CSE13s Fall 2022
Assignment 5
Professor Miller
George Sono

Final Design Document

Number Theory Functions (numtheory.c, numtheory.h)
General Idea: In numtheory.c, we will implement the following functions: gcd (greatest common divisor), mod_inverse (compute the inverse of a modulo n), pow_mod (modular exponentiation), is_prime (checks if a number is prime) and make_prime (creates a prime number given certain parameters). These functions will be to help aid us when doing RSA calculations for encrypting, decrypting and creating the keys.

General Idea for GCD:
We will be implementing Euler's algorithm in order to find the GCD of two numbers. Euler's algorithm relies on modular arithmetic to find the GCD. We successively take the modulo of either of the larger numbers (a or b), until the two numbers become equal.

Pseudocode for GCD (gcd)

1. Create a void call by reference-like function called gcd, that also takes in three parameters all of the mpz type: destination variable, a variable, b variable
2. Create variables og_a, og_b that will keep track of a and b before we update values of a and b
3. While loop that runs while b != 0
    a. temp = b
    b. b = a mod b
    c. a = temp
4. Set destination variable with the value of variable a
5. Set a = og_a and b = og_b
6. Free any extra mpz_variables used
7. Return nothing (exit the function)

General Idea for Modular Inverse:
The modular inverse will be used for calculating if two integers are coprime. It is also used in the RSA library implementation for function rsa_make_priv().

Pseudocode for Modular Inverse (mod_inverse)

1. Create a void call by reference-like function called mod_inverse, that also takes in three parameters all of the mpz type: destination variable, a variable, n variable
2. Create variables r, r', t, t'
3. r = n, r' = a, t = 0, t' = 1
4. while loop that runs while r' != 0
    a. q = floor(r/r')
    b. prev_r = r
    c. r = r'
    d. r' = r - (q x r')m
    e. prev_t = t
    f. t = t'
    g. t' = t - (q x t')
5. If r > 1

    a. Set destination variable to 0
    b. Return nothing (exit the function)
6. If t < 0
    a. t = t + n
7. Set destination variable as t
8. Free any extra mpz_variables used
9. Return nothing (exit the function)

General Idea for Modular Exponentiation:
To calculate modular exponentiation faster, we will use repeated squaring and modular reduction.

Pseudocode for Modular Exponentiation (pow_mod):
1. Create a void call by reference-like function called pow_mod, that also takes in four parameters all of the mpz type: destination variable, a variable, d variable, n variable
2. og_a = a
3. og_b = b
4. If n = 0
    a. Assert an error as we cannot divide by zero
5. Initialize variables v, p
6. v = 1, p = a
7. while loop that runs while d > 0
    a. Initialize and declare is_odd variable
    b. is_odd = d modulo 2
    c. If is_odd is greater than 0
        i. v = (v x p) modulo n
    d. p = (p x p) modulo n
    e. d = floor(d/2)
8. Set destination variable to v
9. Set a and b to their original values with og_a and og_b (as we changed them in the while loop)
10. Free any extra mpz_variables used
11. Return nothing (exit function)

General Idea for Primality Testing:
To make sure that a number is prime, we will use the Miller-Rabin algorithm. However, there are different types of primary testing that give false positives, in this case Miller-Rabin tests pseudoprimes.

Pseudocode for Is Prime (is_prime)
1. Create a boolean call by reference-like function called is_prime, that also takes in two parameters of the mpz type and integer types: n variable (mpz), iters variable (unsigned integer)
2. Create variables s = 0, r =0
3. Calculate the values of s, r with the following
   Calculating s and r pseudocode:
    1. If (n-1) mod 2 = 0 (if n-1 is even)
        a. Return false (in is_prime)
    2. Else:

    a. r = n-1
    b. m = r
    c. while (r mod 2 == 0) (r is even) //help from TA Zack Jorquera
        i. if (m mod 2 == 1) (if m is odd):
            1. r = m
            2. break
        ii. else:
            1. s += 1
            2. m = m/2 (theoretically m can never be less than 1 because 1 is odd and if its odd we break)
4. For i in range iters (with i starting at 1)
    a. a = random variable in the range [2,n-2] inclusive (make sure to include randstate.h and initialize RNG with seed)
    b. y = pow_mod(a, r, n)
    c. If y != 1 and y != (n-1)
        i. j = 1
        ii. while (j <= (s-1)) and (y != (n-1))
            1. y = pow_mod(y, 2, n)
            2. if y == 1
                a. return false
            3. j = j +1
        iii. if y != (n-1)
            1. return false
5. Free any extra mpz_variables used
6. return true

