Implementation of an Unconditionally Secure Public Key Exchange

Ryan Olsen

2019

1 Introduction

This paper details a bit string sending protocol, and the steps taken to implement its computer program. The programs that this paper will discuss are based upon a theoretical cryptographic scheme created between Mariya Bessonov, Dima Grigoriev, and Vladimir Shpilrain [1]. This type of cryptographic scheme is founded upon the idea of trying to create a key exchange that does not rely on computational hardness assumptions. Many current key exchanges utilize factorization of large integers or the discrete logarithm problem as the basis of encryption, as our computers at the present have no way of quickly solving these problems to attack the schemes. However, in the event that one day someone develops a powerful enough quantum computer, these types of schemes would quickly be broken. Thus, this form of post quantum cryptography does not rely on these hardness assumptions, and instead uses unconditionally secure public-key encryption; the trade-off for circumventing these assumptions is that there is a very slim probability that the legitimate parties involved in the exchange do not end up with the same shared key.

The programs that this paper will look at, as with most things, still have a lot of room for improvement. Places where code can possibly be revamped or replaced will be pointed out. Examples of this are the inclusion of potential data structures, or ways to automate certain results. Changes that occurred during the course of creating the programs shall also be explained, as to point out why one solution may be favored over another. Furthermore, as the programs are written in Python, certain methods will be explained, as well as compared to potential implementation in C++.

As one last note, notice that the tone of this paper lacks formality typically found in mathematical writing. This is for a number of reasons. First, this text should serve as a "cookbook" or step by step guide on a possible solution to programming the protocol in the source paper. Ideally, a novice coder should be able to read through this paper and have a grasp on how to make a program of this nature, and one should be able to quickly understand the flow of steps taken without getting bogged down by the details. Additionally, there will be parts of the code that need to be explained effectively, and this can require dropping the formality in favor for the more conversational tone found in certain introductory computer science texts or forums online. And finally, there is no real need for a formal tone in this paper. This paper is not focused on the proofs or theory covered in the source reference, and is instead only focused on the computer science aspect, providing an explanation for the process.

For full transcripts of the programs with notes, please see [2].

2 Bit Transmission

The public key transport protocol is focused on the transmission of bits between two parties in the presence of an adversary. The protocol is made up of two components: the first is where one party, Bob, acts as a receiver and is able to modulate his probability of correctly receiving a bit, which is transmitting from the sender, Alice. Bob's probability, denoted by q, is dependent on a number of parameters, but is ultimately modulated by his private parameter σ of the normal distribution of his public key. The second part of the scheme involves Alice attempting to determine q, which will then allow her to derive σ as the two parties' secret shared key. This section will focus on the first portion of the protocol, beginning with walking through how the protocol works, and then following along the code for the respective program. The goal of this section is that Alice will be able to transmit to Bob a secret bit c that he will receive correctly with probability q privately modulated by him and Alice. The basic parameter n is the length of an interval, and will form the basis of where the protocols "take place." Namely, Alice will be labelling sections of this interval with bits, and Bob will be retrieving points from the labelled intervals.

Although this protocol has Alice as the sender and Bob as the receiver, the proposed protocol actually begins with Bob as the one to first produce a bit string as a public key. This bit string Bob produces is created as follows: Bob selects a private parameter value σ at random from the set $\{0.3n, 0.4n, 0.6n, 1.5n\}$. He then selects an integer b, uniformly at random, from the interval $[0, n-1]$, and uses that b to generate a value B that is normally distributed with parameters $(b,\ \sigma)$. If the point B is not located in the same interval $[0, n-1]$, or it is exactly the midpoint of this interval, he discards the two points b and B; else, Bob records the pair (b, B). He repeats this generation until he has $m = n^2$ pairs of points, and then enumerates them into a sequence

$$(b_1, B_1), (b_2, B_2), \ldots, (b_m, B_m).$$

Bob stores this sequence as part of his private key. He then converts the sequence into a bit string as follows: if B_i is to the right of the midpoint of the aforementioned interval $[0, n-1]$, he puts a "1" at position i of the string; if B_i is to the left he places a "0". Finally, Bob will select, uniformly at random, an integer $t \in \{1, \ldots, m\}$, which is the number of indices of bits that will be changed as an added layer of security. Thus, t bits in the string are randomly selected and changed (i.e., a bit "1" is changed to "0" and vice versa), and Bob records the indices of these t bits that were changed as part of his private key. Bob publishes this altered bit string.

