Karthi Sankar

kasankar@ucsc.edu

11/21/21

CSE13S Assignment 6 Design Document

Purpose:

The purpose of this assignment is to create a program that performs public key cryptography. Essentially, it is used when a person wants to send an encrypted message to someone else. There is both a public key and a private key that need to be used in order to do this. Any message can be encrypted using a public key, but it must be decrypted using the receiver's private key. This is a bit similar to encoding and decoding a message. The purpose is to be able to privately send and receive any message. The algorithm that we will be using in this assignment for the encryption/decryption is called RSA (for Rivest, Shamir, and Adleman).

Structure:

The RSA algorithm makes use of prime numbers, which will need to be very large for a good encryption. So, this program will have separate files for random number generations as well as some mathematical functions involving prime numbers that will be needed. Then, another file will be used for RSA, namely making, reading, and writing public and private keys, as well as a signature and verification function. Then, there will be three "main" files, one for the generation of both public and private keys, one for the encryption of a message, and one for the decryption of the message.

Note: this program makes uses of the GMP library and its functions involving the use of mpz_t variables

Files:

(pseudocode in Python/English)

randstate.c:

```
void randstate_init(uint64_t seed):
   init the state with gmp_randinit_mt()
   set the random seed with gmp_randseed_ui()

void randstate_clear(void):
   clear state with gmp_randclear()
```

This file is the random state module. It will need to be used in order to generate large numbers for the encryption using the GMP library. This file will only contain two functions, which will both make use an extern (and global) random state variable named state. The first function randstate_init will take as a parameter an unsigned 64-bit integer, which will be the seed of random number generation. It will initialize the state variable with a call to gmp_randinit_mt(), which uses a Mersenne Twister algorithm. Then, the seed will be used in a call to gmp_randseed_ui(). The next function will be randstate_clear. It takes no parameters and is responsible for clearing any memory used for the state variable. This is done using a call to the function gmp_randclear().

numtheory.c:

This file will contain the mathematical functions that need to be used to perform RSA. The functions it will include are power_mod (for modular exponentiation), is_prime (for testing the primality of a number using the Miller-Rabin method), make_prime (for generating large prime numbers), gcd() (to check the greatest common divisor of two numbers), and mod_inverse (to compute the inverse of a modulo number).

```
void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus):
    declare/initialize mpz_t variables
    set v to 1
    set p to a
    while d > 0:
        if d is odd:
            v = v *p % n
        p = p *p % n
        d = d/2
    #end while
    set out to v
```

The function pow_mod performs modular exponentiation. Its return types is void, and it takes 4 mpz_t parameters, out, base, exponent, and modulus. First, the necessary mpz_t variables are initialized, and v, p, and temp_d (which will be used as a temporary exponent are set to the necessary values. The temporary variable is necessary because the values of the parameters exponent will be changed. Then, while temp_d is greater than 0, if temp_d is odd, then v is set to v * p % n. Outside that if statement, p is set to p * p mod % n. Finally, still within the while, temp_d is floor divided by 2. This while loop will result in v being the “result”, and so at the end, the out mpz_t is set to v, and all mpz_t variables are cleared.

```
bool is_prime(mpz_t n, uint64_t iters):
    if n > 2 and even:
        return false
    if n = 0 or n = 1:
        return false
    if n = 2 or n = 3:
        return false
    declare/initialize mpz_t variables
    set n_minus_1 to n -1
    set s to 2
    while n-1 is divisible by 2^s:
        s++
    #end
    s --
    r = (n-1)/2^s
    for i = 1 to k:
        choose random {2, n-2}
        y = pow_mod(a,r, n)
        if y != 1 and y != n - 1:
            set j to 1
            while j <=s and y != n - 1:
                y = pow_mod(a,2, n)
                if y == 1:
                    return false
                #end if
                j = j + 1
            #end while
            if y != n -1:
                clear mpz variables
                return false
        #end if
    #end for
    clear mpz variables
    return true
```

```
void make_prime(mpz_t p, uint64_t bits, uint64_t iters):
    do:
        make random number p with state and bits
    while (p is not prime or pbits < bits or p is not odd)
```

