Contents

Introduction to Information and Network Security Final Project

Abid Azad

December 2023

Note: Due to the extensive length of the coding portion, it has been placed at the end of this report. References to specific sections are made throughout the document.

1 Quick Overview

Considering the comprehensive nature of the report, which includes an interactive Chat User Interface (UI), I've included this section to promptly address questions related to the assignment. The majority of the code used in this segment will be elaborated upon in subsequent sections of the report and is also provided in its entirety at the end.

1.1 Implement the Diffie-Hellman algorithm

```
import random

primeNumber =
    102188617217178804476387977160129334431745945009730065519337094992129677228373
primitiveRoot = 2

class User:
    pass

# Initialize two users
user1 = User()
user2 = User()

# DIFFIE-HELLMAN KEY EXCHANGE

# Users generate their Secret Key
user1.secret_integer = random.randint(2, primeNumber - 2)
user2.secret_integer = random.randint(2, primeNumber - 2)

# Users calculate their public keys using primitive root
```

```
c.DHPublicKey = pow(primitiveRoot, user1.secret_integer,
    primeNumber)
user2.DHPublicKey = pow(primitiveRoot, user2.secret_integer,
    primeNumber)

# Users exchange their public keys
user1.receivedDHPublicKey = user2.DHPublicKey
user2.receivedDHPublicKey = user1.DHPublicKey

# Users calculate their shared secret key
user1.sharedSecret = pow(user1.receivedDHPublicKey,
    user1.secret_integer, primeNumber)
user2.sharedSecret = pow(user2.receivedDHPublicKey,
    user2.secret_integer, primeNumber)
```

The provided Python code implements the Diffie-Hellman key exchange algorithm between two users, user1 and user2. In this algorithm, each user generates a secret key (secret_integer) and calculates a public key (DHPublicKey) using a primitive root, a prime number (primeNumber), and modular exponentiation. The users then exchange their public keys, allowing each to calculate a shared secret key (sharedSecret) using the received public key and their own secret key. The shared secret is used for secure communication between the two users. The algorithm relies on the mathematical properties of modular arithmetic to ensure that even if the public keys are intercepted during the exchange, deriving the secret key without knowledge of the private keys is computationally infeasible.

1.2 Check if the key generated by users U1 and U2 are the same.

```
print("Shared Key generated by User U1:", user1.sharedSecret)
print("Shared Key generated by User U2:", user2.sharedSecret)
print("User 1 is sharing the same key as User 2:
    "+(str(user1.sharedSecret == user2.sharedSecret)))
```

Output:

```
Shared Key generated by User U1: 6619624017324270116489091100918429832043040561887
79307337801948257621594659 69
Shared Key generated by User U2: 6619624017324270116489091100918429832043040561887
79307337801948257621594659 69
User 1 is sharing the same key as User 2: True
```

1.3 LFSR and RSA

```
class LFSR:
    def __init__(self, seed, taps):
        self.state = seed
        self.taps = taps
```

```
    def shift(self):
        feedback = sum(self.state[tap] for tap in self.taps)
            % 2
        self.state = [feedback] + self.state[:-1]
        return feedback

    def generate_key(self, length):
        key = []
        for _ in range(length):
            key.append(self.shift())
        binary_string = ''.join(map(str, key))
        generated_key = int(binary_string, 2)
        return binary_string, generated_key
```

The given code defines a Linear Feedback Shift Register (LFSR) class, which is a simple shift register used in cryptography and error detection. It takes a seed value and a list of taps as input during initialization. The shift method performs a shift operation based on the feedback from the taps, and the generate_key method generates a key of the specified length using the LFSR. The key is returned as both a binary string and its integer representation.

```
# Define the seed and feedback positions for the LFSR
seed = [1, 0, 1, 0]
shiftFeedbackPositions = [0, 2, 3]

# Create an instance of the LFSR class feedback positions
lfsr = LFSR(seed, shiftFeedbackPositions)

# Specify the length of the key
key_length = 16

# Generate a key of the specified length
user1_generated_key_binary, user1_generated_key =
    lfsr.generate_key(key_length)

print("Generated LFSR Key:", user1_generated_key_binary)
```

```
Print Output:
Generated LFSR Key: 0101010101010101
```

```
#RSA Functions#
import math
from sympy import isprime
from gmpy2 import powmod

'''Helper function that generates a large prime number with
    the specified number of bits, in which for this
    assignment is 256.'''
```

```
def generate_large_prime(bits):
    while True:
        num = random.randrange(0, 2**bits - 1)
        if isprime(num):
            return num
'''Calculates the modular exponetiation of a given message,
    power, and basis using the 'powmod' function from the
    gmpy2 library'''
def exponentiation(message, power, basis):
    return powmod(message, power, basis)

'''Helper function that incorporates the extended euclidean
    algorithm to help determine the inverse value within the
    inverse_finder function.'''
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        g, x, y = extended_gcd(b % a, a)
        return g, y - (b // a) * x, x

'''Finds the modular inverse of a given number a modulo n
    using the extended euclidean algorithm.'''
def inverse_finder(a, n):
    g, x, _ = extended_gcd(a, n)
    if g != 1:
        raise ValueError(f"The modular inverse does not exist
            for {a} modulo {n}")
    else:
        return x % n
'''A function that generates RSA public and private keys. It
    takes an optional parameter e for the rsaKeyInput; if not
    provided, it defaults to 3.'''
def RSA_key_generate():
    e = 65537
    while(True):
        p = generate_large_prime(256)
        q = p

        while(p == q):
            q = generate_large_prime(256)
        n = p * q
        euler = (p-1) * (q-1)

        if(math.gcd(euler, e) == 1):
            break
    d = inverse_finder(e, euler)
    publicKey = [e, n]
    privateKey = [d, n]
    return publicKey, privateKey
```

```
'''A function encrypts a numeric message or a string using
    RSA encryption with a given key.'''
def RSA_encrypt(message, key):
    if not isinstance(message, str):
        return exponentiation(message, key[0], key[1])
    elif isinstance(message, str):
        ciphertext = []
        for element in range(0, len(message)):
            ciphertext.append(int(exponentiation(ord(message[element]),
                key[0], key[1])))
        return ciphertext

'''A function decrypts a numeric message or a list of numeric
    values using RSA decryption with a given key.'''
def RSA_decrypt(message, key):
    if not isinstance(message, str) and not
        isinstance(message, list):
        return RSA_encrypt(message, key)
    elif isinstance(message, list):
        decrpyted = ''
        for element in range(0, len(message)):
            decrpyted+= chr(exponentiation(message[element],
                key[0], key[1]))
        return decrpyted
```

The provided code defines a set of functions for RSA encryption and decryption. It includes a function to generate large prime numbers, modular exponentiation using the powmod function from the gmpy2 library, extended Euclidean algorithm for finding modular inverses, and functions for RSA key generation, encryption, and decryption. The RSA key generation function creates public and private keys based on the specified bit size, defaulting to 256 bits, while the encryption and decryption functions handle both numeric and string input. The code incorporates essential cryptographic principles, such as prime number generation, modular arithmetic, and the RSA algorithm, to enable secure communication and data protection.

```
# User 1 generates RSA keys
user1.RSAPublicKey, user1.RSAPrivateKey = RSA_key_generate()

# User 1 encrypts his LFSR
encryptedMessage = RSA_encrypt(user1.generated_key_binary,
    user1.RSAPublicKey)

# Assume User 1 sends his encrypted message and private key
    to User 2.
# User 2 decrypts the message.
user2.receivedMessage = RSA_decrypt(encryptedMessage,
    user1.RSAPrivateKey)
```

```
# Check if the LFSR key was successfully sent through RSA
print("User 2 Recieved the Message: " +user2.receivedMessage)
print("Successfully sent LFSR Key Through RSA: " +
    str(user1.generated_key_binary == user2.receivedMessage))
```

```
Printed Output:
User 2 Recieved the Message: 0101010101010101
Sucessfully sent LFSR Key Through RSA: True
```

1.4 Stream Cipher

```
def text_to_bits(text):
    return ''.join(format(ord(char), '08b') for char in text)

def bits_to_text(bits):
    return ''.join(chr(int(bits[i:i+8], 2)) for i in range(0,
        len(bits), 8))

# Stream Cipher
def encrypt(text, key):
    bits = text_to_bits(text)
    encrypted_bits = [int(bit) ^ int(key[i % len(key)]) for
        i, bit in enumerate(bits)]
    return ''.join(map(str, encrypted_bits))

def decrypt(ciphertext, key):
    decrypted_bits = [int(bit) ^ int(key[i % len(key)]) for
        i, bit in enumerate(ciphertext)]
    return bits_to_text(''.join(map(str, decrypted_bits)))
```

This Python code defines functions for converting text to binary representation (text_to_bits), binary to text (bits_to_text), and implements a simple stream cipher encryption (encrypt) and decryption (decrypt). The text_to_bits function converts each character in the input text to its 8-bit binary representation, and bits_to_text reverses this process. The encrypt function XORs each bit of the binary representation of the input text with the corresponding bit of the key, producing the encrypted result. The decrypt function performs a similar XOR operation to reverse the encryption and retrieve the original text. Note that this stream cipher operates at the bit level, and the key is repeated as needed during encryption and decryption.

```
Messages = ["Hello!",
            "Welcome to Introduction to Information and
                Network Security!",
            "In the vast landscape of technology and
                innovation, the intertwining threads of
                progress and human ingenuity weave a
                narrative of constant evolution. From the
                advent of the internet, a global web
```

```
            connecting minds across oceans, to the
            intricate algorithms that power artificial
            intelligence, we find ourselves in an era
            defined by rapid change and
            interconnectedness."]

#For Demo Purposes, lets use User 1's generated LFSR Key
for message in Messages:
    print("Original Message: "+message)
    encrypted = encrypt(message, user1.generated_key_binary)
    print("Encrypted Message: "+encrypted)
    decrypted = decrypt (encrypted,
        user1.generated_key_binary)
    print(decrypted)
```

```
Printed Content
Original Message: Hello!
Encrypted Message: 000111010011000000111001001110010011101001110100
Decrypted Message: Hello!
Successfully Encrypted and Decrypted Message: True
Original Message: Welcome to Introduction to Information and Network Security!
Encrypted Message: 0000001000110000001110010011011000111010001110000001100000
11101010010000100111010011101010001110000111011001000010 0
1001110011101000110001001000000011011000100001001111000011
1010001110110111010100100001001110100111010100011100001111
10011001100111010001001110011100000110100001000010011110000
11101000111011011101010011010000111011001100010111010100011 0
11001100000001000010010001000111010001001110011111001110101 00
0001100011000000110110001000000010011100111100001000010010 11
0001110100
Decrypted Message: Welcome to Introduction to Information and Network Security!
Successfully Encrypted and Decrypted Message: True
Original Message: In the vast landscape of technology and innovation, the
intertwining threads
of progress and human ingenuity weave a narrative of constant evolution.
Encrypted Message:
0001110000111011011101010010000100111101001100000111010100100011001 1
0100001001100010000101110101001110010011010000111011001100010010011 0
0011011000110100001001010011000001110101001110100011001101110101001 0
0001001100000011011000111101001110110011101000111001001110100011001 0
0010110001110101001101000011101100110001011101010011110000111011001 1
1011001110100010001100110100001000010011110000111010001110110111100 1
0111010100100001001111010011000001110101001111000011101100100001001 1
0000001001110010000100100010001111000011101100111100001110110011001 0
0111010100100001001111010010011100110000001101000011000100100110011 1
0101001110100011001101110101001001010010011100111010001100100010011 1
```

```
0011000000100110001001100111010100110100001110110011000101110101 0011
1101001000000011100000110100001110110111010100111100001110110011 0010
0011000000111011001000000011110000100001001011000111010100100010 0011
0000001101000001000110011000001110101001101000111010100111011001 10100
0010011100100111001101000010000100111100001000110011000001110101 0011
1010001100110111010100110110001110100011101100100110001000010011 0100
0011101100100001011101010011000000100011001110100011100100100000 00010
0001001111000011101000111011011110110111010100010011001001110011 1010
0011100001110101001000010011110100110000011101010011010000110001 0010
0011001100000011101100100001011101010011101000110011011101010010 0111
0011010000100101001111000011000101110101001101110010110001110101 0010
0111001101000010010100111100001100010111010100110110001111010011 0100
0011101100110010001100000111010100110100001110110011000101110101 0011
1100001110110010000100110000000100111001101100011101000111011001 11011
0011000000110110001000010011000000011000100111011001100000001001 100010
011001111011
Decrypted Message: In the vast landscape of technology and innovation, the intertwining thre
of progress and human ingenuity weave a narrative of constant evolution.
Successfully Encrypted and Decrypted Message: True
```

1.5 AES

```python
from cryptography.hazmat.primitives.ciphers import Cipher,
    algorithms, modes
from cryptography.hazmat.backends import default_backend
from base64 import b64encode, b64decode

def pad(text):
    # PKCS7 padding
    block_size = 16
    if isinstance(text, str):
        text = text.encode('utf-8')  # Convert string to bytes
    pad_size = block_size - len(text) % block_size
    return text + bytes([pad_size] * pad_size)

def unpad(text):
    pad_size = text[-1]
    return text[:-pad_size]

def encrypt(plaintext, key, mode):
    plaintext = pad(plaintext)
    cipher = Cipher(algorithms.AES(key), mode,
        backend=default_backend())
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(plaintext) +
        encryptor.finalize()
```

```
    return b64encode(ciphertext)

def decrypt(ciphertext, key, mode):
    ciphertext = b64decode(ciphertext)
    cipher = Cipher(algorithms.AES(key), mode,
        backend=default_backend())
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) +
        decryptor.finalize()
    return unpad(plaintext)
```

This Python code employs the cryptography library to implement AES encryption and decryption in both Electronic Codebook (ECB) and Cipher Block Chaining (CBC) modes. The pad function adds PKCS7 padding to the plaintext, and unpad removes the padding. The encrypt function takes plaintext, a key, and a mode as input, pads the plaintext, encrypts it using AES in the specified mode, and returns the Base64-encoded ciphertext. The decrypt function takes a ciphertext, key, and mode, decodes the ciphertext, decrypts it using AES in the specified mode, and returns the unpadded plaintext. The code is designed to handle both ECB and CBC modes, allowing flexibility in choosing the appropriate mode for encryption.

```
from AES import encrypt as aes_encrypt, decrypt as aes_decrypt
from cryptography.hazmat.primitives.ciphers import modes
#Using User 1's Shared Secret Key
#Format User 1's Key into a Bytes Object
integer_value = user1.sharedSecret
byte_value =
    integer_value.to_bytes((integer_value.bit_length() + 7)
    // 8, byteorder='big')
#Run the same tests using AES ECB
print("TESTS FOR AES ECB:")
for message in Messages:
    print("Original Message: "+message)
    encrypted = aes_encrypt(message, byte_value, modes.ECB())
    print("Encrypted Message: "+encrypted.decode())
    decrypted = aes_decrypt (encrypted, byte_value,
        modes.ECB())
    print("Decrypted Message: "+decrypted.decode())
    print("Successfully Encrypted and Decrypted Message:
        "+(str(message == decrypted.decode())))
```

```
Printed Output:
TESTS FOR AES ECB:
Original Message: Hello!
Encrypted Message: s2w6RuvNk+YRg1azkGxL4A==
Decrypted Message: Hello!
```

```
Successfully Encrypted and Decrypted Message: True
Original Message: Welcome to Introduction to Information and Network Security!
Encrypted Message:
c10iKJ7amSxlHJR7ESIiPKOD1ouFGUfI/wUsc+ma2nqKodOPKquuc2L0/g30Sp/uuzAQ+kSawclnaNCePjXPVw==
Decrypted Message: Welcome to Introduction to Information and Network Security!
Successfully Encrypted and Decrypted Message: True
Original Message: In the vast landscape of technology and
innovation, the intertwining threads of progress and human
ingenuity weave a narrative of constant evolution.
Encrypted Message:
KDOpR5jSTOyAAjWOJru+7gzudA5fSCpTFWDRR/GYpA1qHnCJI8hWjsHjQvDE3SY0I4vq
jdeAEY12v9joWaTiXHGpqXJB2Eschgt+2Zs3dT12I4nGpSogzqZsni0YQMrgEfl66b+Q
oZrqJ4scE29gPqhiLPkerZCscDzADq9JEOO9MNejhtBCifTbnDoPrQ308KbcN7Go5LDz
M3j1clkF2w==
Decrypted Message: In the vast landscape of technology and
innovation, the intertwining threads of progress and human
ingenuity weave a narrative of constant evolution.
Successfully Encrypted and Decrypted Message: True
```

```
import os
#Setup initialization vector for Cipher Block Chaining
iv = os.urandom(16)
print("TESTS FOR AES CBC:")
for message in Messages:
    print("Original Message: "+message)
    encrypted = aes_encrypt(message, byte_value,
        modes.CBC(iv))
    print("Encrypted Message: "+encrypted.decode())
    decrypted = aes_decrypt (encrypted, byte_value,
        modes.CBC(iv))
    print("Decrypted Message: "+decrypted.decode())
    print("Successfully Encrypted and Decrypted Message:
        "+(str(message == decrypted.decode())))
```

```
TESTS FOR AES CBC:
Original Message: Hello!
Encrypted Message: fpnSQ6+Nfh6cfm92EHeRoA==
Decrypted Message: Hello!
Successfully Encrypted and Decrypted Message: True
Original Message: Welcome to Introduction to Information and Network Security!
Encrypted Message: dcxJR/ZCpPUfUBDmY8chu/1/bE10UpeiXvdY4t/qYvU3pzksQX8AbCZ+r7jH
Cd8C29zA3u2HArlM3yB03/eoCA==
Decrypted Message: Welcome to Introduction to Information and Network Security!
Successfully Encrypted and Decrypted Message: True
Original Message: In the vast landscape of technology and innovation, the intertwining
threads
of progress and human ingenuity weave a narrative of constant evolution.
```

```
Encrypted Message:
bKnukP9J80RB6W37zXZmbSj215zoeTEys6EyQQ5UKpezH3rtkBeyH3I87UhPMmBg4zuj
5OJ6xTyd6GT69yKnbbhhqjf9O6TiBriNfqjfD5UulWQ84qsPvyYPsELDVWS6DMs2e3QG
5LafbY0+ArqYv1l++MU/gGHXx273tGy/K6HVPtiT0wepd1oSivjzCndf+xBW5AZuHRkz
+8DzJilF+A==
Decrypted Message: In the vast landscape of technology and innovation, the intertwining thre
and human ingenuity weave a narrative of constant evolution.
Successfully Encrypted and Decrypted Message: True
```

Optional Functionality: Encrypting and Decryption Photos (Assume there is a test image of a cat with sunglasses.)

```python
Within AES.py
from PIL import Image
import io
import matplotlib.pyplot as plt

def encrypt_image(image_path, key, mode):
    with open(image_path, 'rb') as image_file:
        image_data = image_file.read()
    ciphertext = encrypt(image_data, key, mode)
    return ciphertext

def decrypt_image(ciphertext, key, mode):
    decrypted_data = decrypt(ciphertext, key, mode)
    return decrypted_data
def view_image(image_path):
    image = Image.open(image_path)
    plt.imshow(image)
    plt.axis('off')  # Turn off axis labels
    plt.show()

def view_image_from_decrypted(image_data):
    image = Image.open(io.BytesIO(image_data))
    plt.imshow(image)
    plt.axis('off')  # Turn off axis labels
    plt.show()
image_path = 'testImage.jpg'
view_image(image_path)
key = b'sixteen byte key'
ecb_ciphertext = encrypt_image(image_path, key, modes.ECB())
cbc_ciphertext = encrypt_image(image_path, key,
    modes.CBC(b'\x00' * 16))

print(f"Encrypted Image Data (ECB): {ecb_ciphertext}\n\n")
print(f"Encrypted Image Data (CBC): {cbc_ciphertext}")

decrypted_ecb = decrypt_image(ecb_ciphertext, key,
    modes.ECB())
```

```
decrypted_cbc = decrypt_image(cbc_ciphertext, key,
    modes.CBC(b'\x00' * 16))

print(f"Decrypted Image Data (ECB): {decrypted_ecb}\n\n")
print(f"Decrypted  Image Data (CBC): {decrypted_cbc}")

view_image_from_decrypted(decrypted_ecb)
view_image_from_decrypted(decrypted_cbc)
```

This Python code leverages the Pillow library for image processing and Matplotlib for visualization to showcase a simple image encryption and decryption process using the AES algorithm. The encrypt_image function reads an image from a specified file path, encrypts it with a given key using either ECB or CBC mode, and returns the Base64-encoded ciphertext. The decrypt_image function reverses the process by decrypting the ciphertext back to binary image data. The code then prints and displays the original image, encrypts it with both ECB and CBC modes, prints the corresponding ciphertexts, decrypts the ciphertexts, and finally displays the decrypted images.

Output

```
The raw image data is rather large. If you would like to see it,
please visit this google drive-link:

https://docs.google.com/document/d/184qtMwQQzuSAgoej7D7q-D4wg3eK5Xzd7-jTB7hAGtU/
edit?usp=sharing
```


Image encryption was achieved using the AES algorithm in both ECB and CBC modes. The encrypted image data, represented as Base64-encoded ciphertexts, demonstrated varying densities. The ciphertexts were denser than the original image data, reflecting the impact of encryption on file size. Notably, the encrypted images, when subjected to the corresponding decryption processes using the correct keys and modes, successfully yielded the original image data. This retrieval of the original image data reaffirms the reversibility of the AES encryption process, showcasing its effectiveness in preserving data integrity. Furthermore, the experiment emphasized the importance of securely managing encryption keys, as possessing the correct key was essential for decrypting the images and restoring them to their original form.

1.6 DES

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
from base64 import b64encode, b64decode

def encrypt(plaintext, key, mode, iv=None):
    if mode == DES.MODE_ECB:
        cipher = DES.new(key, DES.MODE_ECB)
    elif mode == DES.MODE_CBC:
        if iv is None:
            raise ValueError("IV is required for CBC mode")
        cipher = DES.new(key, DES.MODE_CBC, iv)
    else:
        raise ValueError("Invalid mode")

    plaintext = pad(plaintext.encode(), DES.block_size)
    ciphertext = cipher.encrypt(plaintext)

    return b64encode(ciphertext)

def decrypt(ciphertext, key, mode, iv=None):
    ciphertext = b64decode(ciphertext)

    if mode == DES.MODE_ECB:
        cipher = DES.new(key, DES.MODE_ECB)
    elif mode == DES.MODE_CBC:
        if iv is None:
            raise ValueError("IV is required for CBC mode")
        cipher = DES.new(key, DES.MODE_CBC, iv)
    else:
        raise ValueError("Invalid mode")

    plaintext = unpad(cipher.decrypt(ciphertext),
       DES.block_size)

    return plaintext.decode()
```

This Python code provides a simple implementation of the Data Encryption

Standard (DES) using the PyCryptodome library. The "encrypt" function takes plaintext, a secret key, a specified encryption mode (either Electronic Codebook (ECB) or Cipher Block Chaining (CBC)), and an optional initialization vector (IV). It then initializes a DES cipher object with the provided key, mode, and IV, if applicable. The plaintext is padded to the DES block size using PKCS7 padding, encrypted, and the resulting ciphertext is base64-encoded for readability. The "decrypt" function reverses this process by decoding the ciphertext, decrypting it using the DES cipher, and removing the padding to retrieve the original plaintext. The code ensures proper handling of encryption modes and IV requirements, throwing value errors for invalid inputs.

```
from DES import encrypt as des_encrypt, decrypt as des_decrypt
from Crypto.Cipher import DES

#Using a shorter key for DES
integer_value = random.getrandbits(64)
byte_value =
    integer_value.to_bytes((integer_value.bit_length() + 7)
    // 8, byteorder='big')
#Run the same tests using DES ECB
print("TESTS FOR DES ECB:")
for message in Messages:
     print("Original Message: "+message)
     encrypted = des_encrypt(message, byte_value, DES.MODE_ECB)
     print("Encrypted Message: "+encrypted.decode())
     decrypted = des_decrypt (encrypted, byte_value,
         DES.MODE_ECB)
     print("Decrypted Message: "+decrypted)
     print("Successfully Encrypted and Decrypted Message:
         "+(str(message == decrypted)))

import os
#Setup initialization vector for Cipher Block Chaining
iv = os.urandom(8)
print("TESTS FOR DES CBC:")
for message in Messages:
     print("Original Message: "+message)
     encrypted = des_encrypt(message, byte_value,
         DES.MODE_ECB, iv)
     print("Encrypted Message: "+encrypted.decode())
     decrypted = des_decrypt (encrypted, byte_value,
         DES.MODE_ECB, iv)
     print("Decrypted Message: "+decrypted)
     print("Successfully Encrypted and Decrypted Message:
         "+(str(message == decrypted)))
```

Printed Output:

```
TESTS FOR DES ECB:
Original Message: Hello!
Encrypted Message: EUnGMGy3T7U=
Decrypted Message: Hello!
Successfully Encrypted and Decrypted Message: True
Original Message: Welcome to Introduction to Information and Network Security!
Encrypted Message:
zHPt5+bX5ie82wjwSg0TTh1HnEjS/IX0XYRAmONXD2BTvcZ0mTjX1Nagk6Na6AcbqCfVg
Qx8Nad2q6aG9q4n4g==
Decrypted Message: Welcome to Introduction to Information and Network Security!
Successfully Encrypted and Decrypted Message: True
Original Message: In the vast landscape of technology and innovation, the intertwining
threads of progress and human ingenuity weave a narrative of constant evolution.
Encrypted Message:
LhXjL9sT3syq2ismNDiWVRmzPl0j6CZOMGmVcDKOaHzwI6yhVpcCwNq142QqbkFHp2vhB
PkVtZbysM7gWY8+aP/SUjxuK3jFdwwbtjjUTCwjHgZrDLvvSt/jZv7vDJ+0acNaAHjWgt
B+5mgUy0gBTqiu3UXQ2Yg0qVrO0izuabKuhXfp9TAjH0r7Qs1rzWDL7KExAGa9Sjk=
Decrypted Message: In the vast landscape of technology and innovation, the intertwining
threads of progress and human ingenuity weave a narrative of constant evolution.
Successfully Encrypted and Decrypted Message: True
TESTS FOR DES CBC:
Original Message: Hello!
Encrypted Message: EUnGMGy3T7U=
Decrypted Message: Hello!
Successfully Encrypted and Decrypted Message: True
Original Message: Welcome to Introduction to Information and Network Security!
Encrypted Message:
zHPt5+bX5ie82wjwSg0TTh1HnEjS/IX0XYRAmONXD2BTvcZ0mTjX1Nagk6Na6AcbqCfVg
Qx8Nad2q6aG9q4n4g==
Decrypted Message: Welcome to Introduction to Information and Network Security!
Successfully Encrypted and Decrypted Message: True
Original Message: In the vast landscape of technology and innovation, the intertwining
threads of progress and human ingenuity weave a narrative of constant evolution.
Encrypted Message:
LhXjL9sT3syq2ismNDiWVRmzPl0j6CZOMGmVcDKOaHzwI6yhVpcCwNq142QqbkFHp2vhB
PkVtZbysM7gWY8+aP/SUjxuK3jFdwwbtjjUTCwjHgZrDLvvSt/jZv7vDJ+0acNaAHjWgt
B+5mgUy0gBTqiu3UXQ2Yg0qVrO0izuabKuhXfp9TAjH0r7Qs1rzWDL7KExAGa9Sjk=
Decrypted Message: In the vast landscape of technology and innovation, the intertwining
threads of progress and human ingenuity weave a narrative of constant evolution.
Successfully Encrypted and Decrypted Message: True
```

1.7 AES compared to DES

```
import timeit
import matplotlib.pyplot as plt
import random
```

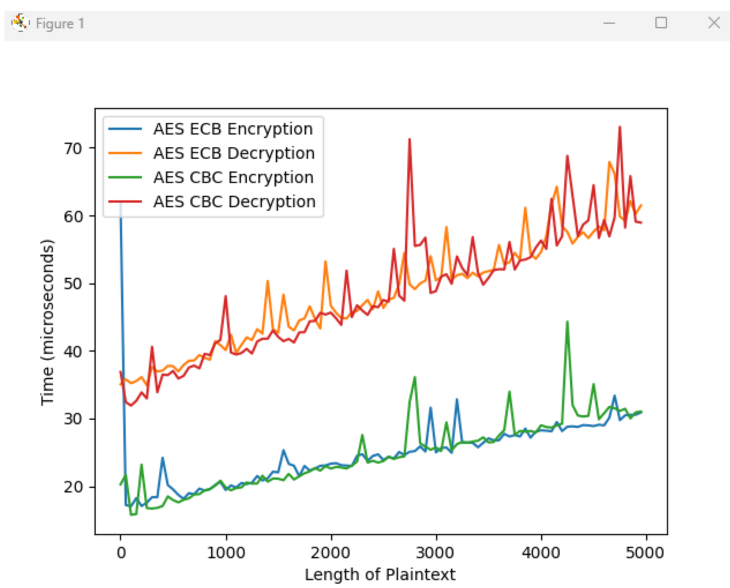
```
import string
from Crypto.Random import get_random_bytes
from AES import encrypt as aes_encrypt, decrypt as aes_decrypt
from DES import encrypt as des_encrypt, decrypt as des_decrypt
from cryptography.hazmat.primitives.ciphers import modes
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

def generate_random_string(length):
    return ''.join(random.choice(string.ascii_letters) for _
        in range(length))

def test_aes():
    key = get_random_bytes(16)  # 128-bit key for AES
    iv = get_random_bytes(16)
    modes_to_test = [modes.ECB(), modes.CBC(iv)]  # Add more
        modes if needed

    for mode in modes_to_test:
        times_encrypt = []
        times_decrypt = []
        lengths = list(range(1, 5001, 50))  # Vary the length
            of the plaintext

        for length in lengths:
            plaintext = generate_random_string(length)

            encrypt_time = timeit.timeit(lambda:
                aes_encrypt(plaintext, key, mode),
                number=1000)  # Increased number of iterations
            times_encrypt.append(encrypt_time * 1e6 / 1000)
                # Convert to microseconds, average time per
                encryption

            decrypt_time = timeit.timeit(lambda:
                aes_decrypt(aes_encrypt(plaintext, key,
                mode), key, mode).decode("utf-8"),
                number=1000)  # Increased number of iterations
            times_decrypt.append(decrypt_time * 1e6 / 1000)
                # Convert to microseconds, average time per
                decryption

            assert aes_decrypt(aes_encrypt(plaintext, key,
                mode), key, mode).decode("utf-8") ==
                plaintext  # Ensure decryption is correct

        # Plot results
        plt.plot(lengths, times_encrypt, label=f'AES
            {mode.name} Encryption')
```

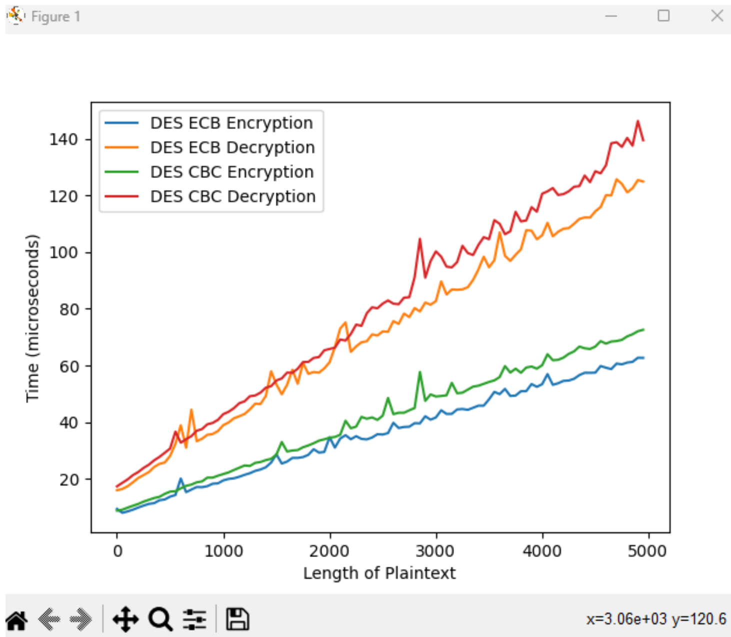
```
        plt.plot(lengths, times_decrypt, label=f'AES
            {mode.name} Decryption')

    plt.xlabel('Length of Plaintext')
    plt.ylabel('Time (microseconds)')
    plt.legend()
    plt.show()

def test_des():
    key = get_random_bytes(8)  # 64-bit key for DES
    iv = get_random_bytes(8)
    modes_to_test = [DES.MODE_ECB, DES.MODE_CBC]  # Add more
        modes if needed
    for mode in modes_to_test:
        times_encrypt = []
        times_decrypt = []
        lengths = list(range(1, 5001, 50))  # Vary the length
            of the plaintext

        for length in lengths:
            plaintext = generate_random_string(length)

            if mode == DES.MODE_CBC:
                encrypt_time = timeit.timeit(lambda:
                    des_encrypt(plaintext, key, mode, iv),
                    number=1000)
                decrypt_time = timeit.timeit(lambda:
                    des_decrypt(des_encrypt(plaintext, key,
                    mode, iv), key, mode, iv), number=1000)
            else:
                encrypt_time = timeit.timeit(lambda:
                    des_encrypt(plaintext, key, mode),
                    number=1000)
                decrypt_time = timeit.timeit(lambda:
                    des_decrypt(des_encrypt(plaintext, key,
                    mode), key, mode), number=1000)

            times_encrypt.append(encrypt_time * 1e6 / 1000)
                # Convert to microseconds, average time per
                encryption
            times_decrypt.append(decrypt_time * 1e6 / 1000)
                # Convert to microseconds, average time per
                decryption

            assert des_decrypt(des_encrypt(plaintext, key,
                mode, iv), key, mode, iv) == plaintext  #
                Ensure decryption is correct

        # Plot results
        if(mode == DES.MODE_ECB):
```

```
        title = "ECB"
    else:
        title = "CBC"
    plt.plot(lengths, times_encrypt, label=f'DES {title}
        Encryption')
    plt.plot(lengths, times_decrypt, label=f'DES {title}
        Decryption')

    plt.xlabel('Length of Plaintext')
    plt.ylabel('Time (microseconds)')
    plt.legend()
    plt.show()

if __name__ == "__main__":
    test_aes()
    test_des()
```

This Python script compares the performance of Advanced Encryption Standard (AES) and Data Encryption Standard (DES) using Matplotlib for visualization. The "test_aes" and "test_des" functions evaluate encryption and decryption times for varying plaintext lengths, displaying the efficiency gap between AES and DES. After executing this code, we get the following results:

1.8 Digital Signatures





In our AES and DES algorithm comparison, our Python implementation revealed insights into their performance across plaintext lengths. AES consistently showed execution times of 20 to 70 microseconds for encryption and decryption, indicating its reliability in balancing security and computational efficiency. On the other hand, DES exhibited slightly higher times, ranging from 20 to 140 microseconds. Despite this, DES displayed efficiency with smaller data sizes, emphasizing its relevance for specific applications. The gradual increase in execution times aligns with expected behavior, highlighting DES's adaptability to varying workloads. Overall, the results provide a nuanced view, guiding users to choose based on specific encryption needs and considerations.

1.9 Digital Signature

For choice of Digitial Signature, I shall be going with RSA Digital Signature. RSA digital signatures are preferred for security due to the mathematical complexity of factoring large primes. They ensure data integrity, sender authentication, and non-repudiation, providing a robust mechanism for secure communication. RSA's versatility, compliance with standards, and support for long key lengths make it widely adopted in various industries for digital signature applications.

```
import hashlib
def hash_message(message):
    # Hash the message using SHA-256
```

```
    sha256 = hashlib.sha256()
    sha256.update(str(message).encode('utf-8'))
    return int(sha256.hexdigest(), 16)

def RSA_sign(message, private_key):
    hashed_message = hash_message(message)
    signature = exponentiation(hashed_message,
        private_key[0], private_key[1])
    return signature

def RSA_verify(message, signature, public_key):
    hashed_message = hash_message(message)
    decrypted_signature = exponentiation(signature,
        public_key[0], public_key[1])

    if hashed_message == decrypted_signature:
        return True
    else:
        return False
```

These functions collectively implement a rudimentary digital signature scheme using the RSA algorithm. The hash_message function employs SHA-256 to produce a hash value for a given message, returning an integer representation of the hash. The RSA_sign function utilizes the private key to exponentiate the hashed message, generating a digital signature as an integer. On the verification side, the RSA_verify function hashes the input message, exponentiates the received signature using the RSA public key, and checks for a match against the hashed message. If the computed signature matches the expected result, the message is deemed authentic. However, a comprehensive and secure implementation would require additional considerations, such as proper padding schemes and robust key management. Additionally, the exponentiation function, referenced within, needs to be implemented securely to ensure the overall security of the RSA-based digital signature scheme.

1.10 How RSA Digital Signature Generally Works:

```
# User 1 wants to send a message to User 2.
Message = "Hi, User 2! Excited for Christmas?"

