Assignment 5 Design

Nhan Nguyen

February 2022

1 Introduction

1.1 Description

The purpose of this program is to perform a traditional RSA encryption to encrypt any kind of messages using generated keys and decrypt those encrypted messages to get the original messages

1.2 List of Files

- **decrypt.c:** This file contains the main() function and also the implementation of a decryption program
- **encrypt.c:** This file contains the main() function and also the implementation of a encryption program
- **keygen.c:** This file contains the main() function and also the implementation of a key generation program
- **rsa.h:** The interface of rsa.c
- **rsa.c:** The implementation of the necessary functions for the rsa encryption
- **numtheory.h:** The interface of numtheory.c
- **numtheory.c:** The implementation of different number theory algorithms that is necessary for generating keys, calculations in the rsa.c
- **randstate.h:** The interface of randstate.c
- **randstate.c:** The implementation of initializing the random state and seed that is necessary for generating any random number generation from any function from gmp and random();
- **Makefile:** This file formats program into clang-format and also compiles it. This Makefile will be responsible for compiling all 3 main() programs and linking these programs each with all 3 files numtheory.c rsa.c randstate.c

- **README.md:** A text file in Markdown format that describes how to build the program, as well as describing the problems encountered while making the program

- **DESIGN.pdf:** This PDF file describes the idea behind the program with visualization and pseudocode

2 randstate.c

This file contains the global variable **state** and 2 functions `randstate_init()` and `randstate_clear()`

2.1 `randstate_init(seed)`

This function initializes the global variable **state** with a seed specified by the parameter and also set the seed for srandom(). Simply use `gmp_randstate_init()` and `gmp_randseed()` will do the trick
Pseudocode:

```
randstate_init(seed):
    initialize state
    set state's seed to seed
    srandom(seed)
```

2.2 `randstate_clear()`

This function clears the initialized **state**, this can be done with a simple `gmp_randclear()`
Pseudocode:

```
randstate_clear()
    clear global variable state
```

3 numtheory.c

This file contains all the necessary mathematical algorithms to generate the keys for the rsa encryption

3.1 `pow_mod(out, base, exponent, modulus)`

This function calculates the $base^{exponent}$ and output the answer to out, however since we're dealing with very big numbers this answer is modded with the modulus in order to get the congruent modulo. This process is achieve through this pseudocode:
Pseudocode:

```
pow_mod(out, base, exponent, modulus)
```

```
v = 1
p = base
while exponent ¿ 0:
    if exponennt mod 2 == 1:
        v = (v * p) mod modulus
    p = (p * p) mod modulus
    expoenent /= 2
out = v
```

3.2 is_prime(n, iter)

This function determines if the input number is a prime or not. The method used for this algorithm is the Miller-Rabin's test, and also filtering out some edge cases like the number is less than or equal to 1, or the number is even
Pseudocode:

```
is_prime(n, iter)
    if n mod 2 == 0 OR n ¡= 1: return
    r = n - 1
    s = 0
    while r mod 2 == 0:
        r /= 2
        s++
        for i from 0 to iter:
        a = random number from 2 to n - 2
        pow_mod(y, a, r, n)
        if y != 1 AND y != n - 1:
            j = 1
            while j ¡= s - 1 AND y != n - 1:
                pow_mod(y, y, 2, n)
                if y == 1: return false
                j++
            if y != n - 1: return false
    return true
```

3.3 make_prime(p, bits, iter)

This function will generate a random prime number that is at least **bits** long, since we need to generate primes the function `is_prime()` is needed and thus will do **iter** number of iterations for the Miller-Rabin test. To generate a number simply use `mpz_urandomb()` to generate the number, however since the number is generated from 0 to 2^{bits-1} you will need to add 2^bits to the randomly generated number to ensure that it has at least **bits** long
Pseudocode:

```
make_prime(p, bits, iter)
```

```
low = $2^{bits}$
while true:
    gen = random number from 0 to $2^{bits-1}$
    gen += low
    if is_prime(gen, iter) == true: break
p = gen
```