The function is_prime performs the Miller-Rabin primality test, which will tell you that a number is probably prime, but not absolutely prime. It returns a boolean telling you if the number is prime or not, and takes in two parameters, n, the number to test, and iters, the number of iterations used to test a number. First, the value of n is checked. If n is greater than 2 and even or equal to 0 or 1, false is returned; if n is equal to 2 or 2, true is returned. These serve as “base” checks on the number to be tested. Then, the mpz_t’s needed are set, including one called n_minus_t (set to n - 1), and a mp_bitcnt_t s is also set to the value of 2. The first part of the function is “write n−1 = 2 s r such that r is odd.” For this, I have a while loop which iterates using the mpz_divisible_2xp_p() function. The loop iterates while n-1 is divisible by 2^s, and s increases by 1 with each iteration. When n - 1 is no longer divisible by 2^s, the last s was valid (in making r odd), so s is decremented. Then r is set to the value of (n-1)/2^s, using truncation division (and it is odd). Then, I have a for loop, and the first thing within it is the generation of a random number between 2 and n-2 using mpz_urandomm(). Then, pow_mod() is called using the appropriate parameters. After that, I use mpz_t’s y and j, whose values are set and compared to see if pow_mod() needs to be called again, or a number can be determined as not true within the for loop (which entails a return of false). Finally, I clear the mpz_t’s initialized and return true, indicating a “probably” prime number.

The function make_prime is used to generate a prime number using the function is_prime within it. Its return type is void, and it takes parameters p (to store the generated prime number), bits (used to check the number of bits in p), and iters (number of iters with which to test p). The structure of make_prime() is a do-while loop. Within the “do” section, I generate a number using the function mpz_urandomb (passing p, state (from randstate.c), and bits as parameters). This generation of numbers should occur while p is not prime (checked using is_prime()), not big enough (number of bits of p less than bits), or not odd (checked using mpz_odd_p()). The result is a large prime number stored in parameter p.

```
void gcd(mpz_t d, mpz_t a, mpz_t b):
    declare/initialize mpz variables
    while b != 0:
        t = b
        b = a % b
        a = t
    #end while
    set d to a
    clear mpz variables

void mod_inverse(mpz_t i, mpz_t a, mpz_t n):
    declare/initialize mpz variables
    r = n, r' = a
    t = 0, t' = 1
    while r' != 0:
        q = r/r'
        r = r', r' = r - q * r'
        t = t', t' = t - q * t'
    #end while
    if r > 1:
        clear mpz variables
        set i to 0
    if t > 0:
        t = t + n
    clear mpz variables
    set i to t
```

The function gcd() computes the greatest common divisor between two numbers a and b. Its return type is void, and it takes 2 parameters, mpz_t's a and b, and mpz_t d, to store the result. First, I initialize temporary variables for a and b (so as not to overwrite the parameters' values) and a variable t. Then, I set temp_a and b to a and b. Then, the while loop in the function iterates while temp_b is not equal to 0. Within it, the variable t is set to b, b is set to a % b, and a is set to t. The loop will result in the GCD being stored in temp_a. Outside the while loop, I clear the mpz_t's and set the parameter d to the value of temp_a, as a "return."

The function mod_inverse() computes the inverse of a number a modulo a number n. Its return type is void, and it takes parameters a and n and an output parameter i. This function would be implemented using parallel assignment, but since that is not allowed in C, values are set line by line. First, I initialize the mpz_t's needed, namely ones like r, r', t, t', and q. First, I set them to the necessary values. Then, there is a while loop which iterates while r' is not equal to 0. Within it, I set q equal to r / r_prime (floor division). Then, r must be set to r', and r' must be set to r - q * r'. However, in the function, this should be done with parallel assignment, but because that isn't possible, a temporary variable is used to accomplish this. Then, the same assignments are done for t and t'. Outside the while loop, if r is greater than 1, no inverse is returned, or the output is set to 0. If t is less than 0, t is set to the value of t + n, which will be the result of the modular inverse calculation. So, afterwards checking the value of t, the output i is set to t ("return t"), and all made mpz_t's are cleared).

