Final Project Report

Development and Implementation of a Secure Instant Point-to-Point Messaging System Using Encryption Techniques

Averi Bates & Yul Castro

Abstract

This report details the design and implementation of a secure instant messaging tool, specifically tailored for point-to-point communication between two parties, referred to as Alice and Bob, akin to platforms like GTalk, Skype, or ICQ chat. Central to the system's design is the use of a shared password or passphrase, which is crucial for the encryption and decryption of messages. Ensuring privacy and security, each message transmitted over the Internet is encrypted using a key with a minimum length of 56 bits. The project leverages Python programming language and integrates open-source cryptographic libraries and tools to realize the system.

Significant design considerations addressed include the selection of a suitable cipher to comply with the 56-bit key requirement, the method of deriving a secure key from the shared password to avoid direct usage of the password as the key, and the choice of padding mechanism. The system showcases a user-friendly graphical user interface (GUI), where the ciphertext of sent messages is displayed, alongside the received ciphertext and its corresponding decrypted plaintext.

Introduction

Our project involves designing a secure chat application. This tool is not just a messaging platform; it embeds cryptographic techniques to ensure that Alice and Bob can exchange messages with the assurance of confidentiality and integrity. The system's cornerstone is the encryption of messages using a key no less than 56 bits, ensuring robust security against potential eavesdropping or interception during internet transmission.

Encryption and Updates

The primary design requirements for this project were clear: develop a system where Alice and Bob can securely send messages to each other using a shared password for encryption and decryption. Our approach involved several key considerations: To meet the requirement of a minimum 56-bit key, we selected a combination of AES (Advanced Encryption Standard). AES is renowned for its robustness and efficiency, making it a popular choice in various security protocols.

Directly using the password as the encryption key was not advisable due to potential security vulnerabilities. Instead, we implemented a key derivation function using PBKDF2HMAC with SHA256 hash algorithm. This approach allowed us to generate a strong, unique key from the shared password. For the encryption, we used PKCS7 padding to ensure that the plaintext is properly aligned to the cipher's block size.

To avoid generating identical ciphertext for repeated messages (like multiple "ok" messages), we incorporated a timestamp with each message, ensuring that each encryption output is unique. We designed a key management protocol to periodically update the encryption key, enhancing the system's security. This mechanism responds to certain triggers, like the number of messages or a specific time.

Graphical User Interface (GUI) and Network Communication

To enhance user experience, we developed a GUI using Python's Tkinter library. This interface not only facilitates easy message sending and receiving but also displays the decrypted plaintext, adding a layer of transparency and user engagement. We utilized socket programming in Python to establish and maintain a connection between Alice and Bob over the internet. This method allowed for real-time, bidirectional communication.

Background

Encryption and Key Generation

AES is a symmetric encryption algorithm widely used across the globe due to its balance of speed and security. It encrypts data in fixed block sizes of 128 bits, but it can have key sizes of 128, 192, or 256 bits. AES operates on entire blocks of data, and it is known for its efficiency in both software and hardware implementations. Its adoption as a standard by the U.S. government and its approval by the National Institute of Standards and Technology (NIST) in 2001 have bolstered its popularity and trustworthiness.

PBKDF2 is a key derivation function used to implement password-based cryptography. It converts passwords into keys that can be used for cryptographic operations. PBKDF2 applies a pseudorandom function, such as HMAC, to the input password along with a salt value and repeats this process many times to produce a derived key. This method helps protect against brute force attacks and rainbow table attacks.

Socket Programming and GUI in Python

The project utilizes Python's socket programming capabilities for network communication, enabling the chat client and server to exchange messages. Python's socket module is instrumental in handling TCP/IP connections, a standard for achieving reliable data transmission over networks.

Additionally, the project employs Python's tkinter library for creating a graphical user interface (GUI). Tkinter is widely used due to its simplicity and effectiveness in building cross-platform applications. It allows for the creation of user-friendly interfaces, enhancing the usability of the application.

Cryptography Hazmat in Python

The project makes use of the cryptography.hazmat module in Python, which stands for "hazardous materials." This module provides a set of low-level cryptographic primitives, often used for building higher-level cryptographic tools. It includes a range of functionalities from basic algorithms like AES and ChaCha20 to padding schemes and key derivation functions like PBKDF2**.**

System Design and Achiteture

Encryption Mechanism and Key Derivation

The AESEncryption class, defined in AES.py, forms the backbone of our encryption strategy. In the updated design, the keys for AES is derived using distinct salts to enhance security.