General Idea for Make Prime:
In conjunction with is_prime, we will generate a prime number of at least bits length by generating a random number and checking if it is prime, and if it's of bits length.
Pseudocode for Make Prime (make_prime):
1. Create a void call by reference-like function called is_prime, that also takes in three parameters of the mpz type and integer types: p variable (mpz), bits (unsigned integer), iters variable (unsigned integer)
2. not_prime = True
3. While loop that keeps running until not_prime is false
    a. rand_num a random number (using mpz_urandomb)
    b. If ((is_prime(rand_num, iters) == True) and of bits length))
        i. not_prime = False
4. Set p = rand_num
5. Free any extra mpz_variables used
6. Return nothing (exit the function)

CSE13s Fall 2022
Assignment 5
Professor Miller
George Sono

RSA Library Implementation (rsa.c, rsa.h)
General Idea: Create RSA Library implementations that has functions 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. We will also create a helper function to find the value of λ (n) (Carmichael of p x q).

General Idea for Carmichael function λ (n):
We will use the Carmichael Function in our RSA functions. This will behave as a helper function and will be used in the following pseudocode for the RSA functions mentioned above.

We will calculate the carmichael function using this identity:

$$\lambda(n) = \mathrm{lcm}(\lambda(p), \lambda(q))$$

For calculating the least common multiple (LCM), we will use this formula:

$$\frac{|\lambda(p) \times \lambda(q)|}{\gcd(\lambda(p), \lambda(q))}$$

λ (n) Pseudocode:

1. Create a void function call-by-reference like function named lambda, that takes in three arguments of the mpz type: a destination variable, p, q
2. Inside, the function, declare variables: totp, totq (tot is short for totient)
3. totp = p - 1, totq = q - 1
4. totp_totq = absolute value of (totp x totq)
5. gcd_totp_totq = GCD(totp, totq)
6. quotient = totp_totq / gcd_totp_totq
7. Free any extra mpz_variables used
8. Set destination variable to the value of quotient

General Idea for rsa_make_pub:
Rsa_make_pub will create two large primes p and q, their product as variable n, and the public exponent. To calculate the public exponent, we will use a while loop that will iterate for nbit iterations, computing a random number each time. In each iteration, we will compute the GCD of the random number and the value of n after being put through the Carmichael function (denoted as λ (n)) is equal to 1 (meaning that they are coprime). The Carmichael function is the equivalent lcm(p-1, q-1). We can calculate the least common divisor with our GCD function in number theory.

rsa_make_pub Pseudocode

1. Create a void call-by-reference like function called rsa_make_pub that takes in 5 arguments: p (mpz), q (mpz), e (mpz), nbits (unsigned integer), iters (unsigned integer)
2. Declare variables: pbits and qbits
3. Set pbits = random number in range of [nbits/4, (3 x nbits)/4) (using random())
4. Set qbits = nbits - pbits (the remaining bits)
5. Set p = a prime number using make_prime() with pbits

6. Set q = a prime number using make_prime() with qbits
7. Counter = 0
8. while (counter < nbits) //we will iterate for "around nbits"
    a. Rand_num = randomly generated number (using seed)
    b. If (GCD(rand_num, λ (n)) == 1))   we use the lambda function mentioned above
        i. Free any extra mpz_variables used
        ii. Set e variable = rand_num (as its coprime with λ (n) since the GCD between both is one)
        iii. return void (exit the function)
    c. Else
        i. Continue to next iteration
    d. Counter += 1