# User 1 encrypts his message using AES
encryptedMessage = aes_encrypt(Message, byte_value,
   modes.ECB())

# User 1 signs his message using his RSA private key
Signature = RSA_sign(Message, user1.RSAPrivateKey)
```

```
# User 1 sends his encrypted message and signature to
   User 2.

# User 2 receives the encrypted message and decrypts
   it.
# Note: User 2 can decrypt using the shared
   Diffie-Hellman key, byte_value.
decryptedMessage = aes_decrypt(encryptedMessage,
   byte_value, modes.ECB())
print("Received the message from User 1: " +
   str(decryptedMessage))

# User 2 verifies the authenticity of the message by
   checking the RSA signature.
verified = RSA_verify(encryptedMessage, Signature,
   user1.RSAPublicKey)

if verified:
    print("This message was indeed sent by User 1!")
else:
    print("This message was not verified!")
```

```
Printed Output:
Received the message from User 1: Hi, User 2! Excited for Christmas?
This message was indeed sent by User 1!
```

The signature procedure depicted in the provided code follows the principles of RSA digital signatures. First, the hash_message function employs the SHA-256 algorithm to produce a secure hash of the original message, ensuring a fixed-size representation. Subsequently, the RSA_sign function takes the hashed message and employs the RSA private key of the sender (User 1) for exponentiation, creating a digital signature unique to both the message content and the private key. This signature serves as proof of the message's authenticity and integrity. On the recipient's side (User 2), upon receiving the encrypted message, the RSA_verify function hashes the decrypted message and then exponentiates the received signature using the sender's RSA public key. The hashed message and the decrypted signature are then compared; a match signifies the authenticity of the original message, providing confidence that it has not been tampered with during transmission. This RSA digital signature scheme ensures data integrity, sender authentication, and non-repudiation in secure communication.

1.11 Hacking the Digital Signature

Consider the scenario mentioned earlier, but this time, an additional word is introduced into the message during the encryption process.

```
# User 1 wants to send a message to User 2.
Message = "Hi, User 2! Excited for Christmas?"

# User 1 encrypts his message using AES
encryptedMessage = aes_encrypt(Message + "GRINCHED",
    byte_value, modes.ECB())

# User 1 signs his message using his RSA private key
Signature = RSA_sign(Message, user1.RSAPrivateKey)

# User 1 sends his encrypted message and signature to
    User 2.

# User 2 receives the encrypted message and decrypts
    it.
# Note: User 2 can decrypt using the shared
    Diffie-Hellman key, byte_value.
decryptedMessage = aes_decrypt(encryptedMessage,
    byte_value, modes.ECB())
print("Received the message from User 1: " +
    str(decryptedMessage))

# User 2 verifies the authenticity of the message by
    checking the RSA signature.
verified = RSA_verify(encryptedMessage, Signature,
    user1.RSAPublicKey)

if verified:
     print("This message was indeed sent by User 1!")
else:
     print("This message was not verified!")
```

```
Printed Output:
Received the message from User 1: Hi, User 2! Excited for Christmas?GRINCHED
This message was not verified!
```

As the RSA signature was linked to the unaltered message, it raised a red flag for the message received by User 2, indicating the possibility that the sender might not be User 1 after all.

1.12 Importance of Hashing

Hashing is a crucial step in digital signatures as it enhances security by condensing the message content into a fixed-size hash value. This hash value serves as a unique fingerprint for the original message. During the signing process, only the hash is signed, not the entire message, reducing computational load and increasing efficiency. Any modification to the message, no matter how minor, will result in a completely different hash, making it easy to detect tampering. This contributes significantly to the security of digital signatures, ensuring message integrity and authenticity.

1.13 Optional Functionality: Timestamps

```
import time

def RSA_sign_with_timestamp(message, private_key):
    # Include the current timestamp in the signature
    timestamp = time.time()

    # Convert the string message to bytes
    message_bytes = message.encode('utf-8')

    # Concatenate the message and timestamp as bytes
    combined_data = message_bytes +
       str(timestamp).encode('utf-8')

    # Hash the combined data
    hashed_message = hash_message(combined_data)

    # Create the signature
    signature = exponentiation(hashed_message,
       private_key[0], private_key[1])

    return signature, timestamp

def RSA_verify_with_timestamp(message, signature,
   timestamp, public_key, validity_period=3600):
    # Verify the timestamp
    current_time = time.time()
    if current_time - timestamp > validity_period:
        return False  # Signature is considered
           invalid if the timestamp is too old

    # Convert the string message to bytes
    message_bytes = message.encode('utf-8')

    # Concatenate the message and timestamp as bytes
    combined_data = message_bytes +
       str(timestamp).encode('utf-8')
```

```
    # Hash the combined data
    hashed_message = hash_message(combined_data)

    # Verify the signature
    decrypted_signature = exponentiation(signature,
        public_key[0], public_key[1])

    return hashed_message == decrypted_signature

# User 1 wants to send a message to User 2.
Message = "Hi, User 2! Excited for Christmas?"
encryptedMessage = aes_encrypt(Message , byte_value,
    modes.ECB())
# User 1 signs his message using his RSA private key
    WITH A TIMESTAMP!
Signature, Timestamp =
    RSA_sign_with_timestamp(Message,
    user1.RSAPrivateKey)

# User 1 sends his encrypted message and signature to
    User 2.

# User 2 receives the encrypted message and decrypts
    it.
# Note: User 2 can decrypt using the shared
    Diffie-Hellman key which is associated to
    byte_value.
decryptedMessage = aes_decrypt(encryptedMessage,
    byte_value, modes.ECB())
print("Received the message from User 1: " +
    str(decryptedMessage.decode()))

# User 2 verifies the authenticity of the message by
    checking the RSA signature AND TIME STAMP.
verified =
    RSA_verify_with_timestamp(decryptedMessage.decode('utf-8'),
    Signature, Timestamp, user1.RSAPublicKey)

if verified:
    print("This message was indeed sent by User 1 and
        is within the validity period!")
else:
    print("This message was not verified or is
        outside the validity period.")
```

In this code snippet, a feature is added to enhance the security of the communication between User 1 and User 2. The RSA_sign_with_timestamp function is introduced to sign a message with an RSA private key, and a timestamp is included to provide a time reference. The signed message, along with its encrypted version using the AES algorithm, is then sent from User 1 to User 2. Upon reception, User 2 decrypts the message using the shared Diffie-Hellman key. The RSA_verify_with_timestamp function is employed to verify the authenticity of the decrypted message by checking the RSA signature and ensuring the timestamp falls within a specified validity period. This timestamped signature mechanism adds an additional layer of security, helping to mitigate potential threats such as replay attacks by associating a time constraint with the validity of the signature.

This concludes the broad overview of the underlying logic and encryption processes. Subsequent sections of the report will delve deeper into each component, providing a detailed exploration of the topics briefly introduced here. The focus will particularly intensify within the Chat UI, offering a comprehensive examination of its intricacies and functionalities.

2 How To Run the Chat Server Interface

This project employs two distinct server/client configurations based on the chosen Key Exchange type. Each server and client is equipped with a HOST variable, represented by an IP address.

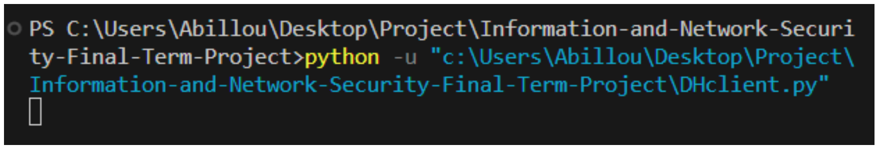
```
HOST = '192.168.1.31'
```

For optimal functionality, it is advisable to set both the server and client HOST variables to your IPv4 Address. You can obtain your IPv4 Address by executing the 'ipconfig' command in your Command Prompt and pressing enter.

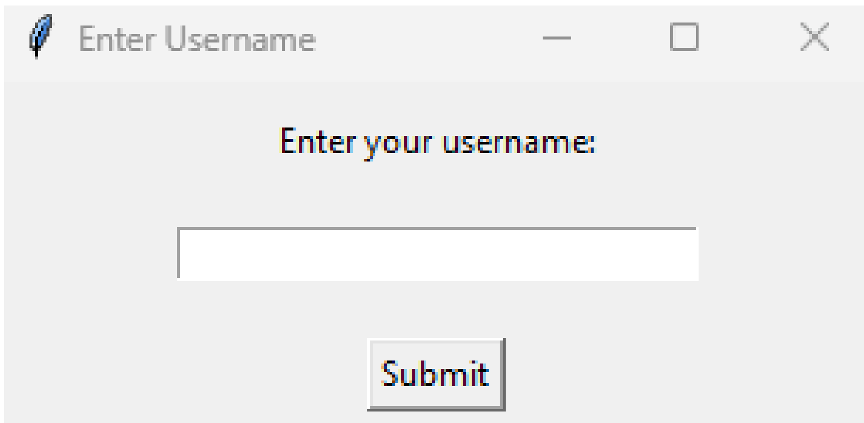
To initiate the server, launch a terminal and execute the server file with the following command:

```
PS C:\Users\Abillou\Desktop\Project\Information-and-Network-Securi
ty-Final-Term-Project> python -u "c:\Users\Abillou\Desktop\Project
\Information-and-Network-Security-Final-Term-Project\DHserver.py"
Server is listening on 192.168.1.31:55555
```

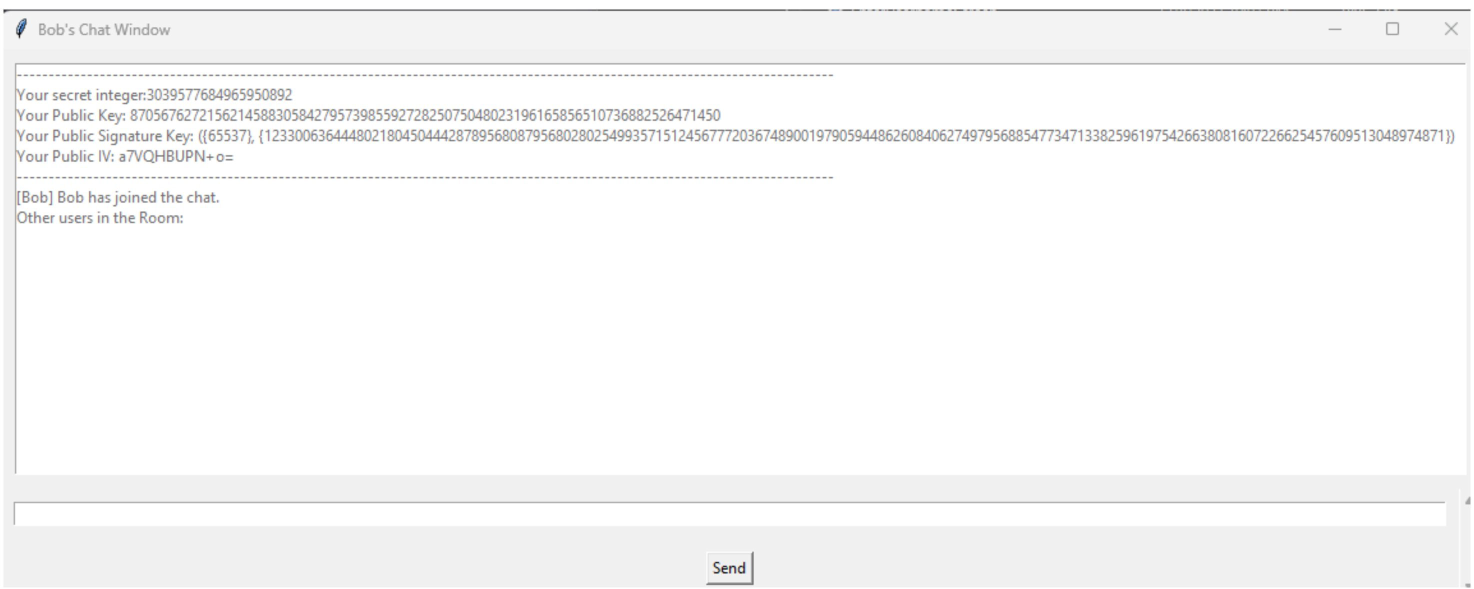
As observed, the server is operational if its output is displayed. Once the server is active, we can proceed to run its corresponding client. To achieve this, open a separate terminal and execute the client as follows:



If the client was executed properly, the following screen should pop up:



After entering a username in this box (using "Bob" as an example), and subsequently clicking the submit button, the ensuing window should be displayed:



The content of the chat log will vary depending on factors such as the timing of a user's joining, the total number of users, and the specific type of key-exchange server in use. Further details on these variations will be discussed in greater depth later in the report.

To simulate additional users joining the chat server, you have the option to either open a new terminal and run the client file again, or alternatively, run the client file on a different computer. In both scenarios, a new user will be added to the chat.

For the purposes of this project, I will not be going too in-depth about the GUI window aspect of the interface nor on how exactly the server parses and send messages, as the primary focus lies on key exchange, encryption/decryption algorithms, and digital signatures. In the same regard, since this is a relatively simple Chat client-server interface, we will make the following assumptions:

- Usernames that are entered are appropriate (i.e., not empty, devoid of special characters, unique compared to existing users, and not named after special commands of the server, which will be discussed later).
- Users join the server one after the other.
- All users within the server agree upon a prime number and primitive root, defined as follows:
  - Prime number: 102188617217178804476387977160129334431745945009730065519337094992129677228373
  - Primitive root: 2
- Users use special commands appropriately.
- We will assume that there are three users for the purpose of examples: Bob, Alice, and Eve.

Considering the information provided, let's commence with the report.

3 Key-Exchange

3.1 Implement the Diffie-Hellman Algorithm

This section will delve into the DH Server and Client files. Let's presume the server is operational. Upon a new user's initial entry to the server (following the submission of their username), they will generate a secret integer using the following code snippet within their chat GUI class:

```
# Generate a secret integer for the client
self.secret_integer = random.getrandbits(64)
```

The secret integer generated is a random number of 64 bits. This was done to simplify its use when running it for other encryption methods which we will talk about later.

Once generated, the user-client transmits their username, public key, and associated attributes to the DH server using the following code:

```
client.send(username.encode('utf-8'))
if ENCRYPTIONTYPE == "AES_CBC" or ENCRYPTIONTYPE == "DES_CBC":
    client.send(f"Public key: {pow(primitiveRoot,
        self.secret_integer, primeNumber)} Public Signature
        Key:
        ({self.publicSignKey[0]},{self.publicSignKey[1]})
        Public IV:
        {b64encode(self.IV).decode()}".encode('utf-8'))
    # Display attribute information to the user pertaining to
        encryption type
else:
    client.send(f"Public key: {pow(primitiveRoot,
        self.secret_integer, primeNumber)} Public Signature
        Key:
        ({self.publicSignKey[0]},{self.publicSignKey[1]})".encode('utf-8'))
    # Display attribute information to the user pertaining to
        encryption type
```

The user-client indeed transmits additional attributes, each varying depending on the encryption type. However, a detailed discussion of these attributes will be deferred until their relevance becomes apparent. It is crucial to note that the user-client conveys a public key using the following format:

```
Public key: {pow(a, x, p)}
```

Here,
a : primitiveRoot
x : self.secret_integer or their own secret integer
p : primeNumber

Additionally, the relationship

$$pow(a, x, p) \equiv a^x \ (mod\ p)$$

defines the public key generation within the DH algorithm.

Following this, once the user sends the message to the server, the server meticulously parses the message, extracting its components, notably the public key. The server diligently stores the public keys corresponding to each user, after which each fragment is broadcast to all other users.

Upon receiving these components, especially the public key segment, other users store this public key associated with the respective user in a dictionary, as such:

```
def handle_public_key(self, message):
    # Parse the public key message
    parts = message.split(": ")
    if len(parts) == 2:
        username = parts[0][14:]
        if(not (username == self.username)):
            public_key = int(parts[1])

            # Store the public key in the dictionary
            self.public_keys[username] = public_key

            # Calculate the shared secret key
            shared_secret_key = pow(public_key,
                self.secret_integer, primeNumber)

            # Display Public and Shared Secret Key of
                associated user to current user
```

It's noteworthy that upon receiving the public key of another user in this function, a recipient calculates their shared secret key using the equation:

```
shared_secret_key = pow(A, y, p)
```

Here,
A : Public Key of the other user
y : self.secret_integer or their own secret integer
p : primeNumber

The relationship

$$pow(A, y, p) \equiv A^y \ (mod\ p)$$

expresses the shared secret key exchange equation within the DH algorithm.

Prior to this process, it's crucial to note that when a user transmits their public key to the server, the server reciprocates by providing the public keys of all other users who have shared theirs. The user then performs the same calculation above using their own secret exchange for every public key sent to them.

The anticipated outcome is that every user in the chat room will now possess a shared secret key with another user.

3.2 Are the Exchanged Keys the Same?

Upon a user's entry into the server, all users receive notifications containing the new user's public key, other pertinent components, and the associated secret key. Conversely, the newly joined user receives similar notifications for each existing user in the server.

For additional testing, a command is available in the following format:

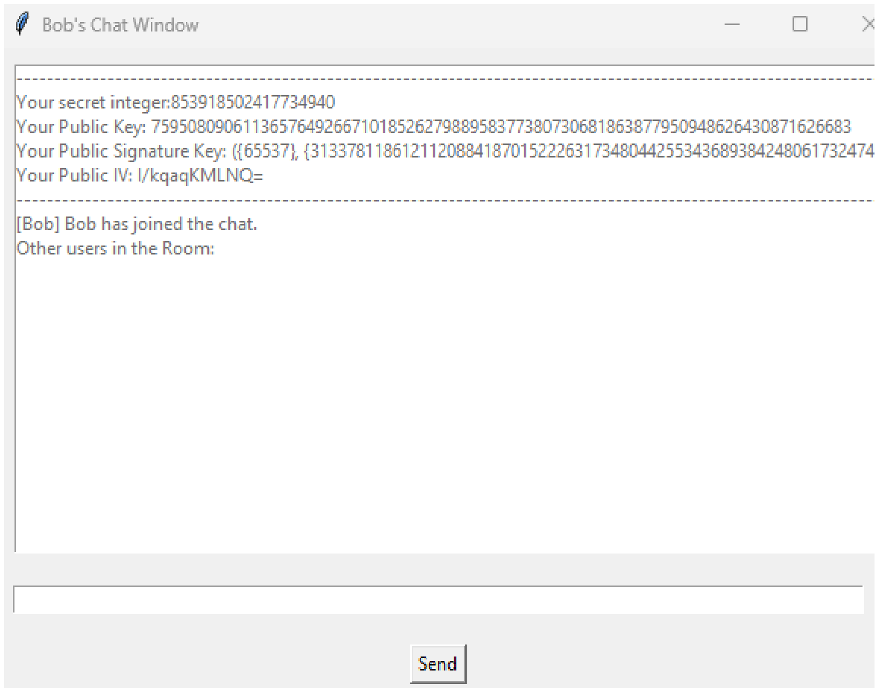
`./checkSharedKey {other username}`

When a user enters this message and sends it, the client will provide them with the received shared key associated with the specified other user. As such:

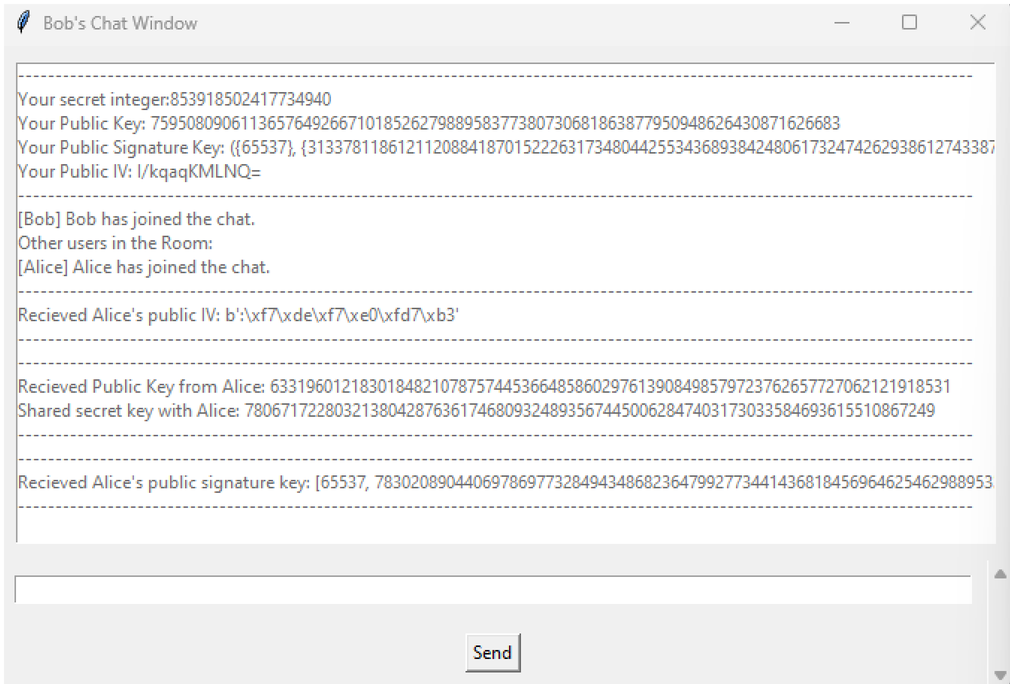
```
def check_shared_key(self, message):
    """Check and output the shared key for a specific
       user."""
    parts = message.split(' ')
    if len(parts) == 2:
        target_username = parts[1]

        # Check if we have the public key for the
           target user
        if target_username == self.username:
            self.display_message(f"Your Public Key
               Is: {pow(primitiveRoot,
               self.secret_integer, primeNumber)}")
        elif target_username in self.public_keys:
            # Calculate the shared secret key
            shared_secret_key =
               pow(self.public_keys[target_username],
               self.secret_integer, primeNumber)

            # Output the
            # shared key for the specific user
            self.display_message(f"Shared secret key
               with {target_username}:
               {shared_secret_key}")
        else:
            self.display_message(f"Public key for
               {target_username} not available.")
```

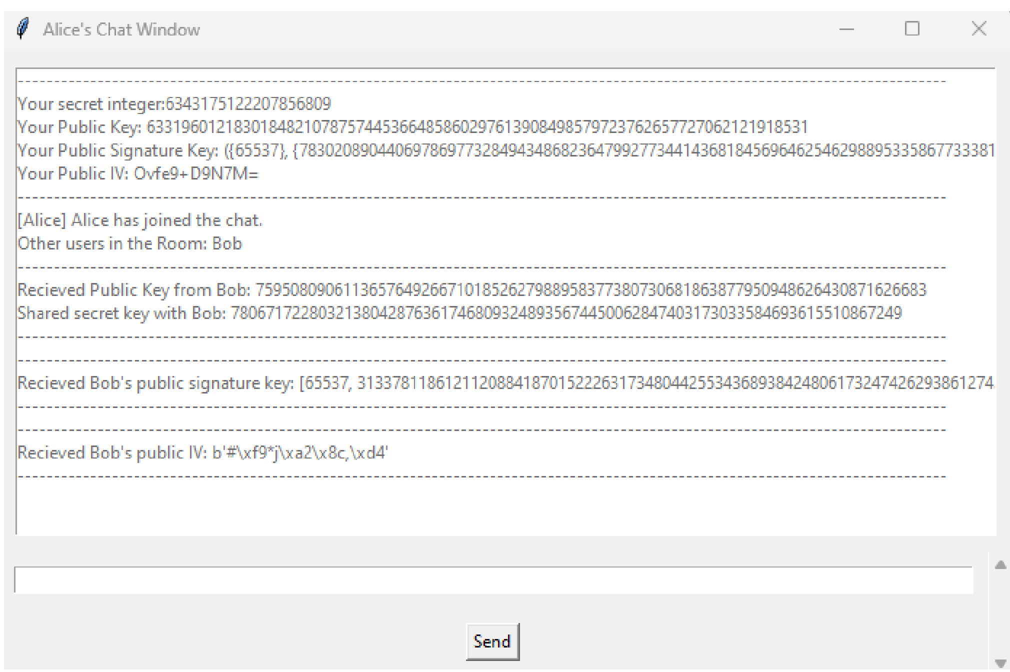
The command runs the same secret key exchange equation mentioned in the earlier subsection and outputs it to the client.

Now, let's assume the server is operational and currently empty. User Bob joins into the server. He is met with the following window:

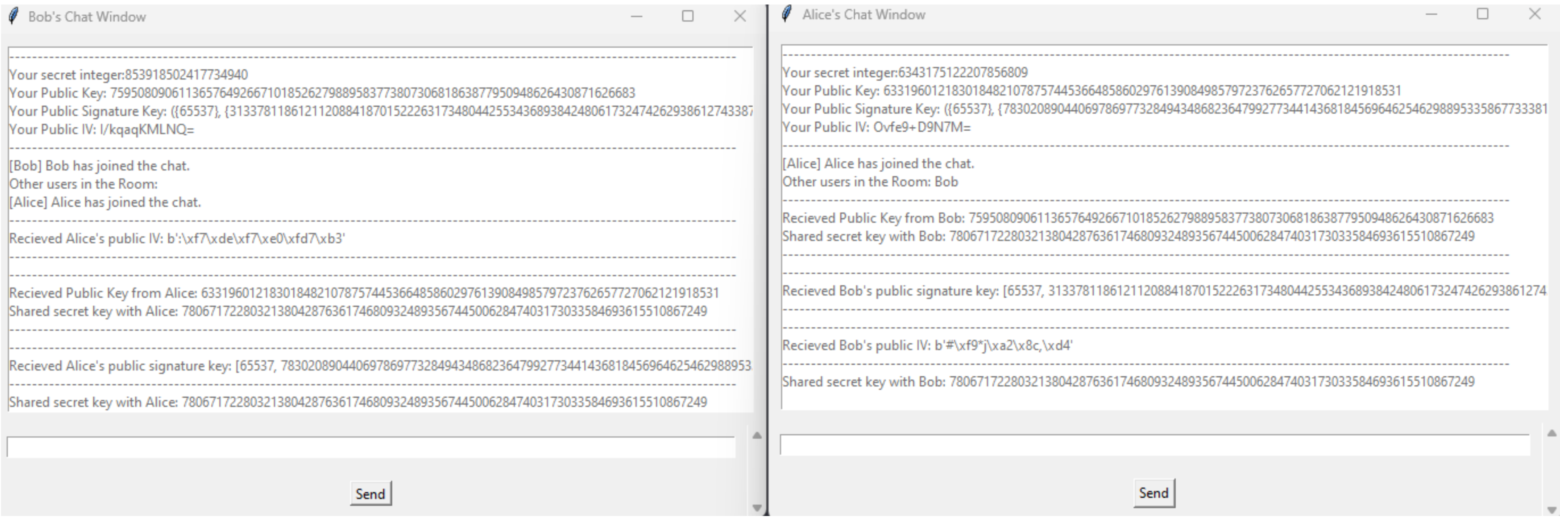


We can see the associated secret integer and public key associated with Bob at the top. Now, let's say Alice joins into the server. Bob and Alice would be met with the following windows:





Notably, the shared secret key transmitted to each user are the same. However, consider a scenario where Alice initiates the command "./checkSharedKey Bob" and Bob issues "./checkSharedKey Alice." The corresponding outputs are as follows:



Clearly, both users can retrieve an identical shared key from each other, housed in their respective storage. This indicates the successful functioning of the DH Key Exchange.

3.3 Implement LFSR Key Generation and RSA Transmission

This section will delve into the RSA Server and Client files. Let's presume the server is operational. Upon a new user's initial entry to the server (following the submission of their username), they will generate a LFSR from the following class embedded :

```
class LFSR:
    def __init__(self, seed, taps):
        self.state = seed
        self.taps = taps

    def shift(self):
        feedback = sum(self.state[tap] for tap in
            self.taps) % 2
        self.state = [feedback] + self.state[:-1]
        return feedback
```

```
    def generate_key(self, length):
        key = []
        for _ in range(length):
            key.append(self.shift())

        # Ensure that the generated key is not zero
        generated_key_int = int(''.join(map(str,
           key)), 2)
        while generated_key_int == 0:
            key = []
            for _ in range(length):
                key.append(self.shift())
            generated_key_int = int(''.join(map(str,
               key)), 2)

        # Return both integer and binary string forms
        generated_key_bin = ''.join(map(str, key))
        return generated_key_int, generated_key_bin
```

The LFSR class, featuring a Linear Feedback Shift Register, plays a pivotal role in generating sequences for cryptographic purposes. Upon instantiation with a user-defined seed and taps specifying feedback positions, the class leverages the shift method to emulate one shift operation. This operation, reminiscent of a recurrence relation, calculates feedback by summing bits at designated taps and subsequently updates the register's state. The dynamic nature of the taps in this process influences the register's behavior, allowing users to tailor output sequences. Furthermore, the generate_key method employs the shift operation iteratively to create a pseudorandom key of a specified length. It ensures the generated key is non-zero for cryptographic security. This key calculation involves generating bits through repeated shifts, converting them into an integer, and returning both integer and binary string representations.

Now, a user client would generate a seed of random length as well as random positions for taps, then using those values, generate an LFSR key and store it within their CHatGUI interface:

```
seed = [random.randint(0, 1) for _ in
   range(random_seed_length)]
shiftFeedbackPositions = random.sample(range(len(seed)),
   k=random.randint(1, len(seed)))
shiftFeedbackPositions.sort()
lfsr = LFSR(seed, shiftFeedbackPositions)
key_length = 256
if(ENCRYPTIONTYPE == 'DES_ECB' or ENCRYPTIONTYPE ==
    'DES_CBC'):
    key_length = 64
self.key_length = key_length
generated_key, generated_keyBin =
```

```
    lfsr.generate_key(key_length)
self.LSFRKey, self.LSFRKeyBin = generated_key,
    generated_keyBin
```

Accordingly, the key length varies based on the type of encryption, a detail we will explore later. The immediate goal is to endeavor transmitting this message to another user using RSA. To accomplish this, we must implement the RSA functions. Fortunately, we possess these functions from a prior assignment, enabling us to employ them within the client class in this context:

```
'''Helper function that generates a large prime number with
    the specified number of bit.'''
def generate_large_prime(bits):
    while True:
        num = random.randrange(0, 2**bits - 1)
        if isprime(num):
            return num
'''Calculates the modular exponetiation of a given message,
    power, and basis using the 'powmod' function from the
    gmpy2 library'''
def exponentiation(message, power, basis):
    return powmod(message, power, basis)

'''Helper function that incorporates the extended euclidean
    algorithm to help determine the inverse value within the
    inverse_finder function.'''
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        g, x, y = extended_gcd(b % a, a)
        return g, y - (b // a) * x, x

'''Finds the modular inverse of a given number a modulo n
    using the extended euclidean algorithm.'''
def inverse_finder(a, n):
    g, x, _ = extended_gcd(a, n)
    if g != 1:
        raise ValueError(f"The modular inverse does not exist
            for {a} modulo {n}")
    else:
        return x % n
'''A function that generates RSA public and private keys.'''
def RSA_key_generate():
    e = 65537
    while(True):
        p = generate_large_prime(256)
        q = p

        while(p == q):
```

```
            q = generate_large_prime(256)
        n = p * q
        euler = (p-1) * (q-1)

        if(math.gcd(euler, e) == 1):
            break
    d = inverse_finder(e, euler)
    publicKey = [e, n]
    privateKey = [d, n]
    return publicKey, privateKey

'''A function encrypts a numeric message or a string using
   RSA encryption with a given key.'''
def RSA_encrypt(message, key):
    if not isinstance(message, str):
        return exponentiation(message, key[0], key[1])
    elif isinstance(message, str):
        ciphertext = []
        for element in range(0, len(message)):
            ciphertext.append(int(exponentiation(ord(message[element]),
                key[0], key[1])))
        return ciphertext

'''A function decrypts a numeric message or a list of numeric
   values using RSA decryption with a given key.'''
def RSA_decrypt(message, key):
    if not isinstance(message, str) and not
        isinstance(message, list):
        return RSA_encrypt(message, key)
    elif isinstance(message, list):
        decrpyted = ''
        for element in range(0, len(message)):
            decrpyted+= chr(exponentiation(message[element],
                key[0], key[1]))
        return decrpyted
```

Moving forward, users gain the capability to generate RSA Keys and perform encryption and decryption as required. Upon a user's initial entry to the server, not only do they create a personal LFSR Key, but they also generate RSA Public and Private Keys. Analogous to the DH Client, the RSA Client of a new user transmits its Public Key to the Server, which stores it, broadcasts its public keys for other users to store, and shares the public keys of other users with the new user as such:

```
self.public_key, self.private_key = RSA_key_generate()
...
...
...
client.send(username.encode('utf-8'))
if(ENCRYPTIONTYPE == "AES_CBC" or ENCRYPTIONTYPE ==
```

```
    "DES_CBC"):
     client.send(f"Public key: {self.public_key[1]}
         Public Signature Key:
         ({self.publicSignKey[0]},{self.publicSignKey[1]})
         Public IV:
         {b64encode(self.IV).decode()}".encode('utf-8'))
     # Display attribute information to the user
         pertaining to encryption type
else:
     client.send(f"Public key: {self.public_key[1]}
         Public Signature Key:
         ({self.publicSignKey[0]},{self.publicSignKey[1]})".encode('utf-8'))
     # Display attribute information to the user
         pertaining to encryption type
...
...
...
def handle_public_key(self, message):
     # Parse the public key message
     parts = message.split(": ")
     if len(parts) == 2:
          username = parts[0][14:]
          if(not (username == self.username)):
               public_key = int(parts[1])
```

However, a notable distinction exists: users do not transmit their LFSR Key automatically. To accomplish this, users need to execute the following command:

`./sendLFSRKey {other username}`

which is associated with the following code snippet:

```
def send_LSFR_key(self, message):
    _, target_username = message.split(' ', 2)

    # Check if the LSFR key has already been sent to the
        target user
    if target_username in self.sentLSFRKeys:
        self.display_message("----------")
        self.display_message(f"You have already sent your
            LFSR Key to {target_username}.")
        self.display_message("----------")
        return

    # Send the LSFR key to the target user
    encrypted_lsfr_key = RSA_encrypt(self.LSFRKey,
        self.public_key)
    client.send(f"./sendLFSRkey {target_username}
        {encrypted_lsfr_key} {self.private_key[0]}
        {self.private_key[1]}".encode('utf-8'))
```

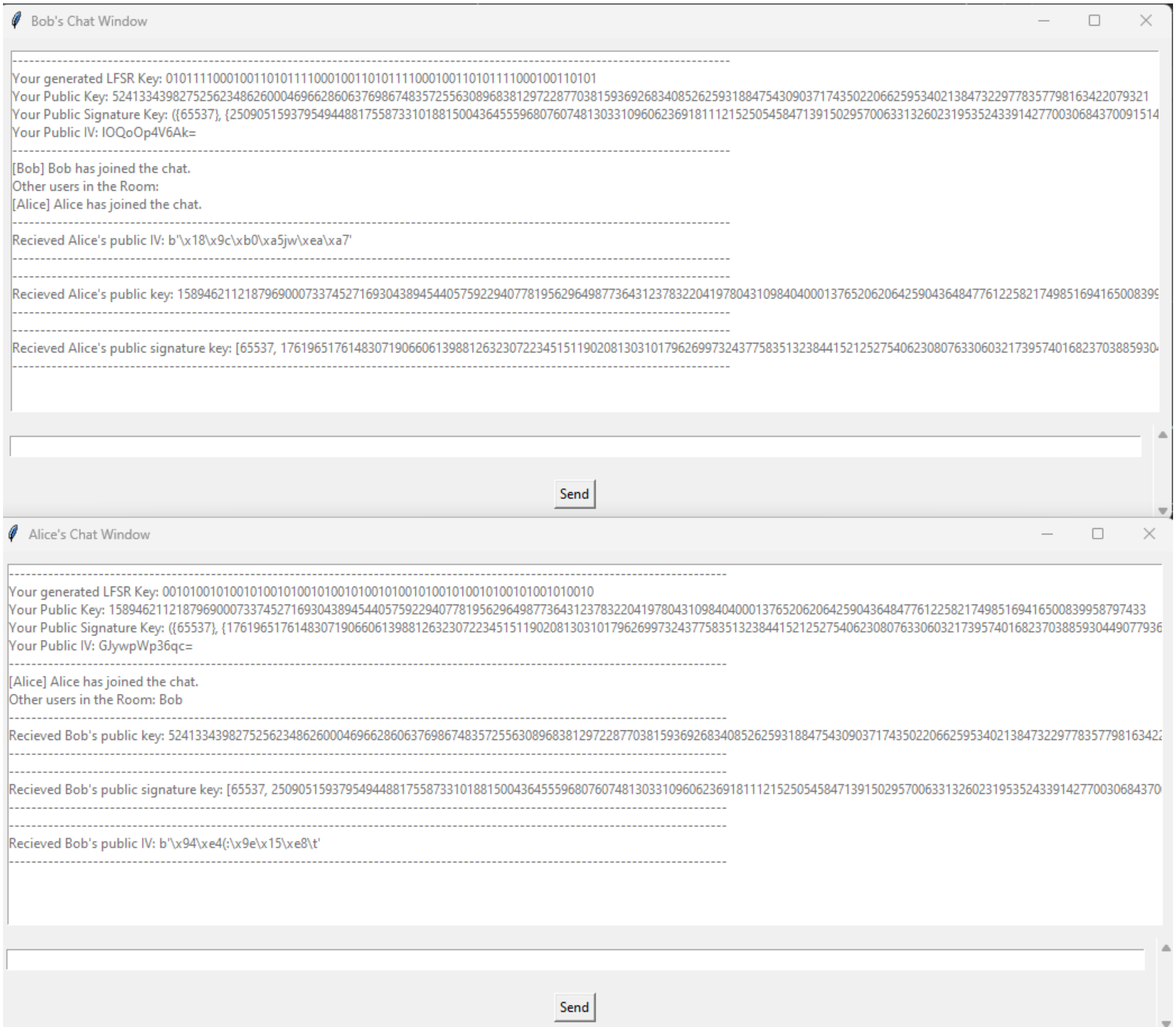
In essence, the user client encrypts their LFSR Key using the RSA encryption function, which executes the equation ($LFSR^e$ mod n). It then formats a message containing the target user, encrypted message, and components of the private key. Subsequently, the user transmits this message to the server.

The server would receive this message and parse out each component. The server then broadcasts the encrypted message to all users. Specifically, for the user targeted by the executed command, the server exclusively transmits the original user's message in its entirety, with the original user's username appended to it, to the target user. The target user client will detect that the server has sent them a special method and run the following method for it:

```
def handle_sendLFSRkey(self, message):
    parts = message.split(' ', 5)
    encryptedLSFRKey = int(parts[2])
    decryptionKey = [int(parts[3]), int(parts[4])]
    fromUser = parts[5]
    sentLSFR = RSA_decrypt(encryptedLSFRKey, decryptionKey)
    self.recievedLSFRKeys[fromUser] = int(sentLSFR)
    key_length = f'0{self.key_length}b'
    self.display_message(f'Received from {fromUser}:
        {format(self.recievedLSFRKeys[fromUser],
        key_length)}')
```

The client extracts specific message components to retrieve the encrypted LFSR Key, decryption key, and sender information. Subsequently, it employs the RSA Decrypt function with the encrypted LFSR Key and decryption key as parameters, executing the equation (EncryptedLFSR^d mod n). The resulting decrypted LFSR Key is stored and associated with the respective user for future use.

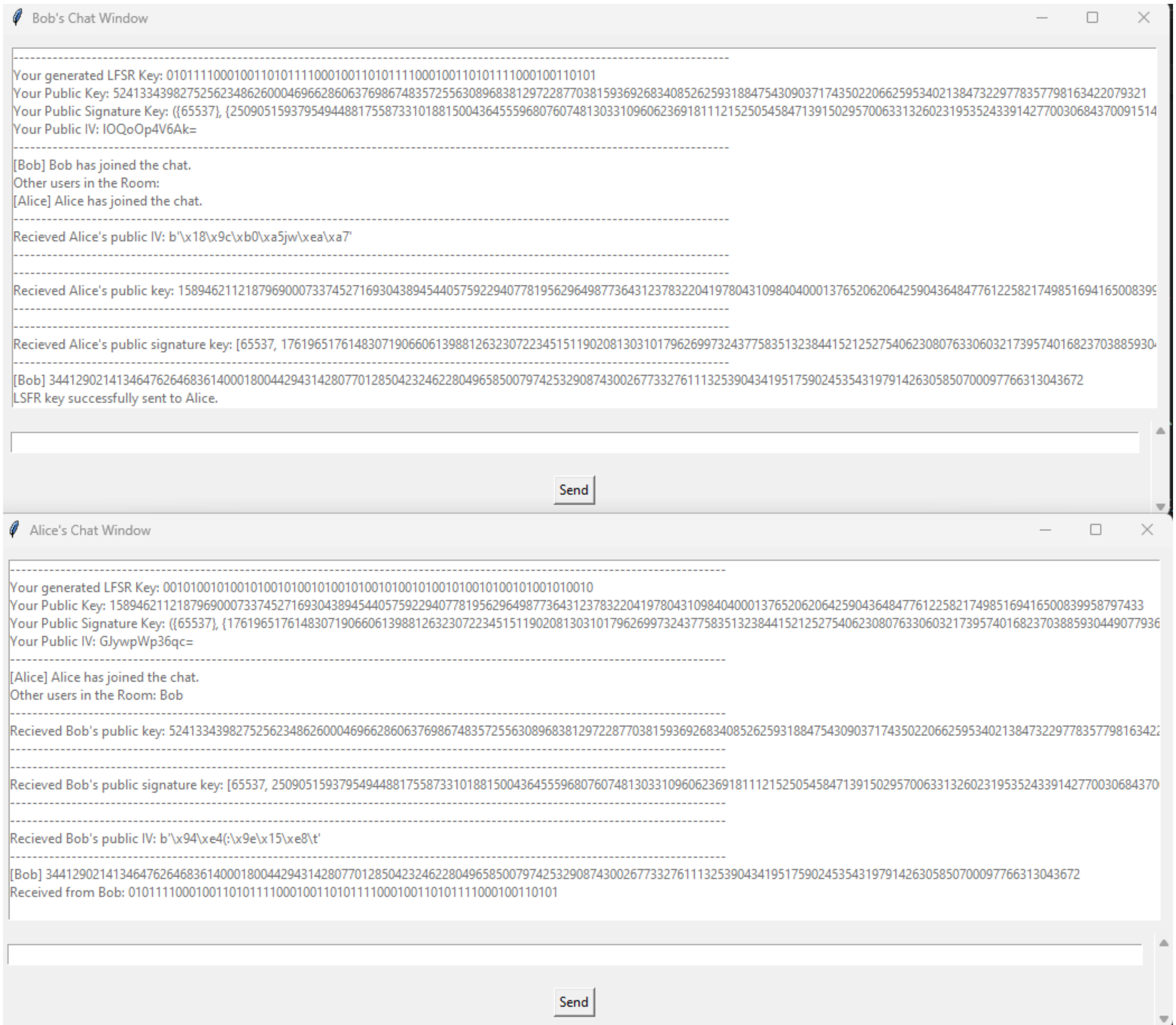
To showcase this happening, let's assume that Bob and Alice have joined the server.



We observe that Alice and Bob have exchanged their public keys through a dialogue, mirroring the DH Server-Client system. Now, suppose Bob intends to transmit his LFSR Key to Alice. In order to do so, he must compose the message:

```
./sendLFSRKey Alice
```

Subsequently, he dispatches this message. Upon completion of this action, the following output ensues:



Both Alice and Bob observe Bob's encrypted message. However, exclusively on Alice's screen, we discern that she received Bob's LFSR Key upon comparing the two displays. With this in mind, we can affirm that the RSA functionality is performing as anticipated.

4 Secure Messaging

Presently, messages sent by a user in either server are transmitted without encryption to all clients. To enable the secure transmission of confidential messages, algorithms for encryption and decryption need to be devised. This would allow users to encrypt their messages using their designated keys, send the encrypted messages to the server where everyone can view them, and enable the intended recipient to decrypt the message using the appropriate decryption method. To do this, we develop the command

`./sendToUser {otherUser} {Message To Send}`

Which is associated with the command for the Diffie-Hellman Client:

```
def send_private_message(self, message):
    """Send a private message to a specific user."""
    # Split the message into parts
    parts = message.split(' ', 2)
    target_username = parts[1]
    message_content = parts[2]

    # Compute shared secret key
    shared_secret_key =
        pow(self.public_keys[target_username],
        self.secret_integer, primeNumber)
    print(bin(shared_secret_key)[2:])

    # Encrypt the message based on encryption type
    ciphertext = self.encrypt_message(message_content,
        shared_secret_key)

    # Create the final message with encryption and digital
        signature
    final_message = f'./sendToUser {target_username}
        {ciphertext} {RSA_sign(message_content,
        self.privateSignKey)}'

    # Send the message to the server
    client.send(final_message.encode('utf-8'))

def encrypt_message(self, message, shared_secret_key,
    target_username):
    """Encrypt the message based on the encryption type."""
    if ENCRYPTIONTYPE == "STREAMCIPHER":
        return encrypt(message,
            str(bin(shared_secret_key)[2:]))
    elif ENCRYPTIONTYPE == "AES_ECB" or ENCRYPTIONTYPE ==
        "AES_CBC":
        key_bytes =
            shared_secret_key.to_bytes((shared_secret_key.bit_length()
            + 7) // 8, 'little')
```

```
        return encrypt(str(message), key_bytes, modes.ECB()
            if ENCRYPTIONTYPE == "AES_ECB" else
            modes.CBC(self.IV)).decode()
    elif ENCRYPTIONTYPE == "DES_ECB" or ENCRYPTIONTYPE ==
        "DES_CBC":
        key_bytes =
            shared_secret_key.to_bytes((shared_secret_key.bit_length()
            + 7) // 8, byteorder='big')
        key_bytes = hashlib.sha256(key_bytes).digest()[:8]
        return encrypt(str(message), key_bytes, DES.MODE_ECB
            if ENCRYPTIONTYPE == "DES_ECB" else DES.MODE_CBC,
            self.IV).decode()
```

Formatting the key slightly differs within the RSA clients, a topic we'll delve into shortly. Regardless, both clients assess the encryption and decryption methods based on the designated ENCRYPTIONTYPE variable. They extract the message components, perform encryption, and subsequently dispatch a special command to the server. This command includes the now-encrypted user message (along with their signature, which we'll discuss later).

`./sendToUser {otherUser} {Encrypted Message} {RSA Digi Signature}`

Upon detecting the special command, the server will broadcast the encrypted message to all clients and subsequently relay the entire command message to the targeted user.

Upon receiving the message, the intended user client identifies the special command and proceeds to execute the decryption algorithm linked to the encrypted message, as follows:

```
def decrypt_message(self, private_message, shared_secret_key,
    target_username):
    """Decrypt the message based on the encryption type."""
    if ENCRYPTIONTYPE == "STREAMCIPHER":
        return decrypt(private_message,
            str(bin(shared_secret_key)[2:]))
    elif ENCRYPTIONTYPE == "AES_ECB" or ENCRYPTIONTYPE ==
        "AES_CBC":
        key_bytes =
            shared_secret_key.to_bytes((shared_secret_key.bit_length()
            + 7) // 8, 'little')
        return decrypt(private_message, key_bytes,
            modes.ECB() if ENCRYPTIONTYPE == "AES_ECB" else
            modes.CBC(self.public_IVs[target_username]))
    elif ENCRYPTIONTYPE == "DES_ECB" or ENCRYPTIONTYPE ==
        "DES_CBC":
        key_bytes =
            shared_secret_key.to_bytes((shared_secret_key.bit_length()
            + 7) // 8, byteorder='big')
```

```
        key_bytes = hashlib.sha256(key_bytes).digest()[:8]
        return decrypt(private_message, key_bytes,
            DES.MODE_ECB if ENCRYPTIONTYPE == "DES_ECB" else
            DES.MODE_CBC, self.public_IVs[target_username])
```

Once again, the process varies based on ENCRYPTIONTYPE and the key exchange method. Nevertheless, it ultimately involves extracting the relevant parameters from the complete message, channeling them through the decryption algorithm, and presenting the user, who sent the encrypted message, with the decrypted content. Now, let's explore the encryption and decryption methods tied to ENCRYPTIONTYPE.

4.1 Stream Cipher

Upon configuring the ENCRYPTIONTYPE as "STREAMCIPHER," we incorporate the following methods from streamCipher.py into the client:

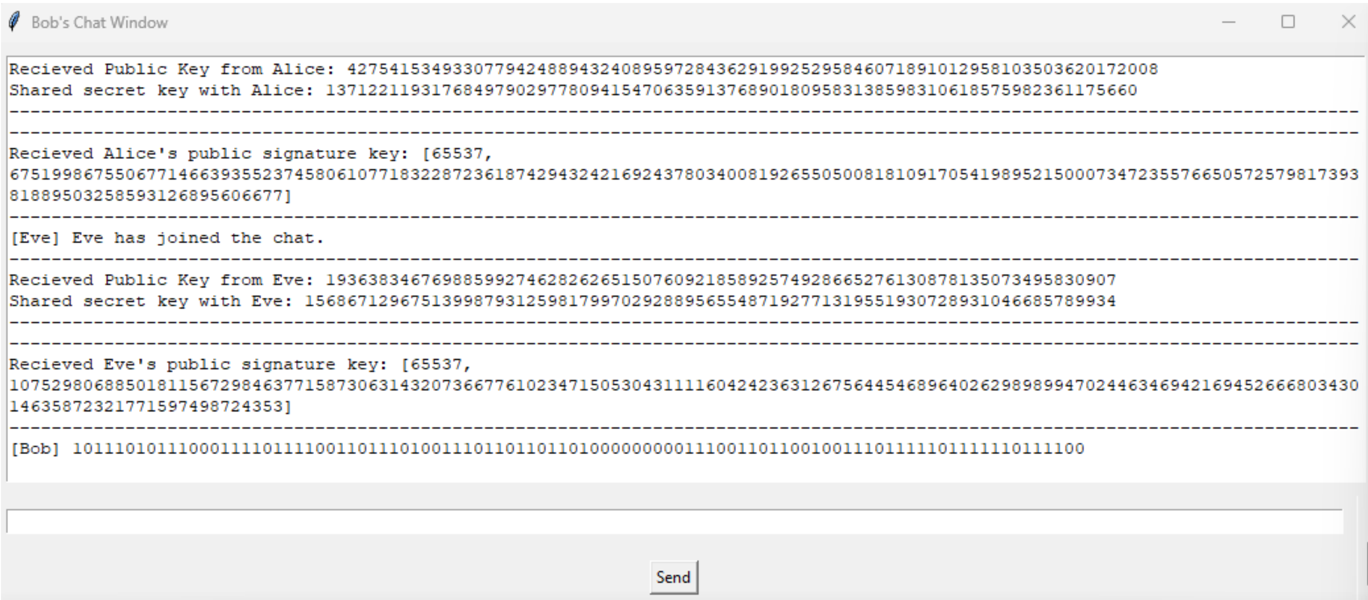
```
def text_to_bits(text):
    return ''.join(format(ord(char), '08b') for char in text)

def bits_to_text(bits):
    return ''.join(chr(int(bits[i:i+8], 2)) for i in range(0,
        len(bits), 8))

# Stream Cipher
def encrypt(text, key):
    bits = text_to_bits(text)
    encrypted_bits = [int(bit) ^ int(key[i % len(key)]) for
        i, bit in enumerate(bits)]
    return ''.join(map(str, encrypted_bits))

def decrypt(ciphertext, key):
    decrypted_bits = [int(bit) ^ int(key[i % len(key)]) for
        i, bit in enumerate(ciphertext)]
    return bits_to_text(''.join(map(str, decrypted_bits)))
```

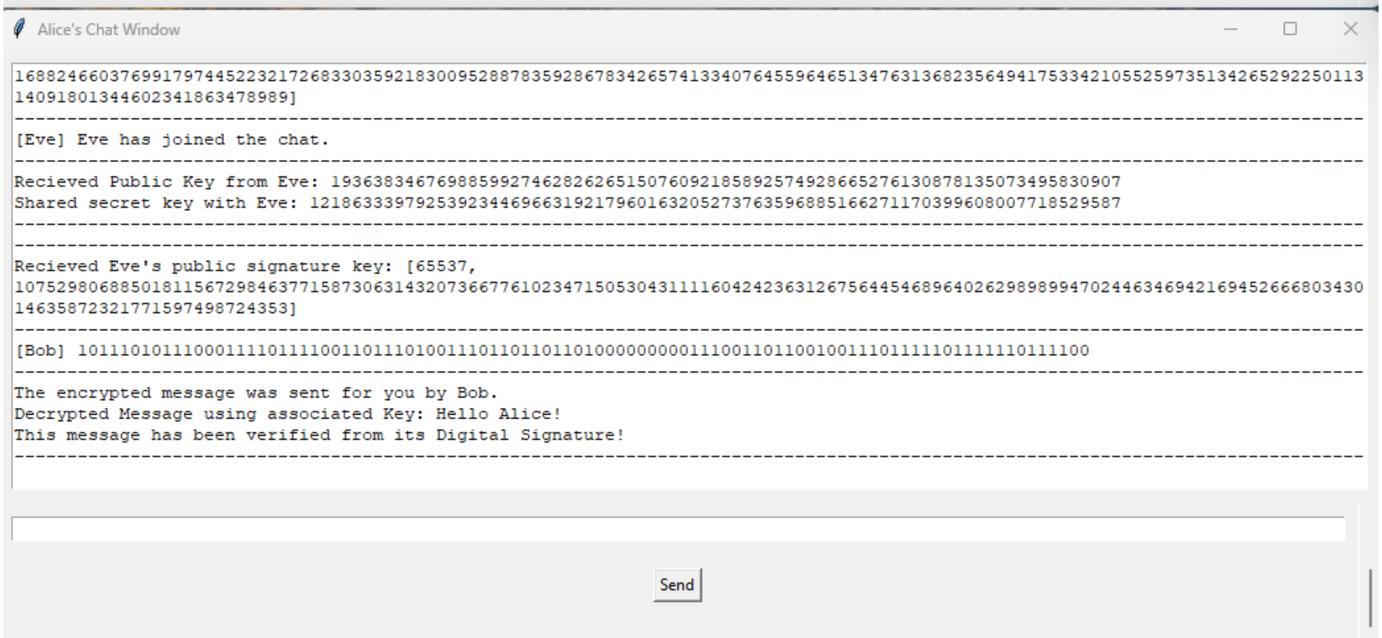
The provided Python methods are part of a stream cipher encryption and decryption process. In the text_to_bits function, it converts a given text into its binary representation, with each character represented by 8 bits. Conversely, the bits_to_text function performs the reverse operation, converting a binary sequence back into its corresponding text. Now, focusing on the stream cipher operations, the encrypt function begins by converting the plaintext message into its binary form using the text_to_bits function. It then XORs each bit of the binary representation of the text with the corresponding bit of the key in a cyclic manner, producing the encrypted binary sequence. The resulting binary sequence is then returned. Conversely, the decrypt function takes a ciphertext and the same key, performs XOR again, and then converts the binary sequence back into the original text using the bits_to_text function.

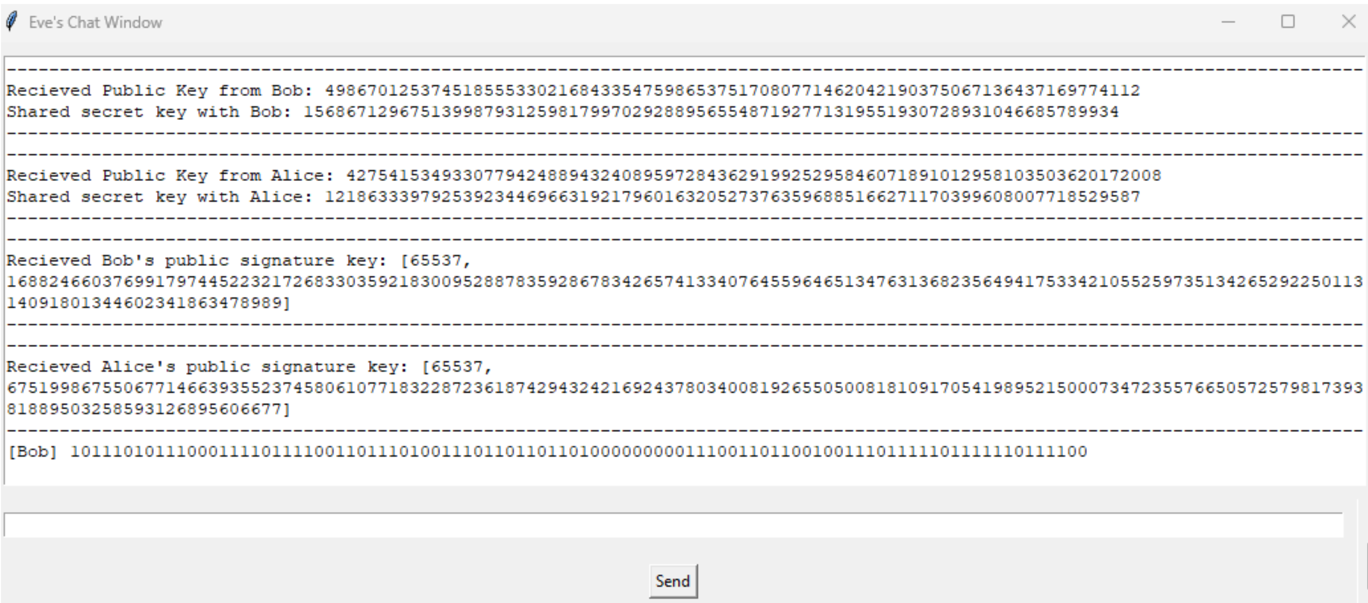
When a user client encrypts a message, they convert the intended plaintext into a binary representation and apply the encryption algorithm using their associated keys. The DH Client employs a shared secret key, while the RSA Client utilizes the generated LFSR Key. In both cases, the key is formatted as a binary value, the message is encrypted using that key, and transmitted to the server. The server subsequently broadcasts the encrypted message and dispatches a special command to the intended recipient. The recipient's user client then decrypts the received message, converts it back to text, and displays it.

Consider a scenario where, for each encryption method mentioned, Alice, Bob, and Eve are present on the server. Assume that Bob intends to convey a private message to Alice. In this context, Bob transmits messages of different lengths – two in the DH server and one in the RSA server – to observe the resulting output. Specifically, let's explore this process using the STREAMCIPHER algorithm.

Test 1: Within the DH Server, Bob Sends the Message "Hello Alice!" to Alice
(./sendToUser Alice Hello Alice!)


Bob's Chat Window
Recieved Public Key from Alice: 42754153493307794248894324089597284362919925295846071891012958103503620172008
Shared secret key with Alice: 13712211931768497902977809415470635913768901809583138598310618575982361175660
Recieved Alice's public signature key: [65537,
6751998675506771466393552374580610771832287236187429432421692437803400819265505008181091705419895215000734723557665057257981739381889503258593126895606677]
[Eve] Eve has joined the chat.
Recieved Public Key from Eve: 193638346769885992746282626515076092185892574928665276130878135073495830907
Shared secret key with Eve: 15686712967513998793125981799702928895655487192771319551930728931046685789934
Recieved Eve's public signature key: [65537,
1075298068850181156729846377158730631432073667761023471505304311116042423631267564454689640262989899470244634694216945266680343014635872321771597498724353]
[Bob] 1011101011100011110111100110111010011101101101101000000000111001101100100111011111011111110111100
Send

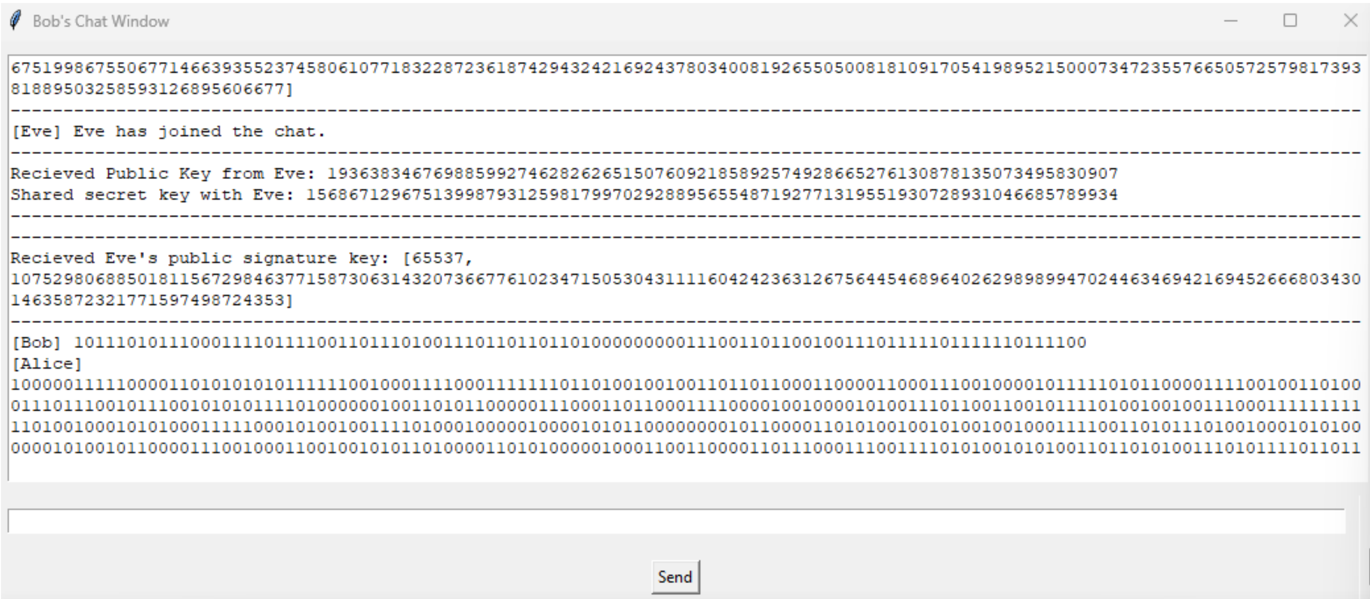
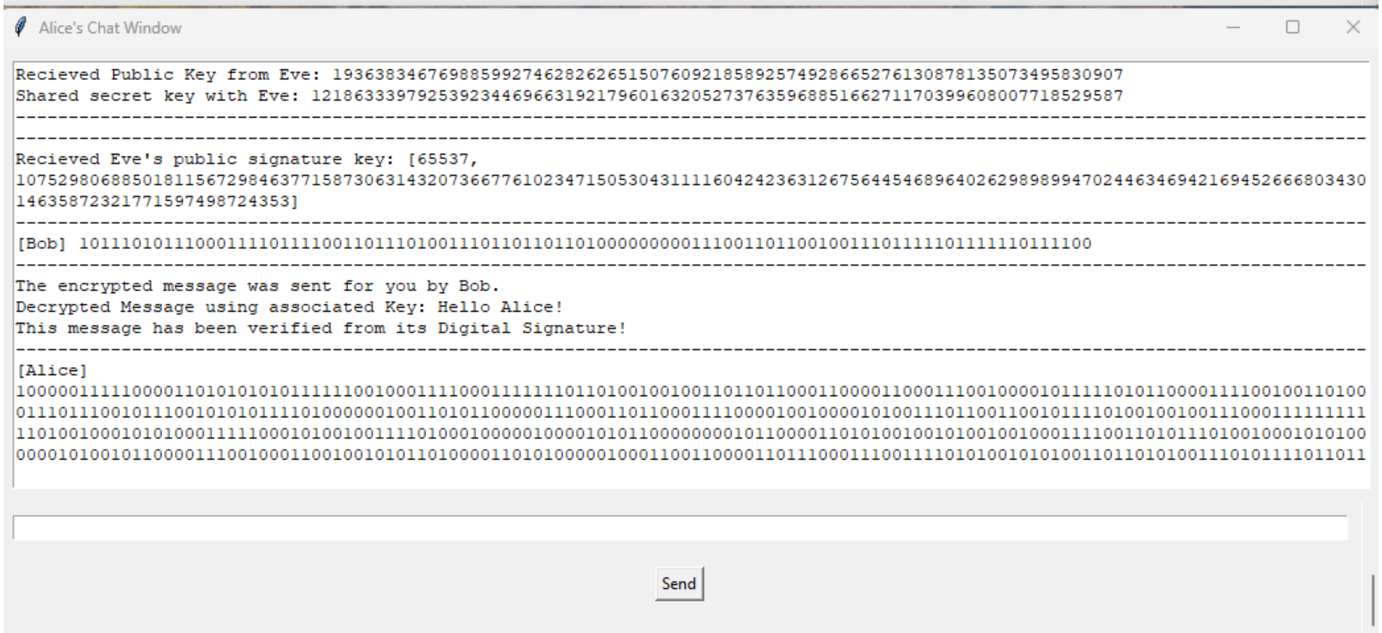

Alice's Chat Window
1688246603769917974452232172683303592183009528878359286783426574133407645596465134763136823564941753342105525973513426529225011314091801344602341863478989]
[Eve] Eve has joined the chat.
Recieved Public Key from Eve: 193638346769885992746282626515076092185892574928665276130878135073495830907
Shared secret key with Eve: 12186333979253923446966319217960163205273763596885166271170399608007718529587
Recieved Eve's public signature key: [65537,
1075298068850181156729846377158730631432073667761023471505304311116042423631267564454689640262989899470244634694216945266680343014635872321771597498724353]
[Bob] 1011101011100011110111100110111010011101101101101000000000111001101100100111011111011111110111100
The encrypted message was sent for you by Bob.
Decrypted Message using associated Key: Hello Alice!
This message has been verified from its Digital Signature!
Send

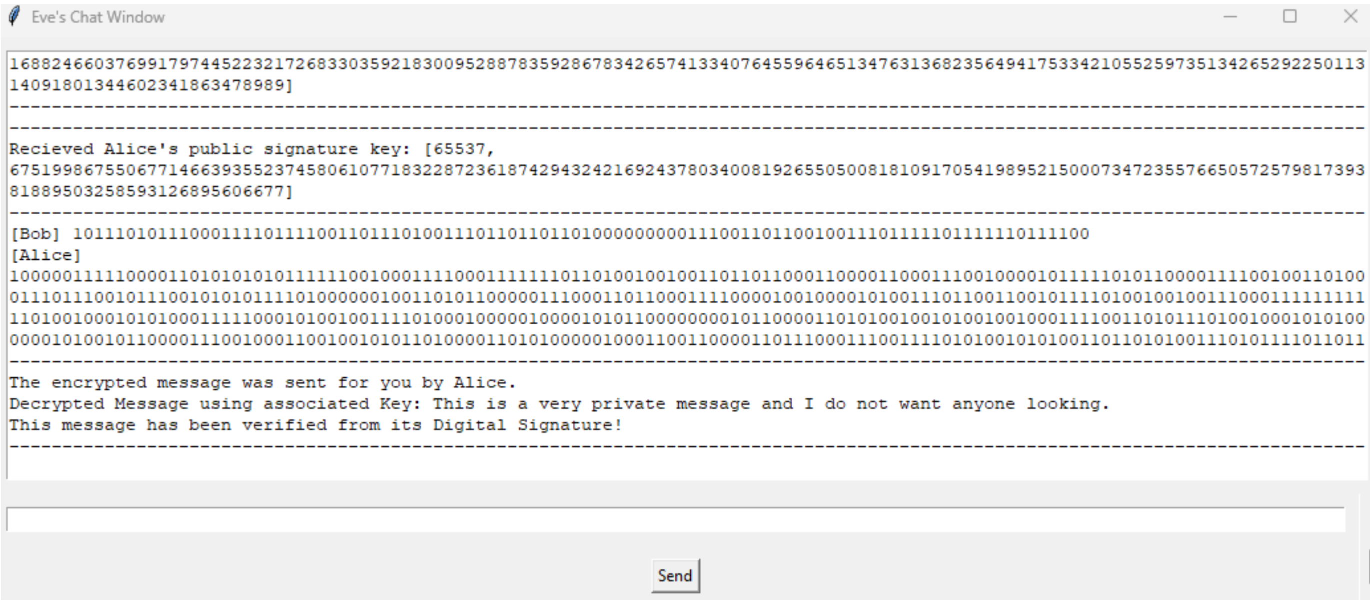


Bob and Eve both receive an encrypted message represented as a binary value. In contrast, Alice not only receives the identical message but also sees that it was sent to her by Bob. Moreover, Alice successfully decodes the message, revealing the content as "Hello Alice!"

Test 2: Within the DH Server, Alice Sends the Message "This is a very private message and I do not want anyone looking." to Eve

(./sendToUser Alice This is a very private message and I do not want anyone looking.)


Bob's Chat Window
6751998675506771466393552374580610771832287236187429432421692437803400819265505008181091705419895215000734723557665057257981739 3
818895032585931268956066 77]
[Eve] Eve has joined the chat.
Recieved Public Key from Eve: 19363834676988599274628262651507609218589257492866527613087813507349583090 7
Shared secret key with Eve: 15686712967513998793125981799702928895655487192771319551930728931046685789934
Recieved Eve's public signature key: [65537,
107529806885018115672984637715873063143207366776102347150530431111604242363126756445468964026298989947024463469421694526668034 30
14635872321771597498724353]
[Bob] 101110101110001111011110011011101001110110110110100000000011100110110010011101111101111110111100
[Alice]
1000001111100001101010101011111100100011110001111111011010010010011011011000110000110001110010000101111101011000011110010011010 0
0111011100101110010101011110100000001001101011000001110001101100011110000100100001010011101100110010111101001001001110001111111 11
1101001000101010001111100010100100111101000100000100001010110000000001011000011010100100101001001000111100110101110100100010101 00
0000101001011000011100100011001001010110100001101010000010001100110000110111000111001111010100101010011011010100111010111101101 1

Alice's Chat Window
Recieved Public Key from Eve: 193638346769885992746282626515076092185892574928665276130878135073495830907
Shared secret key with Eve: 121863339792539234469663192179601632052737635968851662711703996080077185 29587
Recieved Eve's public signature key: [65537,
107529806885018115672984637715873063143207366776102347150530431111604242363126756445468964026298989947024463469421694526668034 30
14635872321771597498724353]
[Bob] 101110101110001111011110011011101001110110110110100000000011100110110010011101111101111110111100
The encrypted message was sent for you by Bob.
Decrypted Message using associated Key: Hello Alice!
This message has been verified from its Digital Signature!
[Alice]
1000001111100001101010101011111100100011110001111111011010010010011011011000110000110001110010000101111101011000011110010011010 0
0111011100101110010101011110100000001001101011000001110001101100011110000100100001010011101100110010111101001001001110001111111 11
1101001000101010001111100010100100111101000100000100001010110000000001011000011010100100101001001000111100110101110100100010101 00
0000101001011000011100100011001001010110100001101010000010001100110000110111000111001111010100101010011011010100111010111101101 1
Send

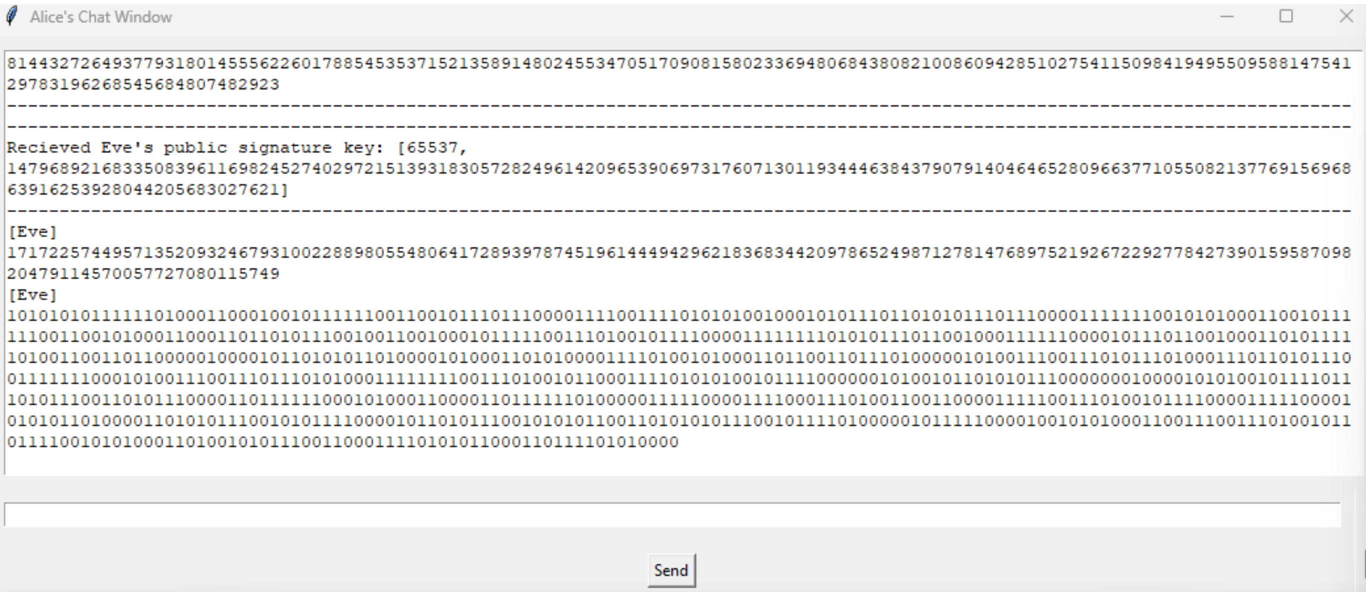


Bob and Alice both receive an encrypted message represented as a binary value. In contrast, Eve not only receives the identical message but also sees that it was sent to her by Alice. Moreover, Eve successfully decodes the message, revealing the content as "This is a very private message and I do not want anyone looking."

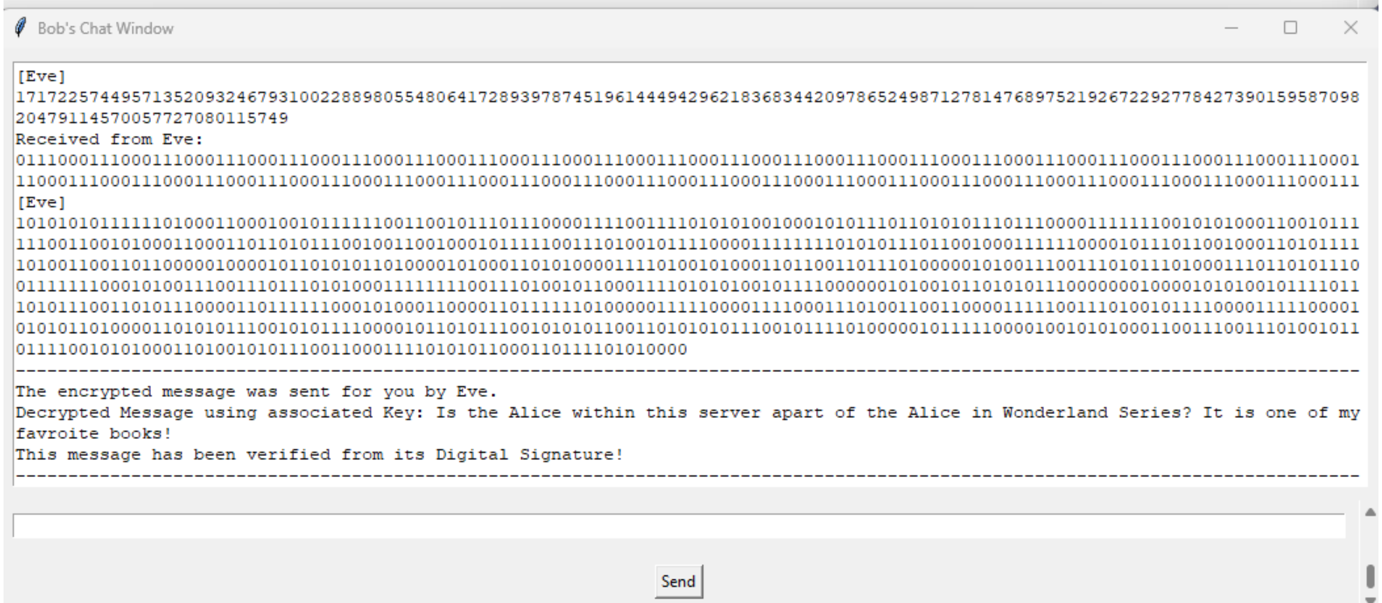
Test 3: Within the RSA Server, Eve sends her LFSR Key to Bob. Then she sends a message saying "Is the Alice within this server apart of the Alice in Wonderland Series? It is one of my favroite books!" to Bob.

(./sendLFSRKey Bob)

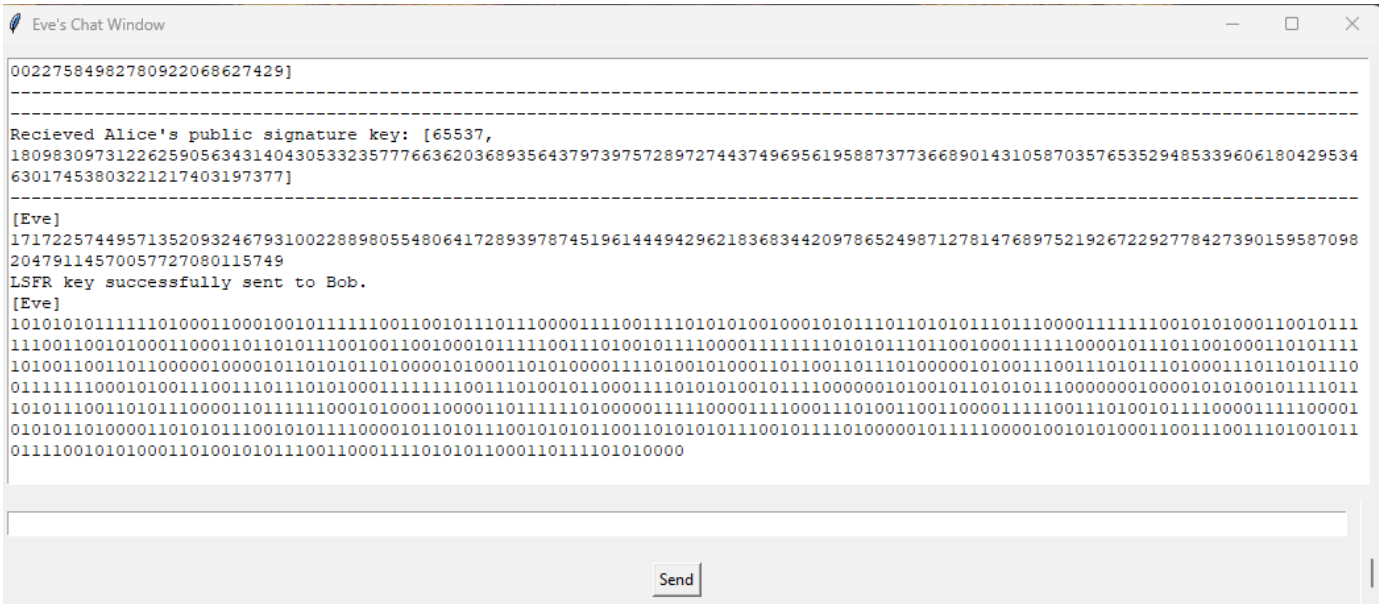
(./sendToUser Bob Is the Alice within this server apart of the Alice in Wonderland Series? It is one of my favroite books!)



Alice's Chat Window
8144327264937793180145556226017885453537152135891480245534705170908158023369480684380821008609428510275411509841949550958814 7541
29783196268545684807482923
Recieved Eve's public signature key: [65537,
147968921683350839611698245274029721513931830572824961420965390697317607130119344463843790791404646528096637710550821377691569 68
639162539280442056830276 21]
[Eve]
171722574495713520932467931002288980554806417289397874519614449429621836834420978652498712781476897521926722927784273901595870 98
20479114570057727080115749
[Eve]
1010101011111101000110001001011111100110010111011100001111001111010101001000101011101101010111011100001111111001010100011001 0111
1110011001010001100011011010111001001100100010111110011101001011110000111111110101011101100100011111100001011101100100011010 1111
1010011001101100000100001011010101101000010100011010100001111010010100011011001101110100000101001110011101011101000111011010 1110
0111111100010100111001110111010100011111111001110100101100011110101010010111100000010100101101010111000000001000010101001011 1011
1010111001101011100001101111110001010001100001101111110100000111110000111100011101001100110000111110011101001011110000111110 0001
0101011010000110101011100101011110000101101011100101010110011010101011100101111010000010111110000100101010001100111001110100 1011
011110010101000110100101011100110001111010101100011011110101 0000



Send
Bob's Chat Window
[Eve]
171722574495713520932467931002288980554806417289397874519614449429621836834420978652498712781476897521926722927784273901595870 98
20479114570057727080115749
Received from Eve:
0111000111 0001
1100011100 0111
[Eve]
1010101011111101000110001001011111100110010111011100001111001111010101001000101011101101010111011100001111111001010100011001 0111
1110011001010001100011011010111001001100100010111110011101001011110000111111110101011101100100011111100001011101100100011010 1111
1010011001101100000100001011010101101000010100011010100001111010010100011011001101110100000101001110011101011101000111011010 1110
0111111100010100111001110111010100011111111001110100101100011110101010010111100000010100101101010111000000001000010101001011 1011
1010111001101011100001101111110001010001100001101111110100000111110000111100011101001100110000111110011101001011110000111110 0001
0101011010000110101011100101011110000101101011100101010110011010101011100101111010000010111110000100101010001100111001110100 1011
011110010101000110100101011100110001111010101100011011110101 0000
The encrypted message was sent for you by Eve.
Decrypted Message using associated Key: Is the Alice within this server apart of the Alice in Wonderland Series? It is one of my
favroite books!
This message has been verified from its Digital Signature!
Send



Bob successfully acquires the LFSR Key from Eve, enabling Eve to send a message to Bob. Both Eve and Alice receive an encrypted message from Eve. However, when Bob receives the message, he is able to decode it using Eve's LFSR key, revealing the original message: "Is the Alice within this server a part of the Alice in Wonderland series? It is one of my favorite books!" Since the stream cipher is working as intended for all three tests, we can confirm that it is working as appropriately.

When a user client encrypts a message, they convert the intended plaintext into a binary representation and apply the encryption algorithm using their associated keys. The DH Client employs a shared secret key, while the RSA Client utilizes the generated LFSR Key. In both cases, the key is formatted as a binary value, the message is encrypted using that key, and transmitted to the server. The server subsequently broadcasts the encrypted message and dispatches a special command to the intended recipient. The recipient's user client then decrypts the received message, converts it back to text, and displays it.

4.2 AES

Upon configuring the ENCRYPTIONTYPE as "AES_ECB" or "AES_CBC", we incorporate the following methods from AES.py into the server:

