Deeraj Gurram

dgurram@ucsc.edu

11 November 2021

CSE 13S Fall 2021

Assignment 6 - Public Key Cryptography

Design Document

Description of the Program:

This assignment consists of 3 different programs that accomplish different tasks. The first program file is keygen.c, and is used as the key generator which produces RSA public and private key pairs. The second program file is encrypt.c which will encrypt files using the public key. The last program file is decrypt.c which will decrypt the encrypted files using the corresponding private key. Keygen.c accepts command line input to specify various values in and inputs. The encryptor and decryptor functions similarly take in command line inputs that specify values like the input and output files to encrypt/decrypt and enable verbose printing.

Deliverables:

1. decrypt.c
    - Functions unknown - I haven't coded this program file yet
    - This program file accepts command line input and uses the given specifications to open the private key file, read the private key from the opened file, Print more info if verbose output is enabled, decrypt the file, and close the private key file.
2. encrypt.c
    - Functions unknown - I haven't coded this program file yet
    - This program file accepts command line input and uses the given specifications to open the public key, read the public key, print more info if verbose output is enabled, convert the username into an mpz_t, encrypt the file, and close the public key file.
3. keygen.c
    - Functions unknown - I haven't coded this program file yet
    - This program file accepts command line input and uses the given specifications to open the public and private key files, make sure the private key permissions are set to 0600, Initialize the random state, make the public and private keys, Get the current user's name, convert the user's name into an mpz_t, and write the computed public and private keys to their files. If verbose output is enabled, a variety of information is printed before closing the public and private key files and clearing the random state.
4. numtheory.c

- void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus) performs fast modular exponentiation, computing base raised to the exponent power modulo modulus, and storing the computed result in out.
- bool is_prime(mpz_t n, uint64_t iters) conducts the Miller-Rabin primality test to indicate whether or not n is prime using iters number of Miller-Rabin iterations.
- void make_prime(mpz_t p, uint64_t bits, uint64_t iters) generates a new prime number stored in p.
- void gcd(mpz_t d, mpz_t a, mpz_t b) computes the greatest common divisor of a and b, storing the value of the computed divisor in d.
- void mod_inverse(mpz_t i, mpz_t a, mpz_t n) computes the inverse i of a modulo n.

5. randstate.c
    - void randstate_init(uint64_t seed) initializes the global random state named state with a Mersenne Twister algorithm, using seed as the random seed.
    - void randstate_clear(void) clears and frees all memory used by the initialized global random state named state.
6. rsa.c
    - void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters) Creates parts of a new RSA public key: two large primes p and q, their product n, and the public exponent e.
    - void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile) writes a public RSA key to pbfile.
    - void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile) reads a public RSA key from pbfile.
    - void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q) creates a new RSA private key d given primes p and q and public exponent e.

- void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile) write a private RSA key to pvfile.
- void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile) reads a private RSA ey from pvfile.
- void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n) performs RSA encryption, computing ciphertext c by encrypting message m using public exponent e and modulus n.
- void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e) encrypts the contents of infile, writing the encrypted contents to outfile.
- void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n) performs RSA decryption, computing message m by decrypting ciphertext c using private key d and public modulus n.
- void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d) decrypts the contents of infile, writing the decrypted contents to outfile.
- void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n) performs RSA signing, producing signature s by signing message m using private key d and public modulus n.
- bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n) performs RSA verification, returning true if signature s is verified and false otherwise.

7. numtheory.h
    - This contains the implementations of the number theory functions.
8. randstate.h
    - This specifies the interface for initializing and clearing the random state
9. rsa.h
    - This specifies the interface for the RSA library
10. Makefile

- Builds the encrypt, decrypt, and keygen executables, with each one able to be individually built. Also removes all files that are compiler generated, and formats all source code including the header files.

11. README.md
    - Gives a brief description of the program and Makefile and also lists the command line options that the program accepts. Also notes any errors or bugs found in the code and gives credit to sources of code or help.
12. DESIGN.pdf
    - Provides a description of the assignment and program files, the deliverables, and pseudocode for each of the .c files. Also describes the design and design process for the program.

Pseudocode:

numtheory.c

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

Initialize a variable v and set it equal to 1

Initialize a variable p and set it equal to argument a

Create a while loop where d must be greater than 0

Check if d is odd

Set v equal to (v multiplied by p) modded by n

Set p equal to (p multiplied by p) modded by n

Set d equal to d divided by 2

Return v

// This function performs fast modular exponentiation where the base is raised to the

// modulus power, and the result is stored in out

bool is_prime(mpz_t n, uint64_t iters)

Initialize a variable r and set it equal to 1

Initialize a variable s

Set n - 1 equal to 2 raised to the s power multiplied by r

Create a for loop where i starts at 1 and ends when it reaches k

Choose a random number between the values 2 and n - 2

Set the variable y equal to pow_mode(a,r,n)

Check if y doesn't equal 1 and y doesn't equal n - 1

Set variable j equal to 1

Create a while loop where j must be less than or equal to s - 1 and y

doesn't equal n - 1

Set y equal to power_mode(y, 2, n)

Check if y equals 1

Return false

Increment j by 1

Check if y doesn't equal n - 1

Return false

Return true

// This function tests a number stored in argument n and goes through iters

// number of iterations to check if a number is prime

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

Create a while loop where iters must be greater than 0

Set variable w equal to a randomly generated prime number of size bits number of bits

Check if function is_prime(w, iters) returns true

Set p equal to w

Set iters equal to 0

Else

Decrement iters by 1

// This function generates a prime number of size bits and stores the result in

// argument p. The function makes sure to check if a number is prime using

// function is_prime.

