James Alford-Golojuch

CS 378

April 21, 2016

El Gamal System Documentation

The purpose of this project was to create a working El Gamal encryption system in which the system takes input from a user about what message they want to send and encrypt along with desired requirements concerning the size of the prime number p used in the system and the confidence level that the number p is prime. This system also provides a decryption module which decrypts the encrypted message outputting the correct decrypted version of the message.

This system is made up of seven different modules. Of those seven different modules one of them is the main module which calls the different modules that make up the El Gamal system. This main module gets the user input for the desired size of the prime numbers being generated, n, and the desired confidence level that the prime number used is prime, t. It uses this input to call the key generation module of the El Gamal system. This is one of three modules that make up the El Gamal system. This module takes the input from the main module and calls the module for generating random prime numbers. It generates a random number and tests it for primality until a prime number, x, is generated. It then calculates another number p, which is equal to 2x+1. It tests the primality of this number and if it is not prime generates a new x until both x and p are prime. It outputs this prime number p. This prime number module also outputs a primitive number for the prime number p and outputs this back to the key generation module. The key generation module uses these values of p and g along with a random integer x which is used to calculate integer h where h=g^x(mod p). This module then outputs two files, one containing a public key, K1, and the other containing the private key, K2. The public key consists of the values for p, g and h while the private key consists of the values for p, g and x.

The second module of the El Gamal system is responsible for the encryption of a user string message. It reads from the file K1 to get the public key. It converts the string message to a series of integer values after first converting it to a binary string. The integer values for the message needed to be as large as possible in order to save space when storing the cipher text resulting from encrypting the message. The integers are desired to be as large as 2^n which was 200 in this test case. As such knowing a single character is an 8 bit binary number we can see that up to 25 characters can be used to create one of the integer values for the message, z. Once a vector of z values if found for each z value this module creates a pair of big integers which make up the ciphertext. These pairs are stored together on the same line of the cipher text and will later be used to decrypt the message in the next module. This module finally outputs the pairs of every element in vector z into a cipher text file which will be passed on to the next module.

The third and final module of the El Gamal system reads the files for K2 to get the private key and from the ciphertext in order to get the pairs of integers for each vector element in z. Using the private key this module decrypts the big integer values of the pairs in order to recreate the array of integers, z. This array of integers is then sent to the module for modular exponentiation, which is used throughout these modules to do fast modular exponentiation problems. This module also contains a function which takes as input a vector of integers z and outputs a string of bits. This string of bits is then converted to a string of characters in the decryption module and finally this message is output to a file for the plaintext.

As far as any problems I had occur during the creation of this project I didn't really encounter many errors. One of the bigger issues I had involved the random generation of numbers. I had the random numbers being output just to have a look at the numbers being generated as I was testing the module for prime numbers. I noticed that consecutive numbers would be the same random number. I realized this was due to my seed generation for the random number being based upon the system's time and that the clock must not have been updating fast enough that each number was always different. I solved this by multiplying the time with some other variable which in this case was a loop counter which incremented every time a new number was generated and as such meant that each number had a different seed. I was able to notice that now every single random number was different so the problem was fixed. I applied this solution of multiplying a value that changes with time every time a random number needs to be generated in order to prevent similar issues arising. Another problem I had difficulty with was a forgotten line of code in the decryption module where I forgot to take the mod p value for one of the numbers so the integer values z was way off.

I was able to fix this issue by strategically placing different outputs in different functions in order to be able to visually follow the message through the various functions and modules in order to determine which module and which functions were the cause of this error. I also used this strategy in order to test my modules one by one as I finished them. For the prime number testing I would manually put in numbers I knew were either prime or not prime and see whether the tests confirmed this. I also used this method to test whether the primitive roots of a selected prime were being correctly generated. I did this by placing using for prime x=11 and p=23 since 23=2*11+1 and 23 has only a small amount of primitive roots which I was able to quickly check to make sure correct primitive roots were being generated. I performed similar tests throughout each module using inputs that I knew what the outputs should be and comparing what I know they should be to what they were in order to ensure the system was working or to see where the system's flaws were.

One last major problem I had involved the generation of the output files for K1, K2, cipher and plain text. For some reason when I didn't give it an exact folder path I was unable to find the files being generated even though they must have been generated since I was not receiving the error I included for if a file is unable to open. My workaround to this was providing the exact filepath for the file so that it was easily accessible for the modules and for me to check whether the output to those files were what I had expected based on inputs into the modules.