General Idea for rsa_write_pub:

rsa_write_pub will write the information into pbfile (provided file pointer) from the results of running rsa_make_pub (n and the public exponent (e)), as well as the private key encrypted username (s) and the username, all in a text file. Regarding the signature, this will be done through calling rsa_sign function (will be defined later) All information will be written as hex strings.

rsa_write_pub Pseudocode:

1. Create a void call-by-reference like function named rsa_write_pub, that takes in 5 parameters: n (product of p x q), e (public exponent), s (signature), username (so when file is read, it can be cross checked with decrypted signature), pbfile (file which we will write the public key data to)
2. Write n (hex) into pbfile followed by a new line (using gmp_fprintf with pbfile)
3. Write e (hex) into pbfile followed by a new line (using gmp_fprintf with pbfile)
4. Write s (hex) into pbfile followed by a new line (using gmp_fprintf with pbfile)
5. Write username into pbfile followed by a new line (using gmp_fprintf with pbfile)

General Idea for rsa_read_pub:

rsa_read_pub is a void call by reference like function which will update the values of n (p*q), e (public exponent), s (signature encrypted with private key), signature and username.

rsa_read_pub Pseudocode:

1. Create a void call-by-reference like function named rsa_write_pub, that takes in 5 parameters: n (product of p x q), e (public exponent), s (signature), username (so when file is read, it can be cross checked with decrypted signature), pbfile (file which we will read the public key data from)
2. set n = first line of pbfile (use gmp_fscanf)
3. set e = second line of pbfile (use gmp_fscanf)
4. set s = third line of pbfile (use gmp_fscanf)
5. set username = fourthline of pbfile (use gmp_fscanf)
6. Return nothing (void)

CSE13s Fall 2022
Assignment 5
Professor Miller
George Sono

General Idea for rsa_make_priv:
rsa_make_priv is a void call by reference-like function which will create the values (through updating the provided values) for d (key). Given the values of the public exponent (e) and the two large primes (p,q), we will calculate d, by computing the inverse of e modulo (λ (n)), where n is p x q. Like in public key generation, λ (n) is equivalent to lcm(p-1, q-1).

Pseudocode for rsa_make_priv:

1. Create a void call-by-reference like function named rsa_make_priv, that takes in 4 parameters: d (private key destination variable), e (public exponent), p (prime number), q (prime number)
2. Set d = inverse of (e mod lambda(n) where n = p x q), using mod_inverse

General Idea for rsa_write_priv:
Rsa_write_priv will write the private key into pvfile (provided file pointer). The format written into the pvfile will be n (p x d) and d (the key itself). They will be written as hex strings.

Pseudocode for rsa_write_priv:

1. Create a void call-by-reference like function named rsa_make_priv, that takes in 3 parameters: n (product of p * q), d (private key), pvfile (private key file we will write to)
2. Write n (hex) into pvfile followed by a new line (using gmp_fprintf with pvfile)
3. Write d (hex) into pvfile followed by a new line (using gmp_fprintf with pvfile)

General Idea for rsa_read_priv:
Rsa_read_priv will read the private RSA key from pvfile (provided file pointer). We will update the provided values to the values of n and d from pvfile.

Pseudocode for rsa_read_priv:

1. set n = first line of pvfile (use gmp_fscanf)
2. set e = second line of pvfile (use gmp_fscanf)

General Idea for rsa_encrypt:
Rsa_encrypt will allow us to encrypt one block of text. We will use the formula of E(m) = c = m^e (mod n), where E is the encryption function, c the cipher, e the public exponent, m the message (plain text) and n the product of primes p and q. It will be used as a helper function for encrypt_file and rsa_sign.