- **AES Encryption**:
    - **Function**: **encrypt(self, plaintext: str) -> bytes**
        - This function takes plaintext, pads it using PKCS7 padding to match AES's block size, and encrypts it using AES in CBC mode with a derived 256-bit key.
- **AES Decryption:**
    - Function: decrypt(self, ciphertext: bytes) -> str
        - This function first decrypts the input using ChaCha20Poly1305, followed by AES in CBC mode. The decrypted text is then unpadded using PKCS7 to retrieve the original plaintext.
- **Key Derivation Function**: **derive_key(password: str, salt: bytes, key_length: int) -> bytes**
    - This static method generates a cryptographic key from a given password using PBKDF2HMAC with SHA256.

Client-Server Communication

Python's socket programming is used to establish and maintain TCP connections with clients. This choice ensures reliable, ordered, and error-checked transmission of encrypted messages, which is paramount in a secure messaging application.

Client implementation is done using **client_updated.py** script. The client-side application uses a GUI (Graphical User Interface) built with Tkinter. It allows users to connect to the server, send, and receive encrypted messages.

- **Socket Programming:** Python's socket library is used for network communication. Clients connect to the server over TCP, ensuring reliable, ordered, and error-checked delivery of messages.
- **Message Encryption/Decryption**: Every message sent is first timestamped, then encrypted using the AESEncryption class, and finally transmitted to the server. Incoming messages are decrypted in a similar manner.
- **Key Functions**
    - connect(self): Establishes a connection to the server and initializes the encryption mechanism.
    - send_message(self): Encrypts and sends messages to the server.
    - receive_messages(self): Constantly listens for incoming messages from the server and decrypts them.

Server Implementation is done using the **server_updated.py** script. The server-side application mirrors the client's structure. It listens for incoming connections and facilitates message exchange between clients.

- **Key Functions**:
    - **accept_connections(self)**: Listens for and accepts incoming client connections.
    - **handle_client(self, client_socket)**: Manages communication with connected clients, including receiving and decrypting messages. Updates the password and key after every 5 messages. This provides enhanced security by ensuring that the key derivation parameters are always fresh, further protecting against advanced cryptanalytic attacks that may attempt to exploit static salt values.

Justification of Design Choices

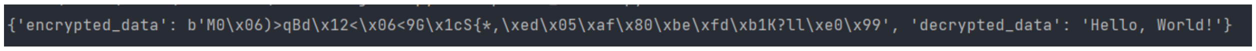
The use AES in our application establishes a comprehensive and resilient security framework. AES, known for its robustness and widespread acceptance in the cryptographic community, serves as the primary encryption standard, ensuring a high level of

Utilizing PBKDF2 with SHA256 for key derivation represents a conscious decision to align with industry best practices in key management. This method not only reinforces the strength of the cryptographic keys derived from user passwords but also adds a layer of protection against brute-force attacks. Moreover, the incorporation of a dynamic key updating mechanism is a proactive security measure. It addresses the potential risks associated with long-term key exposure, ensuring that even if a key is compromised, its effective duration and impact are minimized.

