Encryption, a Key Aspect of Electronic Communication

Isabella Phung

January 2023

1 Introduction

Encryption is the only way that we can safely communicate across the internet, without it, there's little point in trading information when anyone can read your messages. This document describes the basic pseudocode and structure of a library written to generate keys, encrypt, and decrypt files based off of the Schmidt-Samoa algorithm. The README.md file contains information on how to compile and use this library while the WRITEUP.pdf file describes some of the difficulties and the approximate process of how this library was written.

2 Key Generation

Create primes p and q via random selection. Size of p should be within [bits/5, (2*bits)/5], this will ensure pq will be at least equal to n in size. With the size of p and q, start generating p and q values until you get primes that aren't identical and p is not a factor of q-1 and q is not a factor of p-1. Create n = p * p * q. compute gcd(p-1, q-1) in order to find lcm(p-1, q-1), because lcm(a, b)= (a * b)//gcd(a, b). We'll call this lcm lambda. All of these aspects make up the public key.

Now using these values, we create the private key. We compute (inverse n)mod(lambda)p*q = d.

3 Encryption function

Block size k = floor of ($\log_2$(sqrt(n)-1)/8). Front of each block has 0 prepended to it to shift indices up to start from 1. A block will be an allocated array of type uint8_t *(char). Until end of file, while less than k-1 bytes, put byte in block. These values will be converted into a number using mpz_import() from the GMP library. ss_encrypt() will be applied to this value. The outputted number will be written to the outfile as hexstring then newline.

<u>ss_encrypt() details:</u>

Computes ciphertext by encrypting plaintext using the public key. ciphertext = ((plaintext)ˆ(n))mod n where n is the public key which was computed from hidden primes p and q.

4 Decryption function

Much like the encryption function, the ciphertext must be decrypted in blocks. Calculate k = floor of ($\log_2$(p*q-1)/8). Why pq instead of n? There's no need to calculate the exact amount blocks, we can instead over estimate it a bit using p*q instead.

Iterate over the file, scan in a hex string, use ss_decrypt to convert this cipher text back into plaintext, store in block. use mpz_export which will change the long string into smaller values.

Write the values, starting from index 1 to outfile.

ss_decrypt details:

Decrypt ciphertext into plaintext using the private key d and n (which is p*q). plaintext = ((ciphertext)ˆ(d))mod n where n is the public key computed from the hidden primes p and q.

5 Executable Structure and Pseudocode

The basic structure of the program itself is quite simple.

Keygen.c:

- check command line options
- prep files for keys (ensure permissions are correct and exit if error)
- use randstate_init() to generate a random number
- feed into ss_make_pub() and ss_make_priv() to generate the public and private keys
- get user's name
- write keys to files
- output information if verbose option is enabled
- clean up any GMP values and close files.

Encryptor:

- check command line options
- open public key file and read

- print username and public key if verbose option enabled
- use ss_encrypt_file() to encrypt file
- close file and cleanup any GMP values

Decryptor:

- check command line options
- open and read private key
- print pq and d if verbose is enabled
- decrypt using ss_decrypt_file()
- file and cleanup any GMP values

6 Number Theory Pseudocode

gcd(g, a, b): store b in temporary value, put a mod b in b, put temp value in a. Continuously searches for greatest factor a and b share.

mod_inverse(o, a, n): set r to n, r' to a. Set t to 0 and t' to 1. while r' isn't 0, q = floor(r/r'), r is r', r' is r-q*r', t is t', and t' is t-1*t', loop. If r is greater than 1, there's no inverse and 0 put in o and returned. if t is less than 0, then t = t+n. return t.

pow_mod(o, a, d, n): v is 1, p is a. while d greater than 0, if d is odd, v is (v*p) mod n. p is (p*p) mod n, d is floor(d/2), loop. put v in o, return o.

is_prime(n, iters): s is greater than 0, r is an odd value greater than 0. These values must be selected such that n-1 = (2^s) * r. For the number of iters, choose random a between 2 and n-2, this is our witness. y is the power mod of a, r, and n. If y doesn't equal 1 and doesn't equal n-1, j is our iteration variable and starts at 1, while j is less than or equal to s-1 and y doesn't equal n-1, y equals the power mod of y, 2, and n, if y equals 1, the value isn't prime. iterate j. Loop. If y doesn't equal n-1, the value isn't prime. Return true.

make_prime(p, bits, iters): Generates a random number stored in p of a given length (bits). It then calls is_prime to check if the randomly generated value is prime. If it isn't generate a new random number and try again, otherwise return the generated number.

7 Function Glossary

There's a lot of different functions in this library some that seem to have an overlapping purpose, this section lists and quickly describes their purpose.

Modular Inverse: Only numbers coprime to C have modular inverses to C. Ex: ($A * A^{-1}$) mod C = 1

Coprime/relatively prime: Two values are coprime if they don't share any common factors. Ex: 22 and 21 are coprime but 22 and 24 are not.

Power Mod: Suppose we had an extremely large exponential we had to mod. We can separate it and perform mod on the separate parts before combining to get the same value. Ex:

$$2^{90} = 2^{50} * 2^{40}$$

$$2^{90} \bmod 13 = (2^{50} \bmod 13) * (2^{40} \bmod 13)$$

$$2^{50} \bmod 13 = 4, 2^{40} \bmod 13 = 3$$

$$4 * 3 = 12 = 2^{90} \bmod 13$$

NumTheory.c functions: **gcd(g, a, b):** finds greatest common divisor of a and b, stores value in g.

mod_inverse(o, a, n): finds the modular inverse of a mod n and stores it in o. If the modular inverse cannot be found, o will be 0.

pow_mod(o, a, d, n): evaluates (aˆd) mod n and stores the value in o.

is_prime(n, iters): checks if n is prime or not using the Miller-Rabin test, has increased accuracy with more iterations (iters).

make_prime(p, bits, iters): generates a random number of a specified bit length, more iterations means the higher likelihood the value is actually prime.

randstate.c functions:

randstate_init(seed): uses GMP's randstate in order to create a random value.

randstate_clear(void): clears the randstate.

ss.c functions: **ss_make_pub(p, q, n, nbits, iters):** creates a public key.

ss_make_priv(d, pq, p, q): creates a private key.

ss_write_pub(n, username, pbfile): writes a public key to the inputted file.

ss_write_priv(pq, d, pvfile): writes a private key to the inputted file.

ss_read_pub(n, username, pbfile): reads a public key.

ss_read_priv(pq, d, pvfile): reads a private key.

ss_encrypt(c, m, n): encrypts plaintext m and stores the cipher text in c using the public key n.

ss_encrypt_file(infile, outfile, n): performs SS encryption on an inputted file and writes the ciphertext to the output file using the public key n.

ss_decrypt(m, c, d, pq): decrypts the given ciphertext c and stores the output in m using the private key elements d and pq.

ss_decrypt_file(infile, outfile, d, pq): performs decryption on the inputted file and writes the decrypted plaintext to the output put file uisng d and pq private key elements.

8 Resources

What is a modular inverse?

What is modular exponentiation?

Miller-Rabin Primality Test Wikipedia Page