Edison Chen

Assignment 6: Public Key Cryptography

Program Description:

The purpose of this assignment is to learn about and implement public key cryptography. Public-key cryptography is a system that uses pairs of keys consisting of both public and private private keys. The system will function by having a message be encrypted using a user's public key and decrypted only by use of the user's private key. For this assignment three programs will be created: a key generator, an encryptor, and a decryptor. The key generator program will produce RSA public and private key pairs. The encryptor will encrypt files using the public key and the decryptor will decrypt the file using the corresponding private key.

Files Included:

decrypt.c - implementation and main() function for the decrypt program

encrypt.c - implementation and main() function for the encrypt program

keygen.c - implementation and main() function for the keygen program

numtheory.c - implementation of numtheory functions

numtheory.h - interface for numtheory functions

randstate.c - implementation of the random state interface for the RSA library and number theory functions

randstate.h - interface for the randstate functions, initialization and clear

rsa.c - implementation of the RSA library

rsa.h - interface for the RSA library

Makefile - file that builds the programs and formats all files to clang format

README.md - text file in markdown format that describes: the program, how to build the program, and how to run the program

DESIGN.pdf - pdf file that describes the program and its design

Notes and Pseudocode:

randstate.c:

- Install gmp to use large integers, use pkg-config to link gmp
- Need two functions, one to initialize the state and one to clear it
- Initialize the global random state using the Mersenne Twister algorithm for initializing
- State is a global function defined in randstate.h
- A random state variable is necessary for random integer functions in GMP

void randstate_init(uint64_t seed):

Initialize state using a call to gmp_randinit_mt() with the global variable state as an arg

Call gmp_rand_seed_ui() using state and seed as args

void randstate_clear(void):

Free all memory of the state by calling the function gmp_rand_clear(state)

numtheory.c:

- The gcd function computes the greatest common divisor of a and b and stores the value into d
- The method used to find the gcd in this function is by using mod rather than subtraction, which is much slower
- mpz means a multi precision integer, used to represent very large numbers

void gcd(mpz_t d, mpz_t a, mpz_t b):

while b is not equal to 0:

set d equal to the value of b

set b equal to the value of a mod b

return once the while loop is finished

- The mod_inverse function computes the inverse i of a modulo n
- The function uses parallel assignment of variables, which means that the assignments happen simultaneously in the code
- Temporary variables will be required to store the original values of some variables in order to do parallel assignment
- If the modular inverse cannot be found, set i to 0

void mod_inverse(mpz_t i, mpz_t a, mpz_t n):

set variables r and temp r (r') to n and a respectively

set variables t and temp t(t') to c and 1 respectively

while temp r is not equal to 0:

set variable q to floor of (r / temp r)

set temp r to (r - (q * temp r))

set r to the value of temp r

set temp t to (t-(q * temp t))

set t to the value of temp t

if r is greater than 1:

set mpz_i to 0

if t < 0:

set t to the value of (t + n)

- The pow_mod function computes base raised to the exponent power modulo modulus and performs fast modular exponentiation
- Stores the computed result in out

void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus):

set the value of out to 1

set the value of p to the value of base

while exponent is greater than 0:

if exponent mod 2 is not equal to 0:

set the value of out to ((v * p) mod modulus)

set the value of p to ((p * p) mod modulus)

set the value of exponent to the floor division of (d/2)

- The is_prime function conducts the Miller_Rabin primality test to determine whether or not a number n is prime using iters number of Miller-Rabin iterations
- The Miller-Rabin test is not a guarantee, just with a high probability
- The chances of the Miller_Rabin test being wrong is determined by (¼)^iters, making the test extremely likely with just a small number if iters

bool is_prime(mpz_t n, uint64_t iters):

find r using the formula n - 1 = 2^(s)r such that r is odd

set i to loop from values 1 to k using a for loop:

choose a random number out of (2, 3, ...n-2) using random() and applying an offset to the randomly generated number to manipulate the range

call pow_mod function using a created variable y as the arg for out

if y is not equal to 1 and y is not equal to (n-1):

set variable j to 1

while j is less than or equal to (s-1) and y is not equal to (n-1):

set y to the arg for out in pow_mod()

if y is equal to 1:

return false

set j to the value of j + 1

if y is not equal to (n-1):

return false

return true

- The make_prime function generates a new prime number stored in p
- The generated prime should be at least bits number of bits long
- Primality of the generated number needs to be tested using is_prime function and with iters number of iterations

void make_prime(mpz_t p, uint64_t bits, uint64_t iters):

While the randomly generated number fails the is prime test and its number of bits in p is less than argument bits:

use mpz_urandomb() to generate a random number of nbits using bits as an arg

test the generated number using the is_prime function using iters as the arg for iterations

rsa.c:

- The rsa_make_pub function creates the parts of a new RSA public key: 2 large primes p, q, their product n, and the public exponent e
- The floor of log base 2 n is needed in the function so a mpz function that calculate the bits in a number n is needed

- mpz_sizeinbase(with n and 2) as its arguments is needed to calculate the value k

void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters):

use make_prime() function to create primes p and q

decide the number of bits that go to each prime using the formula log base 2 (n) >= nbits

the number of bits that go into p is specified as a random number in the range of the floor (nbits/4, (3*nbits)/4).