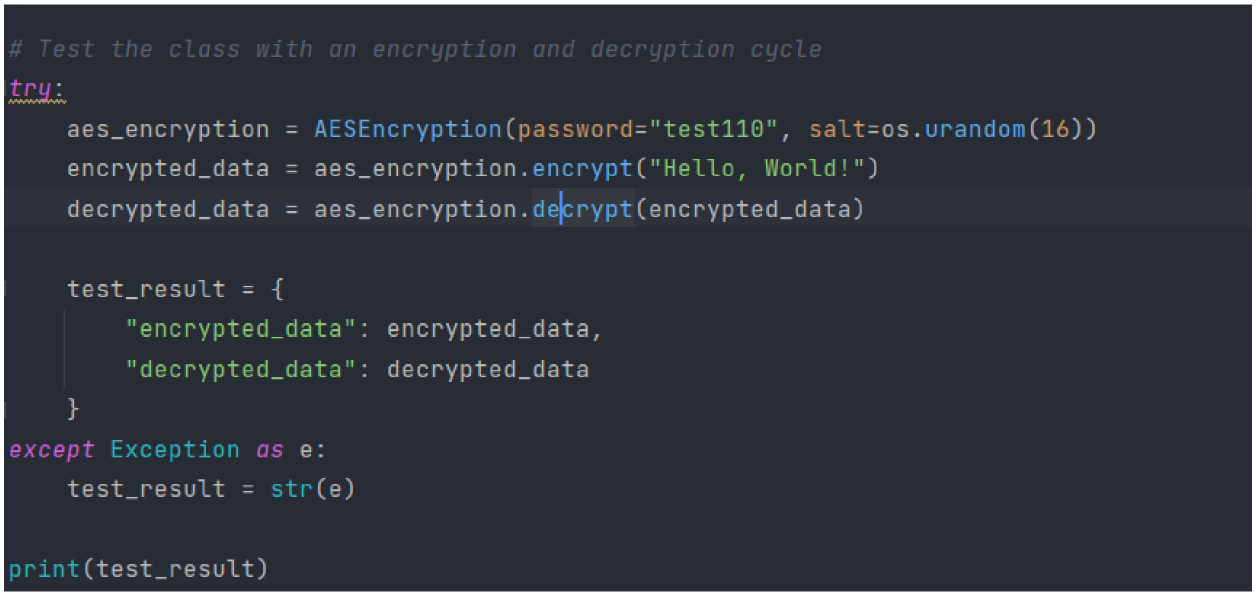
The choice of TCP over UDP for network communication is dictated by the necessity for reliability in message delivery. TCP, with its inherent mechanisms for ensuring the correct order and completeness of the transmitted data, is pivotal in maintaining the integrity of communication in our application. This decision underscores our commitment to providing a secure, reliable, and consistent user experience, where every message is delivered without loss or corruption.

Results

AES Test Encrypt and Decrypt

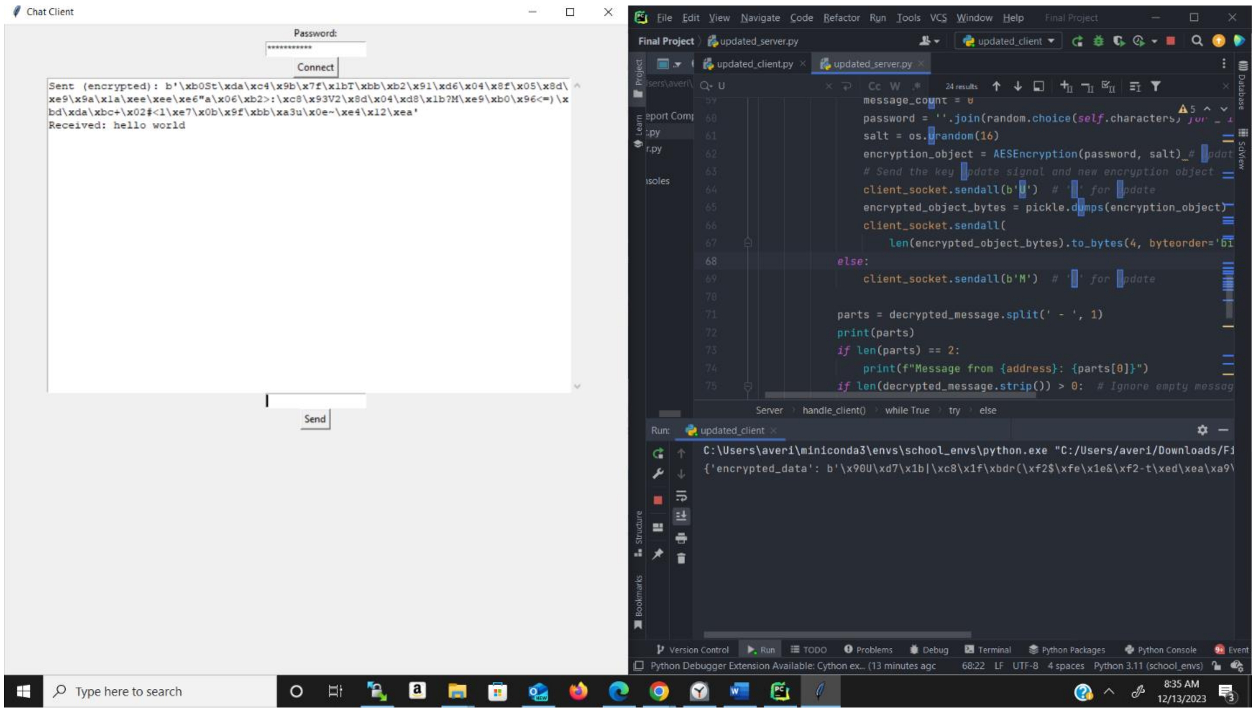


```
{'encrypted_data': b'M0\x06)>qBd\x12<\x06<9G\x1cS{*,\xed\x05\xaf\x80\xbe\xfd\xb1K?ll\xe0\x99', 'decrypted_data': 'Hello, World!'}
```



```python
# Test the class with an encryption and decryption cycle
try:
    aes_encryption = AESEncryption(password="test110", salt=os.urandom(16))
    encrypted_data = aes_encryption.encrypt("Hello, World!")
    decrypted_data = aes_encryption.decrypt(encrypted_data)

    test_result = {
        "encrypted_data": encrypted_data,
        "decrypted_data": decrypted_data
    }
except Exception as e:
    test_result = str(e)

print(test_result)
```

