

Deeraj Gurram

[dgurram@ucsc.edu](mailto:dgurram@ucsc.edu)

21 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. There are also 3 other files called `randstad.c` which is used to initialize a `randstad` and clear a `randstate`, `numtheory.c` which computes the mathematical portion of the code such as `pow_mod`, `is_prime`, and `mod_inverse`, and `rsa.c` which does the reading and writing to files portion as well as the encryption and decryption of files.

## **Deliverables:**

### 1. decrypt.c

- 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

- 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

- 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 key 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 5 mpz variables to hold the values of the parameters and to serve as temporary variables

Set mpz v equal to 1, mpz p equal to argument a, mpz p equal to argument base, mpz exp equal to argument exponent, mpz mod equal to argument modulus

Create a while loop where mpz exp must be greater than 0

    Re set the value of mpz d to mpz exp each time the loop runs

    Check if d is odd

        Set v equal to (v multiplied by p)

        Set v equal to v modded by mod

    Set p equal to (p multiplied by p)

    Set p equal to p modded by mod

    Set exp equal to exp divided by 2

Set the argument out equal to v

Clear the initialized mpz's

// 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 9 mpz's to hold the value of n, n-1, two, and a variety of other tasks

Set mpz n\_copy equal to n

Hardcode some if statements to take care of cases where n = 0 to n = 3. The

Miller-Rabin formula doesn't take care of these values

Set  $n - 1$  equal to  $n\_copy$  minus 1

Create a `mpz_t s` equal to 2

Create a while loop to increment  $s$  if  $n-1$  is divisible by  $2^s$

Decrement  $s$  by 1

Create a for loop where  $i$  starts at 1 and ends when it reaches argument `iters`

Set `mpz_t upper_bound` equal to  $n\_copy$  minus 3

Choose a random number between the values `state` and `upper_bound` and store the result in `mpz_t a`

Call `pow_mode(y,a,r,n_copy)` to store the result in  $y$

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

Set `mpz_t 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 `mpz_t two` equal to 2

Call `pow_mod(y, y, two, n_copy)` to store the result in  $y$

Check if  $y$  equals 1

Clear the `mpz_t`'s

Return false

Increment  $j$  by 1

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

Clear the `mpz_t`'s

Return false

Clear the `mpz_t`'s

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)**

Call `mpz_urandmob(p, state, bits)`

Create a while loop where p has to be prime and the size of p must be less than argument bits

Call `mpz_urandmob(p, state, bits)`

// 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)**

Initialize 3 mpz's to act as temporary variables that hold the values of the arguments

Set `ay` equal to argument `a`

Set `be` equal to argument `b`

Create a while loop where `b` can't equal 0

Set `t` equal to `be`

Set `be` equal to `ay modded by be`

Set `ay` equal to `t`

Set `d` equal to `ay`

Clear the mpz's

// 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)**

Initialize 7 mpz's to act as temporary variables and other tasks

Set `mpz r` equal to `n` and set `mpz r_prime` equal to `a`

Set `mpz_t t` equal to 0 and `mset mpz_t 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 temp equal to r\_prime

Set r\_prime equal to q multiplied by r\_prime

Set r\_prime equal to r minus by r\_prime

Set r equal to temp

Set temp2 equal to t\_prime

Set t\_prime equal to q multiplied by t\_prime

Set t\_prime equal to t minus t\_prime

Set t equal to temp2

Check if r is greater than 1

Set t to 0

Check if t is less than 0

Increment t by n

Set i equal to t

Clear the mpz's

// 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 7 mpz's to store the value of n, totient, p-1, q-1, and more

Set mpz pbits equal to a random number between nbits/4, (3 \* nbits)/4

Set mpz qbits equal to argument nbits minus pbits

Call function make\_prime(p, pbits+1, iters)

Call function make\_prime(q, qbits+1, iters)

Set mpz product equal to p\*q

Set argument n equal to product

Set mpz one equal to 1

Calculate the totient by setting mpz p-1 equal to p minus 1

Set mpz q-1 equal to q minus 1

Set mpz\_t t equal to p-1 multiplied by q-1 to calculate the totient

Set mp\_bitcnt\_t en equal to nbits

Set mpz rand equal to argument e

Create a do while loop where the mpz gcd\_e\_n must be greater than mpz one

    Call function mpz\_urandomb(rand, state, en)

