Assignment 5 DESIGN.pdf

Ananya Batra

February 26, 2023

1 Description

Purpose: Encrypt and decrypt a file using Schmidth Samoa cryptography, which includes generating a public and private key for a user. A file can be encrypted by using this user's public key and can only be decrypted correctly through the user's private key (which in the real world, would only be available to this user, meaning that the encrypted file can only be decrypted by its intended user).

Structure: Before a user can encrypt or decrypt a file, they need to generate their public and private keys. keygen.c generates a user's public and private keys, calling upon ss.c functions, which in turn calls upon numtheory.c functions and randstate.c to do this. ss.c contains all the functions necessary to perform schmidt samoa cryptography(making public and private keys, reading and writing these keys, and encrypting and decrypting files), utilizing numtheory.c which contains the mathematical functions required to generate the necessary components for SS cryptography. Once the keys are generated, we can encrypt a file (pretending to be a new user) using the original user's public key with encrypt.c. This program takes in a file, encrypts that file with the user's public key, and outputs the encrypted message in another file. The orginal user can then take this outputted file, and decrypt it using decrypt.c which uses the user's private key to decrypt the encrypted text. Both decrypt.c and encrypt.c also make calls to ss.c. Makefile compiles all the c program files and builds the keygen, encrypt, and decrypt executables.

2 Design

randstate.c: This program file handles initializing and clearing the global state variable.
randstate_init(seed): Initializes the random state using the seed provided

- Initialize state for a Mersenne Twister algorithm with gmp_randinit_mt(state)
- generate a random seed
- set an inital seed value into state with gmp_randseed_ui(state,seed)

randstate_clear:

- free all the memory used by state with gmp_randclear(state)

numtheory.c:
gcd(g,a,b) - calculates the greatest common divisor of a and b and stores it in g

- while b does not equal 0 (use mpz_cmp(b,0) whick will return 0 if b = 0)
  - initialize mpz_t t (use mpz_init())
  - set t = b (use mpz_set())
  - set b = a mod b (use mpz_mod(a,b,c) which sets a = b mod c)
  - set a = t
  - The idea behind this: the gcd can only be as large as the number, either a or b. If a mod b = 0, then a perfectly divides b and we can stop looking. Otherwise, the euclidean algorithm states that GCD(a,b)=GCD(b,a-b). Computing the modulo (the remainder) is faster and equivalent to computing the difference between the two numbers until the larger number is no longer larger.
- set g = a

mod_inverse(o,a,n) - This function computes the inverse of a mod n. This is relevant because if a and n and coprime, the extended Euclidian algorithm states that at = 1 (mod n) where t is the inverse.

- initialize t, t',r,r',q, and temp
- set r = n, r' = a, t = 0, t' = 1
- while r' is greater than 0
  - set q = r/r'
  - set temp = r, r = r' and r' = r-q*r' (use mpz_submul(a,b,c) which sets a = a -b*c)
  - set temp = t, t = t', and t' = t - q*t'
- if r is greater than 1, set o to 0
- if t is less than 0, add n to t
- set o = t

pow_mod(o,a,d,n) - This function performs modular exponentiation. It is the equivalent of doing ($a^d \% n$)

- initialize v and p as mpz_ts
- set v = 1
- set p = a
- while d is greater than 0
  - if d is odd (AKA it is not divisible by 2), set v = (v * p) mod n (modding by n at each intermediate step yields the same result as modding the final result at the end. The former also has the added benefit of keeping the final result small)
  - set p = p^2 mod n
  - set d = d/2
- set o = v

is_prime(n,iters) - conducts the Miller-Rabin test to test if n is prime using iters iterations