3.4 gcd(d, a, b)

This function calculates the gcd of any number 2 numbers and output it to the answer, the implementation of gcd is according to this pseudocode:
Pseudocode:

```
gcd(d, a, b)
    t = 0
    while b != 0:
        t = b
        b = a mod b
        a = t
    d = a
```

3.5 mod_inverse(t, a, n)

This function finds modular inverse of an integer a mod n. Note that not all a mod n can have a modular inverse. The implementation of the modular inverse algorithm is the following:
Pseudocode:

```
mod_inverse(i, a, n)
    r = n
    r2 = a
    t = 0
    t2 = 1
    while r2 != 0
        q = r / r2
        sub1 = r - q * r2
        sub2 = t - q * t2
        r = r2
        r2 = sub1
        t = t2
        t2 = sub2
    if r > 1:
        i = 0
        return
    if t < 0:
        t += n
    i = t
```

4 rsa.c

This file contains all the necessary functions to perform a RSA encryption

4.1 `rsa_make_pub(p, q, n, e, nbits, iters)`

This function is used to generate the public keys components: two primes p and q, prime n, exponent e. The process will require the use of LCM() which can be converted from GCD through this simple equation.

$$LCM(a,b) = \frac{a \cdot b}{GCD(a,b)}$$

In addition, the p's bits and q's bits are limited with p's bits being a random number from $\frac{bits}{4}$ to $\frac{bits \cdot 3}{4}$ while q's bits being nbits - p's bits. This is to make sure that n's bits is higher than nbits because $n = p \cdot q$. As for the generated values the ways to generate each values are as follows:

- p and q: generated through `make_prime()`
- n = p · q
- e is randomly generated until the `gcd(e, lcm(p - 1, q - 1)` == 1

Pseudocode:

```
rsa_make_pub(p, q, n, e, nbits, iters)
    pbits = random number from $\frac{bits}{4}$ to $\frac{bits \cdot 3}{4}$
    qbits = nbits - pbits
    make_prime(p, pbits, iters)
    make_prime(q, qbits, iters)
    gcd(lcm, p - 1, q - 1)
    lcm = (p - 1) * (q - 1) / lcm
    e = random number from 0 to $2^{nbits-1}$
    gcd(loop, lcm, e)
    while loop != 1:
        e = random number from 0 to $2^{nbits-1}$
        gcd(loop, lcm, e)
```

4.2 `rsa_write_pub(n, e, s, username, pbfile)`

This function will print out prime number n, exponent e, signature s, and the username into the specified pbfile. To do this, you will use gmp_fprintf() since the variable you are dealing with are in `mpz_t`

Pseudocode:

```
rsa_write_pub(n, e, s, username, pbfile)
    print n to pbfile
    print e to pbfile
    print s to pbfile
    print username to pbfile
```

4.3 `rsa_read_pub(n, e, s, username, pbfile)`

This function does the opposite of `rsa_write_pub()` since instead of writing to the file, this function reads the values from the file and assign it to the values in the parameter. Use gmp_fscanf() since the variable you are dealing with are in `mpz_t`

Pseudocode:

```
rsa_write_pub(n, e, s, username, pbfile)
    read n to pbfile
    read e to pbfile
    read s to pbfile
    read username to pbfile
```

4.4 `rsa_make_priv(d, e, p, q)`

This function generates the private key d. We can do this by finding the modular inverse of e mod lcm(p - 1, q - 1)

Pseudocode:

```
rsa_make_priv(d, e, p, q)
    gcd(lcm, p - 1, q - 1)
    lcm = (p - 1) * (q - 1) / lcm

    mod_inverse(d, e, lcm)
```

4.5 `rsa_write_priv(n, d, pvfile)`

Similarly to `rsa_write_pub()`, this function will print the prime n and private d to pvfile. Use gmp_fprintf() to print these variables

Pseudocode:

```
rsa_write_priv(n, d, pvfile)
    print n to pvfile
    print d to pvfile
```

