A narrative report (in pdf format) describing how your program works and what algorithms you implemented. List the major components of the program and explain how they fit together. Describe how you tested the program and how you verified its correctness. Write about the problems that you encountered and how you overcame them. If your program does not work correctly, explain what parts do not work and what parts you believe work correctly. The report should also contain compilation and execution instructions for your program.

1. How your program works

The python program provided works by implementing the standard RSA cryptosystem. It begins by generating two keys, one public and one private. The former consists of two numbers (n, e), where the "module" n is the product of two large primes p and q, and the "exponent" e is a prime hardcoded to equal 65537. The latter, d, is equal to the inverse of e in modulus (p-1)(q-1). These values are all stored in appropriately named text files.

A message m, consisting only of numbers, is then encrypted with the formula

$m' = m^e \pmod{n}$

and stored in a text file. This ciphertext m' is then be decrypted with:

$m = m'^d \pmod{n}$.

2. List the major components of the program and explain how they fit together.

In the file "numbers.py," mathematical operations to be used by the RSA cryptosystem process are implemented. These are:

- ModExp(*b, e, m*): Using the modular exponentiation process described on pages 78 and 79 of the textbook, this algorithm returns the result of b^e (mod m) in $O(2\log_2(b))$ time.
- DecimalToBinaryList(*n*): Returns a list, where each digit in the binary representation of *n* is an entry in the list. This is used only by the ModExp() function, as the ModExp() function, as described in the book, sums congruences which correspond to digits in the binary representation of *n* which are 1.
- GetPrime(*n*): Returns a random prime number of length *n*. This is accomplished by generating random numbers of length *n* with RandIntLength() and checking if they are prime using IsPrime().
- IsPrime(*n, k*): Determines if *n* is prime and returns a Boolean value accordingly. This is accomplished using Fermat's primality test, in which *k* random numbers between 2 and n-2 are generated, and it is checked for each number if it is congruent to 1 when raised to the power *n-1* in mod *n*. If this is ever the case, *n* is determined not to be prime. This function is used by GetPrime() to determine the primality of randomly generated integers.
- RandIntLength(*n*): Using python's random library's randint() function, returns a random integer of length *n*. For example, if a 3-digit number is desired, RandIntLength() calls randint() to return a number between 100 and 999. This is used by GetPrime() to generate prime numbers of a specified length.
- EEAInv(a, b): Returns the inverse of *a* in mod *b* via the extended Euclidian algorithm.
- EEA(a, b): Returns a tuple representing (gcd(*a, b*), *x, y*) as used in the extended Euclidian algorithm, which finds an *x* and a *y* such that gcd(*a, b*) = $ax + by$. The function EEAInv() simply returns x mod b.

In the file "rsa.py," the actual operations involved in the RSA cryptosystem are implemented. These are:

- MakeKeys(): Generates valid values of *e, d,* and *n* to be used in RSA and writes them to their respective text files.
- EncryptMessage(): Performs the operation $m^e \ (mod\ n)$ using values of *m, e,* and *n* retrieved from the aforementioned text files.
- DecryptMessage(): Performs the operation $c^d \ (mod\ n)$ using values of *c, d,* and *n* retrieved from the aforementioned text files.

3. Describe how you tested the program and how you verified its correctness. Write about the problems that you encountered and how you overcame them.

I tested my program with known correct examples of RSA encryption/decryption sourced from online resources which perform RSA encryption/decryption given values of *m, c, d,* and *n.*

The only issue I encountered in my testing was in my ModExp() function. I found that the final decrypted message was incorrect. This, I assumed, would lie either in the encryption function or the decryption function. I determined that it was the encryption function, as the decrypted text was as expected for the incorrect ciphertext. I traced this issue back to ModExp(), which worked correctly when the base was 2, but not for larger bases.

4. Compilation and execution instructions for your program.

As this is a python program, execution is performed simply with the command "python3 main.py" within the scripts' folder.