CSE13S Assignment 5 Design

Zion Kang

February 14 2023

1 Description

In this assignment, we create three programs: a key generator, an encryptor, and a decryptor. The keygen program will be in charge of key generation, producing SS public and private key pairs. The encrypt program will encrypt files using a public key, and the decrypt program will decrypt the encrypted files using the corresponding private key. There will also be implementations of two libraries, a random state module and one with logic regarding the math behind SS cryptography.

2 Pseudocode

randstate.c:

randstate_init:
Initialize global random state with a Mersenne Twister algorithm passing in a seed
- Use calls to srandom(), gmp_randinit_mt(), and gmp_randseed_ui().

randstate_clear:
Clear and free all memory used by initialized global random state.
- Single call to gmp_randclear().

numtheory.c:

pow_mod:
initialize and set mpz variables to be used in function
set out as 1
set p as passed in param a

while d (pow) is greater than 0
    if d is odd
    - set out as (v x p) mod n
    set p as (p x p) mod n
    set d as floor div (d / 2)
clear mpz vars used

is_prime:
initialize mpz vars to be used
when trying to find $n - 1 = 2^s r$ such that r is odd
we start from s = 0, so init r as (n - 1)
make loop that checks for if r is odd
    update s by incrementing by 1
    update r by r/2 (floor div here)

for i = 1 to k
- generate random num for a using gmp_urandomm_ui such that range is from (2, 3, ..., n - 2)
- y = pow_mod(a, r, n)
- if (y != 1) and (y != n - 1)
    set j as 1
    while (j ¡= s - 1) and (y != n - 1)
    - set y as pow_mod(y, 2, n)
    - if y == 1, return false
    - update j by incrementing by 1
    if (y != n - 1), return false
clear mpz vars used
otherwise, return true (indicating passed in num is indeed prime)

make_prime:
create a loop that doesn't end until prime is made
generate random num
check if generated random num is prime
once prime is generated, end loop

gcd:
create mpz var copies passed in so as to not alter them
create a loop that constantly updates b as (a mod b) and a as b
in order to do this, set a temp var for b
the break condition for this loop should be if b isn't 0
clear mpz vars initialized
return a

mod_inverse:
init and set mpz vars used in function
set (r, r') = (n, a)
set (t, t') = (0, 1)
use mpz_init_set()/_ui() here
while (r' != 0)
    q = floordiv(r/r')
    set (r, r') = (r', r - q x r')
    set (t, t') = (t', t - q x t')
    when swapping above vars need to set temp var to hold value of r/t respectively
if r > 1
- set out as 0
- return
if t < 0
- set out = t + n
clear used mpzs

SS.C
ss_make_pub:
set bits for p in range [nbits/5, (2 * nbits)/5]
set bits for q as nbits - p, so p + q = nbits
make primes with specified bits for p and q as well as iters
compute lambda(n) = lcm(p - 1, q - 1)
In order to find lambda(n), take phi_n = (p - 1)(q - 1) and divide it by gcd(p - 1)(q - 1)

ss_write_pub:
print n as hexstring into pbfile
print username string into pbfile
both should be written with trailing newlines

ss_read_pub:
This function reads in a public SS key from passed in pbfile
scan in hexstring n and username from pbfile, both followed by trailing newlines

ss_make_priv:
This function creates a new SS private key d given primes p and q and public key n
to compute d, compute the inverse of n modulo lambda(pq)
use the mod_inverse() function here

ss_write_priv:
This function write a private SS key to passed in pvfile
The format of the priv key should be pq then d, each followed by trailing newlines and written as hexstrings.

ss_read_priv:
This function reads a private SS key from pvfile.
The format of a priv key should be pq then d, both following the format specifics above.

ss_encrypt:
This function performs SS encryption, which is simply just computing the ciphertext (c) by encrpyting message (m) using the public key (n).
This is done by performing pow mod on the message.
E(m) = c = $m^n(modn)$.

