Assignment 6 - Public Key Cryptography

Teresa Joseph

CSE 13S - Professor Long

Fall 2021 - RD November 11/ FD November 21

Purpose

The purpose of this assignment is to implement encryptors and decryptors that can manipulate public and private RSA keys, which are generated in this assignment as well. numtheory.c contains the implementation of modular math functions that will be used for this purpose. It finds the greatest common divisor, modular inverse, power modulus, determines if a variable is prime, and makes a variable prime. randstate.c generates the state needed to create the large values for the keys and the values they need to be calculated. rsa.c reads and writes from/to files and encrypts and decrypts the keys to their respective files, which will be used a lot in the main test files. keygen.c creates these keys, encrypt.c writes the public key to the outfile, and decrypt.c writes the private key to the outfile. All three of these files can run with command line options.

Pseudocode

numtheory.c

```
gcd
    while b is not 0:
        store b in d
        store a mod b in b
        store d in a
mod_inverse
    store n in r and store a in inv_r
    store 0 in i and store 1 in inv_t
    while inv_r is not 0:
        store floor of r/inv_r in q
        store inv_r in r
        store r-q*inv_r in inv_r
        store inv_i in i
        store i-q*inv_i in inv_i
```

```
        if r>1, set i to 0
        if i<0, store i+n in i
pow_mod
        let v be 1 (to return if exponent is 0)
        let p be base
        while exponent is greater than 0:
                if the exponent is odd (use mpz_odd_p not equal to 0):
                        store v*p mod modulus in v
                else:
                        store p*p mod modulus in p
                        make the exponent half of what it originally was
        set out to v
is_prime
        set s = 0 and r = n-1
        while r is even (use mpz_even_p not equal to 0):
                increment s by 1
                half r by 2
        a valid odd r would be found upon exiting
        for i from 0 to iters number of iterations:
                choose random value from 2 to n-2 and store it in a
                call pow_mod(y, a, r, n)
                if y isn't 1 and isn't n-1:
                        set j to 1
                        keep track of primality with boolean flag = false
                        while j is <= s:
                                call pow_mod(y, y,  2, n)
                                if y is 1, break
                                if y is n-1, set the flag to true and break
                                increment j by 1
                        if flag is still false, break
        return flag (represents primality)
```

make_prime

while true:

call mpz_urandomb on variable

call is_prime on the variable

if it is true, break

else, continue looping until true

randstate.c

randstate_int

initialize state for MT algorithm with gmp_randinit_mt()

set initial seed value of state and given seen with gmp_randseed_ui()

randstate_clear

clear memory of state with gmp_randclear()

rsa.c

rsa_make_pub

generate random number that represents the number of bits in p, store it in p_bits

check that the number is prime and in range [nbits/4, (3*nbits)/4]

if not, generate new number until true

call make_prime() with p_bits and set result to p

call make_prime() with nbits-p_bits and set result to q

set totient_n = (p-1)*(q-1)

while true:

call mpz_urandomb() with size nbits to make random number e

find gcd of random number and totitent_n

if gcd of e and totient_n = 1:

set current random number to e

break out of loop

rsa_write_pub

using gmp_fprintf() to pbfile with new line character after each:

write n as hexstring (%Zx)

write e as hexstring (%Zx)

write s as hexstring (%Zx)

write username as string (%s)

rsa_read_pub

using gmp_fscanf() to pbfile with new line character after each:

read n as hexstring (%Zx)

read e as hexstring (%Zx)

read s as hexstring (%Zx)

read username as string (%s)

rsa_make_priv

create totient_n variable and set it = (p-1)*(q-1)

set d = e mod totient_n using mod_inverse()

rsa_write_priv

using gmp_fprintf() to pvfile with new line character after each:

write n as hexstring (%Zx)

write d as hexstring (%Zx)

rsa_read_priv

using gmp_fscanf() to pvfile with new line character after each:

write n as hexstring (%Zx)

write d as hexstring (%Zx)

rsa_encrypt

set c = m^e mod n using pow_mod()

rsa_encrypt_file

set k = the floor of (log base 2 of n -1)/8

use malloc to allocate k size of memory of type uint8_t pointer (this is the block)

set the zeroth index of the block to 0xFF

let j = number of bytes converted (starts at 0)

while true:

call fread() starting from block[1] and save value to j

if j is less than or equal to 0 (meaning we reached EOF):

break from while loop

call mpz_import from block to message m as document describes

convert c to bytes using rsa_encrypt() with m, e, and n print c to outfile with gmp_fprintf

rsa_decrypt

set m = c^d mod n using pow_mod()

rsa_decrypt_file

set k = the floor of (log base 2 of n -1)/8

use malloc to allocate k size of memory of type uint8_t pointer (this is the block)

set the zeroth index of the block to 0xFF

let j = number of bytes converted (starts at 0)

