Zion Kang
zkang5@ucsc.edu
27 October 2022

CSE13S Fall 2022
Assignment 5: Public Key Cryptography
Design Document

I. Explanation

In this assignment, there are three programs: keygen, encryp, and decrypt. The keygen program produces RSA public and private key pairs. The encrypt program encrypts files using a public key while the decrypt program decrypts the encrypted files using the corresponding private key.

II. Pseudocode

randstate.c

// initializes the random state needed for RSA key generation operations
// must be called before any key generation or number theory operations are used
 # randstate_init(seed)
  # set rand seed for stdlib rand funcs
  # initalize global random state with name state //using gmp_randinit_mt()
  # set seed value // using gmp_randseed_ui()
// frees any memory used by initialized random state
// must be called after all key generation or number theory operations are used
 # randstate_clear()
  # free memory used by initialized random state // use gmp_randclear()

numtheory.c

// performs fast modular exponentiation, computing base raised to the exponent power modulo modulus, and storing the computed result in out
 # pow_mod(o, a, d, n)
  # initialize mpzs
  # set variable v as 1 // mpz_init, mpz_add
  # set p as a // mpz_init, mpz_set
  # while d > 0
    # if d is odd
      # set v = (v x p) mod n // mpz_mul, mpz_mod
    # set p = (p x p) mod n // same mpz as above
    # set d = d/2 // mpz_fdiv_q
  # store v in o (out)
  # clear mpzs

// conducts the Miller-Rabin primality test to indicate whether or not n is prime using iters number of Miller-Rabin iterations. This function is needed when creating two large primes p and q in RSA, verifying if a large integer is a prime.

```
# bool is_prime(n, iters)
    # initialize mpzs
    # while loop checking if r is odd // s is number of times n - 1 is divisible by 2 r is remainder (odd))
        # if (n - 1)/2^s = r is odd //mpz_odd_p
            # set r
        # increment s
    # while i < k:
        # choose random a = {2, 3, …, n - 2} // using mpz_urandomm
        # set y = pow_mod(a, r, n) // using r from first line in fxn
        # if y != 1 and y != n - 1 // mpz_cmp
            # set j = 1
            # while j <= s - 1 and y != n - 1
                # set y = pow_mod(y, 2, n)
                # if y == 1
                    # return False
                # j += 1 // inc j by 1
            # if y != n - 1
                # return False
    # return True
```

// generates new prime number stored in p (this # should be at least *bits* number of bits long)

```
# make_prime(p, bits, iters)
    # while true
        # generate random num of bits number of bits stored in p // using mpz_randomb
        # check if is_prime
            # store in p
```

// computes the greatest common divisor of a and b, storing value in d

```
# gcd(d, a, b)
    # while b != 0 // mpz_cmp
        # set t = b
        # set b = a mod b //mpz_mod
        # set a = t
    # store a in d
```

// computes inverse i of a modulo n and stores output in o (if modular inverse cannot be found, o = 0)

 # mod_inverse(o, a, n)

```
    # set r1 = n // mpz_set_init
    # set r2 = a
    # set t1 = 0
    # set t2 = 1
    # while r2 != 0 // mpz_cmp
        # set q = r1/r2 (floor div) // mpz_fdiv_q
        # set tmp var = r1
        # set r1 = r2
        # set r2 = r1 - (q * r2)
        # set tmp var = t1
        # set t1 = t2
        # set t2 = t1 - (q * t2)
    # if r > 1 // mpz_cmp
        # store 0 in o
        # clear mpzs
        # return
    # if t1 < 0
        # set t1 = t1 + n
    # store t1 in o
    # clear mpzs
```

rsa.c

// creates parts of a new RSA public key: two large primes p and q, their product n, and the public exponent e

