# CS5173 Project Report

Victor Sandi, Josh Hamilton, Connor Corbin, Yuki Zheng

*Gallogly College of Engineering, University of Oklahoma*
*Norman, Oklahoma, United States of America*

*CS-4173/5173 Computer Security*

***Abstract*— This document gives a report on the project for our Computer Security class. This report analyzes the problems overcome in this project and details the design choices made to solve them.**

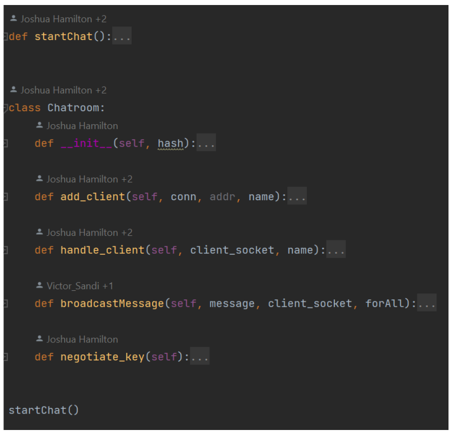
***Keywords*— AES, Diffie-Hellman, Tkinter, Microsoft Azure, P2P, Chat Room**

## I. Project Description

The goal of this project was to design a secure instant messaging app that can allow Alice and Bob to communicate through point-to-point messaging. The messaging should work as follows: Alice and Bob share a password with one another. Then Alice and Bob use the password on the messaging app to connect to each other and begin messaging. A key is then generated on both Alice and Bob's clients. This key will match on both clients to follow symmetric-key cryptography. Once this key is created, Alice and Bob can encrypt their messages before sending, and on receiving a message, they can decrypt the message using the shared key. As an additional requirement, the key must be updated periodically. The key must also be at least 56 bits in length. Another requirement is for repeated messages to produce different resulting ciphertexts. Therefore, the challenges to overcome in this project are to establish a way to form a connection over the internet using a shared password, a way for the same periodically updating key to be generated for Alice and Bob, an encryption/decryption method for the messages, and a graphical user interface (GUI) which the application integrates.

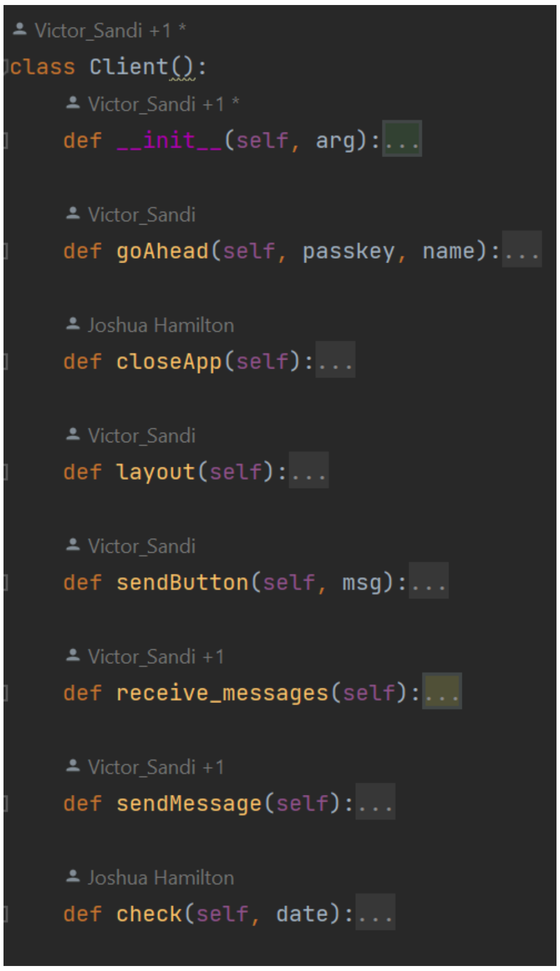
## II. Design Architecture

The design of the project was simple, we had a main client code and a main server code. This is because despite making a P2P server we wanted to make sure that the clients would be able to connect. Below will be two figures showing the architecture of the program for the Server then client code respectively. In Fig 1 and 2.



```
Joshua Hamilton +2
def startChat():...


Joshua Hamilton +2
class Chatroom:
    Joshua Hamilton
    def __init__(self, hash):...

    Joshua Hamilton +2
    def add_client(self, conn, addr, name):...

    Joshua Hamilton +2
    def handle_client(self, client_socket, name):...

    Victor_Sandi +1
    def broadcastMessage(self, message, client_socket, forAll):...

    Joshua Hamilton
    def negotiate_key(self):...


startChat()
```