```
from cryptography.hazmat.primitives.ciphers import Cipher,
    algorithms, modes
from cryptography.hazmat.backends import default_backend
from base64 import b64encode, b64decode
```

```
def pad(text):
    # PKCS7 padding
    block_size = 16
    if isinstance(text, str):
        text = text.encode('utf-8')  # Convert string to bytes
    pad_size = block_size - len(text) % block_size
    return text + bytes([pad_size] * pad_size)

def unpad(text):
    pad_size = text[-1]
    return text[:-pad_size]

def encrypt(plaintext, key, mode):
    plaintext = pad(plaintext)
    cipher = Cipher(algorithms.AES(key), mode,
        backend=default_backend())
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(plaintext) +
        encryptor.finalize()
    return b64encode(ciphertext)

def decrypt(ciphertext, key, mode):
    ciphertext = b64decode(ciphertext)
    cipher = Cipher(algorithms.AES(key), mode,
        backend=default_backend())
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) +
        decryptor.finalize()
    return unpad(plaintext)
```

The provided code snippet utilizes the Python cryptography library to implement encryption and decryption functionalities using the Advanced Encryption Standard (AES) algorithm. The encrypt and decrypt functions support both Cipher Block Chaining (CBC) and Electronic Codebook (ECB) modes. In the encrypt function, the plaintext is padded using PKCS7 padding (handled by the pad function) before encryption, and then encrypted using AES in the specified mode (either CBC or ECB). The result is base64-encoded ciphertext. Similarly, the decrypt function base64-decodes the ciphertext, decrypts it using AES in the specified mode, and then removes the padding (handled by the unpad function) to retrieve the original plaintext. These padding functions are crucial for ensuring proper block alignment and security in block cipher operations, safeguarding against potential vulnerabilities in cryptographic processes.

The provided code snippet for AES encryption in Cipher Block Chaining (CBC) and Electronic Codebook (ECB) modes involves an Initialization Vector (IV) in the CBC mode. The IV is a random or unique value that is XORed with the first block of plaintext before encryption. This initialization ensures that even if the same plaintext is encrypted multiple times with the same key, the resulting ciphertext will differ due to the uniqueness introduced by the IV. In the encrypt function, the IV is implicitly handled by the Cipher object in CBC mode.

Earlier, it was observed that user clients in both servers transmit various components to the server before joining, one of which is a Public IV value. Prior to joining, a user client generates its unique IV value within its Chat_GUI class:

```
if(ENCRYPTIONTYPE == "AES_CBC"):
    self.public_IVs = {}
    self.IV = os.urandom(16)
elif(ENCRYPTIONTYPE == "DES_CBC"):
    self.public_IVs = {}
    self.IV = os.urandom(8)
```

Subsequently, the user client stores this IV value and shares it with the server, which then broadcasts it to all other users. In a manner analogous to the public key exchange process, the server also sends the user the Public IVs of all other users. This exchange mechanism ensures that each user client possesses both its own IV and the Public IVs of other users.

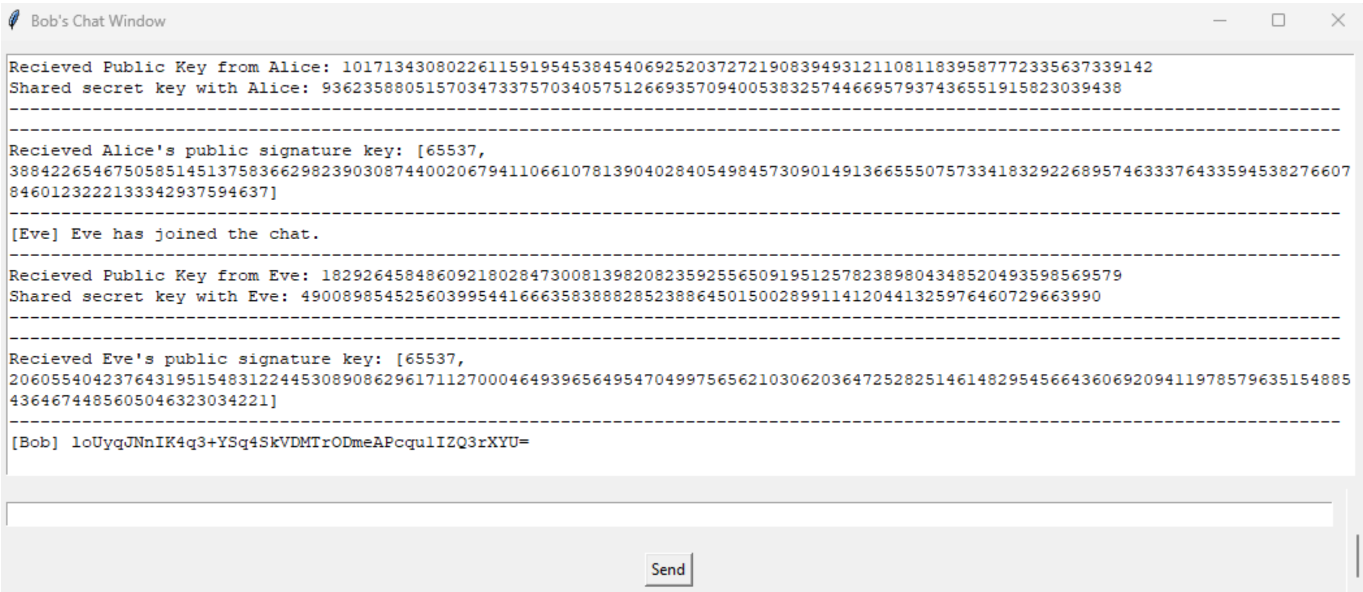
Note, the discrepancy in the size of the Initialization Vector (IV) between AES and DES encryption is due to the block sizes of these encryption algorithms. AES operates with a block size of 128 bits (16 bytes), while DES has a smaller block size of 64 bits (8 bytes).

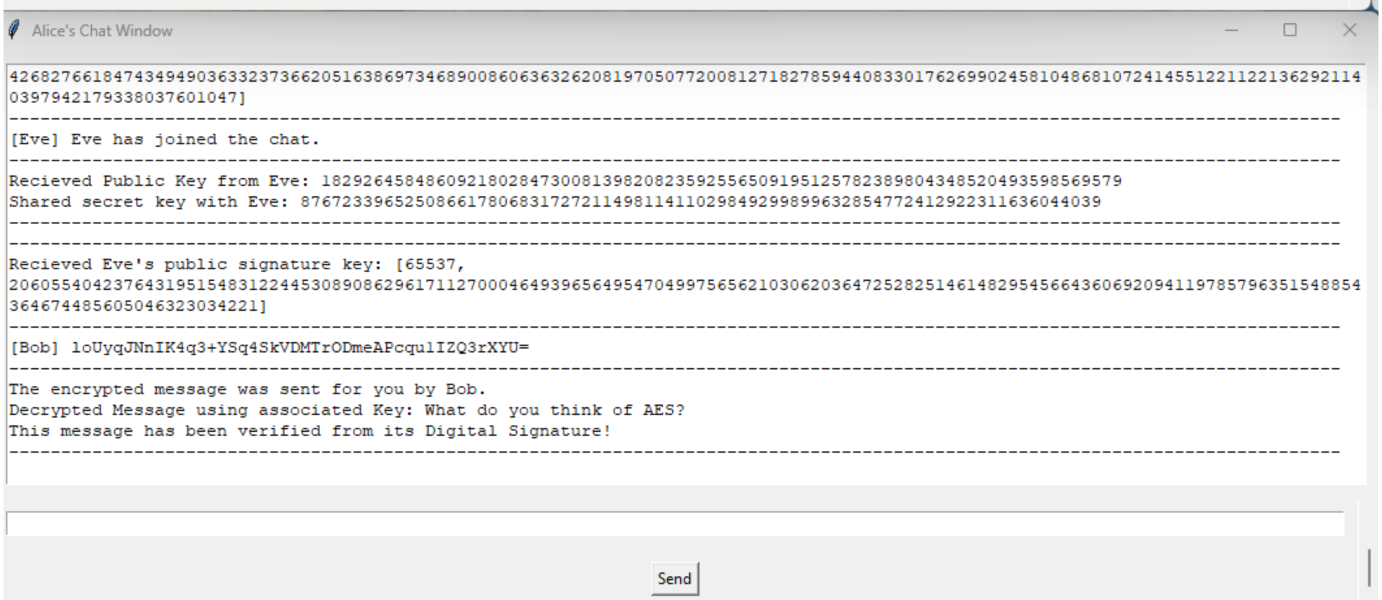
When a user client encrypts a message using the AES encryption algorithm, they employ the provided key and selected mode for encryption. In this context, let's assume the user client parses their plaintext message, encryption key, and the chosen mode into the encryption process. The process begins by checking if the chosen encryption type is AES in Cipher Block Chaining (CBC) mode. If so, the user client uses a unique Initialization Vector (IV), generated as os.urandom(16). Next, the plaintext message is padded using PKCS7 padding to ensure it aligns with the block size (128 bits or 16 bytes) required by AES. Subsequently, the user client encrypts the padded plaintext using the AES algorithm with the specified key and mode. In CBC mode, the IV is XORed with the first block of plaintext before encryption, enhancing the security of the process. The resulting ciphertext, along with the IV, is then sent to the server, which can distribute this encrypted message to other users. The targeted user, upon receiving the encrypted message, can use the shared key and IV to decrypt the ciphertext and retrieve the original plaintext, completing the secure communication process. One last thing to note is that the user clients format their keys into bytes so that it can be parsed through the AES algorithm.

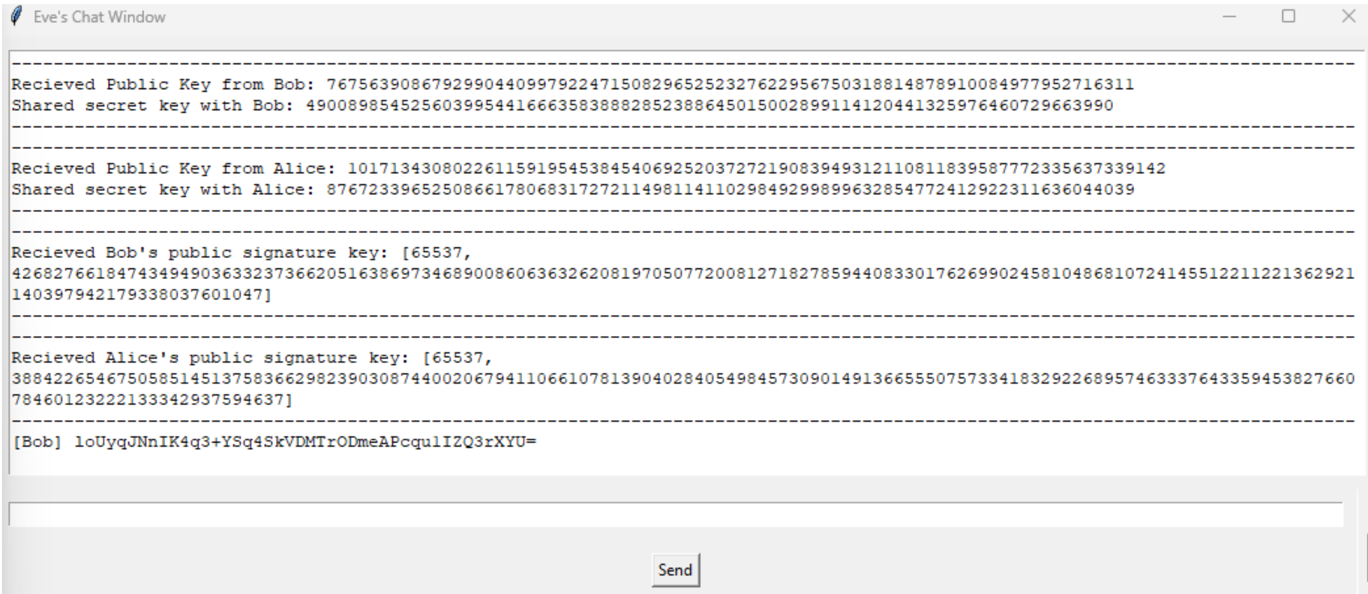
Consider a scenario where, for each encryption method mentioned, Alice, Bob, and Eve are present on the server. Assume that Bob intends to convey a private message to Alice. In this context, Bob transmits messages of different lengths to observe the resulting output. Specifically, let's explore this process using the AES Encryption. algorithm.

Test 1: Within the DH Server and the Encryption Type is set to "AES_ECB", Bob Sends the Message "What do you think of AES?" to Alice.

(./sendToUser Alice What do you think of AES?)



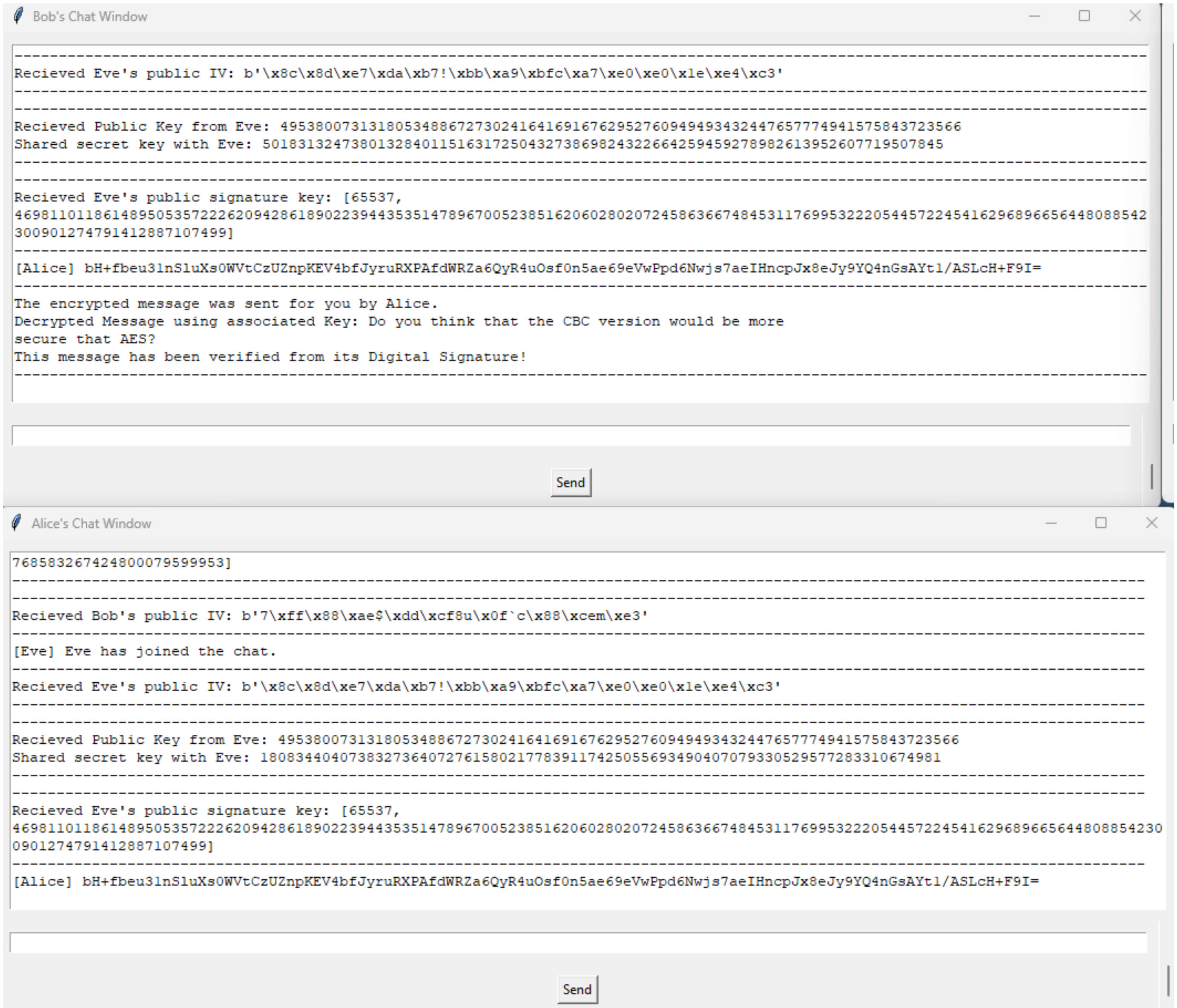


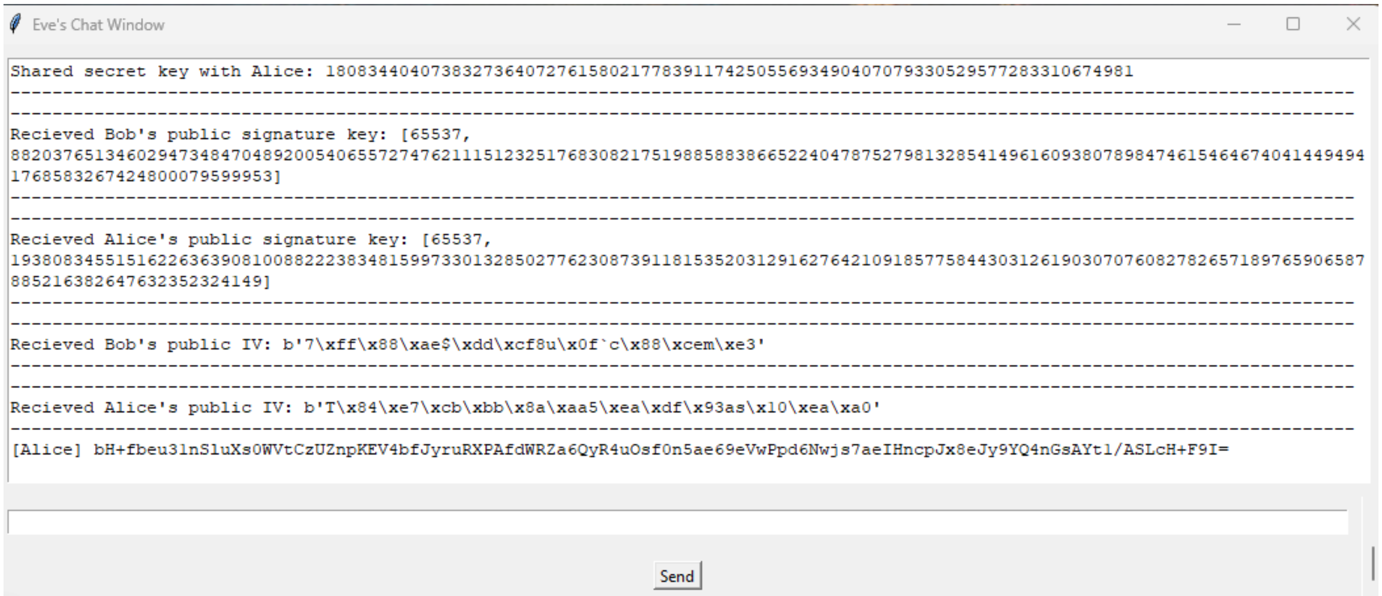


Bob and Eve both receive an encrypted message. In contrast, Alice not only receives the identical message but also sees that it was sent to her by Bob. Moreover, Alice successfully decodes the message, revealing the content as "What do you think of AES?"

Test 2: Within the DH Server and the Encryption Type is set to "AES_CBC", Alice sends the Message "Do you think that the CBC version would be more secure that AES?" to Bob.

(./sendToUser Bob Do you think that the CBC version would be more secure that AES?)


Bob's Chat Window
Recieved Eve's public IV: b'\x8c\x8d\xe7\xda\xb7!\xbb\xa9\xbfc\xa7\xe0\xe0\x1e\xe4\xc3'
Recieved Public Key from Eve: 49538007313180534886727302416416916762952760949493432447657774941575843723566
Shared secret key with Eve: 50183132473801328401151631725043273869824322664259459278982613952607719507845
Recieved Eve's public signature key: [65537,
46981101186148950535722262094286189022394435351478967005238516206028020724586366748453117699532220544572245416296896656448088542300901274791412887107499]
[Alice] bH+fbeu31nSluXs0WVtCzUZnpKEV4bfJyruRXPAfdWRZa6QyR4uOsf0n5ae69eVwPpd6Nwjs7aeIHncpJx8eJy9YQ4nGsAYt1/ASLcH+F9I=
The encrypted message was sent for you by Alice.
Decrypted Message using associated Key: Do you think that the CBC version would be more secure that AES?
This message has been verified from its Digital Signature!
Send
Alice's Chat Window
768583267424800079599953]
Recieved Bob's public IV: b'7\xff\x88\xae$\xdd\xcf8u\x0f`c\x88\xcem\xe3'
[Eve] Eve has joined the chat.
Recieved Eve's public IV: b'\x8c\x8d\xe7\xda\xb7!\xbb\xa9\xbfc\xa7\xe0\xe0\x1e\xe4\xc3'
Recieved Public Key from Eve: 49538007313180534886727302416416916762952760949493432447657774941575843723566
Shared secret key with Eve: 18083440407383273640727615802177839117425055693490407079330529577283310674981
Recieved Eve's public signature key: [65537,
46981101186148950535722262094286189022394435351478967005238516206028020724586366748453117699532220544572245416296896656448088542300901274791412887107499]
[Alice] bH+fbeu31nSluXs0WVtCzUZnpKEV4bfJyruRXPAfdWRZa6QyR4uOsf0n5ae69eVwPpd6Nwjs7aeIHncpJx8eJy9YQ4nGsAYt1/ASLcH+F9I=
Send

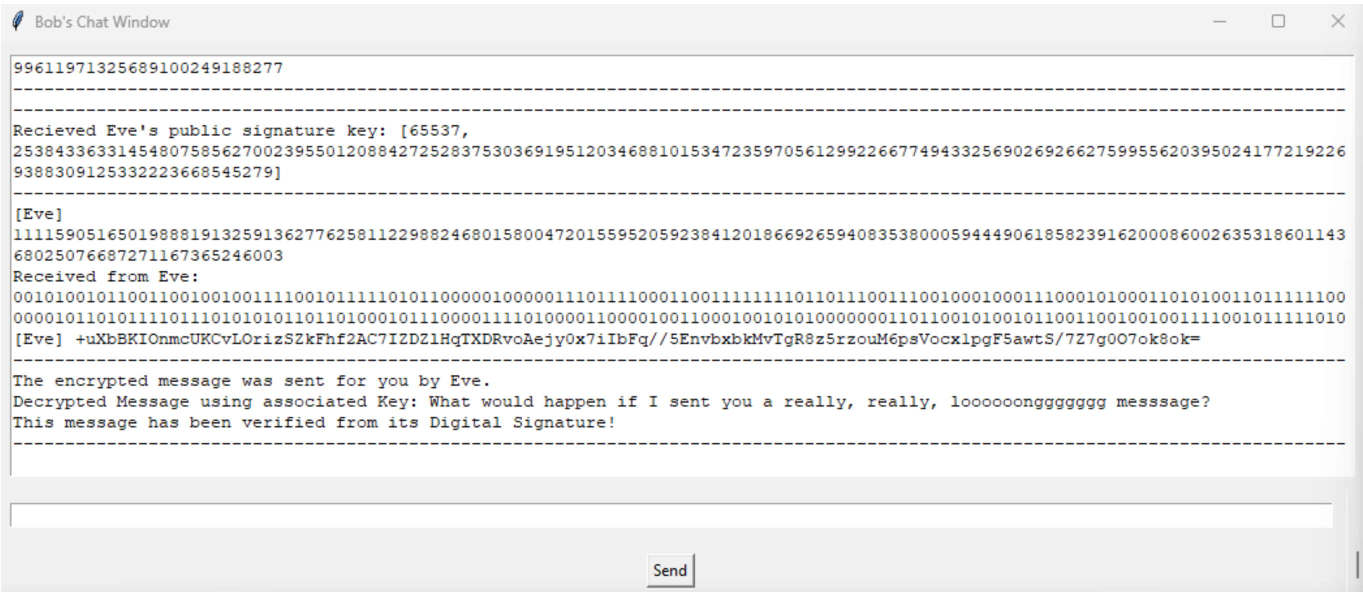


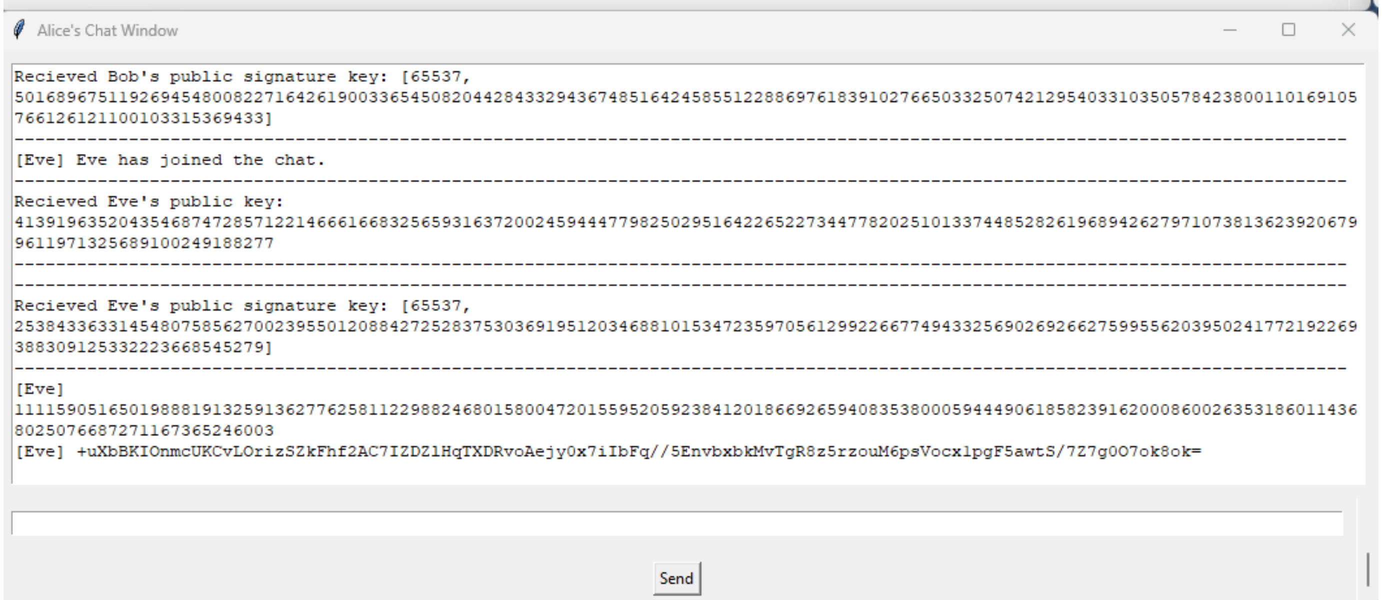
Alice and Eve both receive an encrypted message. In contrast, Bob not only receives the identical message but also sees that it was sent to him by Alice. Moreover, Bob successfully decodes the message, revealing the content as "Do you think that the CBC version would be more secure that AES?"

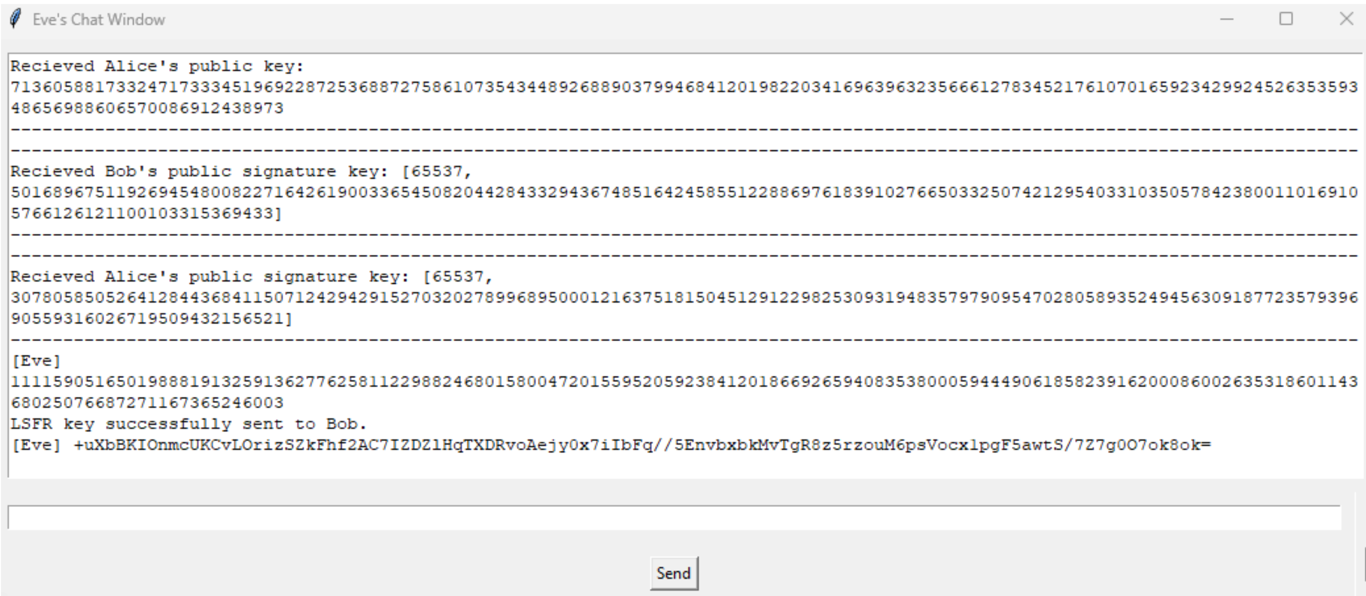
Test 3: Within the RSA Server and the Encryption Type is set to "AES", Eve first sends her LFSR Key to Bob. She then sends the Message "What would happen if I sent you a really, really, loooooonnggggg messsage?" to Bob.

(./sendLFSRKey Bob)

(./sendToUser Bob What would happen if I sent you a really, really, loooooonnggggg messsage?)


Bob's Chat Window
99611971325689100249188277
Recieved Eve's public signature key: [65537,
2538433633145480758562700239550120884272528375303691951203468810153472359705612992266774943325690269266275995562039502417721922 6
93883091253322236685452 79]
[Eve]
1111590516501988819132591362776258112298824680158004720155952059238412018669265940835380005944490618582391620008600263531860114 3
68025076687271167365246003
Received from Eve:
0010100101100110010010011110010111110101100000100000111011110001100111111110110111001110010001000111000101000110101001101111110 0
0000101101011110111010101011011010001011100001111010000110000100110001001010100000001101100101001011001100100100111100101111101 0
[Eve] +uXbBKIOnmcUKCvLOrizSZkFhf2AC7IZDZlHqTXDRvoAejy0x7iIbFq//5EnvbxbkMvTgR8z5rzouM6psVocxlpgF5awtS/7Z7g0O7ok8ok=
The encrypted message was sent for you by Eve.
Decrypted Message using associated Key: What would happen if I sent you a really, really, loooooongggggg messsage?
This message has been verified from its Digital Signature!
Send


Alice's Chat Window
Recieved Bob's public signature key: [65537,
501689675119269454800822716426190033654508204428433294367485164245855122886976183910276650332507421295403310350578423800110169105
7661261211001033153694 33]
[Eve] Eve has joined the chat.
Recieved Eve's public key:
413919635204354687472857122146661668325659316372002459444779825029516422652273447782025101337448528261968942627971073813623920679
961197132568910024918827 7
Recieved Eve's public signature key: [65537,
253843363314548075856270023955012088427252837530369195120346881015347235970561299226677494332569026926627599556203950241772192269
38830912533222366854527 9]
[Eve]
111159051650198881913259136277625811229882468015800472015595205923841201866926594083538000594449061858239162000860026353186011436
8025076687271167365246003
[Eve] +uXbBKIOnmcUKCvLOrizSZkFhf2AC7IZDZlHqTXDRvoAejy0x7iIbFq//5EnvbxbkMvTgR8z5rzouM6psVocxlpgF5awtS/7Z7g0O7ok8ok=
Send

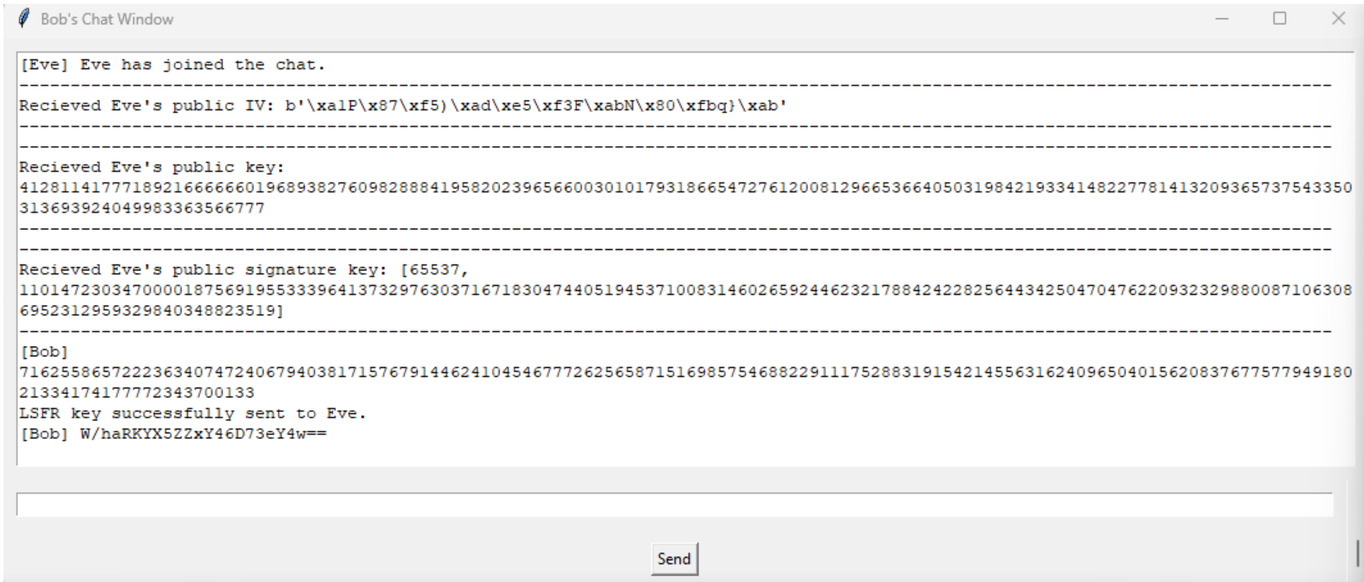


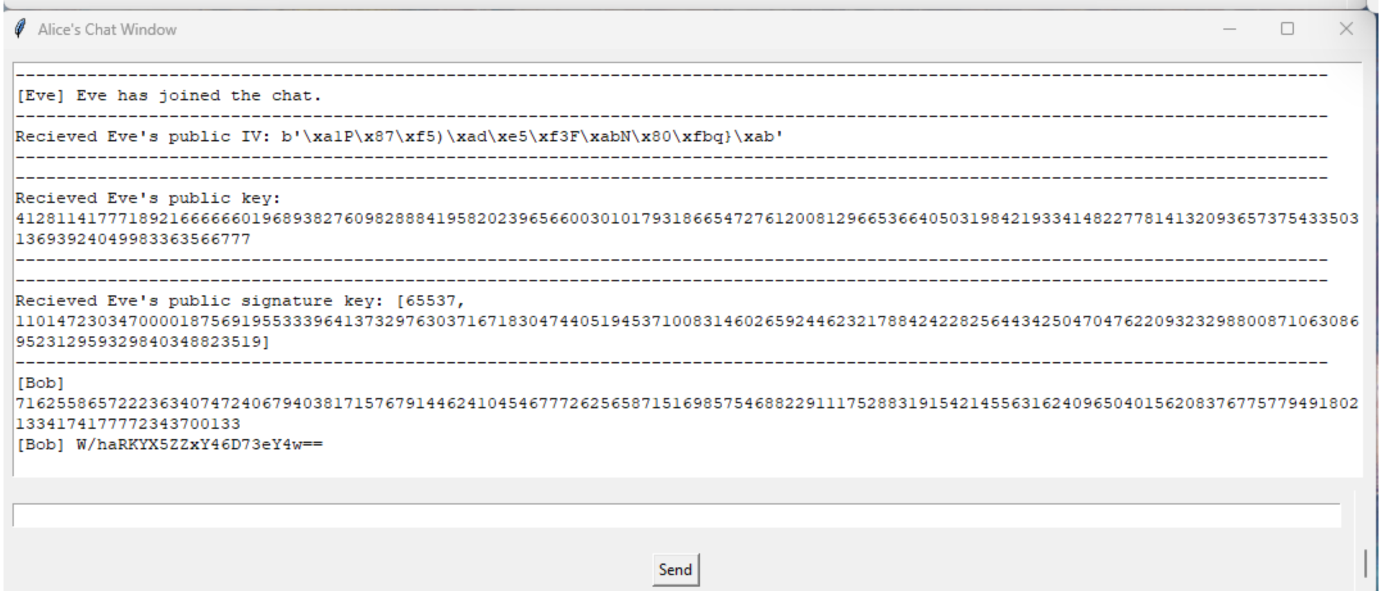
Alice and Eve both receive an encrypted message. In contrast, Bob not only receives the identical message but also sees that it was sent to him by Eve. Moreover, Bob successfully decodes the message, revealing the content as "Do you think that the CBC version would be more secure that AES?"

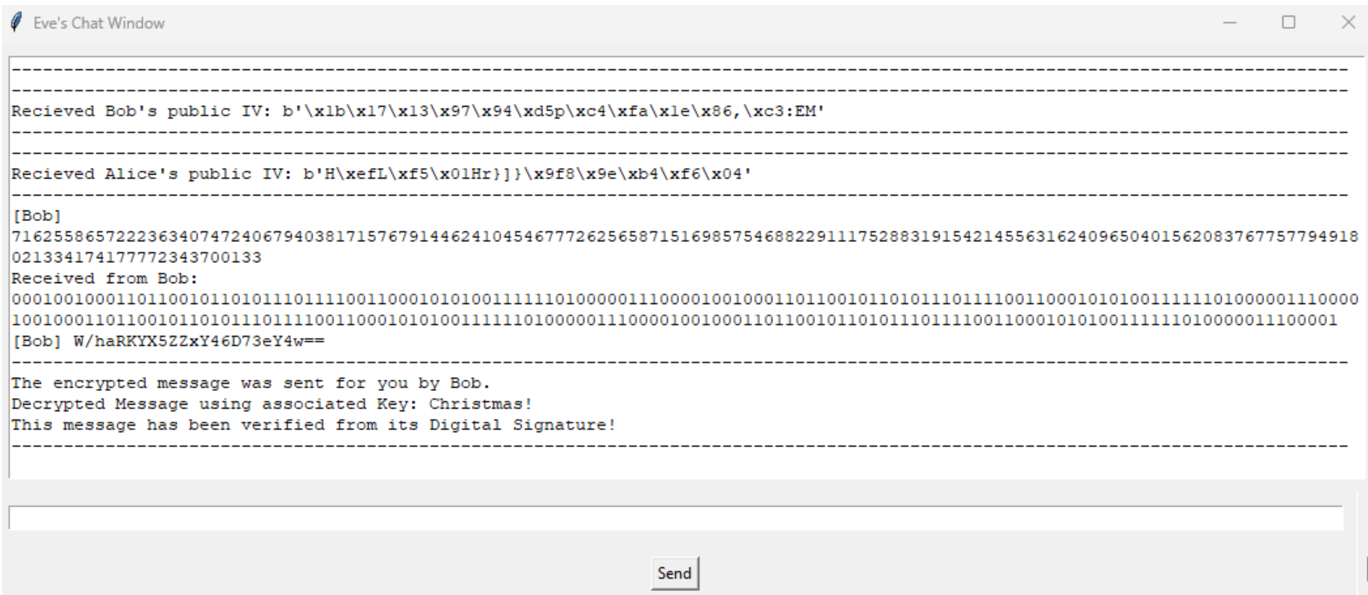
Test 4: Within the RSA Server and the Encryption Type is set to "AES_CBC", Bob first sends his LFSR Key to Eve. He then sends the Message "Christmas!" to Eve.

(./sendLFSRKey Eve)

(./sendToUser Bob Christmas!)


Bob's Chat Window
[Eve] Eve has joined the chat.
Recieved Eve's public IV: b'\xa1P\x87\xf5)\xad\xe5\xf3F\xabN\x80\xfbq}\xab'
Recieved Eve's public key:
4128114177718921666666019689382760982888419582023965660030101793186654727612008129665366405031984219334148227781413209365737543350
313693924049983363566777
Recieved Eve's public signature key: [65537,
1101472303470000187569195533396413732976303716718304744051945371008314602659244623217884242282564434250470476220932329880087106308
6952312959329840348823519]
[Bob]
7162558657222363407472406794038171576791446241045467772625658715169857546882291117528831915421455631624096504015620837677577949180
2133417417777234370O133
LSFR key successfully sent to Eve.
[Bob] W/haRKYX5ZZxY46D73eY4w==
Send


Alice's Chat Window
[Eve] Eve has joined the chat.
Recieved Eve's public IV: b'\xa1P\x87\xf5)\xad\xe5\xf3F\xabN\x80\xfbq}\xab'
Recieved Eve's public key:
41281141777189216666660196893827609828884195820239656600301017931866547276120081296653664050319842193341482277814132093657375433503
136939240499833635667777
Recieved Eve's public signature key: [65537,
11014723034700001875691955333964137329763037167183047440519453710083146026592446232178842422825644342504704762209323298800871063086
952312959329840348823519]
[Bob]
71625586572223634074724067940381715767914462410454677726256587151698575468822911175288319154214556316240965040156208376775779491802
133417417777234370O133
[Bob] W/haRKYX5ZZxY46D73eY4w==
Send



Bob and Alice both receive an encrypted message. In contrast, Eve not only receives the identical message but also sees that it was sent to her by Bob. Moreover, Eve successfully decodes the message, revealing the content as "Christmas!" Since the AES encryption/decryption algorithm is working as intended for all four tests, we can confirm that it is working as appropriately.

4.3 DES

Upon configuring the ENCRYPTIONTYPE as "DES_ECB" or "DES_CBC", we incorporate the following methods from DES.py into the server:

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
from base64 import b64encode, b64decode

def encrypt(plaintext, key, mode, iv=None):
    if mode == DES.MODE_ECB:
        cipher = DES.new(key, DES.MODE_ECB)
    elif mode == DES.MODE_CBC:
        if iv is None:
            raise ValueError("IV is required for CBC
                mode")
        cipher = DES.new(key, DES.MODE_CBC, iv)
    else:
        raise ValueError("Invalid mode")
```

```
    plaintext = pad(plaintext.encode(),
       DES.block_size)
    ciphertext = cipher.encrypt(plaintext)

    return b64encode(ciphertext)

def decrypt(ciphertext, key, mode, iv=None):
    ciphertext = b64decode(ciphertext)

    if mode == DES.MODE_ECB:
        cipher = DES.new(key, DES.MODE_ECB)
    elif mode == DES.MODE_CBC:
        if iv is None:
            raise ValueError("IV is required for CBC
               mode")
        cipher = DES.new(key, DES.MODE_CBC, iv)
    else:
        raise ValueError("Invalid mode")

    plaintext = unpad(cipher.decrypt(ciphertext),
       DES.block_size)

    return plaintext.decode()
```