To begin sending her bit over to Bob, Alice selects her private parameter p from the set $\{0.2, 0.3, 0.7, 0.8\}$. Alice then selects a secret bit, and for simplicity here let us assume she picks this via a coin flip. This bit will indicate which interval Alice selects: if she chooses a 1, she selects the interval $[0, B_i]$, and if she selects a 0 for her secret bit, then she chooses the interval $[B_i, n-1]$. It is important to note that she does not know the point B_i. She labels her interval with her secret bit c with probability p, and instead labels the interval with the bit $1 - c$ with probability $1 - p$.

Yet, how does Alice know which bit to transmit if she cannot see the point B_i? The bit that Bob published for a given i value carries information with it. When the bit is a one, Alice knows that Bob's point B_i is right of the midpoint, and thus the interval on the left of Bob's point is longer than that of the right. Alice's secret bit determines which interval she chooses, so when her chosen interval is the longer of the two, Alice sends her secret bit correctly, else she changes it, doing so with probability p (with probability $1 - p$, she will instead change her bit when her interval is the longer of the two). Alternatively, since Alice would only see a bit, this process can be described in terms of bits. So, if Alice chooses the left interval $[0, B_i]$ for example, then her secret bit is 1. Whenever she encounters a bit 1 from Bob's string, she will send her bit correctly, and change the bit when she comes across a 0, again doing so with probability p. Finally, after Bob receives the labelings, he looks at the interval that b_i lies in. If this bit is one that Bob had changed previously, he now changes this recovered bit to the opposite. After this process is done a number of times, he tallies the number of times b_i is in the correctly labeled interval, and calculates q as the number of correct labels out of the total.

Now that the protocol has been established, the coded program that carries this procedure out this will now be inspected. Following from Section 3.1 of the source paper, this program also finds the ranges of q for specifics values of σ and p so they can be placed into a table. Either party could use this program to determine the ranges, as Bob needs to be able to modulate q via sigma, and Alice needs to build her table of ranges for the second step of the protocol, thus emulating what Bob would be calculating. For now, this program does not convert Bob's pairs into a bit string nor does it change values at a number of the indices. However, the result that Bob gets after he changes the bits back mimics the results he would get had he not changed the bits at all, as it does not influence the probability. Since this program allows for user input for any positive values for p and σ, this program works well for testing purposes, allowing the appropriate sets to be chosen for these variables so that the ranges of q do not conflicts so that searching up a specific q in the table does not produce two different results. User input is received with the following code:

```
inputP = ""
while True:
    try:
        inputP = float(input('Enter input: '))
    except ValueError:
        print("Invalid float")
        continue
    else:
        break
```

This takes in input from the user, and then attempts to convert this input into a float object, and will continue to prompt this until such a value is supplied, at which point it breaks from the loop. This set up is used for p and σ, followed by asserting that the values are positive. It would also be beneficial to restrict these values to a certain range, such as the maximum and minimum values for each set, but this will not break the program as the negative values would, so it does not necessarily have to be checked.

Next, Bob can start building his pairs. This process is fairly straightforward: first, Bob instantiates the list of pairs, then by the protocol outlined above, builds the (b_i, B_i) pairs, rejecting pairs where B either lies outside the interval or is equal to the midpoint, until he has m number of them.

```
pairs = list()
while len(pairs) < m:

        b = random.randint(0, n - 1)
        B = random.normalvariate(b, sigma)

        if ((1 <= B <= n-2) and (B != (n - 1) / 2)):
            pair = (b, B)
            pairs.append(pair)
```

