Assignment 6 DESIGN.pdf

## Description of Program:

This program's purpose is to generate corresponding RSA public and private key pairs, use the public key to encrypt files, and to decrypt encrypted files using the corresponding private key. This will be accomplished via three separate executables: a keygen, an encrypter, and a decrypter. The program will be implemented using the GNU multiple precision arithmetic library.

## Files to be included in directory "asgn6":

1. decrypt.c: This file contains the implementation and main() for the decryption program.

2. encrypt.c: This file contains the implementation and main() for the encryption program.

3. keygen.c: This file contains the implementation and main() for the key generation program.

4. numtheory.c: This file contains the implementation of several number theory functions.

5. numtheory.h: This file specifies the interface for the number theory functions.

6. randstate.c: This file contains the implementation of the random state interface for the other functions.

7. randstate.h: This file specifies the interface for initializing and clearing the random state.

8. rsa.c: This file contains the implementation of the RSA library.

9. rsa.h: This file specifies the interface for the RSA library.

## Pseudocode / Structure:

## Keygen.c:

Write a helper function to identify how many bits make up a mpz_t variable.

Parse command line options with getopt(), then:

If -b was input, set the minimum bits needed for the public modulus to the argument passed.

If -i was input, set the number of Miller-Rabin iterations to the argument passed.

If -n was input, set the public key file pointer to the argument passed. Otherwise, set it to rsa.pub.

If -d was input, set the private key file pointer to the argument passed. Otherwise, set it to rsa.priv.

If -s was input, set the random seed for the random state to the argument passed. Otherwise, set it to time(NULL).

If -v was input, set variable “verbose” to true.

If -h was input, display the help message.

1. fopen() the public and private key files.

    Print an error message if this fails.

2. Use fchmod() and fileno() to ensure the private key’s file permissions are 0600.

3. Call randstate_init() with the specified random seed as input.

4. Call rsa_make_pub() and rsa_make_prv(), passing in all their requested variables.

5. Get the current user’s name using getenv().

Convert it into a mpz_t using mpz_set_str().

Then, use it to make a signature using rsa_sign().

6. Use rsa_write_pub() and rsa_write_priv() to write the public and private keys, in hex, to their specified files.

7. If variable “verbose” is true, print to the terminal:

The user’s name

The signature s

The two prime numbers p and q

The public modulus n

The public exponent e

The private key d

And the number of bits constituting each of these variables. Make sure to print to variable numbers in decimal.

8. Close the files, clear the random state, and clear any active mpz_t variables.

## Encrypt.c:

Write a helper function to identify how many bits make up a mpz_t variable.

Parse command line options with getopt(), then:

If -i was input, set the input file to the argument passed. Otherwise, set it to stdin.

If -o was input, set the output file to the argument passed. Otherwise, set it to stdout.

If -n was input, set the file pointer to the public key to the argument passed. Otherwise, set it to rsa.pub.

If -v was input, set the variable “verbose” to TRUE.

If -h was input, display the health message.

1. fopen() the public key file.

    If this fails, print an error message to the terminal.

2. Call rsa_read_pub() to read the public key from its file.

3. If variable “verbose” is TRUE, print to the terminal:

    The user’s name

    The signature s

    The public modulus n

    The public exponent e

    And the number of bits constituting each of these variables. Make sure to print to variable numbers in decimal.

4. Convert the user’s name to an mpz_t, and then run rsa_verify() on it.

    Print an error and exit the program if the verification fails.

5. Use rsa_encrypt_file() to encrypt the file.

6. Close the public key file and clear the mpz_t variables.

## Decrypt.c:

Write a helper function to identify how many bits make up a mpz_t variable.

Parse command line options with getopt(), then:

> If -i was input, set the input file to the argument passed. Otherwise, set it to stdin.
>
> If -o was input, set the output file to the argument passed. Otherwise, set it to stdout.
>
> If -n was input, set the file pointer to the public key to the argument passed. Otherwise, set it to rsa.priv.
>
> If -v was input, set the variable "verbose" to TRUE.
>
> If -h was input, display the health message.

1. fopen() the public key file.

    If this fails, print an error message to the terminal.

2. Call rsa_read_prv() to read the private key from its file.

3. If variable "verbose" is TRUE, print to the terminal:

    The public modulus n

    The private key e

    And the number of bits constituting each of these variables. Make sure to print to variable numbers in decimal.

4. Use rsa_decript_file() to decrypt the file.

5. Close the file and clear any active mpz_t variables.

## Numtheory.c:

Make a void gcd() function that takes in three mpz_t's, d, a, and b, and finds the greatest common denominator of the latter two, and puts the result in the first variable.

While b is not zero,

set a temp variable to b

set b to a mod b

set a to the temp variable

set d to a

Make a void mod_inverse() function that takes in three mpz_t's, i, a, and n, and finds the mod inverse of a and n, putting the result in i.

