Assignment 5 Design Document

Nicholas Chu

February 5, 2022

1 Purpose

The purpose of this assignment is to create a RSA key pair generator, an encryptor, and decryptor. The files that will be needed to do this include:

- randstate.c: Generates a random state
- numtheory.c: Math functions needed for the RSA algorithm
- rsa.c: Has many of the functions for key generation, encryption, and decryption
- keygen.c: Generates keys
- encrypt.c: Encrypts an input file and prints the message to an output file
- decrypt.c: Decrypts an input file and prints the message to an output file

2 randstate.c

The RSA key generator and encrpytion algorithm will need to pseudo randomly generate number that will be used for generating the primes for the private key (p and q) and the public and the modulo inverse for encryption (e). The file will need to have functions that can create and clear a randomly generated state that will be used in the other files to generate random numbers when needed.

2.1 void randstate_init(uint64_t seed)

Takes a unsigned 64 bit integer which will be used as the seed when using srandom(). It will initialize a stuct called state using gmp_randinit_mt() and seed it using gmp_randseed_ui() in order to set an inital seed value into the initialized state.

2.2 void randstate_clear(void)

Should free any memory allocated by state using gmp_randclear().

3 numtheory.c

The RSA algorithm will require some math functions that will need to be implemented and include modular exponentiation, a primality test, and modular inverse.

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

This will be used for encryption and decryption in order to quickly find the power mod. The function will have four parameters: out, base, exponent, and modulus. Note that even though the function is defined as a void because you can not define a function with type a multiple precision integer(mpz_t). To actually get a value from this function you will need to include a parameter to save the result to. The function will try to calculate: base^{exponent} mod(modulus) and save that to out. It does this by getting the product of $\Pi_{i=0}^{n-1} b^{c_i 2^i}$mod(m) where b is base and m is modulus. Remember that when working with mpz types to use the appropriate built in functions to manipulate them.

```
out = 1
b = base
while (exponent > 0)
    if modulus is odd
        out = out * b % modulus
    b = b * b % modulus
    exponent = exponent floor divided by 2
```

3.2 bool is_prime(mpz_t n, uint64_t iters)

This function uses the Miller-Rabin primality test to see check if a number is prime. It will have two parameters, n which will be the number being checked and iters is the number of iterations of the test will be performed. It will need to use the power mod function defined earlier when checking if for primality over a series of tests where the larger the iterations of the test the higher its accuracy will be. First the function will need to use n-1 = 2^sr to find s and r which will be values used for power mod when checking for primality. Then find a random number 'a' from the range 2 to n-2 by using the random state set up previously and taking that result modding it by n-3 and adding 2 to get it to fit into the range. Using a, it will calculate the value of a^r mod(n) and if it is either 1 or -1 that means the number is likely prime. When it does not, take the calculated power mod and use it to find a new power mod by using: (previously calculated power mod)^{2}mod(n). From this point on wards, if it equals 1 it means that it is not prime and if -1 then it is probably prime and if neither then keep finding (previously calculated power mod)^{2}mod(n) until it equals either 1 to -1.

```
r = n - 1
s = 0
while r is even
    r = r/2
    s += 1

for each test an iters amount of times
```

```
    a = random number % (n-3) + 2
    y = pow_mod(a,r,n)
    if y != 1 or -1
        j = 1
        while j <= s-1 and y != n-1
            y = pow_mod(y,2,n)
            if y == 1
                return false
            j += 1
        if y != n-1
            return false
return true
```

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

This function will later be used for creating the primes for the RSA keys and will have three parameters: p, bits, and iters . It will do this by generating random number using the created state of a bit size that is at least bits long and checking it primality using is_prime with the generated number and iter for iterations. If it is prime then save it to p or just generate another one.

```
p = random number from state of size bits
while is_prime(p,iters) is false
    p = another random number of size bits
```

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

Used to find the greatest common denominator. This would be using in such instances of encrypting and decryption. The parameters are d, a, and b. d would be the value the denominator would be saved to, and a and b are the values that will have their greatest common divisor calculated.

```
while b != 0
    d = b
    b = a % b
    a = d
d = a
```

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

This computes the modular inverse of a i = 1 * mod(n), a and n are given as parameters, and store them in i which is the modular inverse. This uses the extended Euclidean algorithm. The function will need to use temp variables in order replicate parallel assignment.

```
r = n
r' = a
i = 0
i' = 1
while r' != 0
```

```
    quotient = r floor divided by r'
    tmp = r
    r = r'
    r' = tmp - quotient * r'
    tmp = i
    i = i'
    i' = tmp - quotient  * i'
if r > 1
    i = 0
if t < 0
    t += n
```

4 rsa.c

This file is a library of functions that contain in implementation for the RSA algorithm which include key generation, encryption, and decryption.