Finally, Alice needs to send bits, so that Bob can retrieve his points b in order to calculate q. This block of code will be seen again in the next protocol as well, although then the cases will be split apart, implemented as a bit string, and include Alice's random choice of secret bit. Note that here, Alice has her bit fixed as 1.

```
counter = 0
for i in range(N):

        b, B = random.choice(pairs)

        # Out of intervals [0, B) and (B, n-1],
        # label the larger interval with bit 1 with probability P,
        #  and label the larger interval 0 with probability 1 - P.

        if random.random() < P:
            # longer of the intervals is labeled 1
            if ((B < b <= n - 1) and (B < (n - 1) / 2))
            or ((0 <= b < B) and (B > (n - 1) / 2)):

                # Bob retrieves b from interval where bit is 1
                counter += 1

        else:
            # shorter of the intervals is labeled 1
            if ((B < b <= n - 1) and (B > (n - 1) / 2))
            or ((0 <= b < B) and (B < (n - 1) / 2)):

                counter += 1
```

```
q = counter / N
```

Bob first initializes his counter. Then, Alice chooses a random i value, and finds Bob's corresponding bit, as indicated by

$$b, B = random.choice(pairs).$$

Alice will attempt to either correctly mark the interval with probability p, or mark it incorrectly with probability $1-p$. That is to say, she either labels the longer interval with a one with probability p, or marks the shorter interval with a one with probability $1-p$. To emulate this probability, random.random() is used to generate a real value between 0 and 1. Bob then needs to retrieve his point b from whichever interval is labeled one. When this is the longer interval, it is checked if either B is left of the midpoint AND b is right of B, or B is instead to the right of the midpoint, with b to its left. For the shorter interval, this is similar. The furthest nested if statements in this block of code check for these conditions, and adds to the counter of successful retrievals if so; Bob then calculates his q by dividing his counter by the number of attempts N. Running this whole protocol for 100 trials would then produce the maximum and minimum values of q for a given p and σ.

While this program only calculates a singular maximum and minimum q for a given p and σ, it would be far more beneficial to have this code loop through a list of given values and automate this process. Furthermore, someone using this would have to physically write down the maximum and minimum values of q for each pair of p and σ. This leads itself for further implementation improvements. One could set up a self writing tree that writes maximum and minimum values to a file, with each "p" branch split off into separate "σ" branches, each of those branching to a maximum and minimum value. Whenever a new value is encountered for p or subsequently σ is encountered, a new branch would be made, and if a particular q value for an existing branch would be a new maximum or minimum, old value would be overwritten, making the process fully automatic. With this tree, the values could be directly transferred to the next program without the need of some programmer hard coding the values into a table and updating them every time there is a change. Yet, there are two potential pitfalls to look out for: one would include the "looping" through both sets of values, which in itself would take $O(h^2)$ time and make the program run much longer. The other would occur in the self writing tree, which if left to run for too long, outlier values for q might overlap with others from a different σ. Since the next program searches for the first instance of a q value, this could potentially lead to consistent wrong guesses for σ. Accounting for these pitfalls would allow the possibility of such an automated process.

Another possible update is the use of a linked list. This type of data structure involves the use of a node and a pointer in order to store values in a sequence. Typically, the benefits of a linked list include the ability to update the pointers in order to change the sequence, as well as using dynamic storage that grows and shrinks with input size. The downside is that one loses the ability to access values at random indices, and instead one would have to step through the entire list to get to a certain node. For this application, the upsides of using a linked list does not outweigh the effort to encode it. Use of dynamic storage is not necessary, as our data does not need to grow and shrink, and the computer will allocate enough space for the long bit strings during compilation. Furthermore, while most of the time it is not necessary to not access random bits in the string (except for in the next program where Alice chooses her secret bit) and instead loop through the whole list, it is far quicker and easier to understand being able to access the bits via indices rather than a node's data members. Even so, it remains a possibility for the program.

3 Bit String Protocol

As mentioned before, the desired goal of this second protocol is that Alice determines an approximation of q. With the previous program and her table of values, Alice would then be able to find the corresponding σ and become the two parties' shared secret key. The previous program sets the framework in place for this bit string protocol, as the bit string transmission involves sending bits one at a time, utilizing this first program as a subroutine. Even so, the program to be discussed is independent and does not require the previous protocol in order to work. All that is required from the the prior scheme is the mechanism to transmit bits so that Bob recovers them correctly with probability q, but there may be other schemes that set this up as well. Additionally, most of this program can be done offline, only requiring to be online when bits are transmitted and Bob sends the difference in transmitted bits μ.

