Sam Holmes

14307915

FCC Assignment 2 Report

1. RSA Algorithm Implementation

My implementation of the RSA algorithm involves a set of methods to produce the values required to create a public and private key that are used in the encryption and decryption process.

1.1 Prime Number Generation

The process begins with the generation of two randomly generated prime numbers that lie within the range 10,000 → 100,000. Since these numbers are quite large, it becomes increasingly difficult to test the primality of a random number that is generated. The Lehman algorithm helps overcome this issue, by determining the likelihood of whether a number is prime or not. I utilised this in conjunction with an iterative call to find a potentially prime number, then repeated the process ten times to determine if it would still be deemed prime. This provides a multiplicative probability function where the probability is given by 1/2^n . For my implementation this would result in a number being prime, accurate to 1/2^10 or 1/1024 which is 99.9% accurate. This could be improved further, by iterating through the function a larger amount of times, for an increase in CPU time.

1.2 Modulus and Totient creation

The modulus and totient are used as part of the public and private key and are generated through simple manipulation of the two prime numbers generated in the previous step. The modulus is found by multiplying the two prime numbers together, and the totient is found by (prime1 - 1) x (prime2 – 1). In my implementation these numbers are stored as BigInteger numbers, as they could potentially exceed the integer & long variable storage capacity.

1.3 Selecting 1 < e < Totient

I needed to create an e value, which is used as part of the public key. It is created by randomly generating a number between 1 and the totient value generated earlier. I had to create a custom random number generator, as the typical random number generators do not allow numbers to exceed the size of a long, and given that our maximum totient could potentially be 100,000 x 99,000, it had to be stored as a BigInteger. This number is generated and then passed into a function called gcd() which will determine the greatest common divider between the random number and the totient. This is part of the extended Euclidean algorithm. If this function returns a 1, it means that the numbers are co-prime with one another, and is a valid number for e.

1.4 Generating d value

The d value of the RSA algorithm is used as part of the private key, and is generated by finding the multiplicative modular inverse of the e and the totient. BigInteger has an in built function for the called BigInteger.modInverse(), however the assignment specified that we implement one ourselves, so I created a function in the RSACrypto class called modInv that provides the modular inverse of the two numbers. This is the other half of the extended Euclidean algorithm.

1.5 Encoding file

Using these values generated by the algorithm, we can encode a message by using exponential modular arithmetic to determine the transmitted message to be sent to the receiver. I do this through using a function and passing it the e and modulus values generated in the previous steps. This creates a BigInteger value that can be sent onwards to be decoded at the other end.

1.6 Decoding file

Similar to the encoding function, we simply take each BigInteger values and use exponential modular arithmetic, this time using the d value instead of the e value to decode the message. This function provides a integer value which corresponds to a character from the ASCII table, which is then type cast to a character to be displayed to the user, or wtitten to a file.

2. Questions

2.1 Question 1

This question asks us to demonstrate the use of the exponential modular arithmetic. I use this in my code to encode and decode my messages in my program.

2.2 Question 3

DSS or Digital Signature Standard works by utilising a public and private key, similar to RSA. These public key parameters can be shared amongst different users, with the private key representing a single computer user. The algorithm requires a prime number between 2^L-1 & 2^L where L represents the length of the key, a multiple of 64. It then requires q, which is a prime divisor of the original prime number created earlier. The private key (x) is a random number generated between 0 < x < q-1, which is used per user. The public key is equal to g^x mod p where g is the result of a hash function (typically SHA-2) provided with p-1/q.

2.3 Question 4

If the greatest common divisor of the two numbers m & n is 1, then they are said to be co-prime, meaning they have no other divisors. The totient is the number of positive integers, less than n that are co-prime to n . If totient(mn) is co-prime with one another, due to the fact that totient is a multiplicative function, it means that when m and n are co-prime, then the totient of the two will be equal to the the multiplication of the totient of the individual parts.