Assignment 5: *Design*

Sriramya Prayaga

February 17, 2022

Abstract

In this document, I will be describing my design process for Assignment 5. Assignment 5 is essentially simulating an RSA encryption/decryption algorithm — that is, we need to take a text file, encrypt and decrypt it using randomly-generated public and private keys, and display the output (using command-line options).

1 Pseudocode/Structure

Because there are many moving parts to create the encryption and decryption algorithm, there are many files required for this assignment.

1.1 Random State File

In order to construct the keygen/encryption/decryption programs, we need to create a random state generator file (as many functions for this assignment will require the usage of random numbers.) This file is called randomstate.c, and consists of two functions: randstate_init() and randstate_clear(). As we are supposed to use the GMP library for this assignment, the functions in the random state file —and all other files below — should use and make calls to the relevant GMP random state functions (as described below).

- randstate_init(uint64_t seed): This file initializes a gmp_randstate_t type variable with the given uint64 seed. It uses the gmp_randinit_mt() for the random algorithm, and gmp_randseed_ui() for seeding. It also calls srandom() with the given seed.
- randstate_clear(void): frees memory that was used by randstate_init().

The psuedocode for this is as follows:

```
initialize gmp_randstate_t state

void randstate_init(uint64_t seed) {
        initialize random state with the Mersenne Twister algorithm
        seed the function with given state and seed
        seed srandom()
}
```

```
void randstate_clear(void) {
        clear random state
}
```

1.2 Number Theory File

The number theory file calculates important values for the RSA module (described later.) It has the following functions: pow_mod(), make_prime(), is_prime(), gcd(), mod_inverse(). The file with these functions is called numtheory.c.

- pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus): This function calculates the value of $base^{exponent}$, and stores it in out. The gist is that we are calculating the value of a number raised to a power in O($\log_2(n)$) steps and using a modulus value to simplify large calculations.
- is_prime(mpz_t n, uint64_t iters): The pseudocode for this function is given in the Assignment 5 document, however, it essentially uses the Miller-Rabin test to check if a number is prime.
- make_prime(): This function generates random number, and calls is_prime() to check for the number for primality.
- gcd(mpz_t d, mpz_t a, mpz_t b): This function computes the greatest common divisor of a and b. The gcd value is stored in d.
- mod_inverse(mpz_t i, mpz_t a, mpz_t n): This pseudocode for this function has also been provided for us, and it computes the modular inverse of $a \pmod n$.

The pseudocode for this file is as shown below:

```
void gcd(mpz_t d, mpz_t a, mpz_t b) {
        while (b is not 0) {
                mpz set d to b
                mpz set b to a mod b
                mpz set a to d
         }
         mpz set d to a
}

mod_inverse(mpz_t i, mpz_t a, mpz_t n) {
         mpz set t2 to 1
         mpz set r1 to n
         mpz set r2 to a

         while (r2 is not 0) {
                mpz set q to r1/r2
                mpz set temp1 to r1
                mpz set temp2 to r2
                mpz set r1 to r2
                mpz set temp2 to temp2 * q
```

```
                    mpz set temp1 to temp1 - temp2
                    mpz set r2 to temp1
                    mpz set temp1 to t1
                    mpz set temp2 to t2
                    mpz set t1 to t2
                    mpz set temp2 to temp2 * q
                    mpz set temp1 to temp1 - temp2
                    mpz set t2 to temp1
            }

            mpz set 1 to t1

            if mpz r1 is greater than 1 {
                    mpz set 1 to 0
            } else if mpz t1 is less than 0 {
                    mpz set t1 to t1 + n
                    mpz set i to t1
            }
            clear mpz variables
   }

void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus) {
    mpz set v to 1
    mpz set p to base
    mpz set exp to exponent

    while (mpz exp is greater than 0) {
        if (exp is odd) {
            mpz set v to v * p
            mpz set v to v * modulus
        }
        mpz set p to p * p
        mpz set p to p mod modulus
        mpz set exp to exp/2
    }
    mpz set out to v
    clear mpz variables
}

bool is_prime(mpz_t n, uint64_t iters) {
    mpz set r to n - 1
    mpz set two to 2

    mpz set temp to n mod 2
```

```
if (mpz n < 2, n = 0, or n != 2) {
    clear mpz variables
    return false
}

if (mpz n = 2, or n = 3) {
                clear mpz variables
                return true
        }

while (mpz r is even) {
    mpz set r to r/2
    mpz set s to s + 1
}

mpz set s to s - 1

for (i in the range of 1 to iters) {
    set temp to temp - 3
    create random number roll using state and temp with urandomm()
    mpz set temp to temp + 1
    mpz set roll to roll mod temp
    mpz set roll to roll + 2

    mpz set temp to n - 1

    if (mpz y is not equal to 1 and y is not equal to temp) {
        mpz set j to 1
        while (mpz j <= s and mpz y != temp) {
            call pow_mod(y, y, two, n)

            if (mpz y == 1) {
                clear mpz variables
                return false
            }
            mpz set j to j + 1
        }
        if (y != temp) {
            clear mpz variables
            return false
        }
    }
}
clear mpz variables
return true
```

```
}

void make_prime(mpz_t p, uint64_t bits, uint64_t iters) {
    create random number using n, state, and bits with urandomb()
    set test_prime = false;

    while (test primes is false) {
        create random number using n, state, and bits with urandomb()
        mpz set offset to 2^bits
        mpz set n to n + offset
        set test_prime = is_prime(n, iters)
    }
    mpz set p to n
    clear mpz variables
}
```