ss_encrypt_file:
This function encrypts the content of passed in infile, writing the encrypted contents to passed in outfile.
Since we're working in blocks, we should allocate memory for an array (block).
Calculate the block size k = floordiv($log_2((\sqrt{n}) - 1)/8$)
Now that we know the size, dynamically allocate an array that can hold k bytes. This should be of type (uint8_t *).
Set the zeroth byte to 0xFF. This serves as the workaround byte that we need since the value of a block cannot be 0 or 1.
While looping through the file, or while there are still unprocessed bytes in infile
- read at most k - 1 bytes from infile and let j be the number of bytes read.
- using mpz_import(), convert the read bytes, into a message (m).
- Encrypt this message m with ss_encrypt() and write the output to outfile as a hexstring followed by a trailing newline.

ss_decrypt:
This function performs SS decrpytion, computing a message m by decrypting the ciphertext c using private key d and public modulus n.
We do this by simply using our pow_mod function and computing D(c) = m = $c^d(modpq)$

ss_decrypt_file:
This function decrypts the contents of infile then writes the decrypted contents to outfile.
Since we're dealing with blocks, we allocate an array again.

To find the size of this array, compute floordiv($log_2((\sqrt{n}) - 1)/8$)
After dynamically allocating an array of this size, we iterate over the lines in infile.
Scan in a hexstring, saving it as a mpz_t c.
Decrypt this cyphertext c back into a message m. Then convert this message m back into bytes using mpz_export() and store it back in the block.
Let j be the number of bytes converted.
Write out j - 1 bytes starting from index 1 of the block to outfile because we don't want to write the prepended 0xFF.

keygen.c

This key generator program should accept the following command-line options:

- b: specifies the minimum bits for the public modulus n - i: specifies the number of Miller-Rabin iterations for testing primes (default: 50)
- n pbfile: specifies the public key file (default: ss.pub)
- d pvfile: specifies the private key file (default: ss.priv)
- s: specifies the random seed for the random state initialization (default: the seconds since the UNIX epoch, given by time(NULL))
- v: enables verbose output (printing out elements used to get output like p, q, n)
- h: displays program synposis and usage

1. After parsing through command-line options with getopt() and handling them accordingly, open the pub and priv key files
Print a helpful error message and exit the program in the event of failure.
2. Make sure that the private key file permissions are set to 0600, so that reading and writing privileges are only given to the user, and not anyone else. (use fchmod() and fileno())
3. Initialize the random state (use randstate_init()) with set seed
4. Make the pub and priv keys using the respective ss functions
5. Get the current user's name as a string. (use getenv())
6. Write the pub and priv keys to their respective files
7. If verbose output is enabled, print the following, each with a trailing newline
- username
- the first large prime p
- the second large prime q
- the public key n
- the private exponent d
- the private modulus pq
All of the mpz_t values should be printed with info about their number of bits along with their respective values in decimal.
Close files used, clear randstate, and clear mpz variables used.

encrypt.c:

This program should accept the following command-line options: - i: specifies the input file
- o: specifies the output file
- n: specifies the file containing the public key (default: ss.pub)
- v: enables verbose output
- h: displays program synopsis and usage

1. Parse through the command-line options using getopt() and handling them accordingly.
2. Open the public key file. In the case of failure, print a helpful error and exit the program.
3. Read the public key file to get the public key.

4. If verbose output is enabled, print the following:
- username
- the public key n
5. Encrypt the file using the respective ss function.
6. Close the public key file and clear any mpz vars used.

decrypt.c:

This program should accept the following command-line options: - i: specifies the input file
- o: specifies the output file
- n: specifies the file containing the private key (default: ss.priv)
- v: enables verbose output
- h: displays program synopsis and usage

1. Parse through the command-line options using getopt() and handling them accordingly.
2. Open the private key file. In the case of failure, print a helpful error and exit the program.
3. Read the private key file to get the private key.
4. If verbose output is enabled, print the following:
- the public modulus pq
- the private key d
5. Decrypt the file using the respective ss function.
6. Close the private key file and clear any mpz vars used.