    Call the function gcd(gcd\_e\_n, rand, t)

Set mpz e equal to rand

Clear the mpz's

// 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);**

Use function gmp\_fprintf to print the value of n to the pbfile

Use function gmp\_fprintf to print the value of e to the pbfile

Use function gmp\_fprintf to print the value of s to the pbfile

Use function gmp\_fprintf to print the username to the pbfile

**void rsa\_read\_pub(mpz\_t n, mpz\_t e, mpz\_t s, char username[], FILE \*pbfile);**

Use function gmp\_fscanf to scan the value of n from the pbfile and store it in n

Use function gmp\_fscanf to scan the value of e from the pbfile and store it in e

Use function gmp\_fscanf to scan the value of s from the pbfile and store it in s

Use function gmp\_fscanf to scan the value of the username from the pbfile and store it in username

// Reads a public RSA key from pbfile

**void rsa\_make\_priv(mpz\_t d, mpz\_t e, mpz\_t p, mpz\_t q);**

Initialize 3 mpz's to hold the value of p-1, q-1 and (p-1)\*(q-1)

Set mpz p-1 equal to p minus 1

Set mpz q-1 equal to q minus 1

Set mpz n equal to p-1 multiplied by q-1

Call function mod\_inverse to set d equal to the mod inverse of e and n

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

Set d equal to function mod\_inverse(e, p, n)

Clear the mpz's

// Creates a new RSA private key

**void rsa\_write\_priv(mpz\_t n, mpz\_t d, FILE \*pvfile);**

Use function gmp\_fprintf to print the value of n to the pvfile

Use function gmp\_fprintf to print the value of d to the pvfile

// Writes a private RSA key to pvfile

**void rsa\_read\_priv(mpz\_t n, mpz\_t d, FILE \*pvfile);**

Use function gmp\_fscanf to scan the value of n from the pvfile and store it in n

Use function gmp\_fscanf to scan the value of d from the pvfile and store it in d

// 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 pow\_mod(c, m, e, n)

// Performs RSA encryption

**void rsa\_encrypt\_file(FILE \*infile, FILE \*outfile, mpz\_t n, mpz\_t e);**

Set variable size\_t k equal to (mpz\_sizeinbase(n, 2) - 1) / 8

Set variable size\_t j equal to 0

Allocate memory of size uint8\_t to an array of size k called array

Initialize 2 mpz's to hold the message and ciphertext

Set array[0] to 0xFF

Create a while loop where feof(infile) must equal 0

Set j equal to fread(array +1, the size of a uint8\_t, k-1, infile)

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

Call function mpz\_import(m, j+1, 1, the size of a uint8\_t, 1, 0, array)

Use the function rsa\_encrypt(c, m, e, n)

Use function gmp\_fprintf to print the encrypted number c to the outfile

Clear the mpz's

Free the array

// 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 pow\_mod(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  $(\text{mpz\_sizeinbase}(n, 2) - 1) / 8$

Allocate memory of size uint8\_t to an array of size k called array

Create a while loop where feof(infile) must equal 0

Call gmp\_fscanf to read the value of the encrypted number of the infile and store it in mpz c

Check if c is greater than 0 so it's a valid value

Call rsa\_decrypt(m, c, d, n)

Call function mpz\_export(array, memory address of j, 1, size of a uint8\_t, 1, 0, m)

Call fwrite(array+1, the size of a uint8\_t, j-1, outfile)

Clear the mpz's

Free the array

// 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 pow\_mod(s, m, d, n)

// Performs RSA signing

**bool rsa\_verify(mpz\_t m, mpz\_t s, mpz\_t e, mpz\_t n);**

Initialize an mpz

Call pw\_mod(t, s, e, n)

Check if t is equal to m

Clear the mpz

Return true

Else

Clear the mpz

Return false

// Performs RSA verification

## keygen.c

### **Void usage(char \*exec)**

Call function fprintf(stderr, help message, exec);

### **Int main(int argc, char \*\*argv)**

Initialize two char's called pbname and pvname

Set pbname equal to rsa.pub

Det pvname equal to rsa.priv

Initialize the default values for the command line specifications, all values are ints except verbose which is a bool

Opt = 0

Seed = function time(NULL)

Iters = 50

Bits = 256

Verbose = false

Create a while loop to loop through each case of the command line options

Create a FILE called pbfile and set it equal to fopen(pbname, w)