1.3 RSA Library File

Another integral file for this assignment is the RSA Library, which makes a public and private key, writes to and reads from pbfiles/pvfiles, and encrypts and decrypts text files. It also signs the message (and verifies the signature.) The functions in this file are: rsa_make_pub(), rsa_write_pub(), rsa_read_pub(), rsa_make_priv(), rsa_write_priv(), rsa_read_priv(), rsa_encrypt(), rsa_encrypt_file(), rsa_decrypt(), rsa_decrypt_file(), rsa_sign(), rsa_verify().

Note: for this file I created an lcm() helper function to avoid repeated computations.

- rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters): This file discerns two large primes (p and q), calculates their product n, and the value of e, the public exponent. The pseudocode for this function is as below:

```
        set p to a random() number of bits in the range [nbits/4, 3 * (nbits/4)]
        set q = nbits - p

        do {
             call make_prime(p, bits, iters) and make_prime(q, bits, iters)
             mpz set n to p * q
             set size to number of bits n has using mpz_sizeinbase()
        }
        while (size is less than nbits)
        compute lcm_out to (p-1)(q-1)/gcm(p-1, q-1)
        mpz set lcm_copy to lcm_out

        do {
            call mpz_urandomb() with e, state, and nbits
            mpz set e_copy to e
            call gcd (temp, e_copy, e)
            mpz set e_copy to e
            mpz set lcm_out_copy to lcm_out
        }
        while (temp == 1)
        clear mpz variables
```

- rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile): gmp_fprintf() n, e, s (as hextrings), and username to the given public key pb file.
- rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile): Read with gmp_fscanf() n, e, s (as hextrings), and username from the given public key pb file.
- void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q): This function creates the private key d. This is done as so:

```
            call mod_inverse(d, e, ((p-1)(q-1))/gcd((p-1)(q-1)))
```

  As can be seen here, I calculated lcm for the third parameter of the function (using mpz variables). In reality, this will take more lines of code to initialize all the variables.
- rsa_write_priv(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile): Writes using gmp_fprintf() n, e, s (as hextrings), and username to the given private key pvfile.
- rsa_read_priv(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile): Read n, e, s (as hextrings), and username using gmp_fscanf() from the given private key pvfile.
- rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n): This function encrypts m. This is done as follows:

```
            call pow_mod(cipher_message, m, e, n) to store encrypted
            message in cipher_message
```

- rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e): This function encrypts the contents of infile in chunks of text. It is done as follows:

```
        set k = to (mpz_sizeinbase(n,2) - 1) /8
        dynamically allocate k bytes of memory with calloc (pointer named block)
        set block[0] = 0xFF

        do {
            set j = fread(block + 1, size of a uint_8, k-1, infile)
            if j = 0 {
                 break
            }
            call mpz_import(message, j + 1, size of a uint_8, 1, 0, block)
            call rsa_encrypt(ciphertext, message, e, n)
        print ciphertext to outfile with gmp_fprintf()

        }
        while (j equals k - 1)
    clear mpz variables
    free block
```