4.1 rsa_make_pub

This function is used to produce a public key. It has the parameters: p, q, n, e, nbits, and iters. e is the public exponent and p and q are the primes. n bits are for determining the sized of p, q, and e and iters will be used for the number of Miller-Rabin iterations when making the primes. n is the result of p*q. It will first create two primes using make_prime for p and q. The number of bits for p will be in the range [nbits/4,(3×nbits)/4) and q will have the remainder of bits from that range. iters should be random. Then it will need to compute λ(n) = lcm($p-1$,$q-1$). This could be calculated by finding $\frac{|(p-1)(q-1)|}{gcd(p-1,q-1)}$. Then in a loop generate random number of around nbit using mpz_urandomb until a number is generated is co-prime to λ(n).

```
p_nbits = random from range nbits/4 to 3 * nbits / 4
q_nbits = 3 * nbits / 4 - p_nbits
make_prime(p, nbits, rand iters)
make_prime(q, nbits, rand iters)
lambda = ((p-1)*(q-1))/gcd(p-1, q-1)
while gcd(gen_num, lambda) != 1
    mpz_urandomb(gen_num, state, nbits)
e = gen_num
n = p * q
```

4.2 rsa_write_pub

This function writes the information generated for the public RSA key to an output file. It will open the file pbfile which is specified by the parameters and print the parameters n which is (p * q), e which is the exponent, signature s, and a username in this specific order, each on a new line. It should print n, e, and s as hexstrings which is done with gmp_printf().

4.3 rsa_read_pub

The function will read a pbfile specified by the parameters and use gmp_scanf to read each line and saving the lines to n, e, s, and username.

4.4 rsa_make_priv

All this function will do is create a private key, d, by using specified e, p, and q. To do this it will need to find the modular inverse using the equation de = 1mod(λ(n)). This will involve calling mod_inverse(d, e, lambda n).

4.5 rsa_write_priv

Very similar to rsa_read_pub but instead but instead of printing e, s, and username it will only be printing n and d, which is the private key, as hexstrings into a pvfile specified by the parameters.

4.6 void rsa_read_priv(mpz_t c, mpz_t m, mpz_t e, mpz_t n)

Like rsa_read_pub it will use gmp_scanf to read each line from a file specified by the parameter pvfile and saving the values from appropriate lines to the parameters n and d.

4.7 rsa_encrypt

This will encrypt the file by using the equation E(m) = c = m^e(mod n). This will be done by taking the n parameter and using the power mod function like pow_mod(c,m,e,n).

4.8 rsa_encrypt_file

The function will read the contents of infile by blocks and print the encrypted contents that to an outfile. This is because the value of the block that is being encrypted needs to be below n. The value of the block also cannot be 0 or 1. The function first will begin by finding block size by using $\lfloor(log_2(n) - 1)/8\rfloor$ and use that size to allocate memory for for an array that will hold the message. Set the zeroth byte to 0xFF which prepends the workaround byte. The for every unprocessed byte in the file, it will read (block size - 1) bytes of the infile and save that to an array to hold the bytes. It will take the read byte and save it as an mpz_t using mpz_import and envrypt it. It then will print the encrypt it and print to an outfile. The pseudo-code for encrypting:

```
block size = log_2(n)-1 floor divided by 8
*block = malloc((uint8_t *) * block size)
block[0] = 0xFF
while there are still unprocessed bytes
    read and save bytes into block
    process block using mpz_import to an m
    rsa_encrypt(c,m,d,n)
    print encrypted c to outfile as a hexstring
```

4.9 rsa_decrypt

Similar to the encrypt algorithm except this time is finding m = c^d mod n and would use pow_mod(m,c,d,n)