4.6 `rsa_read_priv(n, d, pvfile)`

Similarly to `rsa_read_pub()`, this function reads the value in pvfile and assign it to n and d

Pseudocode:

```
rsa_read_priv(n, d, pvfile)
    read n to pvfile
    read d to pvfile
```

4.7 `rsa_encrypt(c, m, e, n)`

This function encrypts the message in m and puts it into c. To encrypt the message, apply this equation:

$$E(m) = c = m^e (mod\ n)$$

Another detail to node is that m is a message, therefore it can contain characters other than digits, so it's critical to convert m to base 10 (decimal) before performing the equation

Pseudocode:

```
rsa_encryption(c, m, e, n)
    convert = m
    convert **convert** to base 10
    pow_mod(c, convert, e, n)
```

4.8 `rsa_encrypt_file(infile, outfile, n, e)`

This function encrypts a message from a infile and prints out the encrypted messages to the outfile. Note that we're are not reading the entire file, we need to seperate this file in blocks to not make the overall value of the messages in the file exceed n, each block will have a size of $\frac{log_2(n)-1}{8}$. Each block is a dynamically allocated array, this is achieve through calloc().
Then to read the file, we will use `fread()` to read in the bytes in the file until we either reach the EOF, in that case we will transfer the bytes into a `mpz_t` variable thanks to `mpz_import()`. Encrypt the `mpz_t` variable using `rsa_encrypt()` and print it to outfile

Pseudocode:

```
rsa_encrypt_file(infile, outfile, n, e)
    n2 = n
    k = 0
    while n2 > 0:
        k++
        n2 /= 2
    k = ((k - 1) / 8)
    while true:
        block = dynamically allocate the array size k
        actual bytes read = read in k - 1 bytes from the array
        move all the elements forward by 1 index and assign block[0] = 255
        import the bytes from the block to converti
        rsa_encrypt(converto, converti, e, n)
        print converto to outfile in hexstring
        if(actual bytes read != k - 1) break
```

4.9 rsa_decrypt(m, c, d, n)

This function decrypts the message c and puts in into m. Note that the message encrypted when go through this function should return the original. To decrypt the encrypted message, apply this equation:

$$D(c) = m = c^d (mod\ n)$$

Pseudocode:

```
rsa_decrypt(m, c, d, n)
    convert = c
    convert **convert** to base 10
    pow_mod(m, convert, d, n)
```

4.10 rsa_decrypt_file(infile, outfile, n, d)

This function decrypts the encryped data in infile and output the message to outfile. Since the encrypted data from `rsa_encrypt_file()` are split into different blocks with a trailing new line, we can read each line of the encrypted data, decrypt it using `rsa_decrypt()` and export it to the a dynamically allocate array size k = $\frac{log_2(n)-1}{8}$ using `mpz_import()`, after that it's a matter of reading each byte read and print them to outfile

Pseudocode:

```
rsa_decrypt_file(infile, outfile, n, d)
    n2 = n
    k = 0
    while n2 > 0:
        k++
        n2 /= 2
    k = ((k - 1) / 8)
    read line of file and put in converti
    while converti != EOF:
        rsa_decrypt(converto, converti, d, n)
        block = dynamically allocate array with size k
        export converto into the block array and count the bytes
        for i from 1 to bytes counted:
        print block[i] to outfile
```

4.11 `rsa_sign(s, m, d, n)`

This function will create a RSA signing, calculating the RSA signing by using this equation:

$$S(m) = s = m^d (mod\ n)$$

Note that this m should be in base 10 **Pseudocode:**

```
rsa_sign(s, m, d, n)
    convert = m
    convert convert into base 10
    pow_mod(s, convert, d, n)
```

4.12 `rsa_verify(m, s, e, n)`

This function will verify the signature by calculating **t** from the following equation:

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

This function will output true if t is equal to m (note that both t and m are in base 10) and false otherwise.

Pseudocode:

```
rsa_verify(m, s, e, n)
    convert = s
    convert2 = m
    convert convert and convert2 to base 10    pow_mod(t, convert, e, n)
    return t == convert2
```