- rsa_decrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n): This function encrypts m. This is done as follows:

```
call pow_mod(message, c, d, n) to store decrypted message in message.
```

- rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d): This function works similarly to the function above that encrypts the file – that is, it decrypts the file in blocks. The pseudocode for this is as follows:

```
    set k = to (mpz_sizeinbase(n,2) - 1) /8
    dynamically allocate k bytes of memory with calloc (pointer named block)
    initialize size_t j;

    while (the end of infile has not been reached) {
        call rsa_decrypt(message, ciphertext, d, n)
        call mpz_export(block, address of j, 1, size of a uint8_t, 1, 0, message)
        fwrite(block + 1, size of uint8_t j - 1, outfile)
    }
    clear mpz variables
    free block
```

- rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n): This function creates an RSA signature. It is done as so :

```
    call pow_mod(s, m, d, n) and store signature value in s.
```

- bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n): In order to verify that an RSA signature, we need to find the inverse modulus. It can be found by using the mod_inverse() function:

```
    power_mod(val, s, e, n)
    inverse_mod(t, val, n)
    if m == t {
        return true
    }
    else {
        return false
    }
```

1.4 Key Generation File

This file generates the public and private keys for encryption/decryption. This file is called keygen.c. The function parses command line options to discern the parameters on how to generate the keys. This program calls on many functions mentioned in this document. The pseudocode for this function is as follows:

```
set default command-line option values
Parse the command line options "b:i:n:d:s:hv" with getopt()
while command line options are given {
 switch for {
   if cases ’b’ or ’i’ : reinitialize default bit value
   or iteration values to user-given values
   if case ’n’ or case ’d’: replace default key-files with user-given key files
   if case ’s’ replace default seed with new seed
   if case ’h’: print usage message
   if case ’v’: store that verbose mode is true
 }
}
open public and private key files with fopen()
call randstate_init(seed)

call rsa_make_pub(p, q, n, e, bits, iters)
call rsa_make_priv(d, e, p, q)

call getenv("USER") to get user’s name
call rsa_sign()
make usernsme into a mpz_t with mpz_set_str()

write public, private keys to destination file

if verbose mode is true:
   print verbose message

close opened files
clear random state
clear mpz_t variables
```

1.5 Encryptor File

This file encrypts a text file. The name of the file is encrypt.c. The function parses command line options to discern the parameters on how to encrypt a file. This program calls on many functions mentioned in this document. It prints the output of the file to the designated output file. The pseudocode for this file is as follows:

```
set default command-line option values
Parse the command line options "o:i:n:hv" with getopt()
```

```
while command line options are given {
 switch for {
   if case ’i’ : initialize input file name for encryption
   if case ’o’ : initialize output file name
   if case ’n’ : specify public key-file
   if case ’h’: print usage message
   if case ’v’: store that verbose mode is true
 }
}
open public key files with fopen()
call read_pub() to read public key-file

if verbose mode is true:
   print verbose message

make usernsme into a mpz_t with mpz_set_str()
verify signature with rsa_verify()

scan n and e from public key file
call rsa_encrypt_file()

close public key file, input file and output file
clear mpz_t variables
```

1.6 Decryptor File

This file decrypts text in a file that has been encrypted with a certain public key. This program parses command line arguments to get the specifications on how to output the decryption text. This program calls on many functions mentioned in this document. The pseudocode for this file is as follows:

```
set default command-line option values
Parse the command line options "o:i:n:hv" with getopt()
while command-line options are given {
 switch for {
   if case ’i’ : initialize input_file name for decryption
   if case ’o’ : initialize output file name
   if case ’n’ : specify private key-file
   if case ’h’: print usage message
   if case ’v’: store that verbose mode is true
 }
}
open private key file

if verbose mode is true:
   print public exponent, and private key
```

```
scan n and d from private key-file
call rsa_decrypt_file(input_file, output_file, n, private_key )

close private key file, input file, and output file
clear mpz_t variables
```

2 Summary

The above functions and files all contribute to creating an RSA encryption algorithm. The public and private keys for this algorithm are generated in the key generator file (using the RSA Library, Numerical Theory, and Random State files), and the keys are used in the Encryption and Decryption files to encrypt and decrypt the text files.