AES GUI Encrypt and Decrypt



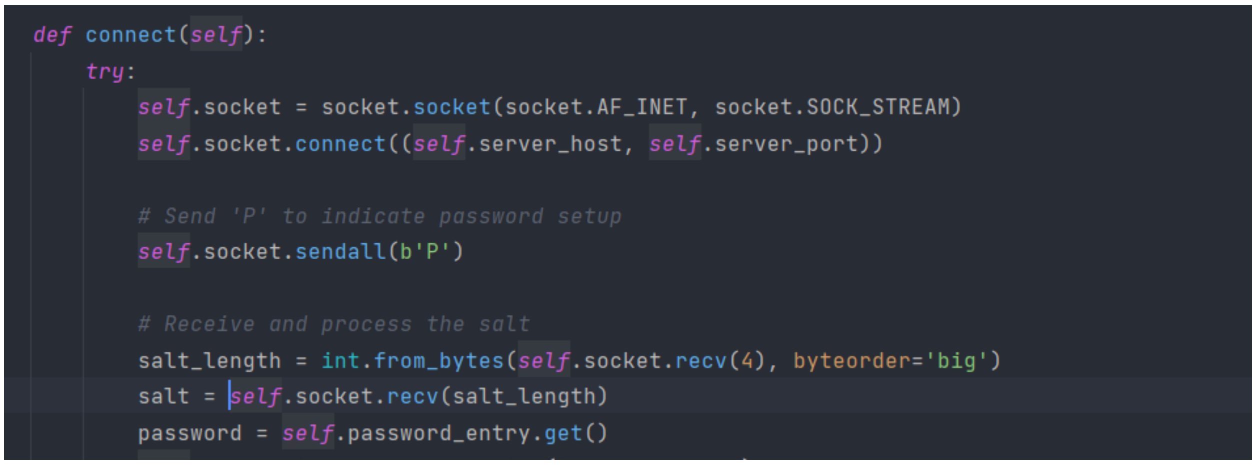
```
Sent (encrypted): b'\xb0St\xda\xc4\x9b\x7f\x1bT\xbb\xb2\x91\xd6\x04\x8f\x05\x8d\
xe9\x9a\x1a\xee\xee\xe6"a\x06\xb2>:\xc8\x93V2\x8d\x04\xd8\x1b?M\xe9\xb0\x96<=)\x
bd\xda\xbc+\x02#<1\xe7\x0b\x9f\xbb\xa3u\x0e~\xe4\x12\xea'
Received: hello world
```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
updated_server.py
updated_client
updated_client.py
updated_server.py
24 results
Server › handle_client() › while True › try › else
Run: updated_client