Pseudocode for rsa_encrypt:

1. Create a void call-by-reference like function which will take in 4 mpz parameters: c (output cipher text), m (message text), n (product of primes p and q), e (public exponent).
2. Set c (cipher text) = pow_mod(m,e,n)
3. Return nothing (exit the function)

CSE13s Fall 2022
Assignment 5
Professor Miller
George Sono

General Idea for rsa_encrypt_file:
Rsa_encrypt_file will take a file (which contains a message), and create a ciphered version of that in another file. We will encrypt in blocks, meaning that we will not encrypt the entire file at one time. The block size is determined by how many bytes n (product of p x q) is.

rsa_encrypt_file Pseudocode:

1. Create a function rsa_encrypt that takes in four arguments: infile (plaintext), outfile (cipher text), n (product of p*q), e (public exponent)
2. k = ceil(log_2_(nbits,)-1)/8)) calculating block size as k (use mpz_sizeinbase(n,2)
3. Kblock = dynamically allocated array of size k (use calloc)
4. Byte_count = # of bytes that infile is
5. Set 0th index of k to be 0xFF
6. while true (we will break out manually)
   a. read at most k-1 bytes from infile (storing the values into kblock, and setting the amount of bytes read to variable j)
   b. turn kblock array into mpz type with mpz_import (use j for size parameter, 1 for most significant word parameter, 1 for endian parameter, 0 for nails parameter), store as mpz variable: tyjess
   c. message = encrypt(tyjess)
   d. Write message (in hex) to outfile followed by a new line (use gmp_fprintf)
   e. If j is less than k-1 (reached end of file)
      i. break
7. Free any extra mpz_variables used

General Idea for rsa_decrypt:
Rsa_decrypt will allow us to decrypt one block of text. We will use the formula of D(c) = m = c^d (mod n), where D is the decryption function, c the cipher, d the private key, m the message (plain text) and n the product of primes p and q.

Pseudocode for rsa_decrypt:

1. Create a void call-by-reference like function which will take in 4 mpz parameters: c (input cipher text), m (output message), n (product of primes p and q), d (private key).
2. Set m (message) = pow_mod(c,d,n)
3. Return nothing (exit the function)

General Idea for rsa_decrypt_file:
Rsa_decrypt_file will take a file (which contains a ciphered message), and create a decrypted version of that in another file. We will decrypt in blocks, meaning that we will not decrypt the entire file at one time. The block size is determined by how many bytes n (product of p x q) is.

rsa_decrypt_file Pseudocode:

1. Create a function rsa_encrypt that takes in four arguments: infile (ciphered text), outfile (message text), n (product of p*q), d (private key)
2. k = ceil(log_2_(nbits,)-1)/8)) calculating block size as k (use mpz_sizeinbase(n,2)
3. Kblock = dynamically allocated array of size k (use calloc)

4. Set 0th index of k to be 0xFF
5. while (true) //there are still unprocessed bytes
    a. c = scanned in hex string from infile
    b. m = decrypt(c)
    c. Export m into bytes, by using mpz_export to export it into kblock
    d. j = amounts of bytes read (defined in mpz_export)
    e. Write kblock into out file with fwrite()
    f. If j < k-1 (reached end of file)
        i. break
6. Free any extra mpz_variables used

General Idea for rsa_sign:
rsa_sign will encrypt the provided username with the private key. This will allow for people to verify if it's the actual user, by decrypting the signature with the public key.

Pseudocode for rsa_sign:
1. Create a void call-by-reference-like function that is called rsa_sign, that takes in 4 parameters: s (signature destination variable), m (message which is being signed), d (private key), n (p x q)
2. Set s = pow_mod(m,d,n) s is calculated using this formula $S(m) = s = m^d \pmod{n}.$

General Idea for rsa_verify:
Rsa_verify will be a boolean function that will return true or false depending on if the decrypted signature matches with the provided username. If false, then people will know that someone may be impersonating someone else.

Pseudocode for rsa_verify:
1. Create a boolean function named rsa_verify, that will take in 4 parameters: m (message), s (signature), e (public exponent), n (p x q)
2. Declare decrypted_signature
3. Decrypted_signature = pow_mod(s,e,n)
4. If decrypted_signature == message:
    a. Return true
5. Else:
    a. Return false

Randstate Implementation (randstate.c, randstate.h)
General Idea: Our randstate.c file will have two functions, randstate_init and randstate_clear. randstate_init will initialize our random number generator (mtwister). Randstate_clear will clear the rand_state from heap memory when we no longer need it.