Alice begins this protocol by generating a random bit string of a large length R offline, and we let $k = Q_1 - Q_0$, with Q_1 as the number of 1 bits in the string, and Q_0 as the number of 0 bits. It is necessary that $|k| < \frac{1}{2}\sqrt{R}$, and Alice should start over of this is not the case. Now online, Alice transmits this bit string, one bit at a time, using the previously described protocol above (including Bob's public key), so that Bob receives the bit with probability q. Bob records $\mu = Q'_1 - Q'_0$ for the altered string he receives, and sends, online as well, this μ value back to Alice. Alice then can use μ to find q with the formula

$$q \approx 0.5 + \frac{\mu}{2k}.$$

With her precomputed table, Alice use q and looks up the corresponding value of σ for her chosen p; now both parties have the shared secret key *sigma* offline.

The implementation of this protocol is far more automated, randomly selecting a p and σ value from each set. The beginning of this program is similar to the last, as Bob needs to create a large bit string as his public key. However, since σ will be randomly selected for Bob, he will need to produce a list of pairs (b_i, B_i) for each value of σ, and then convert this into a bit string. Python's dictionary object, dict(), can store each pair list, and another dictionary can be used to store the converted bit strings. The following code shows the similar nature to the previous pair creation, with these added changes:

```
pairs_dict = dict()
pairs_to_bits = dict()

for val in sigma_set:
    # Initialize empty lists
    pairs_dict[val] = list()          # List used for pair values
    pairs_to_bits[val] = list()      # List used for bits

    # While the number of pairs in the library is < m,
    while len(pairs_dict[val]) < m:

        # (i) Selecting b_i
        b = random.randint(0, n - 1)

        # (ii) Selecting B_i
        B = random.normalvariate(b, val)

        # (iii) Dropping B_i that lie outside the interval [1, n-2]
        if 1 <= B <= n-2 and B != ((n - 1) / 2):
            pair = (b, B)
            pairs_dict[val].append(pair)

            # Create Bit String:
            # If B is greater than the midpoint,
            if pair[1] >= (n - 1) / 2:
                pairs_to_bits[val].append(1)     # Add a 1 to string
            else:
                pairs_to_bits[val].append(0)     # Else add a 0
```

The initialization of the empty lists, pairs_dict[val] = list (), allows us to access each list by the index of σ, such as pairs_dict[0.3 * n]. Yet, Bob needs to change certain bits as detailed in the creation of his public bit string. Bob will create a copy of the strings, and change bits in the copies, and we will refer to these edited strings as satellite strings. The satellite string will need to carry many different pieces of information, however. For each σ value, the respective satellite string contain the string itself, the number of bits that will be changed, and which indices were changed. All of this information can be stored via the use of a class. Below, these three pieces of information are in respectively placed in self.string, self.number, and self.indices. Here is the template used to store this information:

```
class SatelliteString:
        # Note: class must be given an args to make functions work

        def __init__(self, *args):
                self.string = args
                self.number = random.randint(1, m + 1)
                self.indices = random.sample(self.string, self.number)
                self.positions = [0] * m

        # Call to change random indices of the satellite bit string
```

```
def change_and_save_places(self):

    for g in self.indices:
        self.string[0][g] = (self.string[0][g] + 1) mod 2
        self.positions[g] = 1
```

Notice that there is an extra data member position. For ease of coding, a bit string of equal size to the satellite string is also created, which stores a value 1 at position i if the satellite string bit was changed at position i. It is possible to loop through the indices that were changed, produced via self.indices = randmon.sample(...), in order to eventually change the bits back, but a secondary string would instead step through both the satellite string and self.positions at the same time with the same index. The stored indices are random values of indices stored in a list, and changing the bits this way would require random data access, which would not be possible if the strings happened to be coded as a linked list. Either method would work, yet the one chosen for implementation mimics how Alice's bits are naturally distorted, looping through the whole string and changing the bits with certain probabilities. Also above, there is a function included above, change_and_save_places, that walks through the indices list, and changes bits at the specified locations, and then store which locations these were in the bit string self.positions.