Fig. 1  Shows the server side architecture

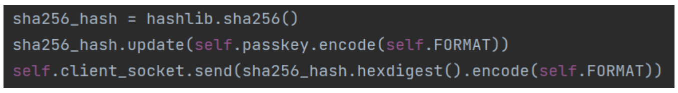


```
Victor_Sandi +1 *
class Client():
    Victor_Sandi +1 *
    def __init__(self, arg):...

    Victor_Sandi
    def goAhead(self, passkey, name):...

    Joshua Hamilton
    def closeApp(self):...

    Victor_Sandi
    def layout(self):...

    Victor_Sandi
    def sendButton(self, msg):...

    Victor_Sandi +1
    def receive_messages(self):...

    Victor_Sandi +1
    def sendMessage(self):...

    Joshua Hamilton
    def check(self, date):...
```

Fig. 2  Shows the client side architecture.

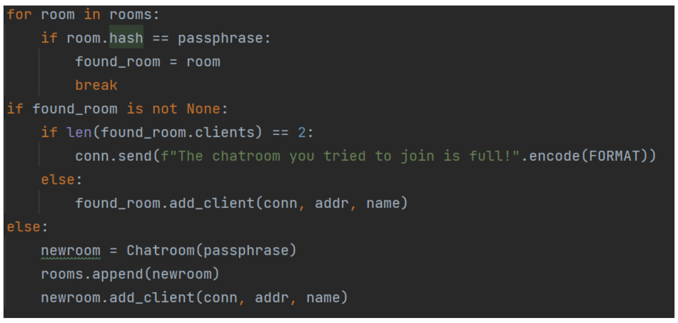
## III. Passphrase

The goal of the passphrase is to ensure that Alice and Bob join the correct room with each other. This is their form of authentication as they trust the mutual rendezvous server. This means that when they input their passphrase, the server can connect them to each other as long as the passphrase is the same. This is further secured by a sha256 hash. The passphrase is hashed on each client instance and given to the server. This is secure as an attacker would not be able to know the passphrase since it is hashed while being transferred to the server. The passphrase is compared to a list of other hashed passphrases on the server side. Once another passphrase is found by the server the two clients are connected to each other. Below in Fig 1. We will show an example of the code from the Client Side.



```
sha256_hash = hashlib.sha256()
sha256_hash.update(self.passkey.encode(self.FORMAT))
self.client_socket.send(sha256_hash.hexdigest().encode(self.FORMAT))
```

Fig. 3  Shows how passphrase is hashed in client code.



```
for room in rooms:
    if room.hash == passphrase:
        found_room = room
        break
if found_room is not None:
    if len(found_room.clients) == 2:
        conn.send(f"The chatroom you tried to join is full!".encode(FORMAT))
    else:
        found_room.add_client(conn, addr, name)
else:
    newroom = Chatroom(passphrase)
    rooms.append(newroom)
    newroom.add_client(conn, addr, name)
```

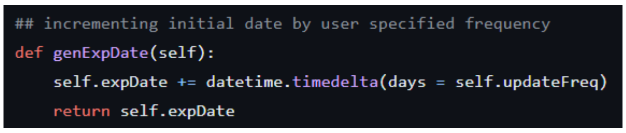
Fig. 4  Shows how passphrase is handled in the server code

In Fig 1 this code can be shown to hash via the hashlib library of python using a sha256 hash on the top of the picture. In Fig 2 we see  some server-side code that handles how rooms are found for users. In the server, when a client joins, it creates a room for them and adds it to a rooms list. When another client joins, it keeps their hash as well, and if two of the hashes are the same, that implies that both of those users had agreed on a passphrase prior to connecting. This means that we can generate a shared room and connect them in a new Chatroom object. As per the description, this is meant to be P2P and only for people like Alice and Bob. Thus, we reject any further connections to any Chatroom object if there are already 2 people in the room. This is shown by the line of code that says “The chatroom you tried to join is full.” Further examples of this functionality are found in Section V. GUI.

