My final project consists of a C++ program developed through the Xcode IDE provided on my macbook. Xcode uses C++ version “C++ 20” and clang is the default C++ compiler in Xcode. The program implements the Rivest-Shamir-Adleman (RSA) public (asymmetric) encryption scheme as well as the Data Encryption Standard (DES) private (symmetric) encryption scheme. Before, RSA or DES are called in my program, two large prime numbers are first generated. To work with arbitrarily large numbers, I included a header file “BigInt” which supports large base 10 numbers, along with operations, assignment, and some methods. One essential method used in my program, which was provided by BigInt, is the big_random(). Big_random(), accompanied with the Miller-Rabin probabilistic primality checker algorithm, allowed me to select large prime numbers. The program should technically work while choosing numbers (p and q) with thousands of bits. However, from my testing, I found that generating two 256-bit numbers takes something around ten minutes or more. Big_random() must be passed as an argument the number of digits in the random large number to be generated. During my testing, I simply passed big_random() the number 78 to represent a 256 bit number; since 78 is the max number of digits in a 256 bit number.

Miller Rabin’s primality checker algorithm required helper functions including modular exponentiation and a function for calculating two numbers “s” and “r” which can be done using a variation of prime factorization. Modular exponentiation in turn required a helper function to execute decimal (base 10) integer conversion into a binary representation. Within the Miller-Rabin function, the results of the prime factorization function must be passed into the modular exponentiation function. This required that that arguments be passed by reference into the Miller-Rabin function so the values can be altered by both functions and these values can act globally within the Miller-Rabin function. This problem required that I use two separate Miller-Rabin functions for each prime number generation, p and q, and in turn required multiple copies of the corresponding helper functions.

RSA key generation phase is executed in the main function, although it would have been neater to abstract it into its own function. Calculations for RSA are simple and therefore essay to do in a few lines of code. Caluclations for DES, on the other hand, are very complicated and are in fact largely based on confusion and diffusion rather than mathematical calculations. While RSA is based on modular arithmetic, DES uses a series of permutation tables, exclusive or operations, and a key stream to transpose the plaintext in sixteen rounds of encryption. Then decryption is done by doing the same permutations and XOR operations using the reverse key stream.

The DES encryption permutations are encapsulated in a DES function. The key generation phase of DES is encapsulated in a separate function and stores the result of each of sixteen rounds of key generation in an array of keys. The keys are then passed to DES in order 1-16 for encryption and 16-1 for decryption.

The hybrid implementation aspect of the project is done by choosing a random binary string to serve as the master key for the DES key generation function. This master key, when applied to the DES key generation function, will produce a key stream that can be used to encrypt or decrypt messages. This master key is selected by the sender and then encrypted using the RSA public key, which was generated by the receiver. Then the encrypted symmetric master key is sent to the receiver. The receiver uses their asymmetric private key to unlock the master key, then generates the reverse key stream. Now messages can be transferred between sender and receiver.