let a size_t variable represent export parameter

while true:

if EOF reached according to gmp_fscanf, break from loop

else:

call rsa_decrypt on m with c, d, and n

call mpz_export from m to block using size_t variable

write from outfile to block from second index, incrementing value to j each time

rsa_sign

set s = m^d mod n using pow_mod()

rsa_verify

set t = s^e mod n using pow_mod()

if t is equal to the message m, return true

else, return false

keygen.c

parse through common line options with getopt

if b, take value as minimum number of bits needed for n

if i, take value as number of iterations for testing primes

if n [pbfile], set as public key (default = rsa.pub)

if d [pvfile], set as private key (default = rsa.priv)

if s, take value as random seed initialization

if v, enable verbose output

if h, display help message (program synopsis and usage) use fopen() to open both public and private key files

    in either case, if unable to open or if files don’t exist, print error message and exit

set private key permission to 0600 with fchmod() and fileno()

use seed and call randstate_init()

make public key using rsa_make_pub()

make private key using rsaa_make_priv()

get user name with getenv() and convert it to mpz_t type using mps_set_str() base 62

use rsa_sign() to compute signature of user name

write public key to its outfile with rsa_write_pub

write private key to its outfile with rsa_write_priv

check if verbose was enabled, and if so:

    print each with number of btis: user name, signature s, p, q, n, e, and d

close public and private files

clear random state with randstate_clear()

clear any extraneous mpz_t variables

encrypt.c

parse through common line options with getopt

    if i, take file as infile (default = stdin)

    if o, take file as outfile (default = stdout)

    if n, set as public key (default = rsa.pub)

    if v, enable verbose output

    if h, display help message (program synopsis and usage)

use fopen() to open public key file

    if unable to open or if file doesn’t exist, print error message and exit

read public key with rsa_read_pub()

check if verbose was enabled, and if so:

    print each with respective mpz_t value: user name, signature s, n, and e

convert user name into mpz_t type (for verified signature)

check signature with rsa_verify()

    if signature couldn’t be verified, print error message and exit

call rsa_encrypt_file()

close public key file

clear any extraneous mpz_t variables

decrypt.c

parse through common line options with getopt

if i, take file as infile (default = stdin)

if o, take file as outfile (default = stdout)

if n, set as private key (default = rsa.priv)

if v, enable verbose output

if h, display help message (program synopsis and usage)

use fopen() to open private key file

if unable to open or if file doesn't exist, print error message and exit

read private key with rsa_read_priv()

check if verbose was enabled, and if so:

print each with number of btis: public modulus n and private key e

convert user name into mpz_t type (for verified signature)

call rsa_decrypt_file()

close private key file

clear any extraneous mpz_t variables

Overall Description

The files for randstate.c, numtheory.c, and rsa.c have multiple functions defined for making up the keygen, encrypt, and decrypt main functions. All of these functions use GNU Multiple Precision Arithmetic (GMP) library operations so as to handle long keys, exponents, modulus, and such.

numtheory.c handles mathematical operations. gcd() finds the greatest common divisor between two variables and stores the result in a third variable. It takes the mod (%) to do so and slowly reduces the variables by common denominators, swapping variables as it progresses. It is really efficient and straightforward. mod_inverse() finds the modular inverse of a mod n and stores it in i. This also requires a lot of swapping and resetting values as the loop progresses. I made use of two temporary variables, one that stored r and another that stored i, so as to not overwrite my values later. This is because C does not allow parallel assignment. Next is pow_mod(), which was straightforward and was similar to the others in the sense that it required modulo as well, only this time, I had to implement an exponent. This also involved using temporary variables. is_prime() was really complicated. I went through a while loop first to determine s and r so as to write n-1 with an odd r. Then, I went through iters number of iterations, manipulating a flag that represented primality. Using plenty of variables from modulo expressions and looking at their relationship to the provided number n, I returned just a single flag at the end of my function. As such, it was a simplified version of the Miller-Rabin pseudocode provided in the assignment document. make_prime() simply looped through a while loop, calling is_prime() until a prime number was found, which was really straightforward.

randstate.c only took on two functions and were implemented in less than five minutes. randstate_int() called gmp_randinit_mt() and gmp_randseed_ui() to initialize the state and the seed respectively. randstate_clear() simply freed memory with gmp_randclear().