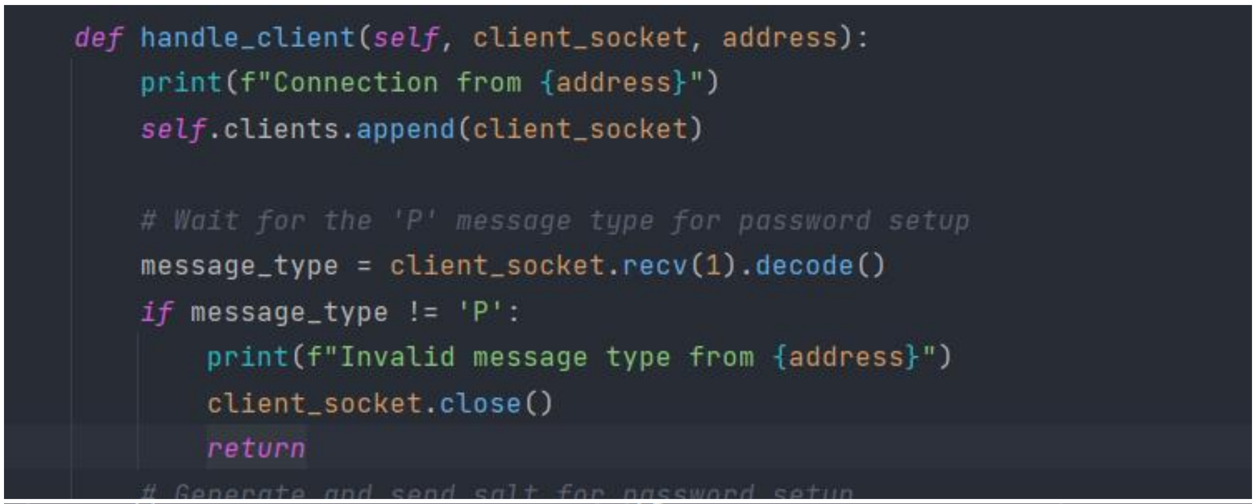
```
        message_count = 0
        password = ''.join(random.choice(self.characters) for _ i
        salt = os.urandom(16)
        encryption_object = AESEncryption(password, salt) # Updat
        # Send the key update signal and new encryption object
        client_socket.sendall(b'U')  # 'U' for update
        encrypted_object_bytes = pickle.dumps(encryption_object)
        client_socket.sendall(
            len(encrypted_object_bytes).to_bytes(4, byteorder='bi
    else:
        client_socket.sendall(b'M')  # 'M' for update

    parts = decrypted_message.split(' - ', 1)
    print(parts)
    if len(parts) == 2:
        print(f"Message from {address}: {parts[0]}")
    if len(decrypted_message.strip()) > 0:  # Ignore empty messag
```

```
C:\Users\averi\miniconda3\envs\school_envs\python.exe "C:/Users/averi/Downloads/Fi
{'encrypted_data': b'\x90U\xd7\x1b|\xc8\x1f\xbdr(\xf2$\xfe\x1e&\xf2-t\xed\xea\xa9\
```