 # rsa_make_pub(p, q, n, e, nbits, iters)

```
    # create primes p and q using make_prime()
        // nbits must be less than log base 2 (n) // mpz_sizeinbase?
        // number of bits for p be a random number in range [nbits/4, 3nbits/4)
        // remaining bits go to q = nbits - pbits
        // number of Miller-Rabin iterations specified by iters
    # compute lambda(n) = (p - 1) * (q - 1) / gcd(p - 1, q - 1), store in var
    # loop for finding suitable public exponent e
        # generate random nums of around nbits // using mpz_urandomb()
        # compute gcd of random num and computed lambda(n)
        # if random num is coprime with lambda(n)
            # set random num as public exponent e
            # clear mpzs
```

// writes a public RSA key to pbfile using rsa_make_pub (format of a public key should be n, e, s, username (each with trailing newline)

 # rsa_write_pub(n, e, s, username[], *pbfile)

write n, e, s (as hexstrings: %Zx) and username (each on new lines) to pbfile. // using gmp_fprintf() %Zx\n

// read public RSA key file from pbfile (formatted as n, e, s, username from rsa_write_pub)

 # rsa_read_pub(n, e, s, username[], *pbfile)

read n, e, s, and username from pbfile // using gmp_fscanf() %Zx

// creates new RSA private key d given primes p and q and public exponent e (compute d by getting the inverse of e modulo lambda(n))

 # rsa_make_priv(d, e, p, q)

compute d = mod inverse e, lambda(n) // lambda(n) = (p - 1)(q - 1) / gcd(p - 1, q - 1)

// write private RSA key to pvfile (format should be n then d, both written as hexstrings and with trailing newlines)

 # rsa_write_priv(n, d, *pvfile)

write n then d // hexstring, trailing newline using gmp_fprintf() %Zx

// reads private RSA key from pvfile (format should be n then d, both written as hexstrings and with trailing newlines)

 # rsa_read_priv(n, d, *pvfile)

read n then d // hexstring, trailing newline using gmp_fscanf() %Zx

// performs RSA encryption, computing ciphertext c by encrypting msg *m* using *e* and *n*

 # rsa_encrypt(c, m, e, n)

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

// encrypts content of infile, writing encrypted content to outfile (done in blocks)

 # rsa_encrypt_file(*infile, *outfile, n, e)

calculate block size k = floordiv[(log base 2 (n) - 1) / 8]

set vars for total num of bytes, bytes left to read in file, bytes actually read, index

while there are still unprocessed bytes in infile:

dynamically allocate an array of size k of type (uint8_t *) which serves as the block

set 0th byte of block to 0xFF // prepends the workaround byte that we need

check for reading at most k - 1 bytes from infile

if bytes in file >= bytes in file - (k - 1)

                    # set bytes left to read as k - 1
            # else
                    # set bytes left to read as bytes in file - index
            # set bytes_read to return of fread starting from block + 1
            # update index with bytes read
            # using mpz_import(), convert read bytes including 0xFF into an mpz_t *m* // set 1 for most significant word first, 1 for the endian parameter, and 0 for the nails parameter
            # encrypt m with rsa_encrypt(), write encrypted number to outfile as hexstring followed by trailing newline
            # print ciphertext in outfile
            # free and clear block
    # clear mpzs

// performs RSA decryption, computes message *m* by decrypting ciphertext *c* using *d* and *n*.

 # rsa_decrypt(m, c, d, n)

    # set m = c^d (mod n) // pow_mod, mpz_mod

// decrypts contents of infile, writing decrypted content to outfile (done inblocks_

