Asgn 5: Final Design Doc
Mika Peer Shalem
Mpeersha

General Overview

Implements public-key and symmetric-key cryptography to understand how to decipher and encrypt messages. This uses a system with a pair of keys: one public and one private. The public key is used to encrypt the message, while the private key is used to decrypt the message. The public is available for everyone, while the private one can be seen only by the owner. The message would be encrypted through a combination of a person's public key and a prime number that is compatible with that. The algorithm depends on factors of large, prime numbers.

I will implement three main functions, two libraries, and a module.
Each main function has command line options that would need user input.
The first main function, the key generator, is used to produce a pair of public and private keys through the use of user input.
The second function, encrypt, would cipher a given file using the public key.
The last function, decrypt, would decipher a given file using the private key.
The first library, numtheory, is used to handle the mathematics behind RSA, while the second one, the RSA library, includes the actual RSA routine.
Lastly, the randstate module would generate random numbers that would later be used to create keys.

Key Generator

Goal: produce RSA public and private key pairs
Create a function that prints the error message to stderr.

Main:
The function would return 0 unless the user inputs an invalid option.

Set default iteration to 50
Set default public_file to rsa.pub
Set default private_file to rsa.priv
Set default bits to 1024
Set default seed to the current seconds (time(NULL))
Get the global variable state

// PARSE COMMAND LINE OPTIONS

```
While getopt argument is not -1 (getopt goes through all the given options and returns -1 when it
runs out of options):

    If the argument is equal to b:
        If the user input is less than 50: print error and exit
        Set the bits var to optarg
    If the argument is equal to i:
        Convert optarg to an int and set iteration to optarg
    If the argument is equal to n:
        Set public_file to optarg
    If the argument is equal to d:
        Set private_file to optarg
    If the argument is equal to s:
        Set seed to optarg
    If the argument is equal to v:
        Enables verbose output
    If the argument is equal to h:
        Displays help message using the function
    Else
        Displays help message using the function
        Exit with a nonzero number

// FILES, used to make sure the files exist and work
Open public and private files
Set two file pointers to the return value of fopen (either NULL or pointer to the file)
If the pointer is NULL:
    Print an error message and exit the program
Set permissions of private key file to 0600

//CALL OTHER FUNCTIONS
Set state by calling randstate_init() with the seed
Call rsa_make_pub() and rsa_make_priv()
Set name by calling getnev()
Set name to mpz_t by calling mpz_set_str() with base 62
Set signature by calling rsa_sign() with name
Call rsa_write_pub() and rsa_write_priv() to send the keys to the files

//VERBOSE
If verbose was enabled:
    Print username, signature, large prime, second large prime, public modulus, public exponent, and private key

//END
Close the public and private files
Clear the random state
Clear the mpz_t variables
```

Encrypt

Goal: encrypt files using a public key

Create a function that prints the error message to stderr.

Main:
The function would return 0 unless the user inputs an invalid option.

```
Set default input_file to stdin
Set default output_file to stdout
Set default pub_file to rsa.pub
Set flags to verbose output, if the user gave an input file, and if the user gave an output file

// PARSE COMMAND LINE OPTIONS
While getopt argument is not -1 (getopt goes through all the given options and returns -1 when it runs out of options):

    If the argument is equal to i:
        Set input file flag
        Set the input_file to getopt
    If the argument is equal to o:
        Set output file flag
        Set the output_file to getopt
    If the argument is equal to n:
        Set pub_file to optarg
    If the argument is equal to v:
        Enables verbose output
    If the argument is equal to h:
        Displays help message using the function
    Else
        Displays help message using the function
        Exit with a nonzero number
```

```
// FILES
Set pub to the return value of fopen of the public key (either NULL or pointer to the file)
If pub is NULL:
    Print an error message and exit the program

If user gave an input file:
    Open it
If user gave an output file:
    Open it
If cannot open the file:
    Print an error message and exit the program

//CALL OTHER FUNCTIONS
Read public key

//VERBOSE
If verbose was enabled:
    Print username, signature, modulus, and the public exponent

// ENCRYPTION
Convert the username to an mpz_t
Check that rsa_verify() returns the same username
If they are not the same:
    Print an error message and exit

Encrypt the file by calling rsa_encrypt_file()

//END
Close the public files
Clear the mpz_t variables
```

Decrypt

Goal: decrypt the encrypted files using the private key

Create a function that prints the error message to stderr.

