GMP Library and Some Cryptography

dlee181

February 2023

1 Introduction

Hello, this assignment goes over public key cryptography which is essentially the system we use to send information between the sender and recipient without having any possible external parties viewing the information. We will be implementing Schmidt-Samoa cryptography takes two keys, a public and a private key, which must be different. These keys are typically inverses and these keys are used to encrypt and decrypt a message. The reason we use systems like SS and RSA is because of the fact that these systems depend on the difficulty of integer factorization. If we attempted to hard break one of these keys, it would take 10^{284} years to break. So attempting to do so is basically impossible without the assistance of ”cheating”.

2 Layout/Psuedo-code

I just want to prefix that all code will be done in c using the GMP Library that allows for the use of an extremely large integer. The reason we need this is that on most systems we can only go up to a long double which although undefined is generally 64-bits or larger.

randstate

1. void randstate_init(seed)
   This function initializes the global randomstate named state, which uses the Mersenne Twister algorithm. in randstate.h. Then we randomly seed this function aswell. This function also calls gmp_randinit_mt and gmp_randseed_ui().

2. void randstate_clear()
   clears and frees all memory initialized by global random state, state. Call gmp_randclear().

numtheory

1. void gcd(mpz_t g, mpz_t a, mpz_t b)
   gcd will compute the greatest common divisor of a and b. We will then store the value in d.

   ```
   while b doesn't equal 0 {
   t equals b
   b equals a mod b
   a equals t }
   input a into d
   ```

2. void mod_inverse(mpz_t o, mpz_t a, mpz_t n)
   mod inverse computes the inverse of a with the value of modulo, n, and store it in o.

   ```
   create values r, r', t, t'
   set (n, a') $\rightarrow$ (r, r')
   set (t, t') $\rightarrow$ (0, 1)
   while r' isn't odd {
   create value q and set it to the floor($r \div r'$)
   Remember to use temporary variables so that you do not lose the intended values
   (r', r - q $\times$ r') $\rightarrow$ (r, r')
   (t', t - q $\times$ t') $\rightarrow$ (t, t') }
   if r is greater than one : return set the value of the output to 0
   if t is less than 0 : set t to t plus n
   set t to output
   ```

3. void pow_mod(mpz_t o, mpz_t a, mpz_t d, mpz_t n)
   pow mod sets output value to the value after modular exponentiating. It computes a^d (mod n)

   ```
   create a value v and p
   set v equal to 1 and p equal to p
   while d is greater than 0{
   if d is odd {
   set v to (v $\times$ p) mod n }
   set p to (p $\times$ p) mod n
   set d to floor(d $\div$ d) }
   set output to v
   ```

4. is_prime(mpz_t n, uint64_t iters)
   is_prime was fairly tedious it required the use of many mpz ints.
   This function essentially implements the Miller Rabin primality test. This test is probablistic in that there are no gaurentees. This test does sometimes give false positives but the probability is really really small given enough iterations.

   ```
   check cases where n is 2 or less
   ```

create integers r, s, y, j, a.
Take a value r which is n - 1 and while that value r is even, set r to r ÷ 2 and iterate s each time. s will give us the number of times we can raise an exponent by 2.

```
for integer i in range (1, k) {
pick a random value for a in ranges [2,n - 2]
set y equal to pow_mod(a, r, n)
if y doesn't equal 1 and y doesn't equal n minus 1 {
set j equal to 1
while j is less than or equal to s minus 1 and y doesn't equal n minus 1 {
set y equal to pow_mod(y, 2, n)
if y equals 1 {
return false }
set j equal to j + 1
if y doesn't equal n minus 1 {
return false } } } }
return true
```

5. make_prime(mpz_t p, uint64_t bits, uint64_t iters)
   This function builds on the previous one and creates prime numbers bits longs and we take iters and put it into our is_prime function.
   Also, keep in mind that prime numbers are about logarithmically distributed
   assert if bits is greater than 0
   create an integer r
   use a do-while loop that breaks on the condition that is_prime of r is false
   within the loop, assign a random value to r with the given bit-length.
   then bit set r at the bit length's bit so that our value is at least the given bits long.
   outside the loop set p to the value r.

SS Library The following functions will become the foundations for our keygen, encrypt, and decrypt files. These functions will allow us to make, read, write, encrypt, and decrypt public and private encryption keys.

1. void ss_make_pub(mpz_t p, mpz_t, q, mpz_t n, uint64_t nbits, uint64_t iters) This function gives us our, prime number n, prime number q, and n. N being the multiple of p times p squared The reason that we do this is because it is difficult to factor two large prime numbers.

   make two variables which will hold p_bits and q_bits
   let p_bits be a random number in range [nbits ÷ 4, (2 x nbits) ÷ 5)
   Afterwards take the random value generated and subtract it from the p value.
   Next take the value of bits and generate a prime number,p and q.
   set the value of n to p^2 x q
   Lastly, make sure that p doesn't divide by q - 1 AND q doesn't divide p

- 1.
Psuedo code: in order to generate two prime numbers, we must specify the amounts of bits allocated to each of the prime numbers. We decide this by taking the given nbits and assign the bits for p to be a random value in the range of [nbits/5,(2 * nbits)/5]. Then we assign the remaining bits to q.
Then we generate a prime number p and prime number q with our pre-made make_prime. Next, because we are using the gmp library, set a value as p - 1 and q - 1. These values will represent the totients because if p and q are prime, we know that the value of totient(p) and totient(q) are p -1 and q - 1, because every value before p and q are co-prime to p and q.
Then I will check if p and q - 1 are divisible by zero and vice versa. If so I generate another prime number for both values.
In better practice, I believe that putting a loop here would be better, but I believe that the probability of it being divisible twice is very unlikely.