 # rsa_decrypt_file(*infile, *outfile, n, d)

    # calculate block size k = floordiv[(log base 2 (n) - 1) / 8]
    # declare size_t j
    # while there are still unprocessed bytes in infile
            # dynamically allocate an array that can hold k bytes of type (uint8_t *) which serves as the block
            # if at end of file
                    # clear and free block
                    # break out of loop
            # scan in a hexstring and save as a mpz_t *c* //each block is written as a hexstr with a trailing newline when encrypting a file
            # decrypt ciphertext and store message in m
            # use mpz_export() to convert c back into bytes, storing them in the allocated block // 1 for most significant word first, 1 for endian, 0 for nails
                    # j gets updated from mpz_export
            # write out j - 1 bytes starting from index 1 of block to outfile (because index 0 must be prepended to 0xFF so DO NOT OUTPUT THE 0xFF)
            # free and null term block
    # clear mpzs

// performs RSA signing, producing *s* by signing msg *m* using priv key *d* and mod *n*

 # rsa_sign(s, m, d, n)

set s = m^d (mod n) // pow_mod, mpz_mod

// performs RSA verification, returning true if sign *s* is verified and false otherwise

 # rsa_verify(m, s, e, n)

set t = s^e (mod n)
if t == m
clear t
return true
else
clear t
return false

<u>keygen.c</u>

getopt()
should accept options -bindsvh
-b: specifies minimum bits needed for public modulus *n*
- i: specifies number of Miller-rabin iterations for testing primes (def: 50)
- n pbfile: specifies public key file (def: rsa.pub)
- d pvfile: specifies the priv key file (default: rsa.priv)
- s: specifies the random seed for random state initialization (def: seconds since UNIX epoch, given by time (NULL))
- v: enables verbose output
- h: displays program synopsis and usage
parse command-line for options
open pub and priv key using fopen() (print helpful error and exit (with non zero error code?) in event of failure)
run fchmod() and fileno() to make sure priv key file permissions are set to 0600, indicating permission for user only and no one else
initialize rand state using randstate_init(), with set seed
make the pub and priv keys using rsa_make_pub() and rsa_make_priv()
get current user's name as string // use getenv()
convert username into mpz_t with mpz_set_str() // base of 62 and use rsa_sign() to compute the signature of the username
write computed pub and priv key to respective files // pbfile, pvfile
if verbose output enabled
print username, signature *s*, first large prime *p*, second large prime *q*, public modulus *n*, public exponent *e*, private key *d* (each with a trailing newline) // print these mpz_t values in decimal along with info about the number of bits that constitute them
close pub and priv key files, clear rand state with randstate_clear(), clear any mpz_t variables used

encrypt.c

getopt()

i, o, n, v, h options should be taken in command-line for respective purposes stated in assignment documentation

- i: specifies input file to encrypt (def: stdin)
- o: specifies output file to encrypt (def: stdout)
- n: specifies file containing the public key (def: rsa.pub)
- v: enables verbose output
- h: displays program synopsis and usage

open pub key file using fopen() // print helpful error and exit program in even of failure

read pub key from opened public key file // rsa_read_pub

if verbose output is enabled print the following, each with a trailing newline:

print *username*, signature *s*, public modulus *n*, public exponent *e* // mpz_t values should be printed in decimal along with info about number of bits that constitute them

convert username into mpz_t // with mpz_set_str? this is the expected value of the verified signature

verify signature using rsa_verify(), reporting an error and exiting with non zero exit code if signature couldn't be verified

encrypt file using rsa_encrypt_file()

close pub key file, clear mpz_t variables used

decrypt.c

getopt()

should take i, o, n, v, h options in command-line for respective purposes stated in assignment documentation

- i: specifies input file to decrypt (def: stdin)
- o: specifies output file to decrypt (def: stdout)
- n: specifies file containing priv key (def: rsa.priv)
- v: enables verbose output
- h: displays program synopsis and usage

open priv key file using fopen(), printing helpful error msg and exiting with non zero exit code in event of failure

read priv key from opened priv key file // rsa_read_priv()

if verbose output is enable print the following, each with a trailing newline:

public modulus *n*, private key *e* // both values should be printed in decimal along with info about the number of bits that constitute them

decrypt file using rsa_decrypt_file()

close priv key file and clear any mpz_t variables used