rsa.c:

This file will contain the implementation of RSA functions, using the function from the file numtheory.c. It will include the functions to make, read, and write both public and private keys. Then, there will be separate functions for performing RSA encryption and decryption, both for a message and a whole file. Finally, the last two functions will be used to perform RSA signing and verification.

```
void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters):
   declare and initialize mpz variables
   int pbits = (random() % (nbits / 2)) + (nbits / 4) + 1
   int qbits = q_bits = nbits - p_bits + 1
   make the primes p and q
   n = p*q
   totient = (p-1)(q-1)
   do:
      generate random number e with state and bits
      gcd of e and totient = gcd(e, totient)
   while gcd of e and totient != 1
   clear mpz variables

void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile):
   print all parts (hexstrings, username) of public key to pbfile

void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile):
   scan in all parts of public key into parameters
```

The function rsa_make_pub() is responsible for creating parts of a new RSA public key. Its return type is void, and its parameters are p and q (two large primes), n (the result of multiplying p and q - public modulus), e (the public exponent), nbits (the size of n), and iters (to specify the number of Miller-Rabin iterations needed). First, needed mpz_t's are initialized. Then p and q are made using the function make_prime(). First, their number of bits is specified to be in a certain range. The number of bits for p should be in the range [nbits/4,(3×nbits)/4), and the remaining bits go to q. This is accomplished using random() and using a modulus and offset. Then, n is set the value of p*q. Then the totient, or the value of (p-1)(q-1) is calculated and set to an mpz_t. Then, a public exponent e is generated until the GCD between e and the totient is one (they are coprime) E is generated using the function mpz_urandomb(), and the GCD is checked using the function gcd(). Finally, all mpz_t's are cleared.

The function rsa_write_pub() writes an RSA public key to an output file pbfile. Its return type is void, and its parameters are parts of a public key (n, e, s), a char[] username (storing the username of the user), and the output file pbfile. This function is done using gmp_fprintf(), with the mpz_t's being written as hexstrings (with "%Zx") and newlines trailing them.

The function rsa_read_pub() reads an RSA public key from a file. Its return type is void, and it takes the same parameters as rsa_write_pub(). This is accomplished by using gmp_fscanf(), with the mpz_t's being read as hexstrings (with "%Zx") and the user name being read as well.

```
void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q):
   declare and initialize mpz variables
   totient = (p-1)(q-1)
   d = mod_inverse(e, totient)
   clear mpz variables

void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile):
   print parts of private key (hexstrings) to pvfile

void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile):
   scan parts of private key into parameters from pvfile
```

The function rsa_make_priv() creates a new RSA public key. Its return type is void, and it takes as parameters d (to store the private key), p and q (primes), and public exponent e. D is computed by taking the inverse of e modulo the totient of n ((p-1)(q-1)). So, within the function, the totient of n is calculated, and then the function mod_inverse() is called to find the inverse of e % totient(n), with d storing the result.

The function rsa_write_priv() writes an RSA private key to an output file pvfile. Its return type is void, and its parameters are parts of a private key (the public modulus n and the private key d) and the output file pvfile. This function is done using gmp_fprintf(), with the mpz_t's being written as hexstrings (with "%Zx") and newlines trailing them.

The function rsa_read_priv() reads an RSA private key from a file. Its return type is void, and it takes the same parameters as rsa_write_priv(). This is accomplished by using gmp_fscanf(), with the mpz_t's being read as hexstrings (with "%Zx").

```
void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n):
   pow_mod(c, m, e, n)

void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n):
   pow_mod(m, c, d, n)
```