Set (r, r') to (n, a)

Do this by creating a new variable for r', and:

Setting r to n;

Setting the new variable to a;

Set (t, t') to (0, 1)

Do this by creating a new variable for t', and:

Setting t to 0;

Setting the new variable to 1;

While r' is not zero,

set q to fdiv(r/r’)

set (r, r’) to (r’, r-q*r’)

Do this by making a temp variable temp_r with value r, and another temp variable with value r’, and:

Setting r to r’;

Setting r’ to temp_r-q*temp_r’

set (t, t’) to (t’, t-q*t’)

Do this by making a temp variable temp_t with value t, and another temp variable with value t’, and:

Setting t to t’;

Setting t to temp_t-q*temp_t’

If r > 1,

return no inverse

If t < 0,

set t to t + n

Set i to t

Make a void pow_mod() function that takes four mpz_t’s, out, base, exponent, and modulus, using the last 3 variables to compute the power mod and put it in out.

Set v to 1.

Set p to a.

```
While exponent > 0:
    if exponent is odd:
        Set v to (v*p) mod modulus
    Set p to (p*p) mod modulus
    Set exponent to fdiv(exponent/2)
Set out to v
```

Make a bool function is_prime() that takes two mpz_t variables, one number n to test the primality of, and one number of iterations.

```
Write n – 1 = (2^s)r such that r is odd
From 1 to iterations:
    choose a random number from 2 to n-2
    Set y to pow_mod(a, r, n)
    if y is not 1 and y is not n – 1:
        set j to 1
        while j is less than or equal to s-1 and y is not n-1:
            set y to pow_mod(y, 2, n)
            if y is 1:
                return FALSE
            add 1 to j
```

if y is not n-1:

return FALSE

return TRUE

Make a void function make_prime() that takes in an mpz_t p, and two uint64_t's, bits and iters.

Generate random numbers and run them through is_prime() with iters iterations until a valid prime number is found.

## Randstate.c:

Make a void randstate_init() function that takes a uint64_t variable seed as input.

Initialize the random state with gmp_randinit_mt().

Seed the random state with gmp_randseed() and seed.

Make a void randstate_clear() function with no input.

Clear the random state with gmp_randclear().

## Rsa.c:

Make a void function rsa_make_pub() that takes in mpz_t's p, q, n, and e, as well as uint64_t's nbits and iters.

Create primes p and q using make_prime(). Make p have a random bit count between nbits/4 and 3nbits/4, and give the remaining bits to q.

Compute the totient: (p-1)(q-1).

Make a loop where numbers of roughly nbits are randomly generated using mpz_urandom().

Stop the loop when a number coprime with the totient is found, and make it the public exponent.

Make a void function rsa_write_pub() that takes in mpz_t's n, e, and s, as well as char username[] and FILE *pbfile.

Write the public RSA key to the pbfile in the format n, e, s, (in hex), then username.

Make a void function rsa_read_pub() that takes in mpz_t's n, e, and s, as well as char username[] and FILE *pbfile.

Read the public RSA key to the pbfile in the format n, e, s, (in hex), then username.

Make a void function rsa_make_priv() that takes in mpz_t's d, e, p, and q as input.

Create d by the equation d = e mod (p-1)(q-1).

Make a void function rsa_write_priv() that takes in mpz_t's n and d, as well as FILE * pvfile, as input.

Write n, then d, to pvfile, both as hexstrings.

Make a void function rsa_read_priv() that takes in mpz_t's n and d, as well as FILE * pvfile, as input.

Read n, then d, from pvfile, both as hexstrings.

Make a void function rsa_encrypt() that takes in mpz_t's c, m, e, and n as input.

Perform RSA encryption, computing c by the equation (m^e) mod n.

Make a void function rsa_encrypt_file() that takes in two FILE *'s, infile and outfile, as well as mpz_t's n and e.

Calculate the block size k with k = fdiv((log2(n)-1)/8).

Dynamically allocate an array that can hold k bytes of type uint8_t *.

Do this by getting the value of k with mpz_get_ui(), and then callocing the array with it.

Set the zeroth byte of the block to 0xFF.

While there are still unread bytes:

Read at most k-1 bytes from infile and place them into the block from index 1.

Using mpz_import(), convert the bytes into an mpz_t m.

Encrypt m with rsa_encrypt(), then write the number to outfile as a hexstring.

Make a void function rsa_decrypt() that takes mpz_t's m, c, d, and n as input.

Decrypt c using the equation (c^d)mod n. Pass the answer out with m.

Make a void function rsa_decrypt_file() that takes FILE *'s infile and outfile, as well as mpz_t's n and d, as input.

Calculate the block size k with fdiv((log(n)-1)/8).

Dynamically allocate an array that can hold k bytes of type uint8_t *.

Do this by getting the value of k with mpz_get_ui(), and then callocing the array with it.

While there are unread bytes:

Scan in a hexstring as a mpz_t.

Use mpz_export() to convert c back into bytes.

Write out j-1 bytes, if j is the number of bytes actually read, to outfile.

Make a void function rsa_sign() that takes mpz_t's s, m, d, and n as input.

Sign m with the equation (m^d) mod n. Pass the result out through s.

Make a bool function rsa_verify() that takes mpz_t's m, s, e, and n as input.

Verify signature s. If (s^e) mod n is the same as m, return TRUE. Otherwise, return FALSE

## Credit:

Thank you to Elmer, the CSE13S helper, for providing Python pseudocode for the bit calculation helper functions.