## IV. KEY SHARING

For key sharing, we chose to implement the symmetric Diffie-Hellman protocol in our code. The users, Alice and Bob, will generate a random value on their clients, a and b. Each user use this value to generate an intermediate value on their random values using the equations: **a* = g^(a)mod p** and **b*=g^(b)mod p**, where p is a public prime number sent by the server, and g is a public primitive root of that prime number. The users will share their intermediate values, and from them, they can find the same key value using a similar equation, but substituting g with b* and a*. For example, Alice can use the value Bob shared **(b*)** to calculate the same value of the key x through **x = (b*)^a mod p**. When Bob does a mirrored calculation of **x = (a*)^b mod p**, the values will match and the two users can message each other securely.

The key management protocol we implemented to update the shared key periodically uses an expiration date. A class was created to keep track of the initial date the key was generated and the frequency rate for which the expiration date must be updated. This ExpirationDate class then sets the expiration date for the key with this specified frequency as the length of time the date is valid. A function can be called to determine if the expiration date has been reached. This design enhances security as it prevents reuse of a leaked key by a client. In general, periodically updating keys is an aspect of good key management practices.



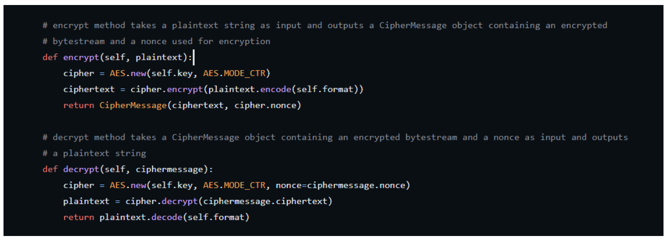
```
## incrementing initial date by user specified frequency
def genExpDate(self):
    self.expDate += datetime.timedelta(days = self.updateFreq)
    return self.expDate
```

Fig. 5 Code for using the datetime module to update the expiration date

In the main function, the check function will check for the key status in two different ways: every time a user sends a message, or every time a message is received. The user should not be able to do either of these actions if the key used for encryption and decryption has already expired. Once the key has expired, it is disabled by being randomized, forcing the user to relaunch the chat lobby to dynamically generate a new key.

## V. MESSAGE ENCRYPTION/DECRYPTION

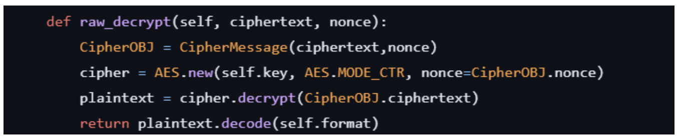
Our code implements an AES (Advanced Encryption Standard) algorithm for encrypting and decrypting messages (plaintext strings). We imported Python's AES module from Crypto.Cipher to implement this encryption and decryption function. The AES Algorithm itself uses CTR mode since it is parallelizable and secure. The encryption method takes the plaintext string (the message) and encrypts it using the AES function, returning a CipherMessage object with the encrypted output and the nonce (salt) used for the encryption. This salt is vital to achieving the defined objective of encrypting the same plaintext input to different ciphertext outputs. The decryption method takes the CipherMessage object containing the encrypted bytestream and nonce (salt), decrypting using the same key and mode. Thus, the plaintext is returned. Again, the salt is necessary to ensure correct decryption.



```
# encrypt method takes a plaintext string as input and outputs a CipherMessage object containing an encrypted
# bytestream and a nonce used for encryption
def encrypt(self, plaintext):
    cipher = AES.new(self.key, AES.MODE_CTR)
    ciphertext = cipher.encrypt(plaintext.encode(self.format))
    return CipherMessage(ciphertext, cipher.nonce)

# decrypt method takes a CipherMessage object containing an encrypted bytestream and a nonce as input and outputs
# a plaintext string
def decrypt(self, ciphermessage):
    cipher = AES.new(self.key, AES.MODE_CTR, nonce=ciphermessage.nonce)
    plaintext = cipher.decrypt(ciphermessage.ciphertext)
    return plaintext.decode(self.format)
```

