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 deal with public and private RSA keys, which will be generated in this assignment as well. numtheory.c contains the implementation of modular math functions that will be used for this purpose, and randstate.c will generate the large values needed to create the keys. rsa.c does plenty of reading and writing as well as encrypting and decrypting for 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.

Pseudocode

numtheory.c

```
gcd
    while b is not 0:
        store b in d
        store a mod b in b
        store d in a
    return 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
        return 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 (mod 2 = 1):
                store v*p mod modulus in v
            else:
                store p*p mod modulus in p
                make the exponent half of what it originally was
        return v
    is_prime
        set s = 0 and r = n-1
        while r is even (mod 2 = 0):
            for i from 1 to iters:
                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
                    while j is less than or = to s-1 and y isn't n-1:
                        call pow_mod(i, y,  2, n)
                        if y is 1, return false
                        increment j by 1
                    if y isn't n-1, return false
        return true
    make_prime
        generate random number of length bits using randstate_int() and store in p
        call is_prime on p and store in result
        if result is false, repeat process 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)

in for loop for nbits number of times:

call mpz_urandomb() with size nbits to make random numbers

find gcd of random number and totitent_n

if gcd = 1:

set current random number to e

break out of loop

rsa_write_pub

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

write n as hexstring

write e as hexstring

write s as hexstring

write username as hexstring

rsa_read_pub

read each hexstring from pbfile

set first read line as n, second as e, third as s, and fourth as username

rsa_make_priv

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

set d = e mod totient_n

call mod_inverse() of final result

rsa_write_priv

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

write n as hexstring

write d as hexstring

rsa_read_priv

read each hexstring from pvfile

set first read line as n and second read line as d

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)

while the infile hasn't been fully read:

scan and save hexstring as mpz_t c

convert c to bytes using mpz_export() and store to block

let j = number of bytes converted

use write() to write j-1 bytes from first index to outfile

(note: don't output the zeroth index 0xFF)

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 first (zeroth) byte of the block to 0xFF (all 1's)

while the infile hasn't been fully read:

read k-1 bytes from infile

let j = number of bytes read

add j to block starting from the first byte

convert read bytes and 0xFF to mpz_t type with mpz_import()

call rsa_encrypt() with message m

use write() to write encrypted number to outfile as hexstring

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, 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

Notes

- I plan on working on randstate.c first, then numtheory.c, then rsa.c, and then the main test harnesses in the order I have my pseudocode provided
- I don't know how to write mpz_t values as hexstrings at the moment, so I will do more research on this later and fix this part of rsa.c later
- i'm not entirely sure what read_pub and read_priv entail in rsa.c so I will add onto my pseudocode later once I figure this out

Other (notes)

$\varphi(p) = p-1$ coprime if gcd = 1

$\varphi(q) = q-1$ ↳ "relatively prime" but don't have to be prime #s

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

↳ choose some e value

then $d = e^{-1} \bmod \varphi(n)$ d = private key

& $de \equiv 1 \bmod \varphi(n)$ n & e = public keys

[n = 4096 bit #, p & q must be vvv big] → use GNU

encryption of message = $\text{message}^e \bmod n$

decryption of $E(m) = (E(m))^d \bmod n$

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 $v = y^e \bmod n$