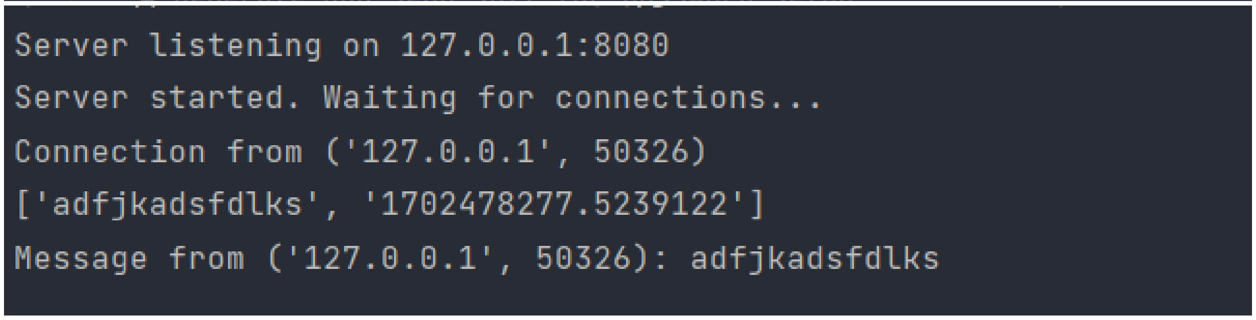
Socket Communication



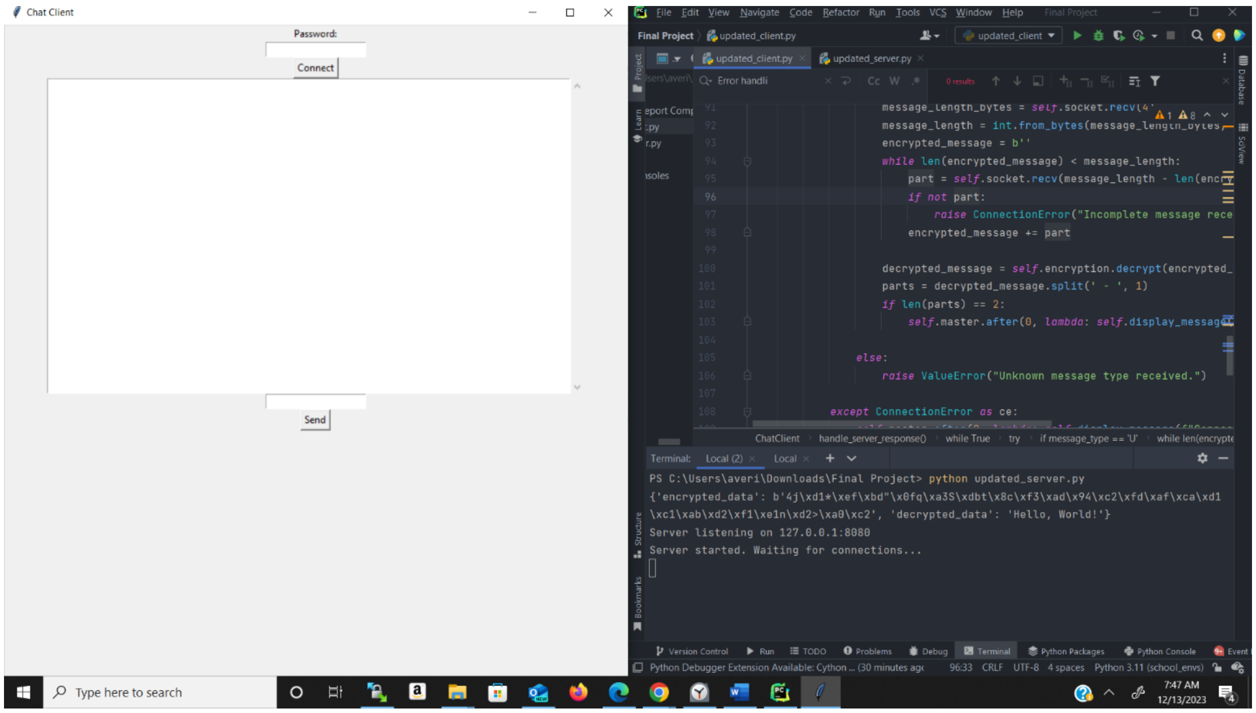
```python
def connect(self):
    try:
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((self.server_host, self.server_port))

        # Send 'P' to indicate password setup
        self.socket.sendall(b'P')

        # Receive and process the salt
        salt_length = int.from_bytes(self.socket.recv(4), byteorder='big')
        salt = self.socket.recv(salt_length)
        password = self.password_entry.get()
```



```python
    def handle_client(self, client_socket, address):
        print(f"Connection from {address}")
        self.clients.append(client_socket)

        # Wait for the 'P' message type for password setup
        message_type = client_socket.recv(1).decode()
        if message_type != 'P':
            print(f"Invalid message type from {address}")
            client_socket.close()
            return
        # Generate and send salt for password setup
```



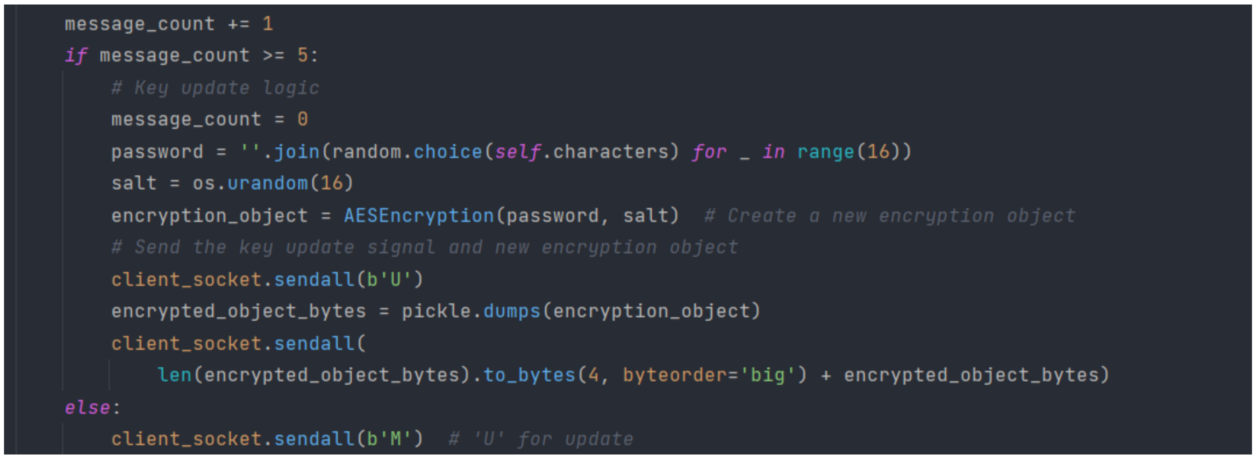
```
Server listening on 127.0.0.1:8080
Server started. Waiting for connections...
Connection from ('127.0.0.1', 50326)
['adfjkadsfdlks', '1702478277.5239122']
Message from ('127.0.0.1', 50326): adfjkadsfdlks
```



