Group members: William Bak, Nicholas Laboy, Michael Carlucci, Christian Lizaso, Prasanth Yedlapalli, Alex Podolsky

Instructions: These are just instructions necessary to run the cyrptogame:

Our program uses numpy in one of the ciphers to store a matrix of information. If you do not have numpy installed, please use 'pip install numpy' in order to make full use of our program. Also note this will not work for python versions earlier than python 3.

Open four terminal windows and navigate to the directory where the folder is stored. Once inside the folder run 'python server.py' in one terminal to start the server, 'python UserProfile.py' in two windows(one will be Alice/the first user and prompted for inputs mentioned later), and finally 'python AttackerProfile.py' for Chuck.

When using Chuck, be aware the program will quit once Chuck enters 'n' when asked if more information is needed. This means Chuck needs to be restarted but is mostly a product of our issues implementing Chuck, explained throughout the following two sections of the report.

Design:

Client: The client's[UserProfile.py] only purpose is to generate random sentences based on the lists inside the gen_random_sentences method and try to connect with the server using network sockets. We choose to go with the automated version of the client rather than manual inputs.

Server: The server[Server.py] is the control center of the project, as it handles sending messages between users, encryption and decryption, as well as giving keys to the user when necessary. When connecting to the server, each user gets their own thread in order to make communication possible and distributes a key depending on which key exchange protocol the user chooses out of SKE and PKE. The server also handles calling the encryption and decryption methods listed in the 'Ciphers' section whenever messages are exchanged between users. We accomplish the actual messaging exchange by using a global buffer called buf_ciphertext, which when added to signals to the system that current user has sent a message and is going to wait until receiving one. This type of exchange repeats once the bots stop or the users are manually stopped.

A step-by-step of our design is as follows:

1. Server accepts the first user and creates a thread for them, server then prompts the first user for encryption key type and encryption method. Finally first user sends message before server waits or switches to next client

2. Second user also gets their own thread and if another person has already connected but not sent a message, the server will keep spinning through the recv() call until the server sends something back.

3. First user will send a message to the server if they have not already, the server will then store the message as plaintext into one list and ciphertext into two others, one of which being buf_ciphertext to signal the system to switch.

4. Server decrypts the newest message in buf_ciphertext and sends it to the second user, at this point the second user is able to send their message.

5. Second user sends a message to the server which is stored in the 3 buffers mentioned in step 3, signals the server to switch threads and decrypt the newest message in buf_ciphertext for the first user.

6. Repeat steps 3-5 until the users or the server is manually shut down or the client runs out of input.

Chuck: We designed Chuck to be able to work with the 4 required attack modes, which include ciphertext-only, known plaintext-only, chosen ciphertext, and chosen plaintext. Each mode provides Chuck with varying outputs:

· Ciphertext-only: Chuck can input 5 strings which are stored in the server

· Known Plaintext: Chuck gets 5 pairs of ciphertext and plaintext messages from the server and stores them in his own lists.

· Chosen Ciphertext: Chuck can input his own string and ask the server to give back the encrypted ciphertext (using the first user's key).

· Chosen Plaintext: Similar to Chosen Ciphertext, except Chuck receives back the decrypted ciphertext(again using the first user's key).

After the user who works with Chuck decides what attack mode they would like to use, the user can keep requesting for information as needed. All this information is stored in two different lists, one for ciphertext and one for plaintext, and are frequently updated based on the attack role.

Additionally, Chuck also has access to brute force and frequency analysis as a part of his toolset. We wanted to include more except our Ciphers proved difficult to solve with either of those tools, as well as having both tools be useful within the constraints of how the server assigns keys to users depending on the encryption method. Because of this, we decided Chuck could win if he guessed the first user's key, even though in practice Chuck would be required to solve for all keys in order to win against the system.

Key Generation: Because some of our ciphers need specific types of keys or combinations of keys, we needed to tailor our key generator accordingly. The LaBoy cipher uses two numeric keys while Porta and PlayFair use string keys.

Ciphers:

· Porta Cipher:

- This cipher generates a 13x26 table by writing out the second half of the alphabet (N-Z) in the first 13 columns. Each row of these columns is the previous row shifted to the left by 1 letter each time. The other 13 columns are written out with the first half of the alphabet (A-M), shifted to the right by 1 letter each time it goes down the rows. The table rows correspond to the key letters in pairs (Row 1 corresponds to A and B, Row 2 corresponds to C and D, etc.), and the columns correspond to the plaintext letters. The cipher copies the key multiple times until it produces a string that is of equal length to the plaintext to be encrypted. Then it looks up in the table, key letter for row and plaintext letter for column, and generates the encrypted text based on the table values. Plugging this encrypted text back into the algorithm with the same key decrypts the message.

· PlayFair Cipher:

This cipher was invented in 1854 by Charles Wheatstone and was greatly utilized by the British to encrypt and send secret messages during World War I and World War II. The cipher consists of creating a 5x5 matrix. The first x cells in the matrix are filled with the x unique letters that are present in the given key. Then the rest of the matrix is filled in with the remaining letters of the alphabet excluding j since there are only 25 total cells and 26 total letters. Once the matrix is built and populated, the cipher takes the message given by the user and splits it into pairs of letters. All non letters and the letter j are ignored during this process and simply written the same way in the ciphertext. If there is an odd number of letters, then a Q is placed at the end of the word to ensure every letter is in a pair. The cipher then takes the pairs of letters and encrypts them in the following way, if they are both in the same row in the matrix, then they are substituted for the letter directly to the right of them, if they are in the same column then they are substituted for the letter directly below them. If the letters are neither in the same column nor the same row, then we form a rectangle out of the two letters in the matrix, and the other two corners that complete this rectangle are what we substitute our two letters for. This results in a substitution cipher that is very difficult to break by hand since multiple letters can encrypt to the same letter.