After the specific σ and p values are chosen, the core of this protocol can begin. Alice must make a bit string so that

$$k = Q_1 - Q_0.$$

There are two ways to achieve this. Originally, Alice would randomly select a bit, via a coin flip, and append it to the string. That way, k is naturally derived, resulting from whichever bits are added. However, the problem with this bit string is that looping a large number of times and adding a bit to the string takes a very long time. Furthermore, k needs to be within a certain range, and if it is too small, Alice needs to start over with a brand new string. Since the outcome of k cannot be controlled in this manner, there will be many times where k is too small, and this slow bit string creation will start all over again. Instead, Alice should randomly select an integer k instead. Then, Alice should create a bit string of zeroes, and then, using the same random.sample() function from the satellite string distortion, change a number of bits to 1, so that the formula for k holds. Because

$$k = Q_1 - Q_0,$$

and

$$R = Q_1 + Q_0,$$

via substitution it can be shown that the number of bits should be

$$Q_1 = \frac{k+R}{2},$$

and

$$Q_0 = R - Q_1.$$

Below is how this second method can be implemented. Alice first randomly selects an integer for k, and ensures that k is even so that Q_1 will be an integer. Then, via a coin flip, k will either be positive or negative. She creates the "empty" bit string, and uses random.sample() to select Q_1 number of positions where the corresponding value at each position will be switched to 1.

```
k = random.randint(int(math.sqrt(R)), R / 2)

          # Guarantees k is even
          if k mod 2 == 1:
               k += 1

          # Coin flip to decide if k should be negative
          flip = random.randint(0, 1)
          if flip == 1:
               k = k * -1

          # Initializes an empty bit string
          bit_string = [0] * R

          # Q1_holder is the number of 1's randomly placed in string
          Q1_holder = int((k + R) / 2)

          # Selects Q1_holder number of indices to be changed to 1
          position_list = random.sample(range(R), Q1_holder)
          for position in position_list:
                   bit_string[position] = 1

          Q1_original = sum(bit_string)
          Q0_original = R - Q1_original
```

Once Alice has her bit string created, the transmission of bits strings can begin. Alice selects her secret bit via the following: she selects a random index from Bob's string, and whichever bit that index holds, her secret bit shall be the opposite (again, this bit reflects which interval Alice chose). Then, Alice transmits the bits similar to the last program, now as a string of bits instead. She looks at the Bob's bit in position i, if it matches her secret bit, she sends the bit from her randomly generated string as is, and if it does not match her secret bit, she changes the bit in her string, both with probability p. With probability $1-p$, she will either change or keep the bit based on the opposite condition: if it matches, change the bit, else keep it. Then, Bob will retrieve the bits as before based on how Alice marked the intervals, and change the bits based on which positions were changed in the satellite string. Below documents this procedure.

```
 secret_bit = -1 # Initialize Alice's secret bit to an impossible value
 pick = random.randint(1, m + 1) # Choose a random bit from Bob
 if pairs[pick][1] == 1:   # Alice chooses the interval [0, Bi]
         secret_bit = 0
 else:
         secret_bit = 1      # Alice chooses the interval [Bi, n−1]

 assert (secret_bit >= 0)

 for i in range(R):

          # Probability to do the correct bit transmission
          if random.random() < P:

                 if convert_pairs[i] == (secret_bit + 1) mod 2:

                        bit_string[i] = (bit_string[i] + 1) mod 2

            # 'Incorrectly' transmit bit
            else:

                 if convert_pairs[i] == secret_bit:

                        bit_string[i] = (bit_string[i] + 1) mod 2

# Bob's retrieval of b
for i in range(R):

        # 0 < b < B and Left interval chosen
         if (pairs[i][0] < pairs[i][1]) and (bit_string[i] == 0):
                bit_string[i] = secret_bit # Bob retrieves Alice's labels

          # B < b < n−1 and Right interval chosen
          elif (pairs[i][0] > pairs[i][1]) and (bit_string[i] == 1):
                  bit_string[i] = secret_bit

          # other two cases:
          # B < b < n−1 w/ Right interval and 0 < b < B w/ Left interval
          else:
                  bit_string[i] = (secret_bit + 1) mod 2

# This code is used to change a satellite string's values back
for i in range(R):
        if convert_pairs.positions[i] == 1:
                bit_string[i] = (bit_string[i] + 1) mod 2
```