Next, I am uncertain why, but when multiplying p * p * q and assigning this value to n, the number of bits generated was signifcantly greater than nbits. So instead I made a variable pq and assigned p * q to it, next I assigned p * pq to n and created my n value.

2. void ss_make_priv(mpz_t d, mpz_t p, mpz_t q) Personally if I feel like this function should take the previously generated public key and reuse the n that we made because we are already reusing the p and q, but not the n so it feels wasteful.
   This function generates the private key, d which is essentially just the inverse of n mod(lcm of (p - 1, q - 1)). The way we get the lcm is by using Carmichaels's function.
   Psuedo code: Again, because we are dealing with the gmp library, we will assign a value as p - 1 and q - 1. We will call these values t_p and t_q which will essentially represent the totient of p and the totient of q. Next we will create the value t_n which will represent the totient of n. Then assign the values of t_p * t_q to t_n. Next take the absolute value of t_n. Next inorder to generate the lcm, we actually need the the greatest common divisor of t_p and t_q which we will call gcd_t. Next divide t_n by gcd_t and we now have the lcm of (p - 1, q - 1). Then we will create a value pq and set it to p * q. Then we make another n which will be pq * p. Lastly to get the d value, we will use our pre-made mod-inverse function and take the inverse of n modulo lcm.

3. void ss_write_pub(mpz_t n, char username[], FILE *pbfile) This will take Essentially this will take the public key, n and the username which we specified in our keygen.c file and write fprint them to a file *pbfile. Remember that the format is the public key followed by a new line, then the username followed by a new line. Also don't forget that mpz_t have

unique format specifiers

4. void ss_write_priv(mpz_t pq, mpz_t d, FILE *pvfile) The public key takes the provided pq and d and writes them into a file pvfile. The way to put it in the file is the same as ss_make_pub. The order is fprintf pq followed by a new line then d followed by a new line.

5. void ss_read_pub(mpz_t n, char username[], FILE *pbfile) Essentially we just need to scan a file and take the file's lines and turn them into n and username so that we can return those values when this function is called. To do so scanf the pbfile then assign the first line of the file to n and the second line to username. Don't set username to NULL when declaring the username in another file because username won't have any bytes to get allocated to.

6. void ss_read_priv(mpz_t pq, mpz_t d, FILE *pvfile) We will scan a file, pvfile, and take the first line and assign it to pq and the second line to d.

7. void ss_encrypt(mpz_t c, mpz_t m, mpz_t n) The purpose of this function is to take our message which is basically just a bunch of numbers and encrypts it and turns it into cypher text for our decrypter. This function simply uses our premade pow_mod function and when we take our message, square it by n, and mod it by n, our message turns into encrypted cypher text.

8. void ss_encrypt_file(FILE *infile, FILE *outfile, mpz_t n) encrypting a file is much harder than it seems, when encrypting a file, one needs to make sure that the data is chopped up into blocks which need buffers. The reason we make blocks is because of the fact that we are working with modulos. Imagine that you have a number mod 5 = 0. For all you know, the number could be 5, 10, 15, 20 and because of this, the same logic applies. If we had more information than n, we would lose anything after n. Furthermore we have to be wary to not have our block be 0 or 1. To avoid this problem, we prepend a byte at the front of the array at index 0.

   Psuedo code To begin, we must calculate the value of the block size, k. Normally the formula is floor(log2(sqrt(n) - 1) / 8), but because we are again working with mpz_t values, I recommend working inside out. start by calculating the sqrt of n and work your way out.
   We will create a value called BUFFER and set it to the value of k. Next dynamically allocate an array of type (uint8_t*) of uint8_ts that can hold BUFFER bytes. Each of these will represent an ASCII character. Next prepend the zeroth byte of the block to 0xFF. Next set the value of j as the amount of bytes read in each block and it will read starting at arr[1],

9. void ss_decrpyt(mpz_t m, mpz_t c, mpz_t d, mpz_t pq) The purpose of this function is to return the value of m which was previously encrypted. We do this by using our pow_mod function and take the cypher text squared to the power of d (mod pq)

10. void ss_decrpyt_file(FILE *infile, FILE *outfile, mpz_t d, mpz_t pq)

3 Conclusion/What I learned

4 Credits

1. Stuart gave me pseudo code for the is_prime function.
2. Dev helped with mpz_imports, reading a file, and finding the condition in my is_prime function where it breaks with the test he made. I also used Dev's example for reading files.
3. Miles helped with the ss_encrypt_file function by elaborately explaining and visualizing how the function works
4. Lev for helping with the Makefile.
5. Credit Darrell Long for the psuedo code on the spec sheet for basically every function.