My responsibilities for the project were to implement the public key cryptosystem, Goldwasser-Micali, and the symmetric block cipher, Triple Data Encryption Standard. The Goldwasser-Micali implementation was fairly standard in it's implementation, using a combination of the class notes and wikipedia as sources for its implementation. The Triple Data Encryption Standard had first been built on the backbone of a ToyDES implementation, expanded into the full DES. From there, the full DES was used as the basis to create the simplest form of blockchain using the electronic code book mode of operation, which was then further expanded upon to create an ECB-mode triple DES implementation.

The first step of Goldwasser-Micali was to generate the private and public keys. The first substep of the first step was to generate two suitably large prime numbers. The easiest way to accomplish this was to have the algorithm generate a random odd number of a minimum of keySize bits and then check if that number was prime by using the fermat primality test with a probability less than 1/10000 of being false, with the number 1/10000 being chosen as a compromise between accuracy and the sake of runtime speed. If that number wasn't prime than we added two and checked again, rinse and repeat. Now that we have two suitably large most likely prime numbers, p and q, we use them to generate N from N = p * q. From here, we generate a QNR value (x) such that it's legendre symbols against prime numbers p and q are both -1. This is done to make sure that there is no difference in Jacobi symbols when comparing two or more plaintext. This is easily implemented by simply generating random numbers less than p and q that the equation $x^{(p-1)/2} = -1$ is true. If the generated x didn't pass this test, then we simply generated a new x and tried again. Now we have a private key of (p, q) and a public key of (x,

N). And example set of these numbers from a run can be found below in the format ((x,N),(p,q)).

((1704290068, 19158814513186398367), (3924131527, 4882306921))

Now that the cryptosystem has it's key, we can go through the process of encrypting and decrypting messages. For the sake of our group's cohesiveness and making sure that all of our code worked together, all inputs and outputs from functions were agreed to be strings. That's why the encryption algorithm takes in specifically strings and immediately encodes them into individual bits that we'll call m_i. Then for each of these bits, we generated a random number less than N that we'll call y_i. With these pieces in place, for every bit in the plaintext, m_i we can create a ciphertext number, c_i, by using the equation $c_i = y_i^2 x^m$ (The m here is supposed to be m_i). The ciphertext appears to be a wall of random number, just as it should be.

Now when we want to decipher this ciphertext, first we have to figure out whether or not the number is a quadratic residue of both p and q. This is simply using the Legendre symbol

equation on the number and making sure that the number has a symbol of +1 for both p and q. With this information, we know that if the number is a quadratic residue, that the plaintext bit for the number was 0, and vice versa for 1. Intuitively, the reason that this works is because multiplying a quadratic residue by a non-quadratic residue will give a non-quadratic residue number for it product. Once this is done, we've now retrieved the plaintext, which in this case was: Hello World! I am Luthgar, the destroyer of worlds!!!!

For implementing the full DES, I will note that I got the values for the various permutation boxes and the S-boxes from the Wikipedia article about the supplemental materials for DES. Like Goldwasser-Micali above, the first thing to do is to generate the subkeys by using the key schedule. The key schedule begins by taking in a 64 bit key and shuffling those bits around while simultaneously dropping every either bit in a permutation. The result of that permutation is then split into two halves that have their bits rotated. The rotated halves are then input into another permutation that outputs a subkey. Take the two halves we had previously, rotate them again, input them into the permutation and now we have the second set of subkeys. Rinse and repeat 14 more times and now we have 16 subkeys.

```
>>> for subkey in PermKeys(key):
        subkey.to01()


'010000001010110001100101001000100010000110010011'
'110010001010010000110100101001100010000110000011'
'110001001000011000101110001001100000001101000011'
'111000101001001000100010010101101000000101000010'
'101010001001101001100010010001001000010101001000'
'101000000111001001011010010010001011010001001000'
'001001000101011101010000011010001101010000101000'
'010001100101100101010001000010000101110000101010'
'000011100100000101110001000010000101110100110000'
'100011110100000100011101100010010100100001110000'
'000011110000001110001001110000010100101000010000'
'000110110001100010101001100100010000001000011100'
'100110010010100011001000100100010001001010000100'
'000100000110111010001100000100000010001010100101'
'010100000011110100000100001100100010100010000101'
'010100100010110000100100001000100010000010000111'
```

When going through the process of actually encrypting a block of 64 bit, first the input is put through an initial permutation that shuffles the bits around. From here the output of the initial permutation gets split in half with one half being left alone and the other half being put through a one way function. In this case, the one way function first expands its 32 bit input into 48 bits by using an expanding permutation. It does this so that it can then exclusive or that permutation's output with this round's subkey. Then the 48 bit output of that exclusive or is then split into 8 different 6 bit long segments. Then each of those segments are input into an S-box, that is a 4 by 16 table of nearly evenly distributed values. The first and last bit of the are used to inform which row from the table will be used with the remaining four bits are used to select the column. With the row and column selected, the value in the table that is associated with both the row and column is then output from the S-box. This process gets repeated across all 8 S-boxes and the output from those S-boxes is collected into another permutation that shuffles all of the outputs together.

Now that we're done with the one-way function, the output from said one-way function get's exclusive or'd with the other half of the input. The output of that exclusive or then takes the place of the half of the input that did not go through the one-way function. To clarify, if the half that did not go through the one-way function was called the, "left half," and the half that did go through the one-way function was called the, "right half", then the output of the exclusive or would now be considered the new, "left half." Switch the positions of the, "left half," and the, "right half," and we have just completed a, "Round." Now do that 15 more times and run the output of that through the inverse initial permutation and now we have a ciphertext.

Of course, that only allows us to encrypt 64 bits of information, and even then with a rather weak 56 bit key. So to make this into more useful cryptographic system, we make it so that system can encrypt more than 64 bits at a time by implementing the Electronic CookBook (ECB) mode of the system. This simply means that a longer input will get split up into 64 bit blocks that are all encrypted separately. We know that this is not the most secure decision, but this was a compromise between time constraints and functionality. The end of any message would then have, "ENDMESS," appended to the end of it to signify that the end of the message. This is necessary because the final block of 64 bits in the input would often have to be padded in order to be large enough so it was imperative that we add a mechanic to differentiate the end of a message. Naturally, the decryption code has a function added that truncates both the, "ENDMESS," and everything after it from a plaintext.

Now, to deal with the issue of the weak keyspace of 56 bits, we decided to implement triple DES. The implementation of this was actually rather simple. First, instead of generating one key, we now need to generate three different keys. Now that we have three keys, we can run the DES three times and effectively triple the key space to create a much more powerful 168 bit key. This is accomplished by running the code in encryption mode over the plaintext with key_1 to get $ciphertext_1$. Then we run the **decryption** code over $ciphertext_1$ with key_2 to get $ciphertext_2$. Then we run the encryption code again over $ciphertext_2$ with key_3 to get $ciphertext_3$, which will be our final ciphertext. In order to decrypt this, we simply do the reverse. Run the code in decryption mode over $ciphertext_3$ with key_3 to get $ciphertext_2$. Then we run the **encryption** code over $ciphertext_2$ with key_2 to get $ciphertext_1$. Then we run the decryption code again over $ciphertext_1$ with key_1 to get the plaintext. Honestly, the most important thing about this is that the

DES gets run three times with three keys. The only reason that we bother changing between encryption mode and decryption mode in the triple DES is because it makes the algorithm backwards compatible with normal DES by simply making all three keys the same value.