rsa.c took a long time to implement, especially the first function which I am still struggling with. rsa_make_pub() is meant to generate the primes p and q, the public exponent e, and the modulus n. The primes were made with calls to make_prime, taking on iters number of Miller-Rabin iterations. Once found, they were multiplied to find n. The totient was then found in a similar way, but it required using temporary variables so as to not overwrite anything. With this, I then looped through a while loop to find e. In it. I compared the e values with its totient to find the gcd. If the gcd was 1, like the assignment document said, went with that e value. For rsa_make_priv, I found the totient again and used it to call mod_inverse() to find d. This directly relates to the formulas on cryptography I learned from class and section. rsa_read_pub() and rsa_read_priv() were merely calls to gmp_fscanf, and likewise, rsa_write_pub() and rsa_write_priv() were calls to gmp_fprintf. Next, rsa_encrypt was just a call to pow_mod(), as was rsa_decrypt, to find ciphertext c and message m respectively. rsa_encrypt_file() and rsa_decrypt_file() were similar in the sense that I had to find valid k values, allocate enough memory for a block array, and set the first zeroth byte of the block to 0xFF so as to be able to find the start and end of my values. rsa_encrypt_file() then called mpz_import and printed encrypted content to the outfile. rsa_decrypt_file(), on the other hand, scanned from the infile, made use of mpz_export, and wrote to the outfile. rsa_sign() and rsa_verify() go hand in hand. The signature found in sign by calling pow_mod() would be compared to the verified message in verify. If it is equal to the message, then the signature is verified.

keygen.c, encrypt.c, and decrypt.c have a similar format, and this is especially true for the latter two files. All three of them parse through command line arguments, as I have been doing for several assignments now. I also realized that setting FILE pointers to NULL at the start, before parsing, would make choosing the optarg file or sticking with the default files much easier, so I implemented this in all main functions. So after opening my respective files, I called the functions that I made in numtheroy, randstate, and rsa as needed, checked my verbose flag, closed my files, and cleared any allocated memory. This is quite similar to the main functions I made for encode.c and decode.c in our previous assignment.

Notes

- All mpz_t variables are initialized with mpz_init or mpz_inits with a NULL term
- All mpz_t variables are cleared with mpz_clear or mpz_clears with a NULL term at the end and/or before return statements
    - All extraneous variables are cleared like this
- randstate_init() and randstate_clear() are generally called in the main functions (not in the individual functions themselves)
- EOF represents the end of the infile

Other

$\phi(p) = p-1$

$\phi(q) = q-1$

$\phi(pq) = (p-1)(q-1) = \phi(n)$

↳ choose some e value

then $d = e^{-1} \bmod \phi(n)$

& $de \equiv 1 \bmod \phi(n)$

d = private key

n & e = public keys

coprime if gcd = 1

↳ "relatively prime" but don't have to be prime #s [n = 4096 bit #, p & q must be vvv big] → use GNU

encription of message = message^e mod n

decripition of E(m) = $(E(m))^d$ mod n

- need

~~keygen~~ keygen (generate keys)

encryptor (encrypt files)

decryptor (decrypt files)

numtheory

randstate

rsa

encrypt_file: values must be < n to mod, use blocks

use 0xFF in front

- $E(m) = c = m^e \bmod n$

$D(c) = m = c^d \bmod n$

sign $s = x^d \bmod n$

verify $v = y^e \bmod n$

IS PRIME

Ex) $n = 23$

$n-1 = 2^s r$

$22 = 2^s r$

$s \to 0$, $r \to n-1$

$22 = 1 \cdot 22$ ✓

in loop, s++ and r/2

↳ $s = 1$, $r = 11$ not even

mod $2 \neq 0$

$22 = 2^1 (11)$

$22 = 22$ ✓

↳ ~~$s = 2$, $r = 5$~~ non-float??

won't occur ~~$22 = 2^2 \cdot 5$~~

~~$22 = 4 \cdot 5$~~ X

↳ must use $s = 1$ and $r = 11$

$n = 13$

$12 = 2^s r$

$12 = 2^0 \cdot 12$ ✓

↳ $s = 1$, $r = 6$, r is even ✓

$12 = 2 \cdot 6$ ✓

↳ $s = 2$, $r = 3$, r not even

$12 = 2^2 \cdot 3$

$12 = 4 \cdot 3$ ✓

↳ $s = 2$ & $r = 3$ solution

MOD INVERSE

If $(8, 5, 13) \leftrightarrow (i, a, n)$

start: $r = n = 13$ $q = 0$

r_inv $= a = 5$ qr_inv $= 0$

$i = 1$ qi_inv $= 0$

i_inv $= 1$

while $r\_inv \neq 0$:

① $q = \lfloor 13/5 \rfloor = 2$

$r = r\_inv = 5$

$qr\_inv = q \cdot r\_inv = 2 \cdot 5 = 10$

$r\_inv = r \cdot qr\_inv = 5 \cdot 10 = 50$

$i = i\_inv = 5$

$qi\_inv = q \cdot i\_inv = 2$

$i\_inv = i \cdot qi\_inv = 10$

② $q = \lfloor 5/50 \rfloor = 0$

$r = 50$

$qr\_inv = 0$

$r\_inv = 0$

$i = 10$

$qi\_inv = 0$

$i\_inv = 0$

③ undefined?

$\Rightarrow$ overwriting

Will need temp variables

- store r
- store i