Jonathan Redwine
CS789 - Cryptography
Final Project
Fall, 2019

This document outlines the steps needed to act as each of Alice, Bob, and Eve for both ElGamal and RSA. While many of the programs were written in Python, some were written in C++. The terminal commands that look like "python programName.py" are the commands for running the Python files, and the terminal commands that look like "ProgramName/main" are the commands for running the C++ files.

ElGamal

From within the terminal, navigate to the CS789_Jonathan_Redwine_Final_Project/ElGamal/ directory, and run the following commands.

Alice

"python BlumBlumShub.py"

- This will run the program for randomly generating the order of the group N, and will suggest a generator g that is a safe prime. The program uses the Blum Blum Shub algorithm for the pseudorandom generation, and uses Miller Rabin for determining if the random number is prime. The program will ask for a number of bits that the user wants for the order of the group N. To get a value in the 6-digit range, enter 20 for the number of bits.

"python chooseRandomNumberInRange.py"

- This is for Alice to choose her A value. This is a simple Python function that asks the user for the maximum of the range to choose from, and will give the user a random integer in that range. For the maximum of the range, enter the order of the group N to get Alice's A value. Alternatively, Alice could just manually choose her own A.

"python fastExponentiation.py"

- This will perform a fast exponentiation of the form $base^{exp}$ mod M. Alice should enter the generator g for the base, A for the exp, and the order of the group N for the mod M. The last value of y (the right column) is Alice's g^A value.

Alice will then send three values to Bob: g, N, and g^A . Alice will receive Bob's g^B value back.

"python fastExponentiation.py"

- Alice should not calculate hers and Bob's shared key k. To do this, she needs to run fast exponentiation again to calculate $(g^B)^A$ mod N. This k value is now the encryption key for communication between Alice and Bob.

"python findInverse.py"

- Alice then needs to find the inverse of k in Z_N*. This is essentially just another fast exponentiation where the exponent is N-2 (since N is prime, and according to Euler's little theorem). This k^{-1} value can now be used for decoding messages encrypted with k.

"python modularMultiply.py"

- If Alice now wants to send a message x to Bob, she can run this program to encrypt her message. She should enter her message x, the encryption key k, and the order of the group N. The program will give the user the result of e(x) = x * k mod N.

"python modularMultiply.py"

- Alice can also use this program to decode received messages from Bob. Given an encrypted message e(x) from Bob, Alice can run the program to determine Bob's original message x. She should enter the encrypted message e(x), the decryption key k^{-1}, and the order of the group N. The program will give the user x = e(x) * k^{-1} mod N.

Bob

Bob will first receive three values from Alice: g, N, and g^A .

"python chooseRandomNumberInRange.py"

- Bob first needs to choose his B value. Bob should enter N for the maximum value of the range from which to randomly choose from, and the program will give Bob his B value. Alternatively, Bob could just manually select his B value.

"python fastExponentiation.py"

- Bob then needs to calculate g^B using fast exponentiation. By entering g, B, and N into the program, Bob will be given g^B .

Bob can then send g^B to Alice.

"python fastExponentiation.py"

- Bob should then calculate the encryption key k with fast exponentiation, to determine the value of $k = (g^A)^B \bmod N$. This k value is now the encryption key for communication between Alice and Bob.

"python findInverse.py"

- Bob then needs to find the inverse of k in Z_N*. This is essentially just another fast exponentiation where the exponent is N-2 (since N is prime, and according to Euler's little theorem). This k^{-1} value can now be used for decoding messages encrypted with k.

"python modularMultiply.py"

- If Bob now wants to send a message x to Alice, he can run this program to encrypt his message. He should enter his message x, the encryption key k, and the order of the group N. The program will give the user the result of $e(x) = x * k \bmod N$.

“python modularMultiply.py”

- Bob can also use this program to decode received messages from Alice. Given an encrypted message $e(x)$ from Alice, Bob can run the program to determine Alice’s original message x. He should enter the encrypted message $e(x)$, the decryption key k^{-1}, and the order of the group N. The program will give the user $x = e(x) * k^{-1} \bmod N$.

Eve

Eve will be intercepting the values that are being sent between Alice and Bob. These values are g, N, g^A, and g^B . Eve's goal is to figure out either A or B (or both) in order to determine k. Only one is needed as with A she can calculate $k = (g^B)^A \bmod N$ and with B she can calculate $(g^A)^B \bmod N$.

"python babyStepGiantStep.py"

- The program will ask Eve to enter the order of the group, a generator of the group, and a value g^x , and the program will determine x according to the Baby Step Giant Step algorithm. Eve should enter either N, g, and g^A, respectively, to determine A, or N, g, and g^B, respectively, to determine B.

"python fastExponentiation.py"

- Eve can then use the fast exponentiation program to calculate k. Depending on if A or B was determined in the step above, Eve should enter g^A, B, and N or g^B, A, and N for the base, exponent, and the mod. The program will give Eve the encryption key k.