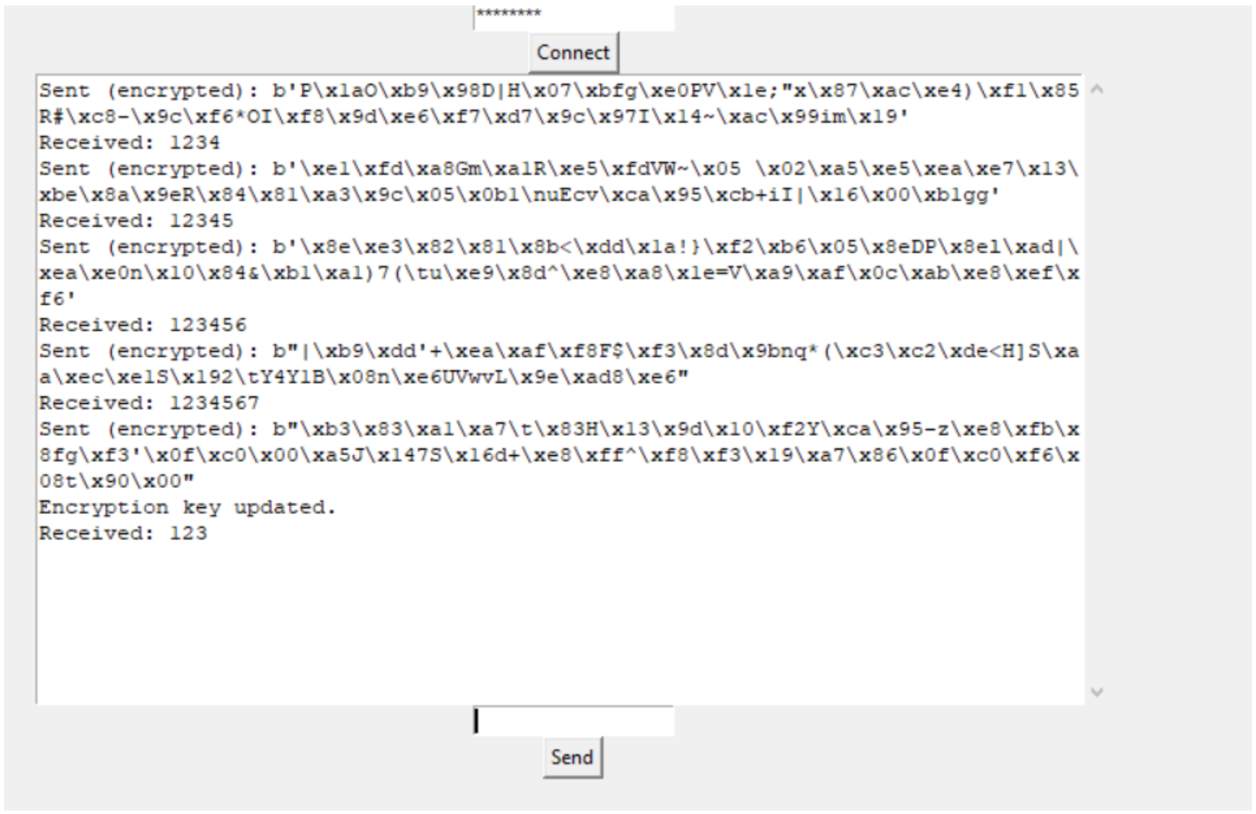
```python
                    message_length_bytes = self.socket.recv(4
                    message_length = int.from_bytes(message_length_bytes,
                    encrypted_message = b''
                    while len(encrypted_message) < message_length:
                        part = self.socket.recv(message_length - len(encry
                        if not part:
                            raise ConnectionError("Incomplete message rece
                        encrypted_message += part

                    decrypted_message = self.encryption.decrypt(encrypted_
                    parts = decrypted_message.split(' - ', 1)
                    if len(parts) == 2:
                        self.master.after(0, lambda: self.display_message

                else:
                    raise ValueError("Unknown message type received.")

            except ConnectionError as ce:
```

```
PS C:\Users\averi\Downloads\Final Project> python updated_server.py
{'encrypted_data': b'4j\xd1*\xef\xbd"\x0fq\xa3S\xdbt\x8c\xf3\xad\x94\xc2\xfd\xaf\xca\xd1
\xc1\xab\xd2\xf1\xe1n\xd2>\xa0\xc2', 'decrypted_data': 'Hello, World!'}
Server listening on 127.0.0.1:8080
Server started. Waiting for connections...
```