The functions rsa_encrypt() and rsa_decrypt() perform RSA encryption and decryption respectively. The function rsa_encrypt()'s return type is void, and it takes parameters c, m, e, and n; it uses the function pow_mod() to calculate c = m^e (% n), with c being the ciphertext, m being the message, e being the public exponent, and n being the public modulus. The function rsa_decrypt()'s return type is void, and it takes parameters m, c, d, and n; it uses the function pow_mod() to calculate m = c^d (% n), with c being ciphertext, d being the private key, m being the message, and n being the public modulus.

```
void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e):
    k = log base 2 (n) -1 / 8
    block = uint8_t array, size = k
    size_t j = 0
    while not at end of file:
        j = bytes read from infile in blocks
        if j > 0:
            import bytes from infile into mpz_t m
            encrypt c with rsa_encrypt(), store in c
            print hexstring mpz_t c to outfile
        #end if
    #end while
    free(block)
    clear mpz variables

void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d):
    k = log base 2 (n) -1 / 8
    block = uint8_t array, size = k
    size_t j = 0
    while not at end of file:
        int bytes_scanned = hexsting scanned from infile, stored in c
        if bytes_scanned > 0:
            decrypt c with rsa_encrypt(), store in m
            export m to convert back into bytes
            write contents of block to outfile
            #end if
    #end while
    free(block)
    clear mpz variables
```

The function rsa_encrypt_file() encrypts a file, writing the output from an input file to an output file. Its return type is void, and its parameters are file pointer infile and outfile and mpz_t's n and e. The data is going to be encrypted in blocks. First, the block size k is calculated; k should equal (log base 2 of (n)−1)/8. The log base 2 of n is calculated using the function mpz_sizeinbase(). Then a array, block, of type uint8_t * is dynamically allocated to be the size of k. The first index in the array block is set to the value of 0xFF. Then the input file is read. While there are still bytes to read in infile (tracked using feof()), k - 1 bytes are read using the function fread(). The number of bytes read is stored in a size_t j. While j is greater than 0, the read bytes are converted into an mpz_t m using mpz_import, m is encrypted using rsa_encrypt(), and the ciphertext result c is printed to the outfile using gmp_fprintf. Finally, the last thing in the function is to free the array block and clear the mpz_t's used.

The function rsa_decrypt_file() encrypts a file, writing the output from an input file to an output file. Its return type is void, and its parameters are file pointer infile and outfile and mpz_t's n and d. The data is going to be encrypted in blocks. First, the block size k is calculated; k should equal (log base 2 of (n)−1)/8. The log base 2 of n is calculated using the function mpz_sizeinbase(). Then a array, block, of type uint8_t * is dynamically allocated to be the size of k. Then the input file is scanned for the hexstring mpz_t's. While there are still bytes to scan in infile (tracked using feof()), the hexstrings are scanned into an mpz_t c using gmp_fscanf(), and the number of bytes scanned is stored in a variable bytes_scanned. While bytes_scanned is greater than 0, the ciphertext c is decrypted into message m using rsa_decrypt(), m is exported back into bytes using mpz_export(), and all the necessary bytes are written to the outfile using fwrite(). Finally, the last thing in the function is to free the array block and clear the mpz_t's used.

```
void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n):
   pow_mod(s, m, d, n)
```

```
bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n):
   initialize mpz_t t
   pow_mod(t, s, e, n)
   if t is the same as m:
      clear t
      return true
   #end if
   clear t
   return false
```

The last two function are RSA signing and verification. The function rsa_sign() has a void return type and it takes in parameters mpz_t's s, m, d, and n. The signature is produced using a call to pow_mod(), because the signature s = m^d (% n). The signature is stored in the mpz_t s. The function rsa_verify() returns a boolean stating if the signature s was verified or not, and it takes in parameters mpz_t's m, s, e, and n. The verification, stored in an mpz_t, is equal to s^e (% n), so it is computed using pow_mod(). Then, if t is the same (or equal to) the message mpz_t m, true is returned, otherwise, false is returned.

keygen.c:

```
int main(int argc, char **argv):
    set default values and booleans
    parse for command line options in getopt() loop (switch + while)
    case 'b', 'i', 's':
        corresponding variable = strtoul(optarg, NULL, 10)
        break
    case 'n', 'd':
        pv/pbfile = optarg
    case 'v', 'h':
        corresponding boolean = true
        break
    default:
        break

    if -h is specified:
        print help message
        exit

    initialize mpz_t variables
    open public/private key files
    set private file permisssions
    initialize random state

    make public key
    make private key

    get username (with getenv())
    convert username to mpz_t user
    rsa_sign(signature, user, d, n)

    if -v option is specified:
        print verbose output

    clear mpz_t variables
    clear random state
    close input file and output file
    return 0
```

This file will be responsible for the generation of public and private keys, by first taking command line options, and then calling the necessary functions.