· LaBoy Cipher:

- This cipher takes in the message to be encrypted and then performs a shift cipher of a random key in the range from 0-10,000. After the shift has been implemented on the message, the shifted message is then turned into a shifted number from their corresponding ASCII numbers. This then outputs the string of the original message but has been shifted using the random key from the shift cipher and the ASCII shift cipher. The decryption then takes the encrypted message and the keys that we used to shift it and then gets the original message. It first uses the shift cipher decryption with its key from the encryption and shifts it back. It then uses the key from the ASCII shift encryption to then revert it back to its original message.

Evaluations:

Testing Server and User: When testing the server and user, we started off by just trying to connect one user and having them send a message to the server. With one user trying to connect, we encountered minimal issues and noted we should probably have the client send something to the server, similar to a private key but more a means of identification, in case other people used the same port number and connection protocol as ours. Although our message is the simple string "hello" it should be thought of as a pseudo distinguishing key, since in practice this kind of key simulates a returning user who has gained trust with the system and would probably be more complicated than our “hello” string.

Once we felt a singular user could connect to the server without issue, we began to try connecting a second user as well and having the server exchange messages between the two. This was when we ran into the bulk of our issues, namely how to ensure asynchronous behavior in the threads. In our case, we ensure asynchronous behavior by making the first user to connect a special case and allow that user to pick the encryption key and method used as well as restricting the second user from sending a message before the first user. We know this is not necessarily the most practical way to create a ‘chat room’ for users, but in our case by giving the first user extra privileges we get around an issue where the socket would be flooded with text from both users. With these restrictions we are basically simulating the first user being the superuser/chat owner, and the second user only has permission to join established chat rooms. Once we felt comfortable with how our server and users interact, we began to incorporate the ciphers and key generator. From there we wanted to include a way for more than two users to be connected to the system, however we were unable to do so since that would have made our system far more complicated to work with and keep asynchronous while maintaining the feel of a normal chat room.

Although the solution sounds simple on the surface, certain edge cases would be very hard to implement for. In our case we figured keeping the users in order would not be difficult, nor would sending the messages to each user, however this is under the assumption that all users will join consecutively rather than the more realistic case of some users may be present from the beginning while others join after several messages have been exchanged. This became an issue mainy when it came to implementing Chuck. Basically, if two users were to join from the beginning and send 4 messages between them we would know messages 1 and 3 were from the first user and messages 2 and 4 were from the second user. If a third user were to join at this point, we would need to also book-keep when the new user joined for indexing purposes in our buffers. This becomes an issue because our users each get their own key when the PKE option is selected, so for Chuck to even have some chance at solving the ciphers, Chuck would at minimum need to be receiving either the ciphertext or plaintext from one user only rather than any random string from the system. If Chuck were to try to query the system for a string to be randomly chosen, there was no other way of guaranteeing the key Chuck has to guess remains the same.

Thus we chose to forego having more than two users, as well as limiting the messages Chuck can query for to come from the same user in order to give Chuck a chance.

Testing the Ciphers: When testing the ciphers we decided to test them independently from the rest of the system in order to ensure that both the encryption and decryption methods in our three ciphers led to expected and checkable ciphertexts in the case of encryption and plaintext in decryption. Essentially each cipher would take in a test string entered manually(either through the terminal or hard-coded in) and the output was verified manually for both encryption and decryption to ensure validity. Once each cipher was validated separately we also tested our key generator with the ciphers to ensure the keys were generated properly and correctly depending on which cipher is being used. Finally, we incorporated the ciphers along with the key generator into the server in order to test how our functioning client/server system would work with encryption and decryption.

This part took little revision and was more a matter of calling the correct functions with the right parameters when necessary since each module was tested prior to being tested together. Essentially the system handles all encryption and decryption, while the users mainly just send plaintext to the server and receive decrypted plaintext from the other user in return. Additionally, we decided to store the messages between users in the system(both the ciphertext and plaintext versions) in order to give Chuck messages when he queries the system during his attack.

Testing Chuck: In order for Chuck to succeed, Chuck would have to successfully guess the key that is used to encrypt/decrypt messages that are stored on the server, effectively giving Chuck full access to chatroom users' messages.

The first problem we ran into was determining how Chuck would be able to check whether his key is correct. As previously mentioned, this issue is because of the way our system assigns unique keys to each user, rather than using one singular (but uniquely constructed) key that would have been further from the goal of this game. So for the sake of trying to mimic a secure chatroom type of environment, we noticed quickly both our implementation for the key exchange and the ciphers themselves would make it difficult to even guess the key.

To try and combat this issue, we initially thought that asking the server to check the key would be a good solution, but this is not something that would happen in real life as even in this fictional scenario, a server should never compromise and let Chuck know if the key is correct. This rules out brute force as a tool to determine the key because Chuck will never be able to determine whether those guessed keys are correct. Another solution we wanted to try was to give Chuck a modified version of each of our ciphers, but this again is not something that parallels a real life scenario as an attacker should never have access to cipher methods directly, only through an oracle. This possible solution also led us to another conclusion that Chuck would require a very complex tool to try and break our ciphers, since they each have unique requirements for the type of key they require.

Our biggest issue though was we were unsure of how much control Chuck had over the hacking process with respect to the user as well as how to handle helping users deal with our ciphers specifically without giving them too much help. We found this issue when trying to implement a translation dictionary for the frequency analysis using the percentages found by [insert citation here] but after trying it ourselves, it proved to not be helpful towards solving the key, and usually mapped letters incorrectly, which may be due to our ciphers not being susceptible to frequency analysis.