Check if pbfile is Null and if so, print an error message

Create a FILE called pvfile and set it equal to fopen(pvname, w)

Check if pvfile is Null and if so, print an error message

Set the permissions of pvfile using fchmod(fileno(pvfile), 0600)



Call the function `randstate_init(seed)`

Initialize the 6 mpz's to hold values for each function call

Call `rsa_make_pub` on `p`, `q`, `n`, `e`, `bits`, `iters`

Call `rsa_make_priv` on `d`, `e`, `p`, `q`

Call `rsa_sign` on `s`, `s`, `d`, `n`

Call `rsa_write_pub` on `n`, `e`, `s`, function `getenv(USER)`, `pbfile`

Call `rsa_write_priv` on `n`, `d`, `pvfile`

Check if `verbose` equals `true`

    Print the key generation statistics

Close the `pvfile` and `pbfile`

Call `randstate_clear`

Clear the mpz's

## encrypt.c

### **Void usage(char \*exec)**

Call function fprintf(stderr, help message, exec);

### **Int main(int argc, char \*\*argv)**

Initialize char called pbname

Set pbname equal to rsa.pub

Initialize two FILE's called infile and outfile

Set infile equal to stdin

Set outfile equal to stdout

Initialize the default values for the command line specifications

Opt = 0

Verbose = false

Create a while loop to loop through each case of the command line options

For case i and o, set infile and outfile equal to fopen(optarg, r) and fopen(optarg, w) respectively

Check if the return value is NULL, and if so print an error message to stderr and return

EXIT\_FAILURE

Create FILE pbfile and set it equal to fopen(pbname, r)

Check if pbfile is NULL

Print an error message

Initialize 4 mpz's to hold values for each function call

Call `rsa_read_pub` on `n`, `e`, `s`, `getenv(USER)`, `pbfile`

Check if `verbose` equals `true` and print out the encryption statistics if so

Call `mpz_set_str` on `m`, `getenv(USER)`, 62

Call `rsa_verify` on `m`, `s`, `e`, `n`

Call `rsa_encrypt_file` on `infile`, `outfile`, `n`, `e`

Close the `infile`, `outfile`, and `pbfile`

Clear the mpz's

## decrypt.c

### **Void usage(char \*exec)**

Call function fprintf(stderr, help message, exec);

### **Int main(int argc, char \*\*argv)**

Initialize char called pvname

Set pvname equal to rsa.priv

Initialize two FILE's called infile and outfile

Set infile equal to stdin

Set outfile equal to stdout

Initialize the default values for the command line specifications

Opt = 0

Verbose = false

Create a while loop to loop through each case of the command line options

For case i and o, set infile and outfile equal to fopen(optarg, r) and fopen(optarg, w) respectively

Check if the return value is NULL, and if so print an error message to stderr and return

EXIT\_FAILURE

Create FILE pvfile and set it equal to fopen(pvname, r)

Check if pvfile is NULL

Print an error message

Initialize 2 mpz's to hold values for each function call

Call `rsa_read_priv` on `n`, `d`, `pvfile`

Check if `verbose` equals `true` and print out the decryption statistics if so

Call `rsa_decrypt_file` on `infile`, `outfile`, `n`, `d`

Close the `infile`, `outfile`, and `pvfile`

Clear the mpz's

### **Credit:**

1. Used the provided pseudocode from the asgn6.pdf from Dr. Long
2. Used Dr. Long's pseudocode for gcd, mod\_inverse, pow\_mod, and is\_prime
3. Used the assignment and program file descriptions from the asgn6.pdf from Dr. Long
4. Got most of is\_prime from tutor Eric Hernandez's section
5. Used the pseudocode for make\_prime from tutor Eric Hernandez's section
6. Used Dr. Long's explanation of each function to base my code off of for rsa.c
7. Used TA Eric Hernandez's pseudocode for rsa\_make\_pub
8. Got the idea of using feof(infile) from Miles on Discord and tutor Eric Hernandez
9. Got the parameters of mpz\_export from tutor Eric Hernandez for rsa\_decrypt\_file
10. Used pseudocode from tutor Eric Hernandez for keygen.c and decrypt.c
11. Used the steps listed by Dr. Long for keygen.c, encrypt.c, and decrypt.c
12. Copied the error messages and help messages from the resources repo
13. Based the structure and setup of the command line options off of Dr. Long's tsp.c in asgn4