Fig. 6 Code for AES algorithm encrypt and decrypt functions

Another decrypt functionality offered in the AES Algorithm takes the ciphertext and nonce directly instead of from a CipherMessage object. This function also returns the plaintext.



```
def raw_decrypt(self, ciphertext, nonce):
    CipherOBJ = CipherMessage(ciphertext,nonce)
    cipher = AES.new(self.key, AES.MODE_CTR, nonce=CipherOBJ.nonce)
    plaintext = cipher.decrypt(CipherOBJ.ciphertext)
    return plaintext.decode(self.format)
```

Fig. 7 Code for AES algorithm raw_decrypt function

## VI. GUI

The goal of the GUI is to act as a front-end for the chat messaging app. It needs to allow the users to log in using a chosen name and their shared password. There also needs to be a window that handles the chat function. The GUI for our program was written using Tkinter which is the standard Python library to generate basic GUIs. When the program is run, the below Passphrase Entry window pops up. This window allows users to type their name and password. They will then be signed in with their entered information, joining the relevant lobby their passphrase is connected to.

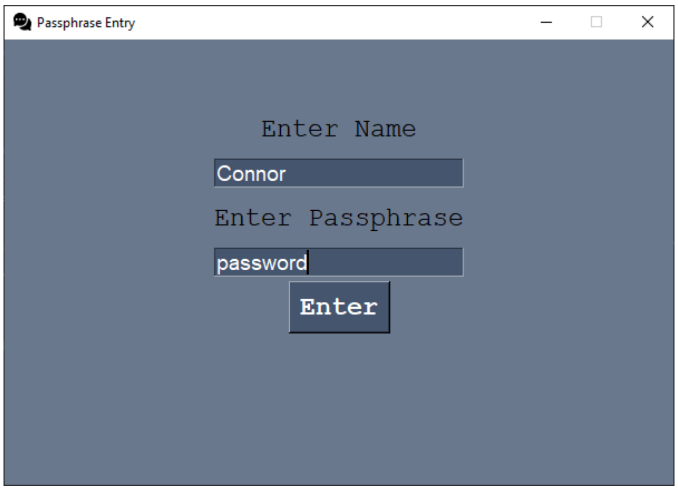
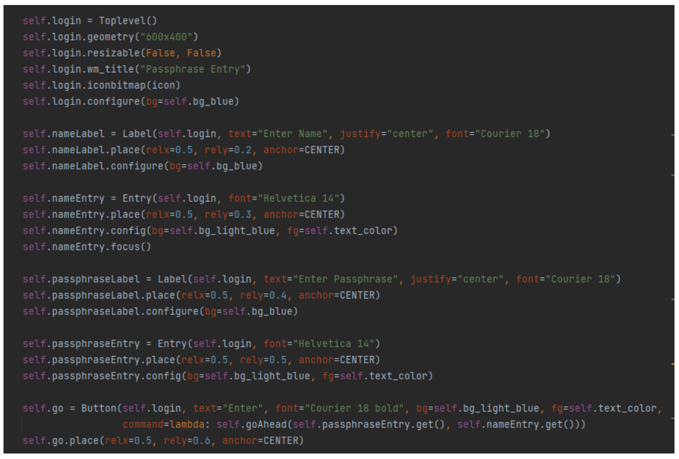