These are the following command line options for keygen, -b, for minimum bits needed for n (default: 256), -i, the number of Miller-Rabin iterations (default: 50), -n pbfile, the public key file (default: rsa.pub), -d pvfile, the private key file (default: rsa.priv), -s the random seed (default: time(NULL)), -v, for verbose output, and -h for the help message. First, I declare necessary variables for defaults and booleans for verbose and help options. Then, I parse command line options accordingly. After that, I open the public and private key file with fopen(). Afterwards, I initialize the seed using randstate_init() and set the private key file permissions to 0600 using fchmod(). Then, I make the private and public keys with the corresponding functions, and get the user's username using get_env(). Then, I convert the username to an mpz_t using mpz_set_str() and use rsa_sign() to find the signature of the username. Then, I write public and private keys to their respective files. Finally, I print the verbose or help messages if specified. At the end, I close all files, clear mpz_t's used, and return 0.

encrypt.c:

```
int main(int argc, char **argv):
   set default values and booleans
   parse for command line options in getopt() loop (switch + while)
   case 'i', 'o':
      infile/outfile = fopen(optarg, 'r' or 'w')
      break
   case 'n':
      pbfile = optarg
      break
   case 'v', 'h':
      corresponding boolean = true
      break
   default:
      break

   if -h is specified:
      print help message
      exit

   open public key file
   initialize mpz_t variables
   initialize username array
   read public key from the public key file

   if -v option is specified:
      print verbose output

   convert username to mpz_t user
   rsa_verify(message, user, e, n)

   rsa_encrypt_file(infile, outfile, n, e)

   clear mpz_t variables
   close input file and output file
   return 0
```

This file will be responsible for the encryption of a message, by first taking command line options, and then calling the necessary functions.

These command line options are used for encrypt: -i for the input file (default: stdin), -o for the output file (default: stdout), -n for the public key file (default: rsa.pub), -v for verbose output, -h for help message. First, I declare necessary variables for defaults and booleans for verbose and help options. Then, I parse the command line options accordingly. Then, I open the public key file using fopen(). Then, I read the public key from the public key file with rsa_read_pub() and encrypt the message using rsa_encrypt_file(). Then, I convert the user's username to an mpz_t with mpz_set_str() and verify it with rsa_verify(). Finally, I print the verbose or help messages if specified. At the end, I close all files, clear mpz_t's used, and return 0.

decrypt.c:

```
int main(int argc, char **argv):
   set default values and booleans
   parse for command line options in getopt() loop (switch + while)
   case 'i', 'o':
      infile/outfile = fopen(optarg, 'r' or 'w')
      break
   case 'n':
      pvfile = optarg
      break
   case 'v', 'h':
      corresponding boolean = true
      break
   default:
      break

   if -h is specified:
      print help message
      exit

   open private key file
   initialize mpz_t variables
   read private key from the private key file

   if -v option is specified:
      print verbose output

  rsa_decrypt_file(infile, outfile, n, d)

   close input file and output file and pvfile
   clear mpz_t variables
   return 0
```

This file will be responsible for the decryption of a message, by first taking command line options, and then calling the necessary functions.

These command line options are used for decrypt: -i for the input file (default: stdin), -o for the output file (default: stdout), -n for the private key file (default: rsa.priv), -v for verbose output, -h for help message. First, I declare necessary variables for defaults and booleans for verbose and help options. Then, I parse the command line options accordingly. Then, I open the private key file using fopen(). Then, I read the private key from the private key file with rsa_read_priv() and decrypt the message using rsa_decrypt_file(). Finally, I print the verbose or help messages if specified. At the end, I close all files, clear mpz_t's used, and return 0.