This Python code provides a simple implementation of the Data Encryption Standard (DES) symmetric encryption algorithm using the Crypto.Cipher module. It defines two functions, encrypt and decrypt, for encrypting and decrypting data, respectively. The encrypt function takes plaintext, a key, a mode (either ECB or CBC), and an optional initialization vector (IV) as input, pads the plaintext to match the block size, and then encrypts it using DES in the specified mode. The result is Base64-encoded ciphertext. The decrypt function reverses this process, decoding the Base64 ciphertext, decrypting it using DES, and then removing the padding to recover the original plaintext. The code employs Electronic Codebook (ECB) or Cipher Block Chaining (CBC) modes, and for CBC mode, it requires an IV to enhance security.

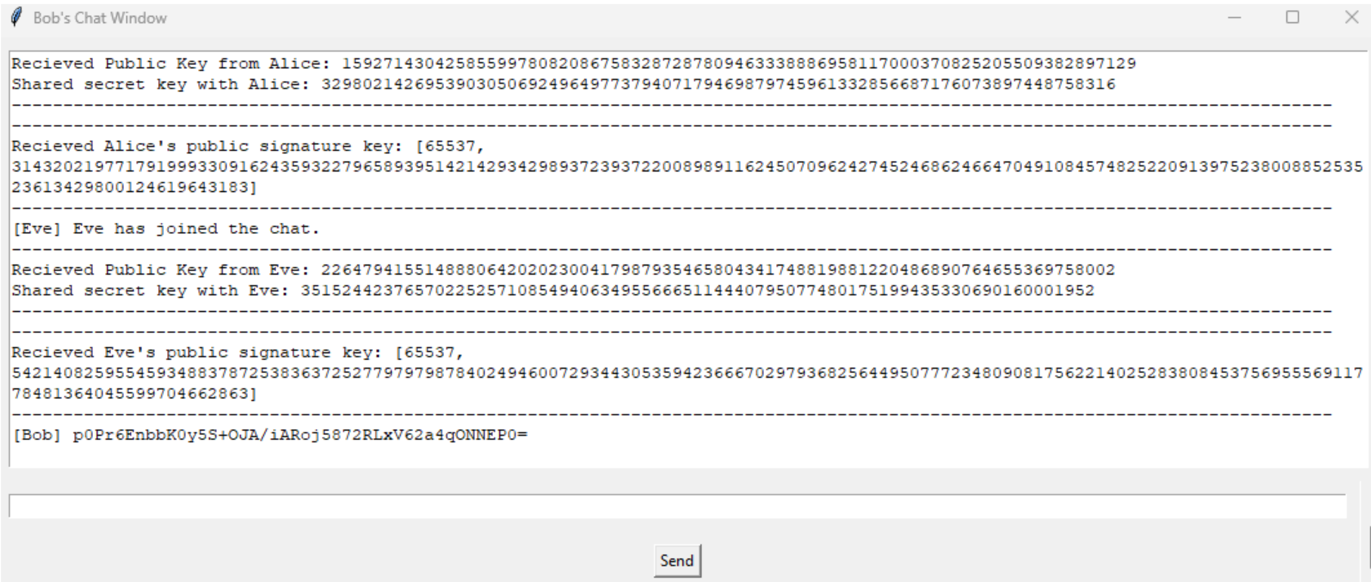
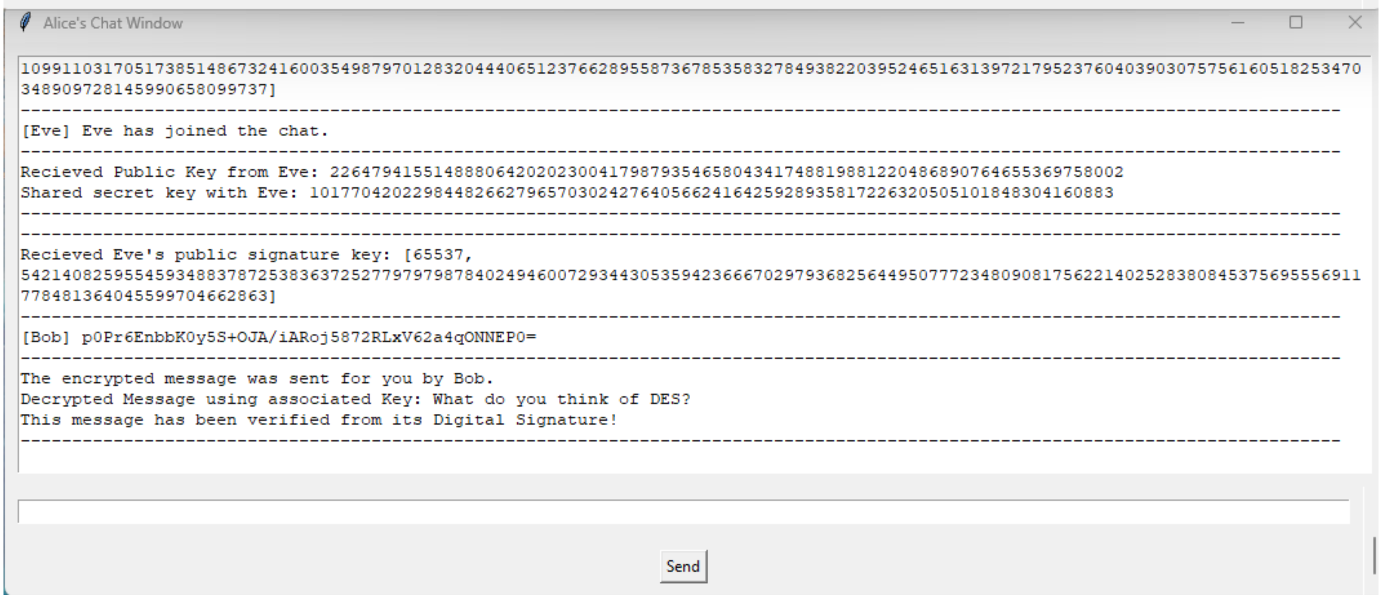
As noted before, the User clients will generate Public IV values for the DES CBC mode, which was sized approriately to fit its block restrictions.

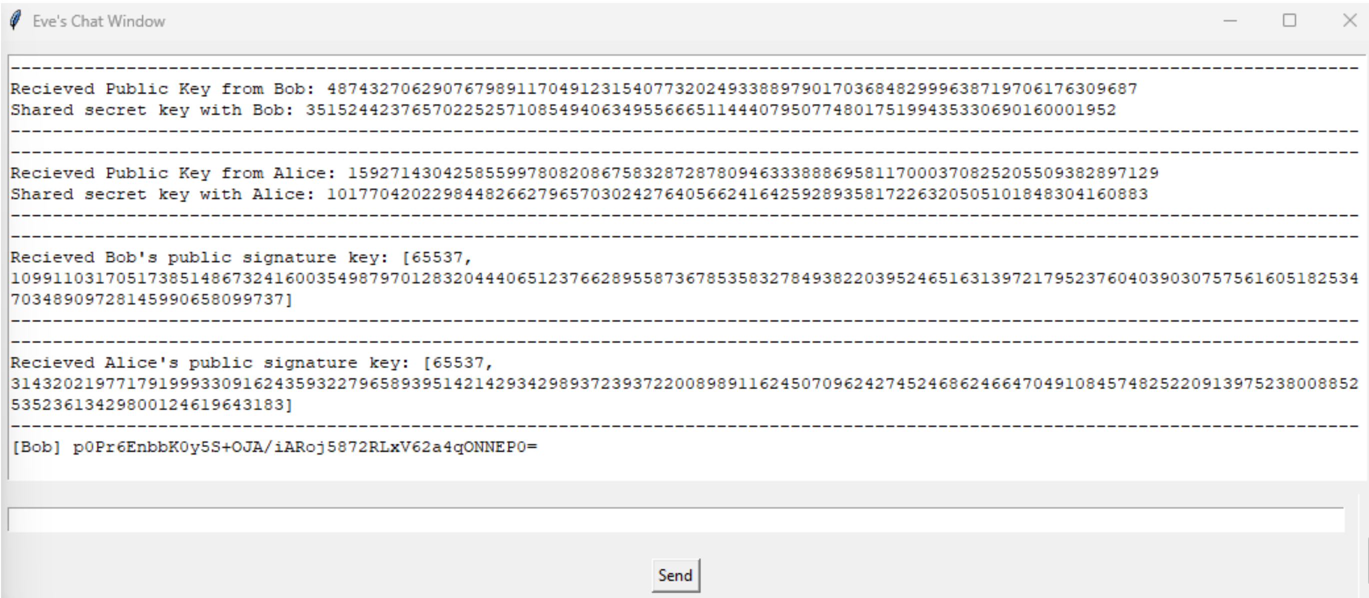
When a user client encrypts a message using the DES encryption algorithm, they employ the provided key and selected mode for encryption. In this context, let's assume the user client parses their plaintext message, encryption key, and the chosen mode into the encryption process. The process begins by checking if the chosen encryption type is DES in Cipher Block Chaining (CBC) mode. If so, the user client uses a unique Initialization Vector (IV), generated as os.urandom(8). Next, the user client invokes the encrypt function from the provided code, passing the plaintext message, encryption key, selected mode (either DES.MODE_ECB or DES.MODE_CBC), and the generated IV in the case of CBC mode. The function internally checks the mode and initializes a DES cipher accordingly, using the provided key and IV for CBC mode. Before encryption, the plaintext is padded to align with the DES block size. For ECB mode, the encryption is straightforward, while for CBC mode, each block of plaintext is XORed with the previous ciphertext block (or IV for the first block) before encryption. The resulting ciphertext is then Base64-encoded to facilitate easy transmission and storage. This resulting ciphertext, along with the IV, is then sent to the server, which can distribute this encrypted message to other users. The targeted user, upon receiving the encrypted message, can use the shared key and IV to decrypt the ciphertext and retrieve the original plaintext, completing the secure communication process. One last thing to note is that the user clients format their keys into bytes so that it can be parsed through the DES algorithm.

As like the AES Algorithms, lets perform the same tests upon the DES Algorithm.

Test 1: Within the DH Server and the Encryption Type is set to "DES_ECB", Bob Sends the Message "What do you think of DES?" to Alice.

(./sendToUser Alice What do you think of DES?)

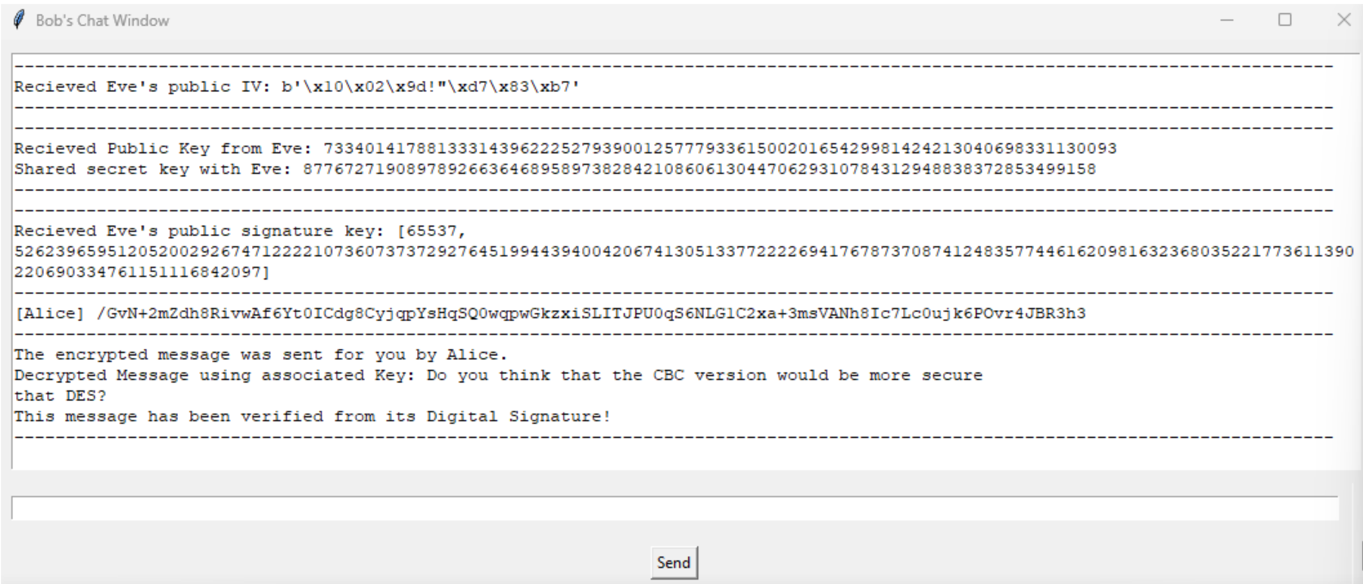

Bob's Chat Window
Recieved Public Key from Alice: 15927143042585599780820867583287287809463338886958117000370825205509382897129
Shared secret key with Alice: 32980214269539030506924964977379407179469879745961332856687176073897448758316
Recieved Alice's public signature key: [65537,
31432021977179199933091624359322796589395142142934298937239372200898911624507096242745246862466470491084574825220913975238008852535
23613429800124619643183]
[Eve] Eve has joined the chat.
Recieved Public Key from Eve: 22647941551488806420202300417987935465804341748819881220486890764655369758002
Shared secret key with Eve: 35152442376570225257108549406349556665114440795077480175199435330690160001952
Recieved Eve's public signature key: [65537,
54214082595545934883787253836372527797979878402494600729344305359423666702979368256449507772348090817562214025283808453756955569117
78481364045599704662863]
[Bob] p0Pr6EnbbK0y5S+OJA/iARoj5872RLxV62a4qONNEP0=
Send

Alice's Chat Window
1099110317051738514867324160035498797012832044406512376628955873678535832784938220395246516313972179523760403903075756160518253470
348909728145990658099737]
[Eve] Eve has joined the chat.
Recieved Public Key from Eve: 22647941551488806420202300417987935465804341748819881220486890764655369758002
Shared secret key with Eve: 101770420229844826627965703024276405662416425928935817226320505101848304160883
Recieved Eve's public signature key: [65537,
5421408259554593488378725383637252779797987840249460072934430535942366670297936825644950777234809081756221402528380845375695556911
778481364045599704662863]
[Bob] p0Pr6EnbbK0y5S+OJA/iARoj5872RLxV62a4qONNEP0=
The encrypted message was sent for you by Bob.
Decrypted Message using associated Key: What do you think of DES?
This message has been verified from its Digital Signature!
Send

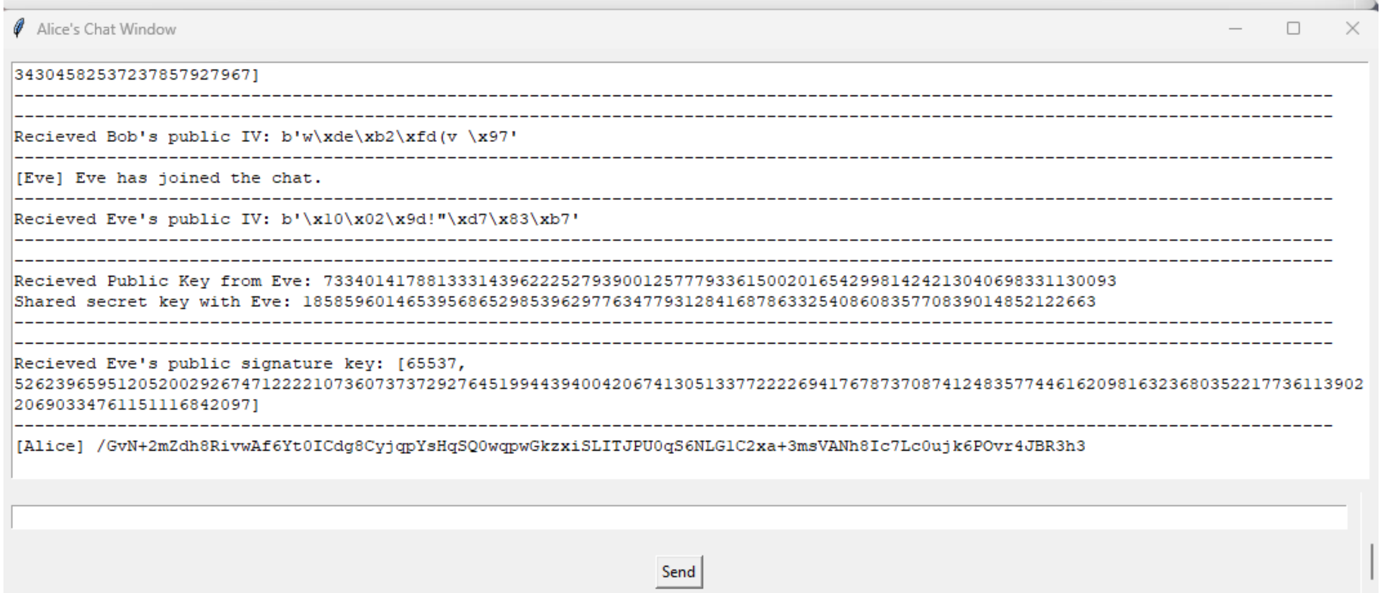


Bob and Eve both receive an encrypted message. In contrast, Alice not only receives the identical message but also sees that it was sent to her by Bob. Moreover, Alice successfully decodes the message, revealing the content as "What do you think of DES?"

Test 2: Within the DH Server and the Encryption Type is set to "DES_CBC", Alice sends the Message "Do you think that the CBC version would be more secure that DES?" to Bob.

(./sendToUser Bob Do you think that the CBC version would be more secure that DES?)