Fig. 8 Shows the GUI for the Passphrase Entry window

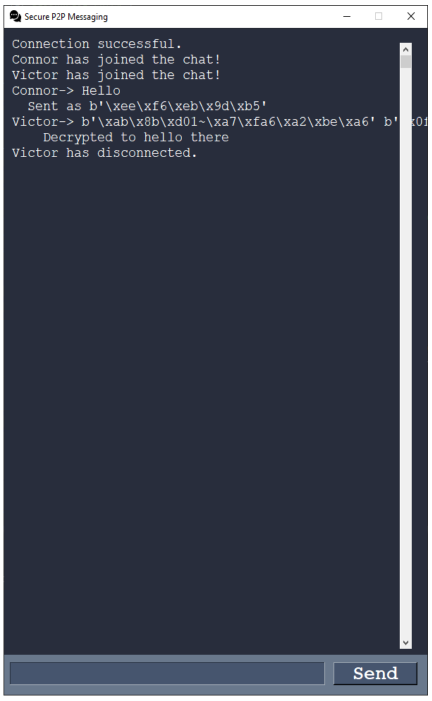
This window is created by creating a TopLevel() object titled Login. Then, the window is populated with the text "Enter name" and "Passphrase Entry" by creating two separate labels for each. The boxes to enter text information are created by using Entry objects. Finally, a button is created and given a function that grabs the values in each entry box and stores them.



```
self.login = Toplevel()
self.login.geometry("600x400")
self.login.resizable(False, False)
self.login.wm_title("Passphrase Entry")
self.login.iconbitmap(icon)
self.login.configure(bg=self.bg_blue)

self.nameLabel = Label(self.login, text="Enter Name", justify="center", font="Courier 18")
self.nameLabel.place(relx=0.5, rely=0.2, anchor=CENTER)
self.nameLabel.configure(bg=self.bg_blue)

self.nameEntry = Entry(self.login, font="Helvetica 14")
self.nameEntry.place(relx=0.5, rely=0.3, anchor=CENTER)
self.nameEntry.config(bg=self.bg_light_blue, fg=self.text_color)
self.nameEntry.focus()

self.passphraseLabel = Label(self.login, text="Enter Passphrase", justify="center", font="Courier 18")
self.passphraseLabel.place(relx=0.5, rely=0.4, anchor=CENTER)
self.passphraseLabel.configure(bg=self.bg_blue)

self.passphraseEntry = Entry(self.login, font="Helvetica 14")
self.passphraseEntry.place(relx=0.5, rely=0.5, anchor=CENTER)
self.passphraseEntry.config(bg=self.bg_light_blue, fg=self.text_color)

self.go = Button(self.login, text="Enter", font="Courier 18 bold", bg=self.bg_light_blue, fg=self.text_color,
                 command=lambda: self.goAhead(self.passphraseEntry.get(), self.nameEntry.get()))
self.go.place(relx=0.5, rely=0.6, anchor=CENTER)
```

Fig. 9 Code for the Passphrase Entry Page

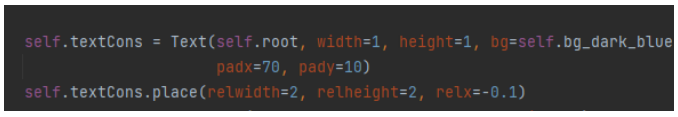
The second portion of the GUI is a window that can be used to display the chat messages. It needs to display all messages whether sent or received, who they were sent by, and the corresponding ciphertext. The window also functions to display when users join the chat and when users disconnect from the chat.



```
Connection successful.
Connor has joined the chat!
Victor has joined the chat!
Connor-> Hello
  Sent as b'\xee\xf6\xeb\x9d\xb5'
Victor-> b'\xab\x8b\xd01~\xa7\xfa6\xa2\xbe\xa6' b' x0f
    Decrypted to hello there
Victor has disconnected.
```

Fig. 10  Shows GUI for the client chat room.