the bits that go into q is specified by nbits - the number of bits going into p

compute the totient given the formula pi (n) = (p-1)(q-1)

for i to nbits:

generate random numbers of nbits using mpz_urandomb()

compute gcd() of each number and the totient

break if a number is found that is coprime with the totient, that is the gcd of the randomly generated number and the totient is one

set e to that coprime number

- The rsa_write_pub function writes a public RSA key to pbfile
- The format of the trailing line should be written with a trailing line after each variable
- The values n, e, and s should be written as hexstrings

void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile):

write the public RSA key to a pbfile using gmp_fprintf statements

format of public key is n, e, s, username each ending with a newline

use gmp_printf() to format for writing hexstrings

The format specifier for writing hexstrings is %Zx

- The rsa_read_pub function reads a RSA key from pbfile
- The formatting is the same as the formatting from rsa_write_pub

void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile):

read the public RSA key from a pbfile using gmp scan functions

format of public key is n, e, s, username each ending with a newline

use gmp functions to format for reading hexstrings

- The rsa_make_priv function creates a new RSA private key d given primes q and p and the public exponent e

void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q):

set the variable d to the value of inverse of e modulo (phi (n) = (p -1)(q-1))

- The rsa_write_priv function writes a private key to pvfile
- The output should be n and s written as hexstrings and each having a trailing newline

void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile):

values n and d should be written as hexstrings using gmp functions

write the format of n, d followed by trailing newline to pvfile

- The rsa_read_priv function reads a private RSA key from pvfile, the format of which is the same as the format in rsa_write_priv

void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile):

values n and d should be read as hexstrings using gmp functions

read the format of n, d followed by trailing newline from pvfile

void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n):

RSA encryption, computes ciphertext c by encrypting m using e and modulus n

Encryption defined by the formula E(m) = c = m^e(mod n)

- The function rsa_encrypt_file encrypts the contents of infile and writes the encrypted contents to outfile

void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e):

encrypt the contents of infile using blocks and write the encryption to outfile

restrict the values of block, value of block must be less than n, block cannot be 0, and block cannot be 1

set block size k equal to floor of ((log base 2 (n) - 1) / 8)

dynamically allocate memory for an array of k bytes type(uint8_t)

set zeroth byte of the block to 0xFF

read from infile until there are no more bytes to read using the fread() function

use mpz_import to convert the read bytes adding 1 to the bytes read to account for the prepended 0xFF

set the order parameter of mpz_import to 1 for most significant word first, 1 for the endian parameter, and 0 for the nails parameter

encrypt the message m using rsa_encrypt(), then output the encrypted number to outfile, printing the number as a hexstring followed by a trailing newline

void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n):

performs RSA decryption and computes the message m by decrypting ciphertext c using private key given in d and public modulus n

decryption is defined by D(c) = m = c^d(mod n)

- the rsa_decrypt_file function decrypts the contents of infile and writes the decrypted contents to outfile

void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d):

set block size k equal to the floor of ((log base 2 (n) - 1) / 8)

dynamically allocate an array of type (uint8_t *)

while there are still unprocessed bytes in infile:

scan in bytes from infile and set j to the bytes scanned

the hex string scanned from infile will be set to an mpz_t c

Use mpz_export to convert c back into bytes and set j to the number of bytes converted (a parameter of mpz_export)

set the order parameter to 1 for the most significant word first, 1 for the endian parameter, and 0 for the nails parameter

write out j - 1 bytes from index 1 of the block to outfile because index 0 contains the prepended byte 0xFF

void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n):

produces signature s by signing message m with private key d and modulus n

formula for signing is S(m) = s = m^d(mod n)

bool rsa_vertify(mpz_t, mpz_t s, mpz_t e, mpz_t n):

performs RSA verification and returns if s signature is verified and false otherwise

t = V(s) = s^e(mod n)

use the pow_mod function from numetheory to calculate t

signature is verified only if t is the same as m

keygen.c:

implement the specified command line arguments: b (minimum bits for modulus n), i (number of Miller-Rabin iterations for testing primes), n (public key file), d (private key file), s (random seed), v (verbose output), and h (program synopsis and usage) parse command line arguments

use fopen to open the public and private key files

use fchmod(fileno(0600)) to set the permissions of the private key file to read and write for the user only

initialize the random state using randstate_ui(seed) and the set seed

use rsa_make_pub (p, q, n, e, nbits, iters) and rsa_make_priv (d, e, p, q) to generate the public and private keys respectively

set a string to the value of getenv("USER") to get the username of the user

convert the username to an mpz_t using mpz_set_str() using a base of 62

use rsa_sign(s, name, d, n) to compute the signature s of the username

write the computed public and private keys to their respective files using rsa_write_pub(n, e, s, username, public key) and rsa_write_priv(n, d, private key)

close the public and private key files

clear the random state with randstate_clear()

encrypt.c:

implement the command line options: i, for the input file, o, for the output file, n, for the public key file, v for verbose printing, and h for help and synopsis message

open the public key using fopen

read the public key from the open file using rsa_read_pub(n, e, s, username, public key)

convert the username that was read into an mpz_t using mpz_str()

verify that the signature using rsa_verify(username, s, e, n)

encrypt the file using rsa_encrypt(input, output, n, e)

close the opened public key file and clear any used variables

decrypt.c:

implement the command line options: i, for the input file, o, for the output file, n, for the private key file, v, for verbose printing, and h, for help and synopsis message

parse through the command-line arguments using optarg

open the private key file using fopen()

read the private key from the opened private key file using rsa_read_priv(n, d, private key)

decrypt the file using rsa_decrypt_file(input, output, n, d);

close the opened private key file and clear used variables

Credit:

- I attended Eugene's section on 11/9/21 and 11/16/21 for general guidance and explanation of the assignment
- I attended Christan's section on 11/12/21 and 11/19/21 in which he provided general pseudocode for some functions and opened breakout rooms for individual help
- I used the pseudocode provided in the assignment 6 doc by Professor Long to implement the mathematical functions in my code