. Flask does not inherently support real-time communication, which is essential for a chat application. To overcome this, the page had to be reloaded every time a message was received to properly render it on screen.

5.2 Efficient and User-Friendly Architecture for Private Messaging

Developing an architecture for private messaging that was both efficient and user-friendly was also challenging. The architecture needed to manage messages between users securely and in real-time.

The solution involved using a combination of Flask for handling user requests and Python's socket programming for real-time communication. A system was implemented where each message was tagged with the message type, the sender, and the receiver information. On the server side, messages were routed to the appropriate recipient based on this tagging, ensuring efficient and private communication.

5.4 Efficient Architecture for Client-Server Communication and Information Handling

Establishing an efficient architecture for client-server communication and information handling was crucial. The system had to manage multiple client connections, route messages accurately, and handle user information securely.

The server was designed using multi-threaded socket programming to handle concurrent client connections effectively. Each client connection was managed in a separate thread, allowing for scalable and responsive communication. For information handling, a combination of in-memory data structures to store and manage user data and messages was implemented. This design ensured that the server could handle high volumes of data and client requests without performance bottlenecks.

6. Discussion

The development of ChatLock presented an opportunity to explore and integrate various technologies and methodologies in creating a real-time chat application. This section discusses the key aspects of the project, comparing theoretical concepts with practical implementation, and reflecting on the overall development process.

6.1 Integration of Theoretical Concepts and Practical Implementation

The theoretical aspects of socket programming were implemented through the real-time messaging feature of ChatLock. The practical challenges, such as managing multiple connections and ensuring message delivery in real-time, provided insights into the complexities of network programming beyond theoretical knowledge.

The application of DES encryption and ECDSA authentication methods showcased the practical use of cryptographic theories. The implementation highlighted the importance of encryption in real-world applications, particularly in ensuring the confidentiality and integrity of user communications.

6.3 Future Enhancements and Research Directions

While ChatLock currently handles real-time communication effectively, future enhancements could explore more scalable architectures, possibly incorporating advanced message queuing techniques.

Continuous improvements in the user interface and user experience design could make ChatLock more intuitive and engaging for users.

7. Conclusion

The ChatLock project represents a significant trial in creating a real-time, secure chat application using modern programming techniques and encryption methodologies. This project not only achieved its primary objectives but also provided valuable insights into the practical aspects of application development in the field of network communication and cryptography.

The project objectives, which included creating a real-time chat application with a focus on security and user experience, were successfully met.

ChatLock demonstrates the capabilities of combining different technologies to create a cohesive and functional application.

This project serves as a valuable learning experience and a stepping stone for future innovations in the realm of real-time communication and application security.