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

Create a while loop where b can't equal 0

Set t equal to b

Set b equal to a modded by b

Set a equal to t

Return a

// Computes the greatest common divisor of a and b and stores the computed divisor in //

d

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

Set r equal to n

Set r prime equal to a

Set t equal to 0

Set t prime equal to 1

Create a while loop where r prime can’t equal 0

Set q equal to r divided by r prime

Set (r, r prime) equal to (r prime, r - q multiplied by r prime)

Set (t, t prime) equal to (t prime, t - q multiplied by t prime)

Check if r is greater than 1

Return the value no inverse

Check if t is less than 0

Set t equal to t plus n

Return t

// Computes the inverse of a modulo

randstate.c

void randstate_init(uint64_t seed)

Call the gmp function gmp_randinit_mt(state)

Call the gmp function gmp_randseed_ui(state, seed)

void randstate_clear(void)

Call the gmp function mpz_clear(state)

rsa.c

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

Initialize boolean check and set it equal to false

Initialize variable rand

Call function make_prime(p, random number(nbits/4, (3 * nbits)/4), iters)

Call function make_prime(q, random number(nbits/4, (3 * nbits)/4), iters)

Set argument n equal to (p - 1) * (q - 1)

Create a while loop where boolean check can't equal true

Set variable rand equal to the gmp function mpz_urandomb(n, state, nbits)

Call the function gcd(e, rand, n)

Check if e is coprime with the totient

Set check equal to true

// This function creates parts of a new RSA public key

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

Open the pbfile

Use the function write(pbfile, nbits, n)

Print a newline

Use the function write(pbfile, nbits, e)

Print a newline

Use the function write(pbfile, nbits, s)

Print a newline

Use the function write(pbfile, nbits, username)

Print a newline

Close pbfile

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

Open the pbfile

Use function read(pbfile, nbits, n)

Print a newline

Use function read(pbfile, nbits, e)

Print a newline

Use function read(pbfile, nbits, s)

Print a newline

Use function read(pbfile, nbits, username)

Print a newline

// Reads a public RSA key from pbfile

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

Set n equal to (p - 1) * (q - 1)

Set d equal to function mod_inverse(e, p, n)

// Creates a new RSA private key

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

Call the function write(pvfile, nbits, n)

Print a newline

Call the function write(pvfile, nbits, d)

Print a newline

// Writes a private RSA key to pvfile

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

Call the function read(pvfile, nbits, n)

Print a newline

Call the function read(pvfile, nbits, d)

Print a newline

// Reads a private RSA key from pvfile

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

Call the function power_mod(c, m, e, n)

// Performs RSA encryption

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

Set variable k equal to (logbase 2 (n) - 1 / 8)

Allocate memory of size uint8_t to an array of size k

Set k[0] to 0xFF

Create a while loop where k must be greater than 0

Read in k - 1 bytes from the infile

Set j equal to the number of bytes read

Place each read bytes into array k[1] and successive bytes into the next indices

Call function mpz_import(___, ___, 1, k, 1, 0, ___)

// Don't know what to put in the ___ positions yet

Use the function write(n, nbits, pbfile)

Set m equal to rsa_encrypt(k, m, e, n)

// Encrypts the contents of the infile and write the encrypted contents to outfile

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

Call the function power_mode(m, c, d, n)

// Performs RSA decryption

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

Set variable k equal to (logbase 2 (n) - 1 / 8)

Allocate memory of size uint8_t to an array of size k

Create a while loop where k must be greater than 0

Call gmp_fscanf(infile, mpz_t c)

Call function mpz_export(c, j, 1, ___, 1, 0, ___)

// Don't know what to put in the ___ positions yet

Start at k[1] and write out j - 1 bytes to the outfile

// Decrypts the contents of infile and writes the decrypted contents to outfile

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

Call function power_mod(s, m, d, n)

// Performs RSA signing

bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n);

Set t equal to power_mod(t, s, e, n)

Check if t is equal to m

Return true

Else

Return false

// Performs RSA verification

keygen.c

Haven’t coded this program file yet

encrypt.c

Haven’t coded this program file yet

decrypt.c

Haven’t coded this program file yet

Credit:

1. Used the program file descriptions and explanations from the asgn6.pdf by Dr. Long
2. Used the pseudocode by Dr. Long from the asgn6.pdf