Bob's Chat Window
Recieved Eve's public IV: b'\x10\x02\x9d!"\xd7\x83\xb7'
Recieved Public Key from Eve: 7334014178813331439622252793900125777933615002016542998142421304069833113009 3
Shared secret key with Eve: 87767271908978926636468958973828421086061304470629310784312948838372853499158
Recieved Eve's public signature key: [65537,
5262396595120520029267471222210736073737292764519944394004206741305133772222694176787370874124835774461620981632368035221773611390
22069033476115111684209 7]
[Alice] /GvN+2mZdh8RivwAf6Yt0ICdg8CyjqpYsHqSQ0wqpwGkzxiSLITJPU0qS6NLG1C2xa+3msVANh8Ic7Lc0ujk6POvr4JBR3h3
The encrypted message was sent for you by Alice.
Decrypted Message using associated Key: Do you think that the CBC version would be more secure
that DES?
This message has been verified from its Digital Signature!
Send

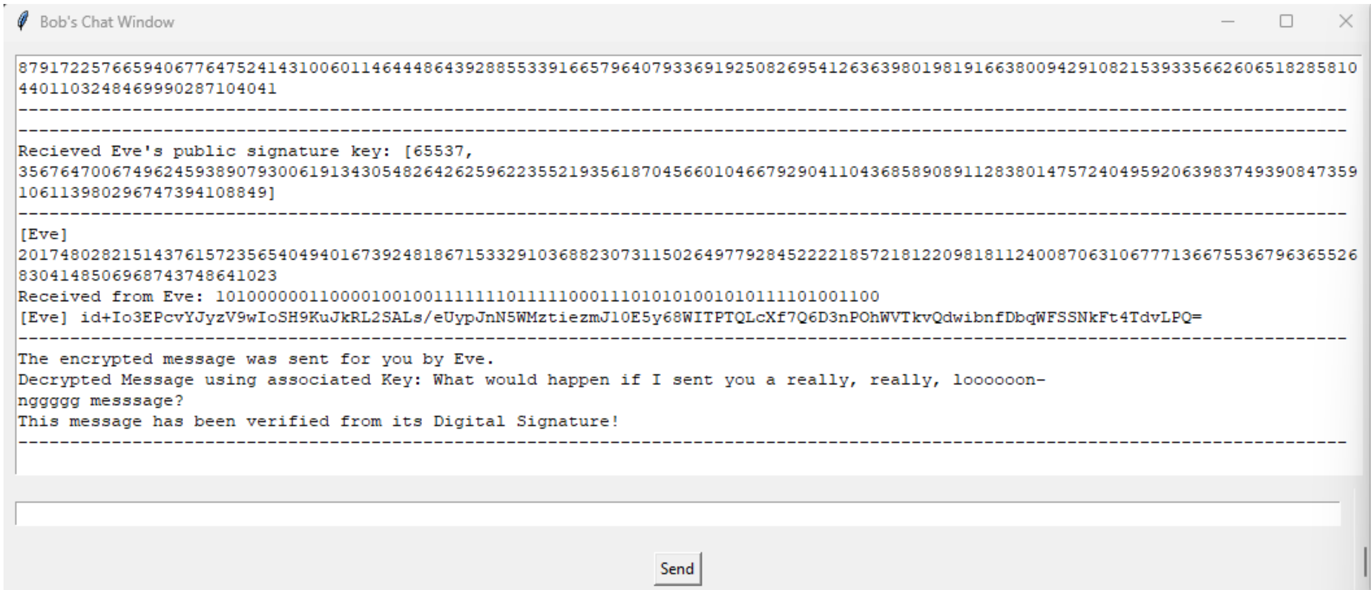
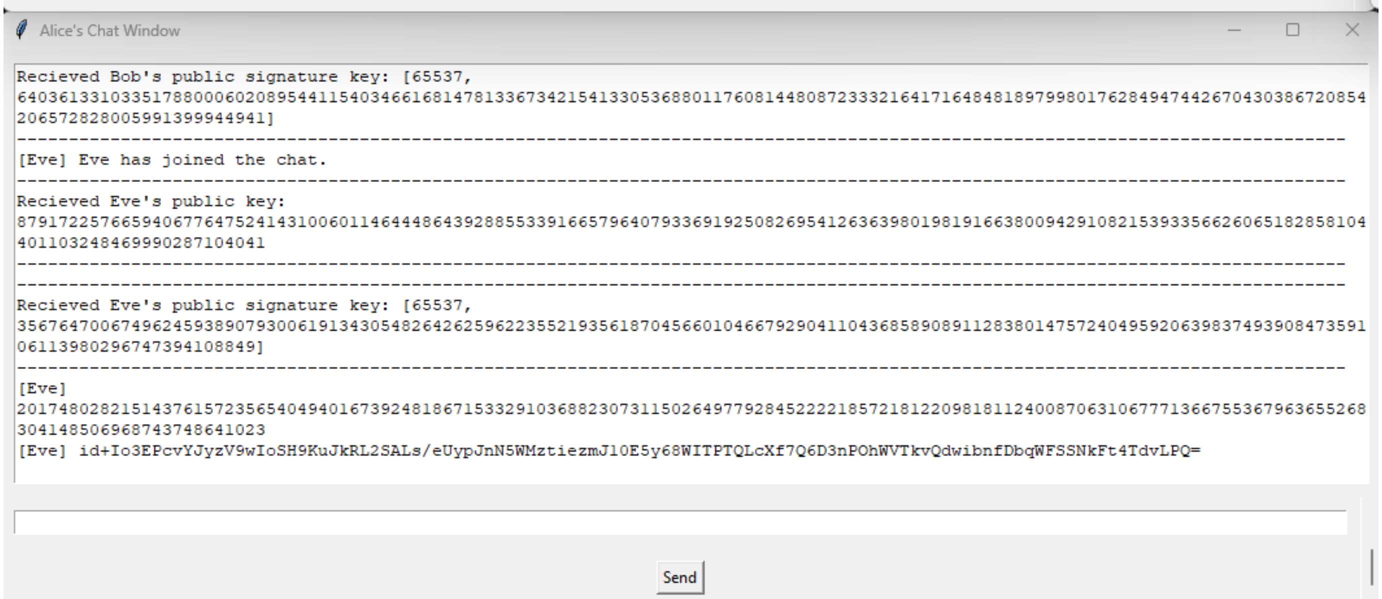

Alice's Chat Window
34304582537237857927967]
Recieved Bob's public IV: b'w\xde\xb2\xfd(v \x97'
[Eve] Eve has joined the chat.
Recieved Eve's public IV: b'\x10\x02\x9d!"\xd7\x83\xb7'
Recieved Public Key from Eve: 73340141788133314396222527939001257779336150020165429981424213040698331130093
Shared secret key with Eve: 18585960146539568652985396297763477931284168786332540860835770839014852122663
Recieved Eve's public signature key: [65537,
5262396595120520029267471222210736073737292764519944394004206741305133772222694176787370874124835774461620981632368035221773611390 2
2069033476115111684209 7]
[Alice] /GvN+2mZdh8RivwAf6Yt0ICdg8CyjqpYsHqSQ0wqpwGkzxiSLITJPU0qS6NLG1C2xa+3msVANh8Ic7Lc0ujk6POvr4JBR3h3
Send

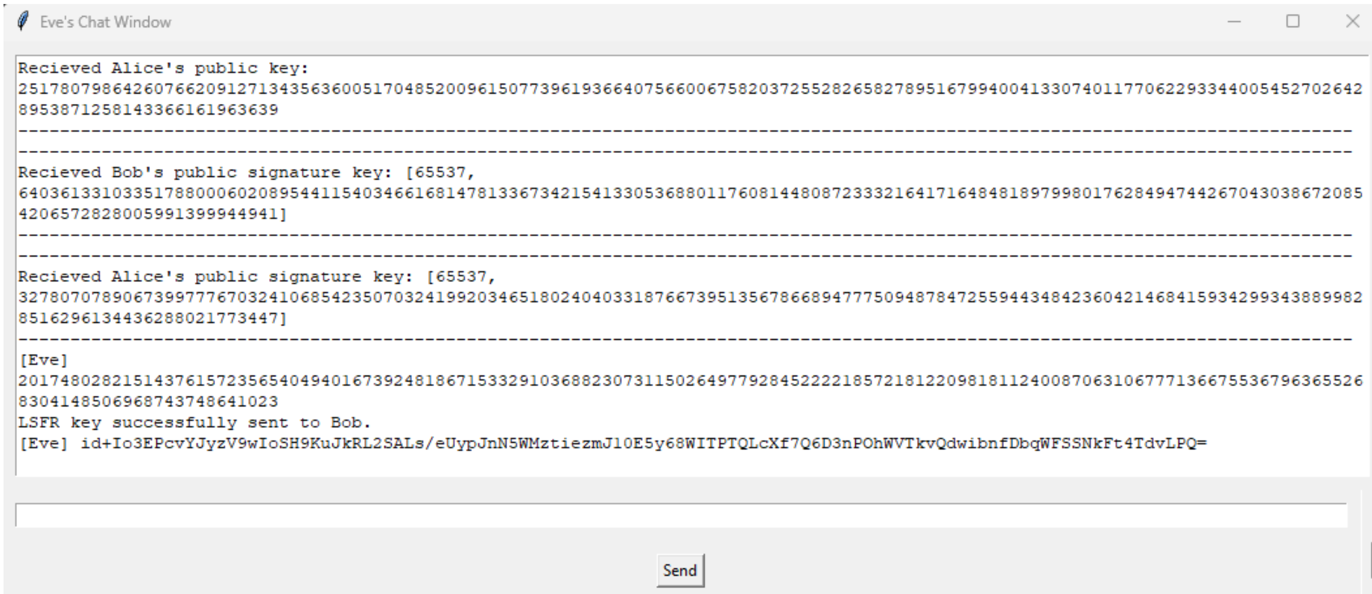
Alice and Eve both receive an encrypted message. In contrast, Bob not only receives the identical message but also sees that it was sent to him by Alice. Moreover, Bob successfully decodes the message, revealing the content as "Do you think that the CBC version would be more secure that DES?"

Test 3: Within the RSA Server and the Encryption Type is set to "DES", Eve first sends her LFSR Key to Bob. She then sends the Message "What would happen if I sent you a really, really, loooooonnggggg messsage?" to Bob.

(./sendLFSRKey Bob)

(./sendToUser Bob What would happen if I sent you a really, really, loooooonnggggg messsage?)


Bob's Chat Window
879172257665940677647524143100601146444864392885533916657964079336919250826954126363980198191663800942910821539335662606518285810
4401103248469990287104041
Recieved Eve's public signature key: [65537,
356764700674962459389079300619134305482642625962235521935618704566010466792904110436858908911283801475724049592063983749390847359
1061139802967473941088491]
[Eve]
201748028215143761572356540494016739248186715332910368823073115026497792845222218572181220981811240087063106777136675536796365526
8304148506968743748641023
Received from Eve: 10100000011000010010011111110111111000111010101001010111101001100
[Eve] id+Io3EPcvYJyzV9wIoSH9KuJkRL2SALs/eUypJnN5WMztiezmJ10E5y68WITPTQLcXf7Q6D3nPOhWVTkvQdwibnfDbqWFSSNkFt4TdvLPQ=
The encrypted message was sent for you by Eve.
Decrypted Message using associated Key: What would happen if I sent you a really, really, loooooon-
nggggg messsage?
This message has been verified from its Digital Signature!
Send

Alice's Chat Window
Recieved Bob's public signature key: [65537,
6403613310335178800060208954411540346616814781336734215413305368801176081448087233321641716484818979980176284947442670430386720854
20657282800591399944941]
[Eve] Eve has joined the chat.
Recieved Eve's public key:
8791722576659406776475241431006011464448643928855339166579640793369192508269541263639801981916638009429108215393356626065182858104
401103248469990287104041
Recieved Eve's public signature key: [65537,
3567647006749624593890793006191343054826426259622355219356187045660104667929041104368589089112838014757240495920639837493908473591
061139802967473941088491]
[Eve]
2017480282151437615723565404940167392481867153329103688230731150264977928452222185721812209818112400870631067771366755367963655268
304148506968743748641023
[Eve] id+Io3EPcvYJyzV9wIoSH9KuJkRL2SALs/eUypJnN5WMztiezmJ10E5y68WITPTQLcXf7Q6D3nPOhWVTkvQdwibnfDbqWFSSNkFt4TdvLPQ=
Send

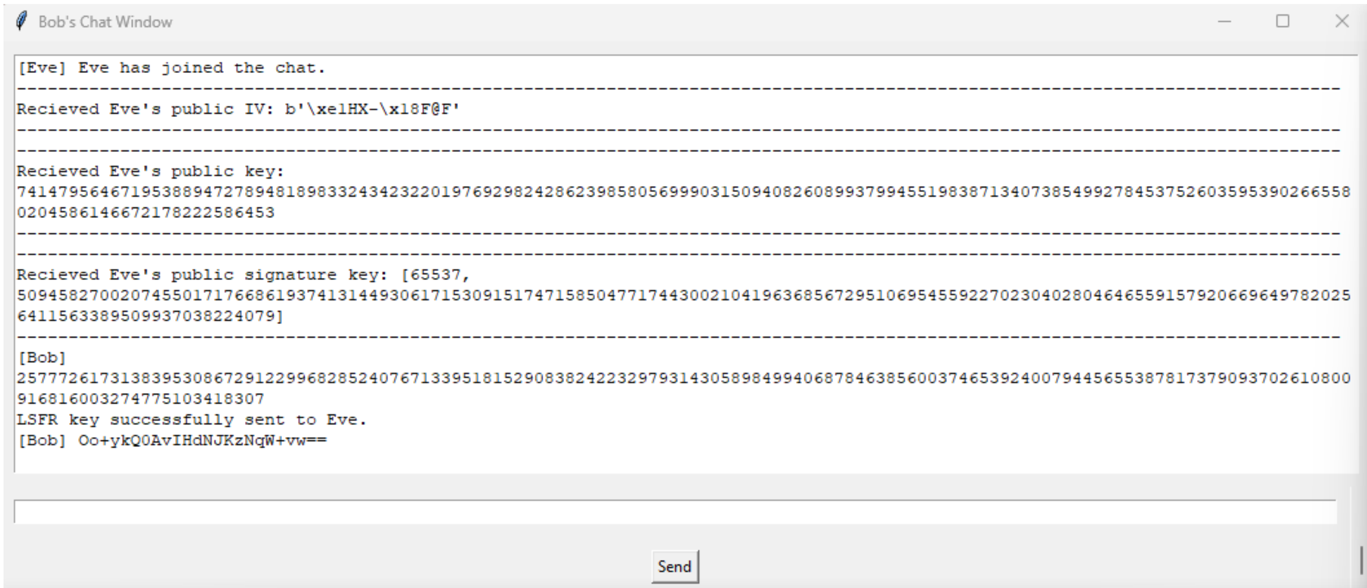


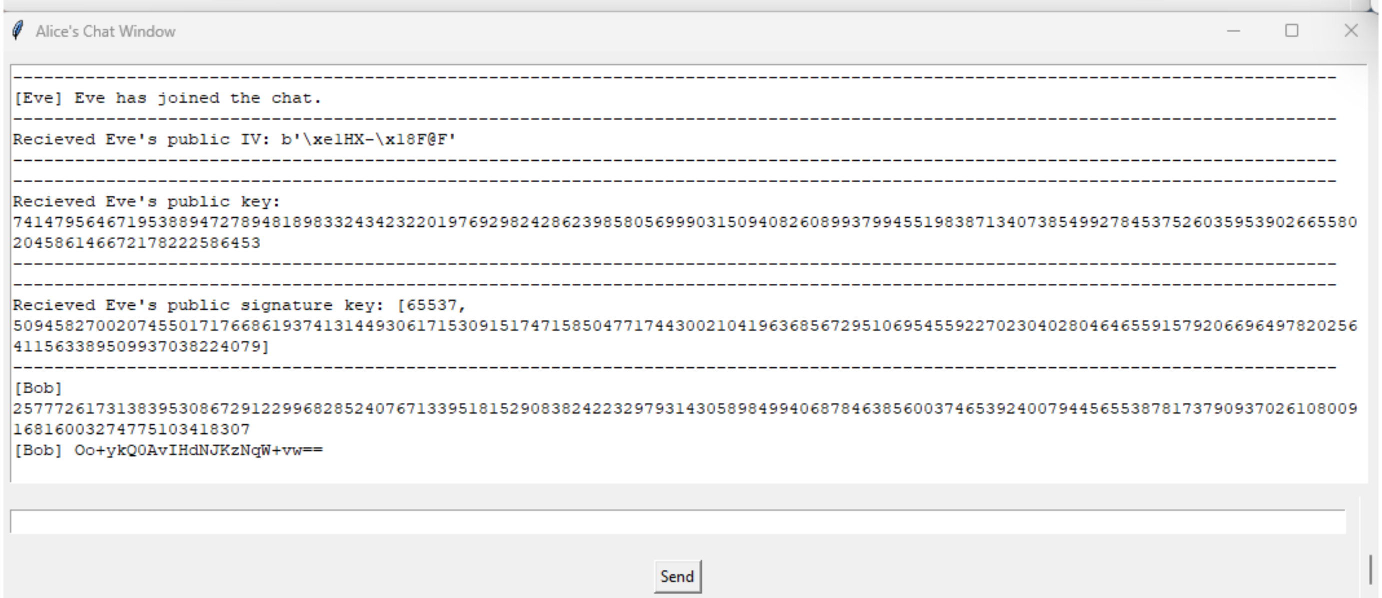
Alice and Eve both receive an encrypted message. In contrast, Bob not only receives the identical message but also sees that it was sent to him by Eve. Moreover, Bob successfully decodes the message, revealing the content as "What would happen if I sent you a really, really, loooooonnggggg messsage?"

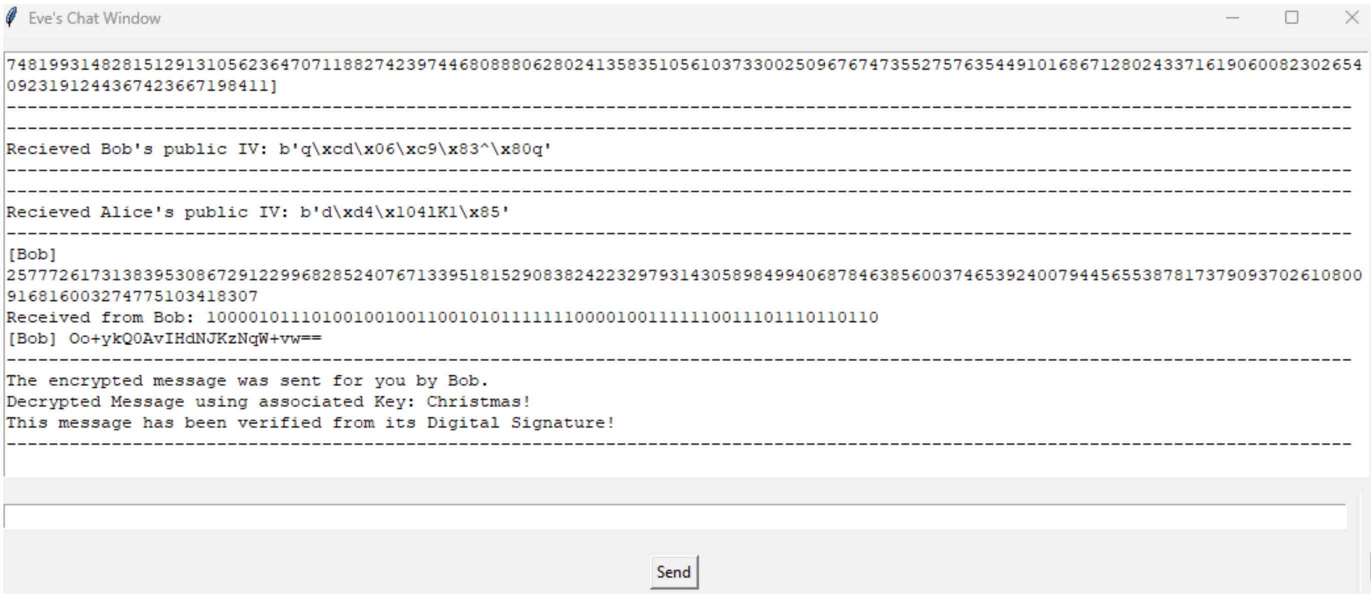
Test 4: Within the RSA Server and the Encryption Type is set to "DES_CBC", Bob first sends his LFSR Key to Eve. He then sends the Message "Christmas!" to Eve.

(./sendLFSRKey Eve)

(./sendToUser Bob Christmas!)


Bob's Chat Window
[Eve] Eve has joined the chat.
Recieved Eve's public IV: b'\xe1HX-\x18F@F'
Recieved Eve's public key:
7414795646719538894727894818983324342322019769298242862398580569990315094082608993799455198387134073854992784537526035953902665580204586146672178222586453
Recieved Eve's public signature key: [65537,
50945827002074550171766861937413144930617153091517471585047717443002104196368567295106954559227023040280464655915792066964978202564115633895099370382240 79]
[Bob]
25777261731383953086729122996828524076713395181529083824223297931430589849940687846385600374653924007944565538781737909370261080091681600327477 5103418307
LSFR key successfully sent to Eve.
[Bob] Oo+ykQ0AvIHdNJKzNqW+vw==
Send


Alice's Chat Window
[Eve] Eve has joined the chat.
Recieved Eve's public IV: b'\xe1HX-\x18F@F'
Recieved Eve's public key:
7414795646719538894727894818983324342322019769298242862398580569990315094082608993799455198387134073854992784537526035953902665580204586146672178222586453
Recieved Eve's public signature key: [65537,
50945827002074550171766861937413144930617153091517471585047717443002104196368567295106954559227023040280464655915792066964978202564115633895099370382240 79]
[Bob]
25777261731383953086729122996828524076713395181529083824223297931430589849940687846385600374653924007944565538781737909370261080091681600327477 5103418307
[Bob] Oo+ykQ0AvIHdNJKzNqW+vw==
Send



Bob and Alice both receive an encrypted message. In contrast, Eve not only receives the identical message but also sees that it was sent to her by Bob. Moreover, Eve successfully decodes the message, revealing the content as "Christmas!" Since the DES encryption/decryption algorithm is working as intended for all four tests, we can confirm that it is working as appropriately.

4.4 AES VS DES

In conducting a comparative analysis between the Advanced Encryption Standard (AES) and the Data Encryption Standard (DES), I implemented Python code to evaluate the encryption and decryption performance of both algorithms. The testing environment utilized the Crypto.Random module for generating random cryptographic keys and initialization vectors (IVs), ensuring a robust assessment. The experiments involved random string plaintexts of varying lengths, ranging from 1 to 5000 characters in increments of 50.

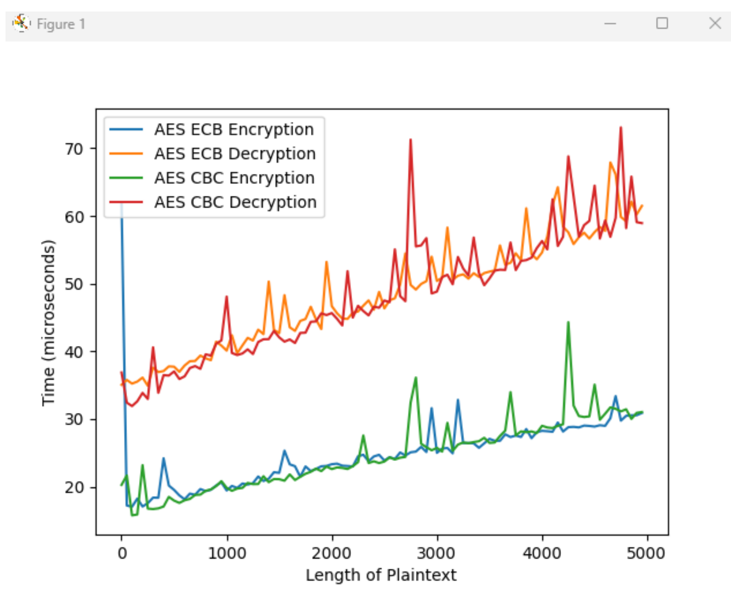
The code, executed within a controlled environment, employed the timeit module to measure the execution times of encryption and decryption functions. For AES, a 128-bit key and a 16-byte IV were used, supporting both Electronic Codebook (ECB) and Cipher Block Chaining (CBC) modes. Similarly, the DES implementation utilized a 64-bit key and an 8-byte IV for CBC mode, accommodating both ECB and CBC modes. The results, presented in microseconds, were plotted using matplotlib to visualize the efficiency and scalability of AES and DES under varying workloads.

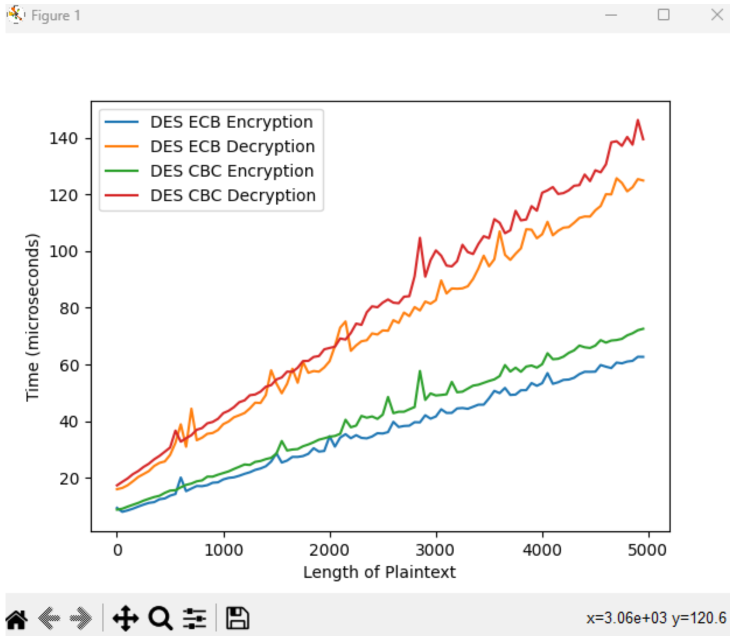
```
import timeit
import matplotlib.pyplot as plt
import random
import string
from Crypto.Random import get_random_bytes
from AES import encrypt as aes_encrypt, decrypt as aes_decrypt
from DES import encrypt as des_encrypt, decrypt as des_decrypt
from cryptography.hazmat.primitives.ciphers import modes
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

def generate_random_string(length):
    return ''.join(random.choice(string.ascii_letters) for _
        in range(length))

def test_aes():
    key = get_random_bytes(16)  # 128-bit key for AES
    iv = get_random_bytes(16)
    modes_to_test = [modes.ECB(), modes.CBC(iv)]  # Add more
        modes if needed

    for mode in modes_to_test:
        times_encrypt = []
        times_decrypt = []
        lengths = list(range(1, 5001, 50))  # Vary the length
            of the plaintext

        for length in lengths:
            plaintext = generate_random_string(length)

            encrypt_time = timeit.timeit(lambda:
                aes_encrypt(plaintext, key, mode),
                number=1000)  # Increased number of iterations
            times_encrypt.append(encrypt_time * 1e6 / 1000)
                # Convert to microseconds, average time per
                encryption

            decrypt_time = timeit.timeit(lambda:
                aes_decrypt(aes_encrypt(plaintext, key,
                mode), key, mode).decode("utf-8"),
                number=1000)  # Increased number of iterations
            times_decrypt.append(decrypt_time * 1e6 / 1000)
                # Convert to microseconds, average time per
                decryption

            assert aes_decrypt(aes_encrypt(plaintext, key,
                mode), key, mode).decode("utf-8") ==
```

```
                plaintext  # Ensure decryption is correct

        # Plot results
        plt.plot(lengths, times_encrypt, label=f'AES
            {mode.name} Encryption')
        plt.plot(lengths, times_decrypt, label=f'AES
            {mode.name} Decryption')

    plt.xlabel('Length of Plaintext')
    plt.ylabel('Time (microseconds)')
    plt.legend()
    plt.show()

def test_des():
    key = get_random_bytes(8)  # 64-bit key for DES
    iv = get_random_bytes(8)
    modes_to_test = [DES.MODE_ECB, DES.MODE_CBC]  # Add more
        modes if needed
    for mode in modes_to_test:
        times_encrypt = []
        times_decrypt = []
        lengths = list(range(1, 5001, 50))  # Vary the length
            of the plaintext

        for length in lengths:
            plaintext = generate_random_string(length)

            if mode == DES.MODE_CBC:
                encrypt_time = timeit.timeit(lambda:
                    des_encrypt(plaintext, key, mode, iv),
                    number=1000)
                decrypt_time = timeit.timeit(lambda:
                    des_decrypt(des_encrypt(plaintext, key,
                    mode, iv), key, mode, iv), number=1000)
            else:
                encrypt_time = timeit.timeit(lambda:
                    des_encrypt(plaintext, key, mode),
                    number=1000)
                decrypt_time = timeit.timeit(lambda:
                    des_decrypt(des_encrypt(plaintext, key,
                    mode), key, mode), number=1000)

            times_encrypt.append(encrypt_time * 1e6 / 1000)
               # Convert to microseconds, average time per
               encryption
            times_decrypt.append(decrypt_time * 1e6 / 1000)
               # Convert to microseconds, average time per
               decryption
```

```
            assert des_decrypt(des_encrypt(plaintext, key,
                mode, iv), key, mode, iv) == plaintext #
                Ensure decryption is correct

        # Plot results
        if(mode == DES.MODE_ECB):
            title = "ECB"
        else:
            title = "CBC"
        plt.plot(lengths, times_encrypt, label=f'DES {title}
            Encryption')
        plt.plot(lengths, times_decrypt, label=f'DES {title}
            Decryption')

    plt.xlabel('Length of Plaintext')
    plt.ylabel('Time (microseconds)')
    plt.legend()
    plt.show()

if __name__ == "__main__":
    test_aes()
    test_des()
```

After executing this code, we get the following results:





In our comparative analysis of the AES and DES cryptographic algorithms, our Python implementation provided empirical insights into their respective performances across a range of plaintext lengths. The AES algorithm demonstrated consistent execution times for encryption and decryption, typically falling within the 20 to 70 microseconds range. This stability suggests that AES is a reliable choice for various encryption scenarios, striking a balance between security and computational efficiency.

Conversely, the DES algorithm exhibited slightly higher execution times, ranging from 20 to 140 microseconds. Despite this, the efficiency displayed by DES, particularly when handling smaller data sizes, highlights its continued relevance for specific applications. The gradual increase in execution times with larger datasets aligns with expected behavior and underscores the algorithm's adaptability to varying workloads.

In summary, both AES and DES present trade-offs in terms of computational efficiency and security, and the choice between them should be informed by specific application requirements and considerations. The results highlight the nuanced strengths of each algorithm, offering a pragmatic perspective for users seeking to make an informed decision based on their unique encryption needs.

5 Digital Signature

Given that clients are already equipped with RSA functions, implementing the RSA algorithm for digital signatures becomes a straightforward and logical choice.

5.1 Implementation

To facilitate message signing, user-clients not only possess the RSA functions discussed in previous sections but also incorporate the following functions:

```
def hash_message(message):
    # Hash the message using SHA-256
    sha256 = hashlib.sha256()
    sha256.update(str(message).encode('utf-8'))
    return int(sha256.hexdigest(), 16)

def RSA_sign(message, private_key):
    hashed_message = hash_message(message)
    signature = exponentiation(hashed_message,
        private_key[0], private_key[1])
    return signature

def RSA_verify(message, signature, public_key):
    hashed_message = hash_message(message)
    decrypted_signature = exponentiation(signature,
        public_key[0], public_key[1])

    if hashed_message == decrypted_signature:
        return True
    else:
        return False
```

The implementation consists of several functions:

The hash_message function employs the SHA-256 hash algorithm to create a fixed-size representation of the input message. This hashing step is essential for maintaining constant-sized signatures and enhancing security.

The RSA_sign function generates an RSA digital signature for a given message using the private key. It begins by hashing the message and then performs modular exponentiation with the private key components to produce the signature.

The RSA_verify function validates the authenticity and integrity of a message by comparing the digital signature against the corresponding public key. It hashes the message, performs modular exponentiation with the public key components, and checks if the result matches the hashed message.

With these established methods in place, the next step is to delve into the integration of digital signatures by user clients.

5.2 Signature Procedure

In the provided examples within other sections, you might have observed references to Public Signature Keys under users and the verification of messages after decryption. These elements collectively form the Digital Signature Process.

Upon a user's initial entry into the server, they not only generate the components mentioned earlier but also create RSA public and private signature keys within their chat GUI:

```
self.publicSignKey, self.privateSignKey =
    RSA_key_generate()
```

Subsequently, the user shares their publicSignKey with the server alongside the other components discussed in previous subsections. The server, in turn, broadcast the user's public signature key to all other users for storage. Simultaneously, the server provides the user with the public signature keys of all other users, which the user retains in their storage.

In the encryption process outlined earlier, the user client employs the previously mentioned RSA_Sign function. This involves utilizing the original message and the user's private signature key, producing a final message formatted for transmission:

```
final_message = f'./sendToUser {target_username}
    {ciphertext} {RSA_sign(private_message,
    self.privateSignKey)}'
```

Upon sending this message, the targeted user receives and decrypts it. In the decryption phase, the recipient retrieves the public signature key associated with the sender. Subsequently, they extract both the signature attached to the message and the decrypted message itself, passing this information through the RSA_verify function:

```
if RSA_verify(DecryptedMessage, signature,
    self.public_signature_keys[target_username]):
     self.display_message(f'This message has been
         verified from its Digital Signature!')
else:
     self.display_message(f'This message is not
         verified from its Digital Signature!')
```

The verification process involves confirming whether the fetched decrypted message, signature, and the sender's public key match the decrypted message performed by the recipient. A successful verification indicates that the message is authentic and indeed sent by the claimed sender. Conversely, a failure in verification signals that the message cannot be trusted.

You can see this process happening within the earlier encryption examples.

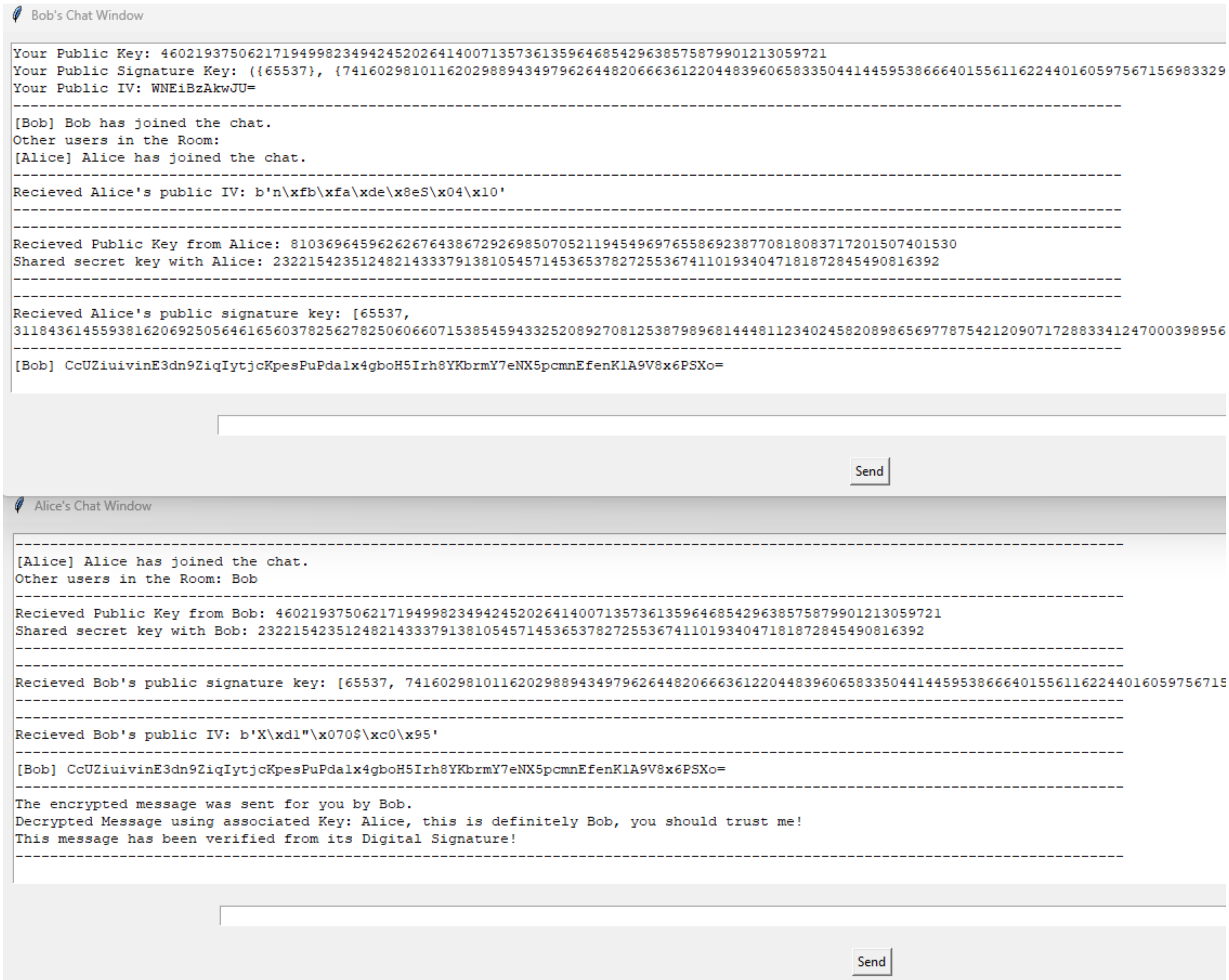
5.3 Altering The Signing

Now, consider a scenario within the chat server where Bob transmits a message to Alice, affirming his identity with the declaration, "Alice, this is definitely Bob, you should trust me!".