Finally, Bob calculates $\mu = Q_1' - Q_0'$ from Alice's distorted bit string, and sends it to her. Alice then uses the formula

$$q = 0.5 + \frac{\mu}{2k}.$$

It should be noted that μ needs to be large so that the value of q is not too close to 0.5, or else Alice cannot retrieve Bob's σ With the value q and her table that was generated during the previous protocol, Alice looks up her chosen value p and finds the range that q lies in, which tells her the value of σ that was used. Now both parties should ideally share the same secret key!

Now that the program has been completed and works as tested, there some improvements that could possibly upgrade the program's efficiency. Such improvements could include storing data better, deciding on a σ value for edge cases, and introducing a more secure form of randomness. In the present version of the program, Alice and Bob actually end up with two bit strings each, one a "true" string and another for its respective satellite string. Bob sends both the string he generates via the pairs (b_i, B_i) as well as the satellite string, and Alice needs to create two bit strings, which are initially copies of each other but become distorted independently during transmission. Yet, using four strings, each with a suggested large R length of 5 million bits seems redundant and a bad use of memory storage. Shortening this to just using the satellite string, and ensuring that everything works as intended would combat this, possibly helping speed up the program. Plus, Bob can simply choose not to change the bits to mimic how it would work without that step. Although, for purposes in debugging, it is handy to be able to look at the results from both of Bob's strings at once.

In terms of obtaining σ, it was previously discussed in the last section to use an automated method to rewrite the ranges. Again, the concern with this approach is that if a certain outlier for a value of q generated via a certain σ happens to overlap with a different range of q from a different $sigma$ value, then the program will only ever check for the first instance of q, which may consistently be wrong. The ranges that are currently used in the program, calculated after about a 100 runs, have "breathing room", so this described problem might not end up being a problem. On the other hand, due to this "breathing room", sometimes q values may lie in the ranges outside those that have already been written, and thus will not return any σ. Currently, the program just counts how often this happens, but in order to genuinely try to use this protocol, there would have to be some approach taken deal with these cases. One could just check the ranges and verify which values q is closest to and choose its respective σ , but if the program is running a great number of times, this would not be feasible. And yet, writing code for the value to "go to the closest range" seems both tedious, and easier said than done. Yet, coming up with such a work around would help improve the accuracy of the outcome, as it would remove the need to disregard certain trials where the computer does not automatically choose σ and instead defaults to 0.

Lastly, both programs make frequent use of Python's random library, as it is used in random number generation, as well as the random.sample() method that it used to pick out a number of indices. However, this pseudo-random number generator used in this library is not entirely secure, so a cryptographically secure pseudo-random number generator should be used in its place. There exists libraries in Python for such cryptographic uses, it would just depend on which of these libraries accomplishes all that the random library does between these two programs. Despite these needed improvements toward a more complete program, the end result is fairly close to what is desired. Out of 100 trials, the same key is derived close to 99 times. In order to truly utilize this result, the probability of successfully sharing the same key should be 0.999 percent. That way, the shared σ key, which functions as a 2-bit key, can be concatenated with 63 other 2-bit keys, to form one 128-bit key; the probability of both parties ending up with the same 128-bit key would become $(0.999)^{64} \approx 0.94$.

4 Conclusion and Next Steps

The success of these two programs working in tandem help to push forward the concept of an unconditionally secure public transport scheme, in the hopes that certain uses for this type of scheme can be found, or that other possible alternatives are produced for the advent of quantum computing. Yet, by no means is this a finished investigation.