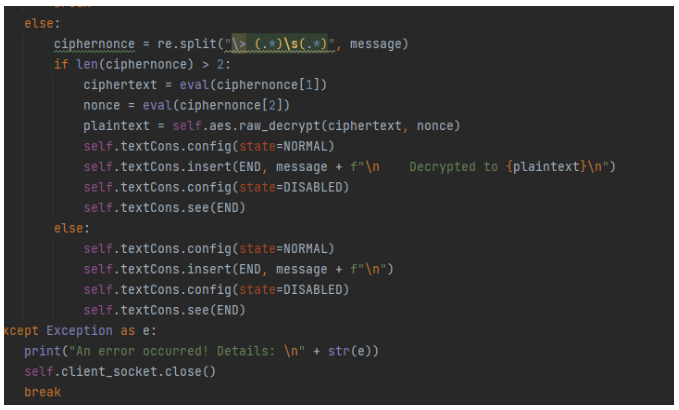
An example of the window running and performing all these different features is shown in Fig. 8. The window also has an entry box on the bottom and a send button which work similarly to the entries and buttons in the Passphrase Entry window. The text being displayed is handled by a Text object. This object is created in the initial initialization of the Text object and then text is added when messages are received and sent. This is then updated with the receive methods in the program so that when a message is sent, it displays differently for each user. In this case, the requirement states that for a sender, the window should show the ciphertext, and for the receiver, it will show the cipher text as well as the decrypted plaintext. This means that the client code can handle how to display messages to fit this requirement.



```
self.textCons = Text(self.root, width=1, height=1, bg=self.bg_dark_blue
                     padx=70, pady=10)
self.textCons.place(relwidth=2, relheight=2, relx=-0.1)
```

Fig. 11  Code that creates the initial Text object

Fig. 9 shows the code that initializes the Text object, and Fig. 10 shows the code that acts to update it when receiving a message. The received ciphertext and the appended salt are then separated. Then using the salt, ciphertext, and the key the plaintext message is formed. Now that the plaintext is acquired it is inserted into the Text object alongside the ciphertext with salt which then updates the window.



```
  else:
      ciphernonce = re.split("\> (.*)\s(.*)", message)
      if len(ciphernonce) > 2:
          ciphertext = eval(ciphernonce[1])
          nonce = eval(ciphernonce[2])
          plaintext = self.aes.raw_decrypt(ciphertext, nonce)
          self.textCons.config(state=NORMAL)
          self.textCons.insert(END, message + f"\n    Decrypted to {plaintext}\n")
          self.textCons.config(state=DISABLED)
          self.textCons.see(END)
      else:
          self.textCons.config(state=NORMAL)
          self.textCons.insert(END, message + f"\n")
          self.textCons.config(state=DISABLED)
          self.textCons.see(END)
xcept Exception as e:
  print("An error occurred! Details: \n" + str(e))
  self.client_socket.close()
  break
```

Fig. 12 Code that receives messages and adds to Text Object

## VII. PROJECT TAKEAWAYS

One major takeaway that we had while working on the project was the difficulty that comes with transmitting data. Both setting up the initial connection and ensuring the data was transmitted correctly were surprisingly tricky processes. When setting up the initial connection, we were initially trying to form a direct connection between both Python clients. This, however, was much more difficult than we anticipated. It would have required the usage of automatic port-forwarding and some other network features to fully implement. Instead, we ended up developing a system where a rendezvous server sets up all connections and organizes interactions between the clients. For transmitting data, there were a lot of challenges to overcome with synchronizing data types. The client and server side programs must send and receive the right data type for strings to display properly and integers to mathematically function correctly. These strings and integers had different protocols for being sent as data over the internet. Additionally, all data sent must be encoded and decoded using the same format, otherwise packets would be interpreted as the wrong characters.

Over the course of the project, several significant bugs arose as a result of the new technologies we were working with. Some examples of these errors were type conversion errors between strings, bytes and ints, networking connection errors using sockets and threads, and transmission errors with receiving data in individual messages instead of one long string. Implementing certain computer security algorithms and project requirements given the restraint of sending messages over threads/sockets via the server was also challenging.

Overall, the project was a useful learning experience. It was interesting to see how the different concepts demonstrated throughout the class apply to a simple real-life scenario. Generating a shared key using the Diffie-Hellman algorithm, securely encrypting and decrypting messages using AES, and transmitting data via IP addresses and ports were all topics covered in class that came to life throughout this program.