In his client application, the system would encrypt the message in the following manner.

```
# Encrypt the private message based on encryption type
ciphertext =
    self.encrypt_private_message(private_message)

# Create the final message with encryption and
    digital signature
final_message = f'./sendToUser {target_username}
    {ciphertext} {RSA_sign(private_message,
    self.privateSignKey)}'
```

When this message is sent out, we get the following windows:

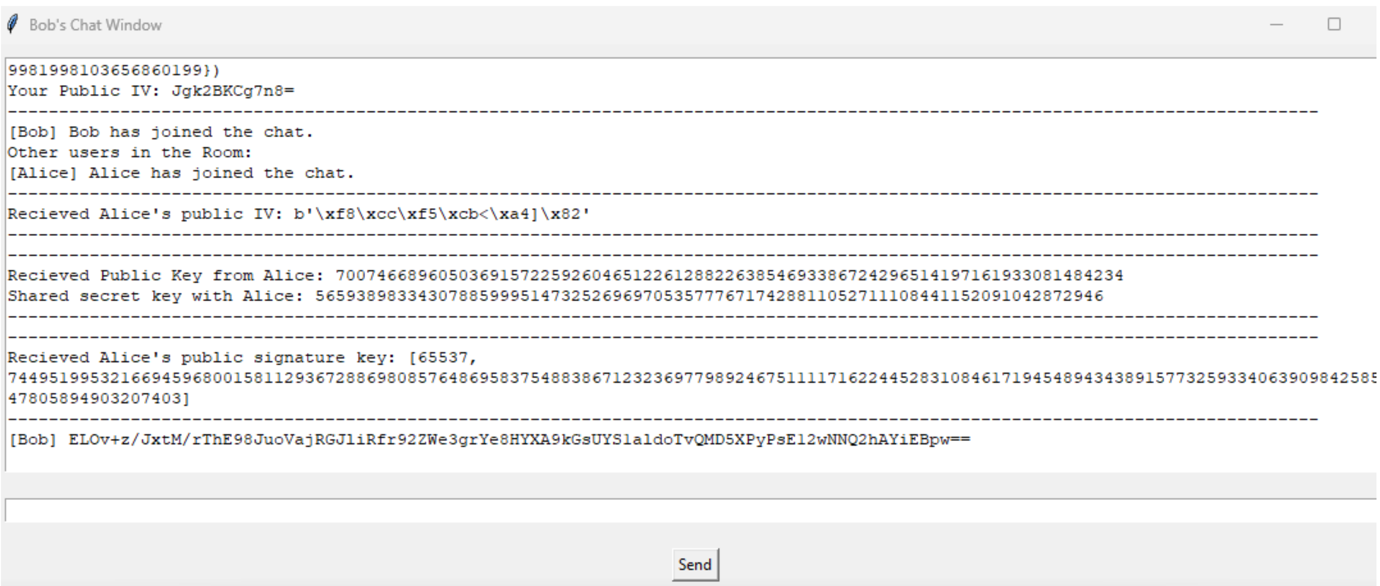


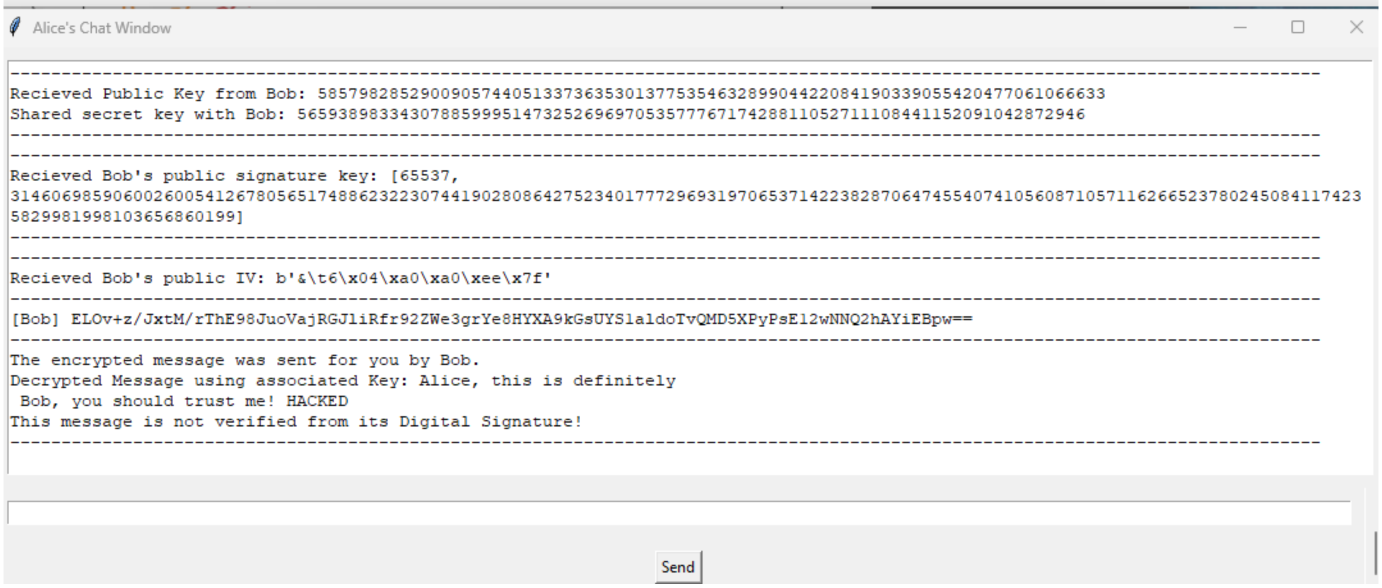
As expected, the message was able to be verified to be from Bob. However, lets say we alter the encryption process in the following manner:

```
# Encrypt the message based on encryption type
ciphertext = self.encrypt_message(message_content +"
   HACKED", shared_secret_key, target_username)

# Create the final message with encryption and
```

```
    digital signature
final_message = f'./sendToUser {target_username}
    {ciphertext} {RSA_sign(message_content,
    self.privateSignKey)}'
```

The initially intended message for encryption has undergone modifications, but the RSA signature remains linked to the original, unaltered message. Upon sending this modified message to the server, the following windows will appear:





The message has not passed verification, leaving Alice uncertain about its authenticity and raising doubts about whether the message truly originates from Bob.

5.4 The importance of Hashing

The hashing process plays a pivotal role in digital signatures, significantly contributing to the security and reliability of the entire signature mechanism. In the context of digital signatures, a hash function serves as a critical component to ensure data integrity and resistance against tampering.

When creating a digital signature, the original message is first subjected to a hash function, such as SHA-256, which produces a fixed-size hash value. This hash value acts as a unique representation, often referred to as the message digest, that is considerably smaller than the original message. The importance of this step lies in its ability to condense variable-length messages into a fixed-size format, irrespective of the message's length or complexity.

By using a hash function, the digital signature system achieves two essential objectives. Firstly, it streamlines the process of signing large messages by creating a compact representation that can be efficiently processed. Secondly, and perhaps more importantly, it provides a means of verification that ensures the integrity of the original message. Even the slightest alteration in the message content results in a vastly different hash value.

The security implications are profound. If an attacker attempts to modify the message without knowledge of the private key, the hash value will change accordingly. Upon verification, the recipient can easily detect any discrepancies between the recalculated hash value and the one included in the digital signature. This cryptographic property makes it computationally infeasible for an adversary to generate a valid signature for a modified message.

In summary, the hashing process in digital signatures serves as a cornerstone for data integrity and tamper detection. By condensing messages into fixed-size hash values, it facilitates efficient signature generation and, more importantly, provides a robust mechanism for verifying the originality and unaltered nature of the transmitted data.

6 Coding Files

Project Link For Easier File Access: https://github.com/AbidAzad/Information-and-Network-Security-Final-Term-Project

6.1 Diffie-Hellman Key Exchange Server (DHServer.py)

```
import socket
import threading
import random
import time
from base64 import b64encode, b64decode

# Server configuration
HOST = '192.168.1.31'
PORT = 55555

# Lists to store connected clients, their usernames
clients = []
usernames = []

# Create a socket for the server
server = socket.socket(socket.AF_INET,
   socket.SOCK_STREAM)
server.bind((HOST, PORT))

public_keys_dict = {}
public_signature_keys_dict = {}
RECIVINGIVs = True

if(RECIVINGIVs):
    public_ivs_dict = {}


def broadcast(message, sender, publicKey=False):
    """Send a message to all clients except the
       sender."""
    for client in clients:
        try:
            if(not publicKey):
                client.send(f"[{usernames[clients.index(sender)]}]
                    ".encode('utf-8') + message)
            else:
                client.send(message)
        except:
            # Remove the client if unable to send a
               message
            handle_disconnect(client)


def handle_disconnect(client):
    """Handle client disconnection."""
    index = clients.index(client)
```

```
    client.close()
    username = usernames[index]
    broadcast(f"{username} has left the
        chat.".encode('utf-8'), client)  # Call
        broadcast directly here
    usernames.remove(username)
    clients.remove(client)

def broadcast_public_keys(client):
    """Send public keys to the newly joined client."""
    for username, public_key in
        public_keys_dict.items():
        key_message = f"Public key of {username}:
            {public_key}"
        time.sleep(0.2)
        client.send(key_message.encode('utf-8'))

def broadcast_public_signature_keys(client):
    """Send public keys to the newly joined client."""
    for username, public_signature_key in
        public_signature_keys_dict.items():
        key_message = f"Signature key of {username}:
            ({public_signature_key[0],public_signature_key[1]})"
        time.sleep(0.2)
        client.send(key_message.encode('utf-8'))

def broadcast_public_IVs(client):
    """Send public keys to the newly joined client."""
    for username, public_iv in
        public_ivs_dict.items():
        key_message = f"Public IV of {username}:
            {b64encode(public_iv).decode()}"
        time.sleep(0.2)
        client.send(key_message.encode('utf-8'))

def handle_client(client, username):
    """Handle individual client connections."""
    try:
        # Broadcast the new user joining the chat
        broadcast(f"{username} has joined the
            chat.".encode('utf-8'), client)
        current_users = ', '.join(user for user in
            usernames if user != username)
        client.send(f"Other users in the Room:
            {current_users}".encode('utf-8'))
        time.sleep(0.1)
```

```
broadcast_public_keys(client)
time.sleep(0.1)
broadcast_public_signature_keys(client)
if(RECIVINGIVs):
    time.sleep(0.1)
    broadcast_public_IVs(client)

# Broadcast the public key of the new user to
   all clients
public_key_message =
   client.recv(1024).decode('utf-8')
if public_key_message.startswith("Public key:
   "):
    parts = public_key_message.split(" Public
       Signature Key: ")
    public_key = int(parts[0][12:])
    signature_key_str = parts[1]

    # Extracting signature key
    if "Public IV: " in signature_key_str:
        # Both signature key and IV are
           present
        signature_key_str, public_iv_str =
           signature_key_str.split(" Public
           IV: ")
        public_iv = b64decode(public_iv_str)
        public_ivs_dict[username] = public_iv
        broadcast(f"Public IV of {username}:
           {b64encode(public_iv).decode()}".encode('utf-8'),
           client, True)

    # Extracting signature key from the new
       format
    signature_key_str =
       signature_key_str.replace("(",
       "").replace(")", "")
    signature_key = list(map(int,
       signature_key_str.replace("(",
       "").replace(")", "").split(',')))

    public_keys_dict[username] = public_key
    public_signature_keys_dict[username] =
       signature_key

    # Broadcasting public key and signature
       key
```

```
        broadcast(f"Public key of {username}:
            {public_key}".encode('utf-8'), client,
            True)
        time.sleep(0.1)
        broadcast(f"Signature key of {username}:
            ({signature_key[0]},{signature_key[1]})".encode('utf-8'),
            client, True)



    else:
        print("Invalid public key format.")
        handle_disconnect(client)

    while True:
        message =
            client.recv(1024).decode('utf-8')

        # Announce all current users to the new
            client

        # Check if the message is a special
            command to send a private message
        if message.startswith('./sendToUser'):
            # Extract the target username and the
                private message from the command
            parts = message.split(' ', 3)
            if len(parts) == 4:
                target_username = parts[1]
                private_message = parts[2]
                signature = parts[3]
                # Find the target client based on
                    the username
                target_client = next((c for c, u
                    in zip(clients, usernames) if
                    u == target_username), None)

                # Send the private message to the
                    target user
                if target_client:
                    # Remove the command and
                        username, leaving only the
                        private message
                    cleaned_message =
                        private_message
```

```
                    broadcast(private_message.encode('utf-8'),
                        client)
                    target_client.send(f"./decrypt
                        {username}
                        {cleaned_message}
                        {signature}".encode('utf-8'))
                else:
                    print(f"User not found.")
            else:
                print("Invalid private message
                    format.")

        else:
            # Broadcast the message to all clients
            broadcast(message.encode('utf-8'),
                client)

    except (socket.error, ConnectionResetError):
        # Handle client disconnection
        handle_disconnect(client)



def start_server():
    """Start the chat server."""
    server.listen()
    print(f"Server is listening on {HOST}:{PORT}")

    while True:
        # Accept a new client connection
        client, address = server.accept()
        print(f"New connection from {address}")

        # Receive the username from the client
        username = client.recv(1024).decode('utf-8')

        # Add the new client and username to the lists
        clients.append(client)
        usernames.append(username)

        # Start a new thread to handle the client
        thread =
            threading.Thread(target=handle_client,
            args=(client, username))
        thread.start()
```

```
if __name__ == "__main__":
    start_server()
```

6.2 Diffie-Hellman Key Exchange Client (DHClient.py)

```
import socket
import threading
import random
from tkinter import Tk, Scrollbar, Listbox, Entry,
   Button, StringVar, DISABLED, NORMAL, Toplevel,
   Label, WORD, Text, END
from sympy import isprime
from gmpy2 import powmod
import math
import os
import hashlib

ENCRYPTIONTYPE = 'DES_CBC'

if(ENCRYPTIONTYPE == 'STREAMCIPHER'):
    from streamCipher import *
elif(ENCRYPTIONTYPE == 'AES_ECB' or ENCRYPTIONTYPE ==
   'AES_CBC'):
    from AES import *
elif(ENCRYPTIONTYPE == 'DES_ECB' or ENCRYPTIONTYPE ==
   'DES_CBC'):
    from DES import *

# Client configuration
HOST = '192.168.1.31'
PORT = 55555

# Agreed Upon Values
primeNumber =
   1021886172171788044763879771601293344317459450097300655193370949921296772
primitiveRoot = 2

'''Helper function that generates a large prime
   number with the specified number of bits, in which
   for this assignment is 512.'''
def generate_large_prime(bits):
    while True:
        num = random.randrange(0, 2**bits - 1)
        if isprime(num):
```

```
            return num
def exponentiation(message, power, basis):
    return powmod(message, power, basis)
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        g, x, y = extended_gcd(b % a, a)
        return g, y - (b // a) * x, x
'''Finds the modular inverse of a given number a
   modulo n using the extended euclidean algorithm.'''
def inverse_finder(a, n):
    g, x, _ = extended_gcd(a, n)
    if g != 1:
        raise ValueError(f"The modular inverse does
            not exist for {a} modulo {n}")
    else:
        return x % n
'''A function that generates RSA public and private
   keys. It takes an optional parameter e for the
   rsaKeyInput; if not provided, it defaults to 3.'''
def RSA_key_generate():
    e = 65537
    while(True):
        p = generate_large_prime(256)
        q = p

        while(p == q):
            q = generate_large_prime(256)
        n = p * q
        euler = (p-1) * (q-1)

        if(math.gcd(euler, e) == 1):
            break
    d = inverse_finder(e, euler)
    publicKey = [e, n]
    privateKey = [d, n]
    return publicKey, privateKey
def hash_message(message):
    # Hash the message using SHA-256
    sha256 = hashlib.sha256()
    sha256.update(str(message).encode('utf-8'))
    return int(sha256.hexdigest(), 16)

def RSA_sign(message, private_key):
    hashed_message = hash_message(message)
```

```
    signature = exponentiation(hashed_message,
       private_key[0], private_key[1])
    return signature

def RSA_verify(message, signature, public_key):
    hashed_message = hash_message(message)
    decrypted_signature = exponentiation(signature,
       public_key[0], public_key[1])

    if hashed_message == decrypted_signature:
        return True
    else:
        return False
# Create a socket for the client
client = socket.socket(socket.AF_INET,
   socket.SOCK_STREAM)
client.connect((HOST, PORT))


class UsernameGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Enter Username")

        # Label and entry widget for entering the
           username
        self.username_label = Label(root, text="Enter
           your username:")
        self.username_label.pack(pady=10)
        self.username_entry = Entry(root, width=30)
        self.username_entry.pack(pady=10)

        # Button to submit the username
        submit_button = Button(root, text="Submit",
           command=self.submit_username)
        submit_button.pack(pady=10)


    def submit_username(self):
        username = self.username_entry.get()
        if username:
            self.root.destroy()  # Close the username
               entry GUI
            # Start the chat GUI with the entered
               username
            chat_root = Tk()
```

```
            chat_gui = ChatGUI(chat_root, username)
            chat_root.mainloop()
        else:
            # Display an error message if the
               username is empty
            error_label = Label(self.root,
               text="Please enter a valid username.")
            error_label.pack(pady=5)


class ChatGUI:
    def __init__(self, root, username):
        self.root = root
        self.root.title(f"{username}'s Chat Window")

        # Create a listbox to display messages
        self.message_text = Text(root, height=20,
           width=200, selectbackground="white",
           exportselection=False, wrap=WORD)
        self.message_text.pack(padx=10, pady=10)
        self.username = username

        # Create a scrollbar for the listbox
        scrollbar = Scrollbar(root)
        scrollbar.pack(side="right", fill="y")

        # Attach the listbox to the scrollbar
        self.message_text.config(yscrollcommand=scrollbar.set)
        scrollbar.config(command=self.message_text.yview)

        # Create an entry widget for typing messages
        self.message_entry = Entry(root, width=200)
        self.message_entry.pack(padx=10, pady=10)

        # Create a Send button to send messages
        send_button = Button(root, text="Send",
           command=self.send_message)
        send_button.pack(pady=10)

        # Generate a secret integer for the client
        self.secret_integer = random.getrandbits(64)

        self.public_keys = {}
        self.public_signature_keys = {}

        self.publicSignKey, self.privateSignKey =
```

```
   RSA_key_generate()

if(ENCRYPTIONTYPE == "AES_CBC"):
    self.public_IVs = {}
    self.IV = os.urandom(16)
elif(ENCRYPTIONTYPE == "DES_CBC"):
    self.public_IVs = {}
    self.IV = os.urandom(8)


# Start a thread to receive messages
receive_thread =
   threading.Thread(target=self.receive_messages)
receive_thread.start()

# Send the username to the server
client.send(username.encode('utf-8'))
if(ENCRYPTIONTYPE == "AES_CBC" or
   ENCRYPTIONTYPE == "DES_CBC"):
    client.send(f"Public key:
       {pow(primitiveRoot,
       self.secret_integer, primeNumber)}
       Public Signature Key:
       ({self.publicSignKey[0]},{self.publicSignKey[1]})
       Public IV:
       {b64encode(self.IV).decode()}".encode('utf-8'))
    self.display_message("-----------------------------------------
    self.display_message(f'Your secret
       integer:{self.secret_integer}')
    self.display_message(f"Your Public Key:
       {pow(primitiveRoot,
       self.secret_integer, primeNumber)}")
    self.display_message(f"Your Public
       Signature Key:
       {({self.publicSignKey[0]},{self.publicSignKey[1]})}")
    self.display_message(f"Your Public IV:
       {b64encode(self.IV).decode()}")
    self.display_message("-----------------------------------------
else:
    client.send(f"Public key:
       {pow(primitiveRoot,
       self.secret_integer, primeNumber)}
       Public Signature Key:
       ({self.publicSignKey[0]},{self.publicSignKey[1]})".encode('ut
    self.display_message("-----------------------------------------
    self.display_message(f'Your secret
```

```
            integer:{self.secret_integer}')
        self.display_message(f"Your Public Key:
            {pow(primitiveRoot,
            self.secret_integer, primeNumber)}")
        self.display_message(f"Your Public
            Signature Key:
            {({self.publicSignKey[0]},{self.publicSignKey[1]})}")
        self.display_message("-----------------------------------------

    def send_message(self):
        message = self.message_entry.get()
        self.message_entry.delete(0, 'end')

        # Check if the message is a special command
            to send a private message
        if message.startswith('./checkSharedKey'):
            self.check_shared_key(message)
        elif message.startswith('./sendToUser'):
            self.send_private_message(message)
        else:
            client.send(message.encode('utf-8'))

    def receive_messages(self):
        """Receive and display messages from the
            server."""
        while True:
            try:
                # Receive message from the server
                message =
                    client.recv(1024).decode('utf-8')

                # Handle different types of messages
                if message.startswith("Public key of
                    "):
                     self.handle_public_key(message)
                elif message.startswith("Public IV of
                    "):
                     self.handle_public_IV(message)
                elif message.startswith("Signature
                    key of "):
                     self.handle_public_signature(message)
                elif message.startswith("./decrypt"):
                     # Split the message into parts
                     parts = message.split(' ', 3)
                     target_username = parts[1]
                     private_message = parts[2]
```

```
    signature = int(parts[3])

    # Display information about the
       encrypted message
    self.display_message("------------------------------------
    display_message = f'The encrypted
       message was sent for you by
       {target_username}.'
    self.display_message(display_message)

    # Compute shared secret key
    shared_secret_key =
       pow(self.public_keys[target_username],
       self.secret_integer,
       primeNumber)

    # Decrypt the message based on
       encryption type
    DecryptedMessage =
       self.decrypt_message(private_message,
       shared_secret_key,
       target_username)

    # Display decrypted message
    display_message = f'Decrypted
       Message using associated Key:
       {DecryptedMessage}'
    self.display_message(display_message)

    # Verify the digital signature
    if
       self.verify_signature(DecryptedMessage,
       signature,
       self.public_signature_keys[target_username]):
        self.display_message('This
           message has been verified
           from its Digital
           Signature!')
    else:
        self.display_message('This
           message is not verified
           from its Digital
           Signature!')
    self.display_message("------------------------------------
else:
    # Handle regular messages
```

```
                self.display_message(message)

        except Exception as e:
            # Handle exceptions
            print(f"An error occurred while
                receiving messages: {e}")
            client.close()
            break

def display_message(self, message):
    """Display a message in the message
        listbox."""
    self.message_text.insert(END, message + '\n')

def decrypt_message(self, private_message,
    shared_secret_key, target_username):
    """Decrypt the message based on the
        encryption type."""
    if ENCRYPTIONTYPE == "STREAMCIPHER":
        return decrypt(private_message,
            str(bin(shared_secret_key)[2:]))
    elif ENCRYPTIONTYPE == "AES_ECB" or
        ENCRYPTIONTYPE == "AES_CBC":
        key_bytes =
            shared_secret_key.to_bytes((shared_secret_key.bit_length()
            + 7) // 8, 'little')
        return decrypt(private_message,
            key_bytes, modes.ECB() if
            ENCRYPTIONTYPE == "AES_ECB" else
            modes.CBC(self.public_IVs[target_username])).decode('utf-8')
    elif ENCRYPTIONTYPE == "DES_ECB" or
        ENCRYPTIONTYPE == "DES_CBC":
        key_bytes =
            shared_secret_key.to_bytes((shared_secret_key.bit_length()
            + 7) // 8, byteorder='big')
        key_bytes =
            hashlib.sha256(key_bytes).digest()[:8]
        if ENCRYPTIONTYPE == "DES_ECB":
            return decrypt(private_message,
                key_bytes, DES.MODE_ECB)
        else:
            return decrypt(private_message,
                key_bytes, DES.MODE_CBC,
                self.public_IVs[target_username])

def verify_signature(self, decrypted_message,
```

```
    signature, public_signature_key):
     """Verify the digital signature of the
        decrypted message."""
     if RSA_verify(decrypted_message, signature,
        public_signature_key):
         return True
     return False

def handle_public_key(self, message):
         # Parse the public key message
         parts = message.split(": ")
         if len(parts) == 2:
              username = parts[0][14:]
              if(not (username == self.username)):
                   public_key = int(parts[1])

                   # Store the public key in the
                      dictionary
                   self.public_keys[username] =
                      public_key

                   # Calculate the shared secret key
                   shared_secret_key =
                      pow(public_key,
                      self.secret_integer,
                      primeNumber)

                   # Store the shared secret key and
                      associated username for later
                      use
                   self.display_message("-----------------------------------
                   self.display_message(f"Recieved
                      Public Key from {username}:
                      {public_key}")
                   self.display_message(f"Shared
                      secret key with {username}:
                      {shared_secret_key}")
                   self.display_message("-----------------------------------

def handle_public_IV(self, message):
     # Parse the public key message
     parts = message.split(": ")
     if len(parts) == 2:
          username = parts[0][13:]
          if(not (username == self.username)):
               IV = b64decode(parts[1])
```

```
            self.public_IVs[username] = IV
            self.display_message("----------------------------------------
            self.display_message(f'Recieved
               {username}\'s public IV: {IV}')
            self.display_message("----------------------------------------

def handle_public_signature(self, message):
    # Parse the public key message
    parts = message.split(": ")
    if len(parts) == 2:
        username = parts[0][17:]
        if(not (username == self.username)):
            signature_key_str = parts[1]
            signature_key_str =
               signature_key_str.replace("(",
               "").replace(")", "")
            signature_key = list(map(int,
               signature_key_str.replace("(",
               "").replace(")", "").split(',')))

            self.public_signature_keys[username]
               = signature_key
            self.display_message("----------------------------------------
            self.display_message(f'Recieved
               {username}\'s public signature
               key: {signature_key}')
            self.display_message("----------------------------------------

def check_shared_key(self, message):
    """Check and output the shared key for a
       specific user."""
    parts = message.split(' ')
    if len(parts) == 2:
        target_username = parts[1]

        # Check if we have the public key for the
           target user
        if target_username == self.username:
            self.display_message(f"Your Public
               Key Is: {pow(primitiveRoot,
                   self.secret_integer,
               primeNumber)}")
        elif target_username in self.public_keys:
            # Calculate the shared secret key
            shared_secret_key =
               pow(self.public_keys[target_username],
```

```
                self.secret_integer, primeNumber)

            # Output the
            # shared key for the specific user
            self.display_message(f"Shared secret
                key with {target_username}:
                {shared_secret_key}")
        else:
            self.display_message(f"Public key for
                {target_username} not available.")

def send_private_message(self, message):
    """Send a private message to a specific
        user."""
    # Split the message into parts
    parts = message.split(' ', 2)
    target_username = parts[1]
    message_content = parts[2]

    # Compute shared secret key
    shared_secret_key =
        pow(self.public_keys[target_username],
        self.secret_integer, primeNumber)
    print(bin(shared_secret_key)[2:])

    # Encrypt the message based on encryption type
    ciphertext =
        self.encrypt_message(message_content +"
        HACKED", shared_secret_key,
        target_username)

    # Create the final message with encryption
        and digital signature
    final_message = f'./sendToUser
        {target_username} {ciphertext}
        {RSA_sign(message_content,
        self.privateSignKey)}'

    # Send the message to the server
    client.send(final_message.encode('utf-8'))

def encrypt_message(self, message,
   shared_secret_key, target_username):
    """Encrypt the message based on the
        encryption type."""
    if ENCRYPTIONTYPE == "STREAMCIPHER":
```

```
            return encrypt(message,
                str(bin(shared_secret_key)[2:]))
        elif ENCRYPTIONTYPE == "AES_ECB" or
            ENCRYPTIONTYPE == "AES_CBC":
            key_bytes =
                shared_secret_key.to_bytes((shared_secret_key.bit_length()
                + 7) // 8, 'little')
            return encrypt(str(message), key_bytes,
                modes.ECB() if ENCRYPTIONTYPE ==
                "AES_ECB" else
                modes.CBC(self.IV)).decode()
        elif ENCRYPTIONTYPE == "DES_ECB" or
            ENCRYPTIONTYPE == "DES_CBC":
            key_bytes =
                shared_secret_key.to_bytes((shared_secret_key.bit_length()
                + 7) // 8, byteorder='big')
            key_bytes =
                hashlib.sha256(key_bytes).digest()[:8]
            if ENCRYPTIONTYPE == "DES_ECB":
                return encrypt(str(message),
                    key_bytes, DES.MODE_ECB).decode()
            else:
                return encrypt(str(message),
                    key_bytes, DES.MODE_CBC,
                    self.IV).decode()
if __name__ == "__main__":
    # Start with the username entry GUI
    username_root = Tk()
    username_gui = UsernameGUI(username_root)
    username_root.mainloop()
```

6.3 LFSR RSA-Key Exchange Server (RSAServer.py)

```
import socket
import threading
import random
import time
from base64 import b64encode, b64decode


# Server configuration
HOST = '192.168.1.31'
PORT = 55555

# Lists to store connected clients, their usernames
```

```
clients = []
usernames = []

# Create a socket for the server
server = socket.socket(socket.AF_INET,
   socket.SOCK_STREAM)
server.bind((HOST, PORT))

public_keys_dict = {}
public_signature_keys_dict = {}

RECIVINGIVs = True

if(RECIVINGIVs):
    public_ivs_dict = {}

def broadcast(message, sender, publicKey=False):
    """Send a message to all clients except the
       sender."""
    for client in clients:
        try:
            if(not publicKey):
                client.send(f"[{usernames[clients.index(sender)]}]
                   ".encode('utf-8') + message)
            else:
                client.send(message)
        except:
            # Remove the client if unable to send a
               message
            handle_disconnect(client)


def handle_disconnect(client):
    """Handle client disconnection."""
    index = clients.index(client)
    client.close()
    username = usernames[index]
    broadcast(f"{username} has left the
       chat.".encode('utf-8'), client)  # Call
       broadcast directly here
    usernames.remove(username)
    clients.remove(client)

def broadcast_public_keys(client):
    """Send public keys to the newly joined client."""
    for username, public_key in
```

```
        public_keys_dict.items():
         key_message = f"Public key of {username}:
            {public_key}"
         time.sleep(0.2)
         client.send(key_message.encode('utf-8'))

def broadcast_public_IVs(client):
    """Send public keys to the newly joined client."""
    for username, public_iv in
        public_ivs_dict.items():
         key_message = f"Public IV of {username}:
            {b64encode(public_iv).decode()}"
         time.sleep(0.2)
         client.send(key_message.encode('utf-8'))

def broadcast_public_signature_keys(client):
    """Send public keys to the newly joined client."""
    for username, public_signature_key in
        public_signature_keys_dict.items():
         key_message = f"Signature key of {username}:
            ({public_signature_key[0],public_signature_key[1]})"
         time.sleep(0.2)
         client.send(key_message.encode('utf-8'))

def handle_client(client, username):
    """Handle individual client connections."""
    try:
        # Broadcast the new user joining the chat
        broadcast(f"{username} has joined the
            chat.".encode('utf-8'), client)
        current_users = ', '.join(user for user in
            usernames if user != username)
        client.send(f"Other users in the Room:
            {current_users}".encode('utf-8'))
        time.sleep(0.1)
        broadcast_public_keys(client)
        time.sleep(0.1)
        broadcast_public_signature_keys(client)
        if(RECIVINGIVs):
             time.sleep(0.1)
             broadcast_public_IVs(client)

        # Broadcast the public key of the new user to
            all clients
        public_key_message =
            client.recv(1024).decode('utf-8')
```

```
if public_key_message.startswith("Public key:
   "):
    parts = public_key_message.split(" Public
       Signature Key: ")
    public_key = int(parts[0][12:])
    signature_key_str = parts[1]

    # Extracting signature key
    if "Public IV: " in signature_key_str:
        # Both signature key and IV are
           present
        signature_key_str, public_iv_str =
           signature_key_str.split(" Public
           IV: ")
        public_iv = b64decode(public_iv_str)
        public_ivs_dict[username] = public_iv
        broadcast(f"Public IV of {username}:
           {b64encode(public_iv).decode()}".encode('utf-8'),
           client, True)

    # Extracting signature key from the new
       format
    signature_key_str =
       signature_key_str.replace("(",
       "").replace(")", "")
    signature_key = list(map(int,
       signature_key_str.replace("(",
       "").replace(")", "").split(',')))

    public_keys_dict[username] = public_key
    public_signature_keys_dict[username] =
       signature_key

    # Broadcasting public key and signature
       key
    broadcast(f"Public key of {username}:
       {public_key}".encode('utf-8'), client,
       True)
    time.sleep(0.1)
    broadcast(f"Signature key of {username}:
       ({signature_key[0]},{signature_key[1]})".encode('utf-8'),
       client, True)
else:
    print("Invalid public key format.\n")
    print(public_key_message)
```

```
while True:
    message =
        client.recv(1024).decode('utf-8')

    # Announce all current users to the new
        client

    # Check if the message is a special
        command to send a private message
    if message.startswith('./sendToUser'):
        # Extract the target username and the
            private message from the command
        parts = message.split(' ', 3)
        if len(parts) == 4:
            target_username = parts[1]
            private_message = parts[2]
            signature = parts[3]
            # Find the target client based on
                the username
            target_client = next((c for c, u
                in zip(clients, usernames) if
                u == target_username), None)

            # Send the private message to the
                target user
            if target_client:
                # Remove the command and
                    username, leaving only the
                    private message
                cleaned_message =
                    private_message
                broadcast(private_message.encode('utf-8'),
                    client)
                target_client.send(f"./decrypt
                    {username}
                    {cleaned_message}
                    {signature}".encode('utf-8'))
            else:
                print(f"User not found.")
        else:
            print("Invalid private message
                format.")

    elif message.startswith('./sendLFSRkey'):
        # Extract the target username and the
            private message from the command
```

```
    parts = message.split(' ', 4)
    if len(parts) == 5:
        target_username = parts[1]
        encrypted_message = parts[2]
        # Find the target client based on
           the username
        target_client = next((c for c, u
           in zip(clients, usernames) if
           u == target_username), None)

        # Send the private message to the
           target user
        if target_client:
            broadcast(encrypted_message.encode('utf-8'),
               client)
            time.sleep(0.1)
            target_client.send(f"{message}
               {username}".encode('utf-8'))
            response = f'./success
               {target_username}'
        else:
            print(f"User not found.")
            response = f'./fail
               {target_username}'

        # Send the response to the client
           initiating the command
        sender_index =
           clients.index(client)
        sender_username =
           usernames[sender_index]
        sender_client =
           clients[sender_index]


        sender_client.send(response.encode('utf-8'))

    else:
        print("Invalid private message
           format.")
        time.sleep(0.1)
        client.send(f'./fail
           {target_username}'.encode('utf-8'))


else:
```

```
                # Broadcast the message to all clients
                broadcast(message.encode('utf-8'),
                   client)

    except (socket.error, ConnectionResetError):
        # Handle client disconnection
        handle_disconnect(client)



def start_server():
    """Start the chat server."""
    server.listen()
    print(f"Server is listening on {HOST}:{PORT}")

    while True:
        # Accept a new client connection
        client, address = server.accept()
        print(f"New connection from {address}")

        # Receive the username from the client
        username = client.recv(1024).decode('utf-8')

        # Add the new client and username to the lists
        clients.append(client)
        usernames.append(username)

        # Start a new thread to handle the client
        thread =
           threading.Thread(target=handle_client,
           args=(client, username))
        thread.start()


if __name__ == "__main__":
    start_server()
```