- initialize variables s,r,i,a,y,temp_n,j,temp_iters,exp,and temp2
- set s = 0, r = n - 1, temp_iters = iters, exp = 2, and temp2 = n-3
- from the assignment sheet: ”write n-1=2^sr such that r is odd”
  - r = (n-1)/(2^s)
  - while r is even (meaning that it perfectly divides 2)
    - r = r/2
    - s = s + 1
- set temp_n = n - 1
- for iterations starting from 1 until iters
  - set a = random_num between 2 and n - 2 (use mpz_urandomm)
  - y = pow_mod(a,r,n)
  - if y is not 1 and y is not n - 1 (temp_n)
    - set j = 1
    - set y = pow_mod(y,2,n)
    - if y is 1, return false
  - if y does not equal n - 1 (temp_n), return false
- return true

make_prime(p,bits,iters) - generates a prime number at least bits number of bits long

- generate a random number and store it in p
  - call mpz_urandomm() to generate a number with max size 2^{bits} (AKA left shift 1 by bits). Then add 2^{bits} + 1 to the random number generated to make sure it's in the right range.
- while p is not is_prime(p,iters)
  - p = another generated random number

ss.c
ss_make_pub(): creates 2 primes (p,q) and n, which are all parts of the public key

- make_prime() p and q
- set the bits range for p be a random number between (nbits/5,2*nbits/5). To do this, generate a random number between (0,nbits/5) and then add nbits/5 to the random number.
- set the number of bits for q to be nbits-2*(bits for p)
- if p/(q-1) is a whole number and q/(p-1) is a whole number, recalculate p and q
- set n = p*p*q

ss_write_pub():

- use gmp_fprintf() to write n as a hex number to the file specified by the file pointer
- use fprintf() to write the username to the file specified by the file pointer

ss_read_pub():

- use gmp_fscanf() to read in n from the file and store it in n
- use fscanf() to read in the username from the file specified by the file pointer and store that value in username[]

ss_make_priv():

- compute totient(n), which is equal to totient(pq)=(p-1)(q-1)/gcd(p-1,q-1)
- set d = mod inverse(n,totient) where n = p*p*q
- set pq = p*q

ss_write_priv():

- use gmp_fprintf() to write pq as a hexadecimal integer to the file specified by the file pointer passed in as a parameter
- write d to the file specified by the file pointer in the same manner

ss_read_priv():

- use gmp_fscanf() to read in pq from the file and store it in pq
- store d in the same manner

ss_encrypt(): helper function for ss_encrypt_file; encrypts a block of information that can be represented in fewer bits than n

- set c = pow_mod(m,n,n)

ss_encrypt_file(): reads in a files contents, encrypts it, and writes it out to the outfile specified

- initialize k. set k (block size of 8 bytes) = $(log_2(sqrt(n)) - 1/8) = log_2(n)/16 - 1/8$ because of log properties. $log_2(n) + 1$ is the number of bits needed to represent n so we can use mpz_sizeinbase() to calculate it
  - We are creating blocks because the SS algorithm used modulo n. For this to work the size of the data being encrypted must be smaller than n.
- initialize a dynamically allocated array of size k and type uint_8 (each element in the array is 1 byte)
- set the array value at index 0 to 0xFF.
- use fgetc() to read in each character from the file
- while it is not the end of file
  - initialize an integer j to keep track of the number of bytes read in

  - while there are still more bytes to read in (AKA we haven't reached end of file yet) and j is less than k
    - set the value of the array at index j = character read in
    - increment j by 1
  - convert the array into an mpz_t with mpz_import()
  - call ss_ecrypt() to encrypt the array
  - write the final encrypted number to the outfile

ss_decrypt():

- set m = pow_mod(c,d,pq)

ss_decrypt_file():

- initialize k. set k (block size of 8 bytes) = $(log_2(sqrt(pq) - 1/8) = log_2(pq)/16 - 1/8$ because of log properties. $log_2(pq)+1$ is the number of bits needed to represent pq so we can use mpz_sizeinbase() to calculate it
- initialize an integer j to keep track of the number of bytes read in
- initialize a dynamically allocated array of size k and type uint_8
- while it is not end of file
  - set c = hexstring scanned in
  - decrypt c back into m
  - convert m from an mpz_t and store the integer into the array (converting it back into bytes). store the number of converted bytes into j. Use mpz_export()
  - write out j - 1 bytes to the outfile starting from array index 1

keygen.c: generates the private and public keys by parsing through the commandline options, initializing the randomstate using the given seed, and calling make_pub() and make_priv() and writing them to their respective files.