On the side of utility, there is much to be added to help turn this into a complete and useable product. Even beyond what was mentioned before about certain data structures or using a more secure random generator, other possibilities exist if these programs were to be practically applied. One major example of this is to make the bit string transmission program work in a network, as opposed to the scripts currently in place; this would help shift towards how the programs can be used in real world scenario, not just to test numbers. This could be achieved through the use of Python's sockets, as data can be sent through a connection between a host and a client. In order to make this work, some knowledge would be required on the various methods that exist for sockets, including how to transmit the data back and forth between the parties, and how to display this data. More or less, the code would still be the same as within the current programs, so this would end up being a learning exercise for computer networks more than about the cryptographic scheme. However, implementing the programs as a network could further investigation on this topic, as it would allow one to see how the data is transferred, how private and public data can be accessed, how an adversary would attempt to attack the scheme in the presence of Bob's decoy keys, and how both parties could achieve the same concatenated 124-bit key along with the experimental probability of successfully doing so.

From a coding standpoint, there is definitely room for improvement. Personally, I started working with this program with only a background in C++, so I had to learn all of Python from scratch to write the programs. Beyond just learning new syntax, Python has its own unique methods and work arounds. There was many points where I would know an approach to a problem in C++, but it required a totally different solution in Python. One example is that C++ allows for header and source files for classes. The first shows the class's member functions as well as data members, while the other ”hidden” source file keeps the implementation of them tucked away. To my knowledge, there is not a way (or at least an easy one) to perform this in Python. The ability to have files interact in this way for C++ directly leads to the networking implementation described above. It turns out that C++ also uses sockets, but being able to call certain pieces of code from separate files makes C++ a better fit for this approach, not requiring a special work around that Python would. C++ can even declare certain values in a class as public or private, whereas Python does not include this as a standard. Also, at one point during the program, I was debugging a certain error that took many hours to find the problem. It turned out that when creating the strings of pairs, the string that converts pairs to bits was being saved as one long string, as opposed to separate strings for every new value of σ. Yet, with C++ and its meticulous syntax, my IDE would have caught the mistake in a heartbeat, saving me from this frustration. Python is very lenient, and objects can be called without a type unlike C++, so this can lead to errors if you as a programmer are not careful. Learning Python was a great exercise and a good instance of self teaching, but sometimes it might be better to stick to what skills you are strongest with. Even so, there may be times in the real world where I am required to work with a new computer language for a certain problem, and being able to start from nothing, teach myself, and build up a brand new skill set in order to overcome such a hurdle is a great skill to have.

In all, despite the hiccups and required future developments to be made, I would mark this as a success. There will always be quality of life improvements that can be made to better a product, yet the key here was to distinguish and prioritize what was necessary to complete a certain build. Since I would be learning and coding these improvements in the presence of a new computer language, even trivial things can ultimately take up a great deal of time. Neglecting these improvements outright, however, would be foolish. Over the course of the build, I came to appreciate how important it is to give the user feedback while running the program. It helps to know if a particular section of code gets hung up on something, or stuck in an infinite loop, or whatever the problem is, yet the user will not know without feedback from the console. I also came to understand how important saving can be while working on a large program. If I was to be reworking a new section, creating a brand new file to work on became necessary, as there is great frustration that occurs when going from a working program to a broken one and not having a save state in place. Many of these realizations came as a result of trial and error, which is key to the learning process. All of these pieces came together in order to make a core final product that allows experimentation of the bit string transport scheme discussed in the paper. As I plan to continue working on the protocol, hoping to add in some of the points discussed throughout this paper, I hope to see how the programs grow and potentially help advance post quantum cryptography and unconditionally secure public key encryption, perhaps one day becoming useful in real-world applications.

References

[1] M. Bessonov, D. Grigoriev, V. Shpilrain, A framework for unconditionally secure public-key encryption (with possible decryption errors), in: International Congress on Mathematical Software – ICMS 2018, Lecture Notes Comp. Sc. 10931 (2018), 45–54.
(Note: This citation is for a former paper on an adjacent subject, as the current paper is not yet in publication)

[2] https://github.com/ryolsen/post-post-quantum