"python findInverse.py"

- Eve should then find the inverse of k to get the decryption key k^{-1}. She should enter the encryption key k and the order of the group N into the program.

Now that Eve has determined the decryption key k^{-1}, Eve is able to intercept encrypted messages e(x) and decrypt them, determining the original message x. She could even encrypt her own messages using the encryption key k if she wished to pretend to be either Alice or Bob.

RSA

From within the terminal, navigate to the CS789_Jonathan_Redwine_Final_Project/RSA/ directory, and run the following commands.

Alice

"python BlumBlumShub.py" x2

- Alice first needs to generate prime numbers p and q. This program should be run twice, each time generating one of p and q. These will be used to calculate N = p * q. If Alice wants N to be about 6 digits, then she should enter 10 for the number of bits for each of p and q.

"python multiply.py" x2

- Alice then needs to calculate N = p * q, and phi(N). Since N has two prime factors p and q, phi(n) = (p-1) * (q-1). This program just makes this calculation simple to do from the command line, but Alice could also easily do this with a regular calculator.

"python chooseRandomNumInRange.py"

- Alice now chooses an encryption key e. This e needs to be in Z_N*, and also needs to be coprime to phi(N). If e is not coprime to phi(N), then it will be much harder to find the inverse of e (which is done in the next step with the Extended Euclidean Algorithm). Alice should enter N and phi(N) into the program, and the program will give a good encryption key e.

Alice can then send to Bob the order of the group N and the encryption key e.

"ExtendedEuclideanAlgorithm/main"

- Alice then needs to find the inverse of encryption key e. This decryption key d can be calculated using the Extended Euclidean Algorithm (written in C++). The program asks for the two values of which to find the greatest common divisor (which will be 1 in this case). The program will also show the coefficients of the two values that make their linear combination equal to 1. Alice should enter the encryption key e and phi(n) for these values. Depending on if e is entered first (the program calls this value x) or second (the program calls this value y), the coefficients will correspond differently. The coefficient a corresponds to y, and the coefficient b corresponds to x. The decryption key d is the coefficient that corresponds to e.

“python modularMultiply.py”

- This is only necessary if the decryption key d is not in the range $1 < d <$ phi(N). If d is outside this range, Alice can find the equivalent within the group by running this program and entering d, the number 1, and phi(N).

Alice can now make the order of the group N and her encryption key e public. Other members of this communication can encrypt a message x to her by calculating $e(x) = x^e \bmod N$.

“python fastExponentiation.py”

- When Alice receives an encrypted message e(x), she can decrypt the message with this program. The original message can be determined by calculating $x = e(x)^d \bmod N$. By entering e(x) as the base, d as the exponent, and N as the mod, the program will give the user the original message x.

“python fastExponentiation.py”

- Alternatively, Alice may wish to send a mass message to all members of the communication network. To do this, she can actually encrypt her message x with her personal decryption key by calculating $e(x) = x^d \bmod N$. To do so, she can enter her message x, her decryption key d, and the order of the group N as the base, exponent, and mod for the program. She can send this encrypted message to everybody in the communication network, which can be decrypted be everyone who has her encryption key e.

Bob

Bob will first receive from Alice the order of the group N and her encryption key e.

“python fastExponentiation.py”

- Bob only needs to run this program to fast exponentiate his message x to encrypt for Alice. With this program, he can enter his message x, the encryption key e, and the order of the group N for the base, exponent, and the mod to calculate e(x) = ex mod N. Bob can then send this encrypted message to Alice.

“python fastExponentiation.py”

- Alternatively, Bob may receive an encrypted message e(x) from Alice that was encrypted with her decryption key and sent to everyone in the communication network. The original message can be determined by calculating $x = e(x)^e$ mod N. By entering e(x) as the base, e as the exponent, and N as the mod, the program will give the user the original message x.

Eve

Eve will intercept from Alice the order of the group N and her encryption key e.

“python pollardRho.py”

- Eve can try to factor Alice’s N using this program. The program asks for the number to factor N, and uses Pollard’s Rho Algorithm to find factors p and q.

“python multiply.py”

- Now that Eve knows p and q, she can calculate phi(N) = (p - 1)*(q-1). This can be done with a regular calculator, but this program makes it easy to calculate at the command line.

“ExtendedEuclideanAlgorithm/main”

- Now having the encryption key e and phi(N), Eve can use the Extended Euclidean Algorithm to find the inverse of encryption key e. The program asks for the two values of which to find the greatest common divisor (which will be 1 in this case). The program will also show the coefficients of the two values that make their linear combination equal to 1. Eve should enter the encryption key e and phi(n) for these values. Depending on if e is entered first (the program calls this value x) or second (the program calls this value y), the coefficients will correspond differently. The coefficient a corresponds to y, and the coefficient b corresponds to x. The decryption key d is the coefficient that corresponds to e.

Eve now has both of Alice’s encryption key e and decryption key d. Similar to as described above with respect to Alice, Eve can read encrypted messages sent to Alice (encrypted with her public key e), and also pretend to be Alice and send messages encrypted with her private key d.