Q1: To run question 1:

g++ server.cpp -o server -lssl -lcrypto && ./server

g++ client.cpp -o client -lssl -lcrypto && ./client

This assignment question requires client.cpp and server.cpp. The configurable values are defined as preprocessor directives at the beginning of each program. Client.cpp has MINIMUM\_TIME 1 second and TIMEOUT\_VALUE 5 seconds. Server.cpp has P\_LENGTH 2 bytes, MINIMUM\_TIME 1 second and TIMEOUT\_VALUE 5 seconds. Please edit these values and rerun the make file to try different values.

The client connects to the running server and sends messages to the server. After, receiving a message called “Anand”, from the client, the server will generate two random numbers from “dev/urandom”. A 128-bit character array is filled and represented as R. Since the length of P is configurable, we need to read from “dev/urandom” at P\_LENGTH bytes and then clip the char array according the specified P\_LENGTH. Since the send API of C++ Sockets will not allow some of the randomly generated characters to be sent, we need to encode them in hex and decode them as well. hex\_encoded() and hex\_decoded() functions were implemented and both the client and server have access to them. R and P are encoded and then concatenated with a “%” to represent a space. Finally, this hex string is sent to the client. The client will use the “%” to split the received string into Rhex and Phex. The client will then decode Rhex and try to create a string in the form R + Y + R, where Y is another randomly generated 128 bit char array (that is converted to a string). The client will then keep calculating the SHA 256 Hash of such string until the hash value starts with Phex. Upon calculating such a value, the client will encode Y and transmit a 384-bit string in the form Rhex + % + Yhex + % + Rhex. Finally, on the server’ side, the received string will be decoded and validated. If the decoded string does not start with R, have a length of 48 (48 bytes \* 8 = 384 bits) or result in a sha256 hash value that starts with Phex, the server immediately closes the client socket connection. Another integral part of this question was the timing aspect. For instance, for both the client and server, the way I checked whether the client was calculating the hash and sending the computed challenge too soon, or whether the server was receiving and processing the computed challenge too quickly was:

time\_t startTime = time(NULL);

while(1) {

if (time(NULL) < startTime + MINIMUM\_TIME) {

// for server, break, for client, sleep until minimum time is met.

}

…

}

Similarly, to force a timeout on the client side used while (time(NULL) < startHashTime + TIMEOUT\_VALUE), for the while loop that was calculating the has values. Forcing a timeout on the server side (if the challenge is not received within a certain time) was more different. The server was stuck at the recv() function in read\_packet(), until coherent data was sent. However, if the client is still calculating the challenge, then no data will be sent. Hence, I used the select() function to force a timeout on the read\_packet() function (subsequently returning a string that said “Timeout Exceeded”) after the TIMEOUT\_VALUE was exceeded.

Q2: To Run (on eceubuntu): g++ client2.cpp -o client -lm && ./client2

This assignment question requires client2.cpp. For this question, I iterated through every lowercase letter in the alphabet and send it to the server on eceubuntu. I use the rdtsc to measure the time for each response. I try each response 100 000 times. This is defined as a preprocessor narrative. It can be changed to speed up or slow down the cracking process, but as the value decreases, accuracy is reduced. I take the mean of all the attempts and calculate the highest mean for each character. I noticed that the character with the highest mean is the correct character. The way the program is set up, it accepts an input from the user and then tries to find the next character in the password. The correct way to use it is to run the executable, press enter during the first prompt (sending an empty string) and wait for the program to pick the first character. FY!, we start with user2. Then exit the executable and run the executable again. This time run the program and provide the first character when prompted. The program should then give you the second character. I output the MEAN of all the characters. Other information such as variance and std deviation is commented out in the code. You can uncomment it if you want. I also output whether this character can be picked with 95% confidence or not.

This is basically finding the margin of error for the highest mean and seeing if there is any overlap between it and the margin of error of any other mean.

Margin of error = Za/2 \* σ/√(n). Za/2 = the confidence coefficient, where a = confidence level, σ = standard deviation, and n = sample size.

I then see if the upper bound of the second highest mean is less than the lower bound of the highest mean. If it is, then we can say that the letter is correct with 95% confidence. If it is not, then we cant make that claim.

The major design decision that I made in this question was to store the letter’s mean, and its upper and lower bound in a struct. The structs were stored in an array[26], and the array was sorted on the mean attribute in descending order.