4.10 rsa_decrypt_file

The function will take an encrypted infile and decrypt its contents, printing it to an outfile. Much of the similar principles from rsa_encrypt apply to the this as well. The main differences are that it will read an infile and saving its contents to a block, and instead of using mpz_import and rsa_encrypt, the function will be using mpz_export and rsa_decrypt. Also when writing out the contents of the decrypted block do not print 0xFF to the outfile.

```
block size = log_2(n)-1 floor divided by 8
*block = malloc((uint8_t *) * block size)
while there are still unprocessed bytes
    scan hexstring and save contents to mpz_t
    rsa_decrypt(m,c,d,n)
    process message using mpz_export to an array
    write decrypted block to outfile as a hexstring
```

4.11 rsa_sign

Produces an RSA signing s which signs message m using private key d and modulus n. This is done through s = m^d(mod n) or pow_mod(s,m,d,n).

4.12 rsa_verify

Checks if signature s is verified. It will return true when verified and false when not verified. Verification is checked using the equation t = s^e mod n and if t is equal to m, m being the expected message.

```
pow_mod(t,s,e,n)
if m == t
    return true
else
    return false
```

5 keygen.c

This will be used to generate the public and private keys based of specifications from command line arguments then print these keys out to 2 separate specified files. The command line arguments that it will take are:

- -b: specify minimum bit size for modulus n
- -i: number of iterations primality testing for making primes, default of 50 iterations
- -n pbfile: file the public key will be printed to, default file is set to rsa.pub

- -d pvfile: file the private key will be printed to, default file is set to rsa.priv
- -s: set the random seed for the random state, if not specified then it will use time(NULL)
- -v: enables output
- -h: help and usage

First comes the set-up where it will need to open the files specified by -n and -d or the default ones when not. If opening the files causes and issue print the usage. Then set up the file permissions are set up so that only the user has permission to access the private key file by setting its permissions to 0600 using fchmod() and fileno(). It will also need to set up the state using randstate_init and use the seed specified by -s, if seed was not specified use the default.

Once set up is complete then you can use the functions rsa_make_pub() and rsa_make_priv(). Get the username by using getenv() and convert is using mpz_set_str() and specify it to base 62. Use this username with rsa_sign() to get the signature of the username. Write the generated keys to the file specified by -n and -d using rsa_write_pub and rsa_write_priv. If -v was specified then with will print:

```
username
signature s
prime p
prime q
public modulus n
public exponent e
private key d
```

It will print out their values in decimal and their size in bits. Once done the file should close the opened files and run randomstate_clear() in order to clean up any allocated memory and avoid memory leaks.

6 encrypt.c

The file will take an un-encrypted input file, encrypts is contents, then prints it to an output file. The command line options for the file will be:

- -i: input file for encryption, default is stdin
- -o: output file for encryption, default is stdout
- -n: file containing public key, default is rsa.pub
- -v: verbose mode
- -h: prints usage

Open the files specified by -i -o and -n or use the defaults if the commands are not specified. If it does not open anything then print an error message with the usage. Then read the files using rsa_read_pub. If -v was specified print:

```
username
signature s
public modulus n
public exponent e
```

It should print out their bit size and and their decimal values. Convert the username to an mpz_t and use it with rsa_verify to verify the signature with the public key in the file specified by -n and returning an error when it is false. If it can be verified then use rsa_encrypt_file() and encrypt the contents of the file specified by -i. Write it to the file specified in -o and once done close all the opened files to prevent memory leaks.

7 decrypt.c

This program will take an encrypted file and print the decrypted contents to an output file. The command line options for this file include:

- -i: input file for encryption, default is stdin
- -o: output file for encryption, default is stdout
- -n: file containing public key, default is rsa.priv
- -v: verbose mode
- -h: prints usage

It will first open all the files given by the command line options or the default ones when not specified. If there is an issue with opening the files it will give an error. It will then need to read the private key file specified by -n with rsa_read_priv() and if -v was specified then print the public modulus n and private key e. These values should be printed in decimal form with their bit sizes. Use rsa_decrypt_file() to decrypt the contents of the file specified by -i and write the contents to the file specified by -o. Once finished close the files opened by -i, -o, and -n.