Pseudocode for randstate_init:
1. Create a void call-by-reference like function named randstate_init that takes in one argument: seed (unsigned integer)
2. Create a variable of the mpz type called mpz_seed
3. Set mpz_seed = seed
4. Initialize the RNG with the global variable, *state* (declared in randstate.h) (use gmp_randinit_mt)
5. Set the seed of the RNG with mpz_seed (use gmp_randseed)

CSE13s Fall 2022
Assignment 5
Professor Miller
George Sono

Pseudocode for randstate_clear:

1. Create a void call-by-reference like function named randstate_init that takes in no arguments
2. Clear state (using gmp_randclear)

Key Generator (keygen.c, keygen.c) Implementation

General Idea: We will create a key generator which creates public and private keys using functions from rsa.c.

Pseudocode for keygen:

1. Create a main function that has parameters which allows us to use command line arguments
2. Seed = time(NULL) (default value and the seconds since the UNIX epoch)
3. Pbfile = rsa.pub (default)
4. pvfile = rsa.priv (default)
5. verbose = 0 (assumed default)
6. Declare bit and iters variables for public modulus n and Miller-Rabin iterations
7. Create a while loop to parse through command line arguments with getopt
    a. Switch case statement
        i. case ‘b’
            1. bit = option argument converted from string to number
            2. Check if bit is in range 50-4096
                a. If not, exit program with non-zero code
            3. break
        ii. case ‘i’
            1. iters = option argument converted from string to number
            2. Check if bit is in range 1-500
                a. If not, and exit program with non-zero code
            3. Break
        iii. case ‘n’
            1. pbfile = option argument
            2. Break
        iv. case ‘d’
            1. pvfile = option argument
            2. Break
        v. case ‘s’
            1. s = option argument converted from string to number
            2. Break
        vi. case ‘v’
            1. Verbose = 1 (verbose is on)
            2. break;
        vii. case ‘h’
            1. Display program synopsis and usage
            2. Return 0.
        viii. Default
            1. Print help message to stderr
            2. Return 1
8. Open public and private key files (if unsuccessful, print help message and exit program)

9. Set private key files permissions to 0600 so the read and write permissions are only for the user
10. Initialize the random state using randstate_init() with seed
11. Make public key with rsa_make_pub()
12. Make private key with rsa_make_priv()
13. Get the username using getenv("USER")
14. Convert the username into an mpz_t (with base 62) and use rsa_sign to compute signature
15. Use rsa_write_pub to write public key info to pbfile (writing n, public exponent, signature, username)
16. Use rsa_write_priv to write private key info to pvfile (writing n, private key d)
17. If verbose is on (verbose == 1), print the information from the program (variables needed for rsa_make_pub, etc.)
    a. (a) username
       (b) the signature s
       (c) the first large prime p
       (d) the second large prime q
       (e) the public modulus n
       (f) the public exponent e
       (g) the private key d
18. Finally, close the private and public key files, clear any mpz variables used and free randstate with randstate_clear()

Encryptor (encrypt.c) Implementation

General Idea:

Using the RSA library functions, we will create a program that will allow us to take in an input file of plain text and output a text file of ciphered text.

Pseudocode for encrypt.c:

1. Create a main function that has parameters which allows us to use command line arguments
2. inputfile = default to standard input
3. outputfile = default to standard output
4. pbfile = rsa.pub
5. verbose = 0 (assumed default)
6. Create a while loop to parse through command line arguments with getopt
    a. Switch case statement
        i. case 'i'
            1. inputfile = option argument
            2. break
        ii. case 'o'
            1. outputfile = option argument
            2. Break
        iii. case 'n'
            1. pbfile = option argument
            2. Break
        iv. case 'v'
            1. Verbose = 1 (verbose is on)