6.4 LFSR RSA-Key Exchange Client (RSAClient.py)

```
import socket
import threading
import random
from tkinter import Tk, Scrollbar, Listbox, Entry,
   Button, StringVar, DISABLED, NORMAL, Toplevel,
   Label, WORD, Text, END
```

```
from sympy import isprime
from gmpy2 import powmod
import os
import time
import math
import hashlib

ENCRYPTIONTYPE = 'DES_CBC'

if(ENCRYPTIONTYPE == 'STREAMCIPHER'):
    from streamCipher import *

elif(ENCRYPTIONTYPE == 'AES_ECB' or ENCRYPTIONTYPE ==
   'AES_CBC'):
    from AES import *
elif(ENCRYPTIONTYPE == 'DES_ECB' or ENCRYPTIONTYPE ==
   'DES_CBC'):
    from DES import *

# Client configuration
HOST = '192.168.1.31'
PORT = 55555

# Agreed Upon Values
primeNumber =
   1021886172171788044763879771601293344317459450097300655193370949921296772
primitiveRoot = 2

# Create a socket for the client
client = socket.socket(socket.AF_INET,
   socket.SOCK_STREAM)
client.connect((HOST, PORT))

class LFSR:
    def __init__(self, seed, taps):
        self.state = seed
        self.taps = taps

    def shift(self):
        feedback = sum(self.state[tap] for tap in
           self.taps) % 2
        self.state = [feedback] + self.state[:-1]
        return feedback

    def generate_key(self, length):
        key = []
```

```
        for _ in range(length):
            key.append(self.shift())

        # Ensure that the generated key is not zero
        generated_key_int = int(''.join(map(str,
           key)), 2)
        while generated_key_int == 0:
            key = []
            for _ in range(length):
                key.append(self.shift())
            generated_key_int = int(''.join(map(str,
               key)), 2)

        # Return both integer and binary string forms
        generated_key_bin = ''.join(map(str, key))
        return generated_key_int, generated_key_bin


seed = [1, 0, 1, 0]
shiftFeedbackPositions = [0, 2, 3]
lfsr = LFSR(seed, shiftFeedbackPositions)

#RSA Functions#
'''Helper function that generates a large prime
   number with the specified number of bits, in which
   for this assignment is 512.'''
def generate_large_prime(bits):
    while True:
        num = random.randrange(0, 2**bits - 1)
        if isprime(num):
            return num
'''Calculates the modular exponetiation of a given
   message, power, and basis using the 'powmod'
   function from the gmpy2 library'''
def exponentiation(message, power, basis):
    return powmod(message, power, basis)

'''Helper function that incorporates the extended
   euclidean algorithm to help determine the inverse
   value within the inverse_finder function.'''
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        g, x, y = extended_gcd(b % a, a)
        return g, y - (b // a) * x, x
```

```
'''Finds the modular inverse of a given number a
   modulo n using the extended euclidean algorithm.'''
def inverse_finder(a, n):
    g, x, _ = extended_gcd(a, n)
    if g != 1:
        raise ValueError(f"The modular inverse does
            not exist for {a} modulo {n}")
    else:
        return x % n
'''A function that generates RSA public and private
   keys. It takes an optional parameter e for the
   rsaKeyInput; if not provided, it defaults to 3.'''
def RSA_key_generate():
    e = 65537
    while(True):
        p = generate_large_prime(256)
        q = p

        while(p == q):
            q = generate_large_prime(256)
        n = p * q
        euler = (p-1) * (q-1)

        if(math.gcd(euler, e) == 1):
            break
    d = inverse_finder(e, euler)
    publicKey = [e, n]
    privateKey = [d, n]
    return publicKey, privateKey

'''A function encrypts a numeric message or a string
   using RSA encryption with a given key.'''
def RSA_encrypt(message, key):
    if not isinstance(message, str):
        return exponentiation(message, key[0], key[1])
    elif isinstance(message, str):
        ciphertext = []
        for element in range(0, len(message)):
            ciphertext.append(int(exponentiation(ord(message[element]),
                key[0], key[1])))
        return ciphertext

'''A function decrypts a numeric message or a list of
   numeric values using RSA decryption with a given
   key.'''
```

```
def RSA_decrypt(message, key):
    if not isinstance(message, str) and not
       isinstance(message, list):
        return RSA_encrypt(message, key)
    elif isinstance(message, list):
        decrpyted = ''
        for element in range(0, len(message)):
            decrpyted+=
                chr(exponentiation(message[element],
                key[0], key[1]))
        return decrpyted
def hash_message(message):
    # Hash the message using SHA-256
    sha256 = hashlib.sha256()
    sha256.update(str(message).encode('utf-8'))
    return int(sha256.hexdigest(), 16)

def RSA_sign(message, private_key):
    hashed_message = hash_message(message)
    signature = exponentiation(hashed_message,
       private_key[0], private_key[1])
    return signature

def RSA_verify(message, signature, public_key):
    hashed_message = hash_message(message)
    decrypted_signature = exponentiation(signature,
       public_key[0], public_key[1])

    if hashed_message == decrypted_signature:
        return True
    else:
        return False

class UsernameGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Enter Username")

        # Label and entry widget for entering the
           username
        self.username_label = Label(root, text="Enter
           your username:")
        self.username_label.pack(pady=10)
        self.username_entry = Entry(root, width=30)
        self.username_entry.pack(pady=10)
```

```
        # Button to submit the username
        submit_button = Button(root, text="Submit",
            command=self.submit_username)
        submit_button.pack(pady=10)


    def submit_username(self):
        username = self.username_entry.get()
        if username:
            self.root.destroy()  # Close the username
                entry GUI
            # Start the chat GUI with the entered
                username
            chat_root = Tk()
            chat_gui = ChatGUI(chat_root, username)
            chat_root.mainloop()
        else:
            # Display an error message if the
                username is empty
            error_label = Label(self.root,
                text="Please enter a valid username.")
            error_label.pack(pady=5)


class ChatGUI:
    def __init__(self, root, username):
        self.root = root
        self.root.title(f"{username}'s Chat Window")
        self.username = username

        # Create a listbox to display messages
        self.message_text = Text(root, height=20,
            width=200, selectbackground="white",
            exportselection=False, wrap=WORD)
        self.message_text.pack(padx=10, pady=10)

        # Create a scrollbar for the listbox
        scrollbar = Scrollbar(root)
        scrollbar.pack(side="right", fill="y")

        # Attach the listbox to the scrollbar
        self.message_text.config(yscrollcommand=scrollbar.set)
        scrollbar.config(command=self.message_text.yview)

        # Create an entry widget for typing messages
        self.message_entry = Entry(root, width=200)
```

```
self.message_entry.pack(padx=10, pady=10)

# Create a Send button to send messages
send_button = Button(root, text="Send",
   command=self.send_message)
send_button.pack(pady=10)


self.public_key, self.private_key =
   RSA_key_generate()
self.recievedLSFRKeys = {}
self.sentLSFRKeys = []
self.public_keys = {}
self.public_signature_keys = {}
self.publicSignKey, self.privateSignKey =
   RSA_key_generate()
if(ENCRYPTIONTYPE == "AES_CBC"):
    self.public_IVs = {}
    self.IV = os.urandom(16)
elif(ENCRYPTIONTYPE == "DES_CBC"):
    self.public_IVs = {}
    self.IV = os.urandom(8)
random_seed_length = random.randint(4, 15)
seed = [random.randint(0, 1) for _ in
   range(random_seed_length)]
shiftFeedbackPositions =
   random.sample(range(len(seed)),
   k=random.randint(1, len(seed)))
shiftFeedbackPositions.sort()
lfsr = LFSR(seed, shiftFeedbackPositions)
key_length = 256
if(ENCRYPTIONTYPE == 'DES_ECB' or
   ENCRYPTIONTYPE == 'DES_CBC'):
    key_length = 64
self.key_length = key_length
generated_key, generated_keyBin =
   lfsr.generate_key(key_length)
self.LSFRKey, self.LSFRKeyBin =
   generated_key, generated_keyBin

# Start a thread to receive messages
receive_thread =
   threading.Thread(target=self.receive_messages)
receive_thread.start()

# Send the username to the server
```

```
    client.send(username.encode('utf-8'))
    if(ENCRYPTIONTYPE == "AES_CBC" or
       ENCRYPTIONTYPE == "DES_CBC"):
        client.send(f"Public key:
           {self.public_key[1]} Public Signature
           Key:
           ({self.publicSignKey[0]},{self.publicSignKey[1]})
           Public IV:
           {b64encode(self.IV).decode()}".encode('utf-8'))
        self.display_message("-------------------------------------------
        self.display_message(f"Your generated
           LFSR Key: {self.LSFRKeyBin}")
        self.display_message(f"Your Public Key:
           {self.public_key[1]}")
        self.display_message(f"Your Public
           Signature Key:
           {({self.publicSignKey[0]},{self.publicSignKey[1]})}")
        self.display_message(f"Your Public IV:
           {b64encode(self.IV).decode()}")
        self.display_message("-------------------------------------------
    else:
        client.send(f"Public key:
           {self.public_key[1]} Public Signature
           Key:
           ({self.publicSignKey[0]},{self.publicSignKey[1]})".encode('ut
        self.display_message("-------------------------------------------
        self.display_message(f"Your generated
           LFSR Key: {self.LSFRKeyBin}")
        self.display_message(f"Your Public Key:
           {self.public_key[1]}")
        self.display_message(f"Your Public
           Signature Key:
           {({self.publicSignKey[0]},{self.publicSignKey[1]})}")
        self.display_message("-------------------------------------------

def send_message(self):
    message = self.message_entry.get()
    self.message_entry.delete(0, 'end')

    # Check if the message is a special command
       to send a private message
    if message.startswith('./checkSharedKey'):
        self.check_shared_key(message)
    elif message.startswith('./sendToUser'):
        self.send_private_message(message)
    elif message.startswith('./sendLFSRKey'):
```

```
            self.send_LSFR_key(message)
        else:
            client.send(message.encode('utf-8'))

    def receive_messages(self):
        """Receive and display messages from the
           server."""
        while True:
            try:
                message =
                   client.recv(1024).decode('utf-8')

                if
                   message.startswith('./sendLFSRkey'):
                    self.handle_sendLFSRkey(message)
                elif message.startswith('./success'):
                    self.handle_success(message)
                elif message.startswith("Public key
                   of "):
                    self.handle_public_key(message)
                elif message.startswith("Public IV of
                   "):
                    self.handle_public_IV(message)
                elif message.startswith("Signature
                   key of "):
                    self.handle_public_signature(message)
                elif message.startswith('./fail'):
                    self.handle_fail(message)
                elif message.startswith("./decrypt"):
                    self.handle_decrypt(message)
                else:
                    self.display_message(message)

            except Exception as e:
                print(f"An error occurred while
                   receiving messages: {e}")
                client.close()
                break

    def handle_sendLFSRkey(self, message):
        parts = message.split(' ', 5)
        encryptedLSFRKey = int(parts[2])
        decryptionKey = [int(parts[3]), int(parts[4])]
        fromUser = parts[5]
        sentLSFR = RSA_decrypt(encryptedLSFRKey,
           decryptionKey)
```

```
        self.recievedLSFRKeys[fromUser] =
           int(sentLSFR)
        key_length = f'0{self.key_length}b'
        self.display_message(f'Received from
           {fromUser}:
           {format(self.recievedLSFRKeys[fromUser],
           key_length)}')

    def handle_success(self, message):
        target_username = message.split(' ', 1)[1]
        self.display_message(f"LSFR key successfully
           sent to {target_username}.")
        self.sentLSFRKeys.append(target_username)

    def handle_fail(self, message):
        target_username = message.split(' ', 1)[1]
        self.display_message(f"Failed to send LSFR
           key to {target_username}.")

    def handle_decrypt(self, message):
        parts = message.split(' ', 3)
        target_username, private_message, signature =
           parts[1], parts[2], int(parts[3])
        self.display_message("--------------------------------------------
        self.display_message(f'The encrypted message
           was sent for you by {target_username}.')

        key_bytes =
           self.recievedLSFRKeys[target_username].to_bytes((self.recievedLSF
           + 7) // 8, 'little')

        if ENCRYPTIONTYPE == "STREAMCIPHER":
             DecryptedMessage =
                 decrypt(private_message,
                 str(bin(self.recievedLSFRKeys[target_username])[2:]))
        elif ENCRYPTIONTYPE == "AES_ECB":
             DecryptedMessage =
                 decrypt(str(private_message),
                 key_bytes, modes.ECB()).decode('utf-8')
        elif ENCRYPTIONTYPE == "AES_CBC":
             DecryptedMessage =
                 decrypt(private_message, key_bytes,
                 modes.CBC(self.public_IVs[target_username])).decode('utf-8')
        elif ENCRYPTIONTYPE == "DES_ECB":
             DecryptedMessage =
                 decrypt(private_message, key_bytes,
```

```
            DES.MODE_ECB)
    elif ENCRYPTIONTYPE == "DES_CBC":
        DecryptedMessage =
            decrypt(private_message, key_bytes,
            DES.MODE_CBC,
            self.public_IVs[target_username])

    self.display_message(f'Decrypted Message
       using associated Key: {DecryptedMessage}')

    if RSA_verify(DecryptedMessage, signature,
       self.public_signature_keys[target_username]):
        self.display_message(f'This message has
            been verified from its Digital
            Signature!')
    else:
        self.display_message(f'This message is
            not verified from its Digital
            Signature!')
    self.display_message("-----------------------------------------------

def display_message(self, message):
    self.message_text.insert(END, message + '\n')


def send_private_message(self, message):
    # Extract username, target_username, and
       private_message from the message
    _, target_username, private_message =
       message.split(' ', 2)

    # Check if the target username is in the list
       of users
    if target_username not in self.sentLSFRKeys:
        self.display_message("-----------------------------------------------
        self.display_message(f"Error: Either user
            {target_username} does not exist OR
            you have not yet sent your LFSR Key to
            them!")
        self.display_message("-----------------------------------------------
        # Handle the error case as needed (e.g.,
            display an error message)
        return

    # Encrypt the private message based on
       encryption type
```

```
        ciphertext =
            self.encrypt_private_message(private_message)

        # Create the final message with encryption
            and digital signature
        final_message = f'./sendToUser
            {target_username} {ciphertext}
            {RSA_sign(private_message,
            self.privateSignKey)}'

        # Send the message to the server
        client.send(final_message.encode('utf-8'))

    def encrypt_private_message(self,
        private_message):
        """Encrypt the private message based on the
            encryption type."""
        if ENCRYPTIONTYPE == "STREAMCIPHER":
            return encrypt(private_message,
                str(bin(self.LSFRKey)[2:]))
        elif ENCRYPTIONTYPE == "AES_ECB" or
            ENCRYPTIONTYPE == "AES_CBC":
            key_bytes =
                self.LSFRKey.to_bytes((self.LSFRKey.bit_length()
                + 7) // 8, 'little')
            return encrypt(str(private_message),
                key_bytes, modes.ECB() if
                ENCRYPTIONTYPE == "AES_ECB" else
                modes.CBC(self.IV)).decode()
        elif ENCRYPTIONTYPE == "DES_ECB" or
            ENCRYPTIONTYPE == "DES_CBC":
            key_bytes =
                self.LSFRKey.to_bytes((self.LSFRKey.bit_length()
                + 7) // 8, 'little')
            if ENCRYPTIONTYPE == "DES_ECB":
                return encrypt(private_message,
                    key_bytes, DES.MODE_ECB).decode()
            else:
                return encrypt(private_message,
                    key_bytes, DES.MODE_CBC,
                    self.IV).decode()


    def send_LSFR_key(self, message):
        _, target_username = message.split(' ', 2)
```

```
        # Check if the LSFR key has already been sent
           to the target user
        if target_username in self.sentLSFRKeys:
            self.display_message("-----------------------------------------
            self.display_message(f"You have already
               sent your LFSR Key to
               {target_username}.")
            self.display_message("-----------------------------------------
            return

        # Send the LSFR key to the target user
        encrypted_lsfr_key =
           RSA_encrypt(self.LSFRKey, self.public_key)
        client.send(f"./sendLFSRkey {target_username}
           {encrypted_lsfr_key} {self.private_key[0]}
           {self.private_key[1]}".encode('utf-8'))

    def handle_public_key(self, message):
        # Parse the public key message
        parts = message.split(": ")
        if len(parts) == 2:
            username = parts[0][14:]
            if(not (username == self.username)):
                public_key = int(parts[1])

                self.public_keys[username] =
                   public_key
                self.display_message("-------------------------------------
                self.display_message(f'Recieved
                   {username}\'s public key:
                   {public_key}')
                self.display_message("-------------------------------------

    def handle_public_IV(self, message):
        # Parse the public key message
        parts = message.split(": ")
        if len(parts) == 2:
            username = parts[0][13:]
            if(not (username == self.username)):
                IV = b64decode(parts[1])

                self.public_IVs[username] = IV
                self.display_message("-------------------------------------
                self.display_message(f'Recieved
                   {username}\'s public IV: {IV}')
                self.display_message("-------------------------------------
```

```
    def handle_public_signature(self, message):
        # Parse the public key message
        parts = message.split(": ")
        if len(parts) == 2:
            username = parts[0][17:]
            if(not (username == self.username)):
                signature_key_str = parts[1]
                signature_key_str =
                    signature_key_str.replace("(",
                    "").replace(")", "")
                signature_key = list(map(int,
                    signature_key_str.replace("(",
                    "").replace(")", "").split(',')))

                self.public_signature_keys[username]
                    = signature_key
                self.display_message("--------------------------------------
                self.display_message(f'Recieved
                    {username}\'s public signature
                    key: {signature_key}')
                self.display_message("--------------------------------------


if __name__ == "__main__":
    # Start with the username entry GUI
    username_root = Tk()
    username_gui = UsernameGUI(username_root)
    username_root.mainloop()
```

6.5 Stream Cipher Encryption (streamCipher.py)

```
def text_to_bits(text):
    return ''.join(format(ord(char), '08b') for char
        in text)

def bits_to_text(bits):
    return ''.join(chr(int(bits[i:i+8], 2)) for i in
        range(0, len(bits), 8))

# Stream Cipher
def encrypt(text, key):
    bits = text_to_bits(text)
    encrypted_bits = [int(bit) ^ int(key[i %
        len(key)]) for i, bit in enumerate(bits)]
```

```
    return ''.join(map(str, encrypted_bits))

def decrypt(ciphertext, key):
    decrypted_bits = [int(bit) ^ int(key[i %
       len(key)]) for i, bit in enumerate(ciphertext)]
    return bits_to_text(''.join(map(str,
       decrypted_bits)))
```

6.6 AES (AES.py)

```
from cryptography.hazmat.primitives.ciphers import
   Cipher, algorithms, modes
from cryptography.hazmat.backends import
   default_backend
from base64 import b64encode, b64decode

def pad(text):
    # PKCS7 padding
    block_size = 16
    if isinstance(text, str):
        text = text.encode('utf-8')  # Convert string
            to bytes
    pad_size = block_size - len(text) % block_size
    return text + bytes([pad_size] * pad_size)

def unpad(text):
    pad_size = text[-1]
    return text[:-pad_size]

def encrypt(plaintext, key, mode):
    plaintext = pad(plaintext)
    cipher = Cipher(algorithms.AES(key), mode,
       backend=default_backend())
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(plaintext) +
       encryptor.finalize()
    return b64encode(ciphertext)

def decrypt(ciphertext, key, mode):
    ciphertext = b64decode(ciphertext)
    cipher = Cipher(algorithms.AES(key), mode,
       backend=default_backend())
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) +
       decryptor.finalize()
```

```
    return unpad(plaintext)

from PIL import Image
import io
import matplotlib.pyplot as plt

def encrypt_image(image_path, key, mode):
    with open(image_path, 'rb') as image_file:
        image_data = image_file.read()
    ciphertext = encrypt(image_data, key, mode)
    return ciphertext

def decrypt_image(ciphertext, key, mode):
    decrypted_data = decrypt(ciphertext, key, mode)
    return decrypted_data
def view_image(image_path):
    image = Image.open(image_path)
    plt.imshow(image)
    plt.axis('off')  # Turn off axis labels
    plt.show()

def view_image_from_decrypted(image_data):
    image = Image.open(io.BytesIO(image_data))
    plt.imshow(image)
    plt.axis('off')  # Turn off axis labels
    plt.show()
'''
image_path = 'testImage.jpg'
view_image(image_path)
key = b'sixteen byte key'
ecb_ciphertext = encrypt_image(image_path, key,
   modes.ECB())
cbc_ciphertext = encrypt_image(image_path, key,
   modes.CBC(b'\x00' * 16))

print(f"Encrypted Image Data (ECB):
   {ecb_ciphertext}\n\n")
print(f"Encrypted Image Data (CBC):
   {cbc_ciphertext}\n\n")

decrypted_ecb = decrypt_image(ecb_ciphertext, key,
   modes.ECB())
decrypted_cbc = decrypt_image(cbc_ciphertext, key,
   modes.CBC(b'\x00' * 16))

print(f"Decrypted Image Data (ECB):
```

```
    {decrypted_ecb}\n\n")
print(f"Decrypted  Image Data (CBC): {decrypted_cbc}")

view_image_from_decrypted(decrypted_ecb)
view_image_from_decrypted(decrypted_cbc)
'''
```

6.7 DES (DES.py)

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
from base64 import b64encode, b64decode

def encrypt(plaintext, key, mode, iv=None):
    if mode == DES.MODE_ECB:
        cipher = DES.new(key, DES.MODE_ECB)
    elif mode == DES.MODE_CBC:
        if iv is None:
            raise ValueError("IV is required for CBC
                mode")
        cipher = DES.new(key, DES.MODE_CBC, iv)
    else:
        raise ValueError("Invalid mode")

    plaintext = pad(plaintext.encode(),
        DES.block_size)
    ciphertext = cipher.encrypt(plaintext)

    return b64encode(ciphertext)

def decrypt(ciphertext, key, mode, iv=None):
    ciphertext = b64decode(ciphertext)

    if mode == DES.MODE_ECB:
        cipher = DES.new(key, DES.MODE_ECB)
    elif mode == DES.MODE_CBC:
        if iv is None:
            raise ValueError("IV is required for CBC
                mode")
        cipher = DES.new(key, DES.MODE_CBC, iv)
    else:
        raise ValueError("Invalid mode")
```

```
    plaintext = unpad(cipher.decrypt(ciphertext),
       DES.block_size)

    return plaintext.decode()
```

6.8 AES DES Plots (AESvsDES.py

```
import timeit
import matplotlib.pyplot as plt
import random
import string
from Crypto.Random import get_random_bytes
from AES import encrypt as aes_encrypt, decrypt as
   aes_decrypt
from DES import encrypt as des_encrypt, decrypt as
   des_decrypt
from cryptography.hazmat.primitives.ciphers import
   modes
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

def generate_random_string(length):
    return
       ''.join(random.choice(string.ascii_letters)
       for _ in range(length))

def test_aes():
    key = get_random_bytes(16)  # 128-bit key for AES
    iv = get_random_bytes(16)
    modes_to_test = [modes.ECB(), modes.CBC(iv)]  #
       Add more modes if needed

    for mode in modes_to_test:
        times_encrypt = []
        times_decrypt = []
        lengths = list(range(1, 5001, 50))  # Vary
           the length of the plaintext

        for length in lengths:
            plaintext = generate_random_string(length)

            encrypt_time = timeit.timeit(lambda:
               aes_encrypt(plaintext, key, mode),
               number=1000)  # Increased number of
```

```
                iterations
            times_encrypt.append(encrypt_time * 1e6 /
                1000)  # Convert to microseconds,
                average time per encryption

            decrypt_time = timeit.timeit(lambda:
                aes_decrypt(aes_encrypt(plaintext,
                key, mode), key,
                mode).decode("utf-8"), number=1000)  #
                Increased number of iterations
            times_decrypt.append(decrypt_time * 1e6 /
                1000)  # Convert to microseconds,
                average time per decryption

            assert aes_decrypt(aes_encrypt(plaintext,
                key, mode), key, mode).decode("utf-8")
                == plaintext  # Ensure decryption is
                correct

        # Plot results
        plt.plot(lengths, times_encrypt, label=f'AES
            {mode.name} Encryption')
        plt.plot(lengths, times_decrypt, label=f'AES
            {mode.name} Decryption')

    plt.xlabel('Length of Plaintext')
    plt.ylabel('Time (microseconds)')
    plt.legend()
    plt.show()

def test_des():
    key = get_random_bytes(8)  # 64-bit key for DES
    iv = get_random_bytes(8)
    modes_to_test = [DES.MODE_ECB, DES.MODE_CBC]  #
        Add more modes if needed
    for mode in modes_to_test:
        times_encrypt = []
        times_decrypt = []
        lengths = list(range(1, 5001, 50))  # Vary
            the length of the plaintext

        for length in lengths:
            plaintext = generate_random_string(length)

            if mode == DES.MODE_CBC:
                encrypt_time = timeit.timeit(lambda:
```

```
                des_encrypt(plaintext, key, mode,
                iv), number=1000)
            decrypt_time = timeit.timeit(lambda:
                des_decrypt(des_encrypt(plaintext,
                key, mode, iv), key, mode, iv),
                number=1000)
        else:
            encrypt_time = timeit.timeit(lambda:
                des_encrypt(plaintext, key, mode),
                number=1000)
            decrypt_time = timeit.timeit(lambda:
                des_decrypt(des_encrypt(plaintext,
                key, mode), key, mode),
                number=1000)

        times_encrypt.append(encrypt_time * 1e6 /
            1000)  # Convert to microseconds,
            average time per encryption
        times_decrypt.append(decrypt_time * 1e6 /
            1000)  # Convert to microseconds,
            average time per decryption

        assert des_decrypt(des_encrypt(plaintext,
            key, mode, iv), key, mode, iv) ==
            plaintext  # Ensure decryption is
            correct

    # Plot results
    if(mode == DES.MODE_ECB):
        title = "ECB"
    else:
        title = "CBC"
    plt.plot(lengths, times_encrypt, label=f'DES
        {title} Encryption')
    plt.plot(lengths, times_decrypt, label=f'DES
        {title} Decryption')

plt.xlabel('Length of Plaintext')
plt.ylabel('Time (microseconds)')
plt.legend()
plt.show()

if __name__ == "__main__":
    test_aes()
    test_des()
```

6.9 QuicKOverallView (QuickOverall.py

```
import random

primeNumber =
   1021886172171788044763879771601293344317459450097300655193370949921296772
primitiveRoot = 2

class User:
    pass

# Initialize two users
user1 = User()
user2 = User()

# DIFFIE-HELLMAN KEY EXCHANGE

# Users generate their Secret Key
user1.secret_integer = random.randint(2, primeNumber
   - 2)
user2.secret_integer = random.randint(2, primeNumber
   - 2)

# Users calculate their public keys using primitive
   root
user1.DHPublicKey = pow(primitiveRoot,
   user1.secret_integer, primeNumber)
user2.DHPublicKey = pow(primitiveRoot,
   user2.secret_integer, primeNumber)

# Users exchange their public keys
user1.receivedDHPublicKey = user2.DHPublicKey
user2.receivedDHPublicKey = user1.DHPublicKey

# Users calculate their shared secret key
user1.sharedSecret = pow(user1.receivedDHPublicKey,
   user1.secret_integer, primeNumber)
user2.sharedSecret = pow(user2.receivedDHPublicKey,
   user2.secret_integer, primeNumber)



print("Shared Key generated by User U1:",
   user1.sharedSecret)
print("Shared Key generated by User U2:",
   user2.sharedSecret)
```

```
print("User 1 is sharing the same key as User 2:
   "+(str(user1.sharedSecret == user2.sharedSecret)))



class LFSR:
    def __init__(self, seed, taps):
        self.state = seed
        self.taps = taps

    def shift(self):
        feedback = sum(self.state[tap] for tap in
           self.taps) % 2
        self.state = [feedback] + self.state[:-1]
        return feedback

    def generate_key(self, length):
        key = []
        for _ in range(length):
            key.append(self.shift())
        binary_string = ''.join(map(str, key))
        generated_key = int(binary_string, 2)
        return binary_string, generated_key

# Define the seed and feedback positions for the LFSR
seed = [1, 0, 1, 0]
shiftFeedbackPositions = [0, 2, 3]

# Create an instance of the LFSR class feedback
   positions
lfsr = LFSR(seed, shiftFeedbackPositions)

# Specify the length of the key
key_length = 16

# Generate a key of the specified length
user1.generated_key_binary, user1.generated_key =
   lfsr.generate_key(key_length)

print("Generated LFSR Key:",
   user1.generated_key_binary)


#RSA Functions#
import math
from sympy import isprime
```

```
from gmpy2 import powmod

'''Helper function that generates a large prime
   number with the specified number of bits, in which
   for this assignment is 512.'''
def generate_large_prime(bits):
    while True:
        num = random.randrange(0, 2**bits - 1)
        if isprime(num):
            return num
'''Calculates the modular exponetiation of a given
   message, power, and basis using the 'powmod'
   function from the gmpy2 library'''
def exponentiation(message, power, basis):
    return powmod(message, power, basis)

'''Helper function that incorporates the extended
   euclidean algorithm to help determine the inverse
   value within the inverse_finder function.'''
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        g, x, y = extended_gcd(b % a, a)
        return g, y - (b // a) * x, x

'''Finds the modular inverse of a given number a
   modulo n using the extended euclidean algorithm.'''
def inverse_finder(a, n):
    g, x, _ = extended_gcd(a, n)
    if g != 1:
        raise ValueError(f"The modular inverse does
            not exist for {a} modulo {n}")
    else:
        return x % n
'''A function that generates RSA public and private
   keys. It takes an optional parameter e for the
   rsaKeyInput; if not provided, it defaults to 3.'''
def RSA_key_generate():
    e = 65537
    while(True):
        p = generate_large_prime(256)
        q = p

        while(p == q):
            q = generate_large_prime(256)
```

```
        n = p * q
        euler = (p-1) * (q-1)

        if(math.gcd(euler, e) == 1):
            break
    d = inverse_finder(e, euler)
    publicKey = [e, n]
    privateKey = [d, n]
    return publicKey, privateKey

'''A function encrypts a numeric message or a string
   using RSA encryption with a given key.'''
def RSA_encrypt(message, key):
    if not isinstance(message, str):
        return exponentiation(message, key[0], key[1])
    elif isinstance(message, str):
        ciphertext = []
        for element in range(0, len(message)):
            ciphertext.append(int(exponentiation(ord(message[element]),
               key[0], key[1])))
        return ciphertext

'''A function decrypts a numeric message or a list of
   numeric values using RSA decryption with a given
   key.'''
def RSA_decrypt(message, key):
    if not isinstance(message, str) and not
       isinstance(message, list):
        return RSA_encrypt(message, key)
    elif isinstance(message, list):
        decrpyted = ''
        for element in range(0, len(message)):
            decrpyted+=
               chr(exponentiation(message[element],
               key[0], key[1]))
        return decrpyted

# User 1 generates RSA keys
user1.RSAPublicKey, user1.RSAPrivateKey =
   RSA_key_generate()

# User 1 encrypts his LFSR
encryptedMessage =
   RSA_encrypt(user1.generated_key_binary,
   user1.RSAPublicKey)
```

```
# Assume User 1 sends his encrypted message and
   private key to User 2.
# User 2 decrypts the message.
user2.receivedMessage = RSA_decrypt(encryptedMessage,
   user1.RSAPrivateKey)

# Check if the LFSR key was successfully sent through
   RSA
print("User 2 Recieved the Message: "
   +user2.receivedMessage)
print("Successfully sent LFSR Key Through RSA: " +
   str(user1.generated_key_binary ==
   user2.receivedMessage))

def text_to_bits(text):
    return ''.join(format(ord(char), '08b') for char
       in text)

def bits_to_text(bits):
    return ''.join(chr(int(bits[i:i+8], 2)) for i in
       range(0, len(bits), 8))

# Stream Cipher
def encrypt(text, key):
    bits = text_to_bits(text)
    encrypted_bits = [int(bit) ^ int(key[i %
       len(key)]) for i, bit in enumerate(bits)]
    return ''.join(map(str, encrypted_bits))

def decrypt(ciphertext, key):
    decrypted_bits = [int(bit) ^ int(key[i %
       len(key)]) for i, bit in enumerate(ciphertext)]
    return bits_to_text(''.join(map(str,
       decrypted_bits)))

Messages = ["Hello!",
            "Welcome to Introduction to Information
               and Network Security!",
            "In the vast landscape of technology and
               innovation, the intertwining threads
               of progress and human ingenuity weave
               a narrative of constant evolution."]

#For Demo Purposes, lets use User 1's generated LFSR
   Key
for message in Messages:
```

```
    print("Original Message: "+message)
    encrypted = encrypt(message,
       user1.generated_key_binary)
    print("Encrypted Message: "+encrypted)
    decrypted = decrypt (encrypted,
       user1.generated_key_binary)
    print("Decrypted Message: "+decrypted)
    print("Successfully Encrypted and Decrypted
       Message: "+(str(message == decrypted)))

from AES import encrypt as aes_encrypt, decrypt as
   aes_decrypt
from cryptography.hazmat.primitives.ciphers import
   modes
#Using User 1's Shared Secret Key
#Format User 1's Key into a Bytes Object
integer_value = user1.sharedSecret
byte_value =
   integer_value.to_bytes((integer_value.bit_length()
   + 7) // 8, byteorder='big')
#Run the same tests using AES ECB
print("TESTS FOR AES ECB:")
for message in Messages:
    print("Original Message: "+message)
    encrypted = aes_encrypt(message, byte_value,
       modes.ECB())
    print("Encrypted Message: "+encrypted.decode())
    decrypted = aes_decrypt (encrypted, byte_value,
       modes.ECB())
    print("Decrypted Message: "+decrypted.decode())
    print("Successfully Encrypted and Decrypted
       Message: "+(str(message ==
       decrypted.decode())))

import os
#Setup initialization vector for Cipher Block Chaining
iv = os.urandom(16)
print("TESTS FOR AES CBC:")
for message in Messages:
    print("Original Message: "+message)
    encrypted = aes_encrypt(message, byte_value,
       modes.CBC(iv))
    print("Encrypted Message: "+encrypted.decode())
    decrypted = aes_decrypt (encrypted, byte_value,
       modes.CBC(iv))
    print("Decrypted Message: "+decrypted.decode())
```

```
    print("Successfully Encrypted and Decrypted
       Message: "+(str(message ==
       decrypted.decode())))

from DES import encrypt as des_encrypt, decrypt as
   des_decrypt
from Crypto.Cipher import DES

#Using a shorter key for DES
integer_value = random.getrandbits(64)
byte_value =
   integer_value.to_bytes((integer_value.bit_length()
   + 7) // 8, byteorder='big')
#Run the same tests using DES ECB
print("TESTS FOR DES ECB:")
for message in Messages:
    print("Original Message: "+message)
    encrypted = des_encrypt(message, byte_value,
       DES.MODE_ECB)
    print("Encrypted Message: "+encrypted.decode())
    decrypted = des_decrypt (encrypted, byte_value,
       DES.MODE_ECB)
    print("Decrypted Message: "+decrypted)
    print("Successfully Encrypted and Decrypted
       Message: "+(str(message == decrypted)))

import os
#Setup initialization vector for Cipher Block Chaining
iv = os.urandom(8)
print("TESTS FOR DES CBC:")
for message in Messages:
    print("Original Message: "+message)
    encrypted = des_encrypt(message, byte_value,
       DES.MODE_ECB, iv)
    print("Encrypted Message: "+encrypted.decode())
    decrypted = des_decrypt (encrypted, byte_value,
       DES.MODE_ECB, iv)
    print("Decrypted Message: "+decrypted)
    print("Successfully Encrypted and Decrypted
       Message: "+(str(message == decrypted)))

import hashlib
def hash_message(message):
    # Hash the message using SHA-256
    sha256 = hashlib.sha256()
    sha256.update(str(message).encode('utf-8'))
```

```
    return int(sha256.hexdigest(), 16)

def RSA_sign(message, private_key):
    hashed_message = hash_message(message)
    signature = exponentiation(hashed_message,
       private_key[0], private_key[1])
    return signature

def RSA_verify(message, signature, public_key):
    hashed_message = hash_message(message)
    decrypted_signature = exponentiation(signature,
       public_key[0], public_key[1])

    if hashed_message == decrypted_signature:
        return True
    else:
        return False


# User 1 wants to send a message to User 2.
Message = "Hi, User 2! Excited for Christmas?"

# User 1 encrypts his message using AES
integer_value = user1.sharedSecret
byte_value =
   integer_value.to_bytes((integer_value.bit_length()
   + 7) // 8, byteorder='big')
encryptedMessage = aes_encrypt(Message, byte_value,
   modes.ECB())

# User 1 signs his message using his RSA private key
Signature = RSA_sign(encryptedMessage,
   user1.RSAPrivateKey)

# User 1 sends his encrypted message and signature to
   User 2.

# User 2 receives the encrypted message and decrypts
   it.
# Note: User 2 can decrypt using the shared
   Diffie-Hellman key which is associated to
   byte_value.
decryptedMessage = aes_decrypt(encryptedMessage,
   byte_value, modes.ECB())
print("Received the message from User 1: " +
   str(decryptedMessage.decode()))
```

```
# User 2 verifies the authenticity of the message by
   checking the RSA signature.
verified = RSA_verify(encryptedMessage, Signature,
   user1.RSAPublicKey)

if verified:
    print("This message was indeed sent by User 1!")
else:
    print("This message was not verified!")

# User 1 wants to send a message to User 2.
Message = "Hi, User 2! Excited for Christmas?"

# User 1 encrypts his message using AES
integer_value = user1.sharedSecret
byte_value =
   integer_value.to_bytes((integer_value.bit_length()
   + 7) // 8, byteorder='big')
encryptedMessage = aes_encrypt(Message + "GRINCHED",
   byte_value, modes.ECB())

# User 1 signs his message using his RSA private key
Signature = RSA_sign(Message, user1.RSAPrivateKey)

# User 1 sends his encrypted message and signature to
   User 2.

# User 2 receives the encrypted message and decrypts
   it.
# Note: User 2 can decrypt using the shared
   Diffie-Hellman key which is associated to
   byte_value.
decryptedMessage = aes_decrypt(encryptedMessage,
   byte_value, modes.ECB())
print("Received the message from User 1: " +
   str(decryptedMessage.decode()))

# User 2 verifies the authenticity of the message by
   checking the RSA signature.
verified = RSA_verify(encryptedMessage, Signature,
   user1.RSAPublicKey)

if verified:
    print("This message was indeed sent by User 1!")
else:
```

```
    print("This message was not verified!")


import time

def RSA_sign_with_timestamp(message, private_key):
    # Include the current timestamp in the signature
    timestamp = time.time()

    # Convert the string message to bytes
    message_bytes = message.encode('utf-8')

    # Concatenate the message and timestamp as bytes
    combined_data = message_bytes +
       str(timestamp).encode('utf-8')

    # Hash the combined data
    hashed_message = hash_message(combined_data)

    # Create the signature
    signature = exponentiation(hashed_message,
       private_key[0], private_key[1])

    return signature, timestamp

def RSA_verify_with_timestamp(message, signature,
   timestamp, public_key, validity_period=3600):
    # Verify the timestamp
    current_time = time.time()
    if current_time - timestamp > validity_period:
        return False  # Signature is considered
           invalid if the timestamp is too old

    # Convert the string message to bytes
    message_bytes = message.encode('utf-8')

    # Concatenate the message and timestamp as bytes
    combined_data = message_bytes +
       str(timestamp).encode('utf-8')

    # Hash the combined data
    hashed_message = hash_message(combined_data)

    # Verify the signature
    decrypted_signature = exponentiation(signature,
       public_key[0], public_key[1])
```

```
    return hashed_message == decrypted_signature

# User 1 wants to send a message to User 2.
Message = "Hi, User 2! Excited for Christmas?"
encryptedMessage = aes_encrypt(Message , byte_value,
   modes.ECB())
# User 1 signs his message using his RSA private key
   WITH A TIMESTAMP!
Signature, Timestamp =
   RSA_sign_with_timestamp(Message,
   user1.RSAPrivateKey)

# User 1 sends his encrypted message and signature to
   User 2.

# User 2 receives the encrypted message and decrypts
   it.
# Note: User 2 can decrypt using the shared
   Diffie-Hellman key which is associated to
   byte_value.
decryptedMessage = aes_decrypt(encryptedMessage,
   byte_value, modes.ECB())
print("Received the message from User 1: " +
   str(decryptedMessage.decode()))

# User 2 verifies the authenticity of the message by
   checking the RSA signature AND TIME STAMP.
verified =
   RSA_verify_with_timestamp(decryptedMessage.decode('utf-8'),
   Signature, Timestamp, user1.RSAPublicKey)

if verified:
    print("This message was indeed sent by User 1 and
       is within the validity period!")
else:
    print("This message was not verified or is
       outside the validity period.")
```