Key Update



```
message_count += 1
if message_count >= 5:
    # Key update logic
    message_count = 0
    password = ''.join(random.choice(self.characters) for _ in range(16))
    salt = os.urandom(16)
    encryption_object = AESEncryption(password, salt)  # Create a new encryption object
    # Send the key update signal and new encryption object
    client_socket.sendall(b'U')
    encrypted_object_bytes = pickle.dumps(encryption_object)
    client_socket.sendall(
        len(encrypted_object_bytes).to_bytes(4, byteorder='big') + encrypted_object_bytes)
else:
    client_socket.sendall(b'M')  # 'U' for update
```



```
Sent (encrypted): b'P\x1aO\xb9\x98D|H\x07\xbfg\xe0PV\x1e;"x\x87\xac\xe4)\xf1\x85
R#\xc8-\x9c\xf6*OI\xf8\x9d\xe6\xf7\xd7\x9c\x97I\x14~\xac\x99im\x19'
Received: 1234
Sent (encrypted): b'\xe1\xfd\xa8Gm\xa1R\xe5\xfdVW~\x05 \x02\xa5\xe5\xea\xe7\x13\
xbe\x8a\x9eR\x84\x81\xa3\x9c\x05\x0b1\nuEcv\xca\x95\xcb+iI|\x16\x00\xb1gg'
Received: 12345
Sent (encrypted): b'\x8e\xe3\x82\x81\x8b<\xdd\x1a!}\xf2\xb6\x05\x8eDP\x8e1\xad|\
xea\xe0n\x10\x84&\xb1\xa1)7(\tu\xe9\x8d^\xe8\xa8\x1e=V\xa9\xaf\x0c\xab\xe8\xef\x
f6'
Received: 123456
Sent (encrypted): b"|\xb9\xdd'+\xea\xaf\xf8F$\xf3\x8d\x9bnq*(\xc3\xc2\xde<H]S\xa
a\xec\xe1S\x192\tY4Y1B\x08n\xe6UVwvL\x9e\xad8\xe6"
Received: 1234567
Sent (encrypted): b"\xb3\x83\xa1\xa7\t\x83H\x13\x9d\x10\xf2Y\xca\x95-z\xe8\xfb\x
8fg\xf3'\x0f\xc0\x00\xa5J\x147S\x16d+\xe8\xff^\xf8\xf3\x19\xa7\x86\x0f\xc0\xf6\x
08t\x90\x00"
Encryption key updated.
Received: 123
```

Discussion

Our secure chat application integrates robust cryptographic practices, such as the use of different salts for key derivation of AES. This method mitigates potential risks associated with key correlation and strengthens the overall security model of our system. However, as with any security system, there is always room for enhancement.

Scalability, an aspect not deeply explored in the current server-client model, is crucial for handling many users simultaneously. The existing socket programming setup in **client_updated.py** and **server_updated.py** could be optimized for better concurrent connections management. Implementing asynchronous communication or a more efficient message queuing mechanism could greatly enhance performance under heavy user loads. Future work could focus on stress testing the application under heavy loads and refining the architecture to support a larger user base while maintaining performance and security.

Another area for improvement is the user interface. While functional, the current GUI, built using tkinter, is quite basic. Upgrading the GUI to be more visually appealing and user-friendly would significantly enhance the overall user experience. This could include modernizing the design, improving the layout for better usability, and adding features like emoticons, file sharing, and more interactive notifications. Technologies like PyQt or web-based frameworks could be considered for a more sophisticated and responsive interface.

Conclusion

In conclusion, while the application adeptly demonstrates the application of encryption for secure communication, it has room for improvement. Enhancements in GUI design, scalability, additional security layers, and refining the key management process, as highlighted in specific code segments, would not only address the current limitations but also significantly advance the application's capabilities. These improvements, grounded in the current codebase, pave the way for transforming this project into a more robust, user-friendly, and secure communication platform.but also prepare the platform for future challenges and advancements in the field of secure digital communication.