            2. Break
        v. case ‘h’
            1. Display program synopsis and usage
            2. Return 0.
7. Open public key file (if unsuccessful, print help message and exit program)
8. Create variables n, e, s, username
9. Use rsa_read_pub with pbfile
10. If verbose is on
    a. Print in this order with each element being followed by a trailing newline: the username, signature s, public modulus n, public exponent e
    b. All mpz_ t values should be printed with number of bits that constitute them, also their decimal values as well
11. Convert username into mpz_t and use rsa_verify with public key to make sure they line up
12. If the decrypted signature and username do not line up
    a. Report error and exit program
13. Else:
    a. Encrypt file using rsa_encrypt_file() setting output file to outputfile
14. Close pbfile, inputfile, outputfile and make sure to clear any mpz_t variables used

Decrypt (decrypt.c) Implementation:

General Idea:
Using the RSA library functions, we will create a program that will allow us to take in an input file of ciphered text and output a text file of deciphered text.

1. Create a main function that has parameters which allows us to use command line arguments
2. inputfile = default to standard input
3. outputfile = default to standard output
4. pvfile = rsa.priv
5. verbose = 0 (assumed default)
6. Declare bit and iters variables for public modulus n and Miller-Rabin iterations
7. Create a while loop to parse through command line arguments with getopt
    a. Switch case statement
        i. case ‘i’
            1. inputfile = option argument
            2. break
        ii. case ‘o’
            1. outputfile = option argument
            2. Break
        iii. case ‘n’
            1. pvfile = option argument
            2. Break
        iv. case ‘v’
            1. Verbose = 1 (verbose is on)
            2. Break

            v. case ‘h’
                1. Display program synopsis and usage
                2. Return 0.
8. Open private key file (if unsuccessful, print help message and exit program)
9. Declare variables n (public modulus), e (private key)
10. Use rsa_read_priv with pvfile
11. If verbose is on
    a. Print in this order with each element being followed by a trailing newline: public modulus n, private key e
    b. All mpz_ t values should be printed with number of bits that constitute them, also their decimal values as well eg. username (*x* bits): *xxxxxxxxxx*…
12. Convert username into mpz_t and use rsa_verify with public key to make sure they line up
13. Decrypt file using rsa_decrypt_file(), setting output file to outputfile
14. Close pbfile, inputfile, outputfile and make sure to clear any mpz_t variables used

CSE13s Fall 2022
Assignment 5
Professor Miller
George Sono

References

1. -Nishant Khannorkar (TA): Nishant Khanorkar (TA) - asked questions and received help regarding make_pub functions and how signatures work
2. Zackary Jorquera (TA): asked questions received advice regarding Miller Rabin algorithn (calculating S and R) and also using fread and fwrite
3. Sanjana Patil (TA): asked questions regarding the gmp_urandomb functions and how they are used in make_prime and also about how nbits worked and how random() is used in make_pub
4. Jessie Srinivas (Tutor): explained how the encrypt and block arrays worked and how import mpz functions can take in an array
5. Lev Teytelman (Tutor): asked questions and checked approach regarding variables inputted for mpz_export and import statements in encrypt_file and decrypt_file
6. Fabrice Kurmann (Tutor): asked questions and received advice regarding the block size in decrypt file and encrypt file
7. Ben Grant (Tutor): Followed his approach on how to generate a number within a given range (from his message on discord) with generating pbits in make_pub

Setting file pointers to stdin/stdout

- Link: https://stackoverflow.com/questions/18505530/how-to-set-a-file-variable-to-stdout
- -How I used it:
    - From the stack overflow article, I learned how to set file pointers to stdin and stdout. I used these methods for encrypt and decrypt where the default arguments for input and output are standard input and standard output.
    -

Freeing dynamically allocating strings in C

- Link: //https://stackoverflow.com/questions/10063222/freeing-strings-in-c
- How I used it:
    - When I was trying to free my dynamically allocated array of characters (for storing the username), I kept on running into an invalid pointer error. Searching the error message on Google, I found that it was because I was changing the pointer address. Meaning the method that I used to store the username actually changed the pointer's address. From the article, I learned that I should use strcpy() to keep the pointer address the same.