- parse through the commandline options using switch statements
    - 'b' sets the minimum number of bits for m (public key)
    - 'i' is the number of iterations for the miller rabin test (used to test if a number is prime)
    - 'n' opens the file specified to write the public key to
    - 'd' opens the file specified to write the private key to
    - 's' sets the seed (needed to initialize a random state)
    - 'v' sets the verbose flag to true
    - 'h' sets the help flag to true
- if help is true, print the synopsis, usage, and options
- otherwise, set private file permissions to 0600 with fchmod() and fileno() so that only the user can access the private key
- initialize a random state with randstate_init()
- initialize p,q,n,d,and pq (private and public key components)
- set name = to the user's username
- generate a public key with ss_write_pub()
- generate a private key with ss_write_priv()
- if verbose is true, print the username, p,q,n,pq,and d
- free the memory used by the random state and clear the mpz_t variables
- close all the file pointers

encrypt.c: encrypts a given file by parsing through commandline options, reading in the public key, and encrypting the specified file by passing in the public key as a parameter to ss_encrypt_file()

- parse through the commandline options using switch statements
    - 'i' opens the file specified to read the message to encrypt from

  - 'o' opens the file specified to write the encrypted message to
  - 'n' opens the file specified to read the public key in from
  - 'v' sets the verbose flag to true
  - 'h' sets the help flag to true
- if help is true, print the synopsis, usage, and options
- initialize n, the mpz_t variable that will store the public key
- read the public_key_file with ss_read_pub(), getting the value of n and the username of the user
- if verbose is true, print the username and n (the public key)
- encrypt the file with ss_encrypt_file()
- clear the mpz_t variable and close all file pointers

decrypt.c: dencrypts a given file by parsing through commandline options, reading in the private key, and decrypting the specified file by passing in the public key as a parameter to ss_decrypt_file() **encrypt.c**: encrypts a given file by parsing through commandline options, reading in the public key, and encrypting the specified file by passing in the public key as a parameter to ss_encrypt_file()

- parse through the commandline options using switch statements
  - 'i' opens the file specified to read the encrypted message in from
  - 'o' opens the file specified to write the decrypted message to
  - 'n' opens the file specified to read the private key in from
  - 'v' sets the verbose flag to true
  - 'h' sets the help flag to true
- if help is true, print the synopsis, usage, and options
- initialize pq and d as mpz_t variables
- read in the private_key_file with ss_read_priv(), getting the value of pq and d

- if verbose is true, print the pq (private modulus) and d (private key)
- decrypt the infile with ss_decrypt_file()
- clear the mpz_t variable and close all file pointers

3 Credit

- I attended Omar's office hours on 2/13. He went over SS and gmp in general and clarified some math concepts and terminology on the assignment sheet.
- I attended Audrey's office hours on 2/16 to understand the role of totient in ss_make_pub() (before it got removed from the assignment sheet
- I attended Dr.Veenstra's office hours on 2/16 to understand what totient is and why it is used in ss_make_priv()
- I attended Ben's office hours on 2/17. They helped me identify that make_prime() was stuck in an infinite loop since I was always adding the random value generated to the random value generated (meaning that the number was always even).
- I attended Varun's office hours on 2/23 because I was having trouble debugging why my ss_encrypt_file() was encrypting the message incorrectly. I was originally using fread() in my function and neither Varun nor I could figure out what was causing the problem. He recommended that I use fgetc() instead. I did that and got it to work.
- I used Ben's guide on compilation and make files : https://hilalmorrar.com/ucsc-guide/docs/majorguides/computerscience/cprogramming/compilation/ to write my Makefile for this assignment.
- I utilized the C Programming book for reading and writing to files.
- I referenced the gmp library to understand what functions I had available to me.