Main:

The function would return 0 unless the user inputs an invalid option.

```
Set default input_file to stdin
Set default output_file to stdout
Set default priv_file to rsa.priv
Get the global variable state
Set flags to verbose output, if the user gave an input file, and if the user gave an output file

// PARSE COMMAND LINE OPTIONS
While getopt argument is not -1 (getopt goes through all the given options and returns -1 when it
runs out of options):

    If the argument is equal to i:
        Set input file flag
        Set the input_file to getopt
    If the argument is equal to o:
        Set outputfile flag
        Set the output_file to getopt
    If the argument is equal to n:
        Set priv_file to optarg
    If the argument is equal to v:
        Enables verbose output
    If the argument is equal to h:
        Displays help message using the function
    Else
        Displays help message using the function
        Exit with a nonzero number

// FILES
Set priv to the return value of fopen of the private key (either NULL or pointer to the file)
If priv is NULL:
    Print an error message and exit the program

If user gave an input file:
    Open it
If user gave an output file:
    Open it
If cannot open the file:
    Print an error message and exit the program

//CALL OTHER FUNCTIONS
Read public key
```

```
//VERBOSE
If verbose was enabled:
    Print modulus and the private key with the number of bits used

// DECRYPTION
Decrypt the file by calling rsa_decrypt_file()

//END
Close the public files
Clear the mpz_t variables
```

randstate.c

Global variable state (pass to random integer functions in GMP)

void randstate_init(uint64_t seed)

Initializes the global random variable named state with a random number.

```
Call gmp_randinit_mt() with the var state
Call gmp_randseed_ui() to initialize the state with the random numbers using the seed.
Set state by calling srandom() with seed as the variable sets seed for a new sequence of random numbers.
```

void randstate_clear(void)

Free memory used by the global random variable state

```
Call gmp_randclear() on the global variable state to clear/free all memory used.
```

numtheory.c

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

Implementation of the modular exponentiation.

It is the reminder of the base to the power of e divided by modulus. Used during the key exchange to encrypt the plain message

```
Set the variable out to 1
Create copies of all arguments since they are passed by reference and we don't want to change the original values
While the exponent is greater than 0:
    If the exponent is odd:
        Set out to out times base with mod modulus
    Set base to base times base mod modulus
    Set exponent to exponent divided by 2
Set t to out
```

bool is_prime(mpz_t n, uint64_t iters)

Checks if a number is prime or not.

```
Set r to n-1
Set s to 0
While r is even:
    Set r to n-1
    S += 1

For i is equal to one, until i is equal to iters, increase i by one each time:
    Choose a random integer a that is in the range (2 to n-2)
    Set y by calling the power mod function with a, r, and n
    If y is not equal to 1 and y is not equal to n-1:
        Set j to 1
        While j is smaller than s-1 and y is not equal to n-1:
            Set y by calling the power mod function with the vars y, 2, and n
            If y is equal to 1:
                Return false
            Set j to j+1
            If y is not equal to n-1:
                Return false
Return true
```

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

Creates a prime number using the previously described function.

```
Set a var to 2^bits-1
While p is not prime: // gets a new number every time until reaching a prime number
    Generate a new random number (in the range 2^bits-1) and save it in p
    Add 2^bits to the random number
    Test if it's prime
```

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

Calculates the greatest common denominator of two numbers

```
While b is not equal to 0: // since we set b to the remainder of a and b, if b is equal to zero then
the number is a denominator of the two numbers
```

```
        Set to t to the number b
        Set b to the number a mod b
        Set a to t
Set d to a
```

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

Calculates the inverse value of a modulo n.

```
Set r to n
Set r' to a
Set t to 0
Set t' to 1
While r' is not equal to 0:
        Set q to r divided by r'
        Set temp_r to r
        Set temp_r' to r'
        Set r to temp_r'
        Set r' to temp_r - q*r'

        Set temp_t to t
        Set temp_t' to t'
        Set t to temp_t'
        Set t' to temp_t - q*t'

If r is greater than 1:
        Set i to 0
        Stop the function
If t is smaller than 0:
        Add n to t
Set i to t
```

RSA library

Note: Convert to a hexstring:

Many of the functions described below need to convert the inputs to hexstrings.
The GMP library in C has existing functions to help format the input and output, so they would be in the form of hexstrings. In other languages, there are other libraries that can help with the formatting.

void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters)

Creates a new RSA public key. This includes two prime numbers, p and q, their product n, and the public exponent e.

```
Set p and q to two different prime numbers using make_prime()
    Bits for p are equal to a random number in the range (nbits/4 to 3nbits/4) // ex. if there are 8 bits, the range is between 2 to 6
    Bits for q are equal to the total nbits minus the number of bits allocated to p
    Need to make sure that log2(n) >= nbits: create a while function that generates new prime numbers and calculates the product until the condition is satisfied

LCM = the absolute value of the totient of n / (GCD of p-1 and q-1))

// coprime of two numbers → gcd of them is equal to 1
// e also needs to be in the rage (2,n)
Infinite while loop:
    Set e to a random number with size nbits
    Compute the gcd and lamda of e
    If e is in the range and gcd is 1:
        Stop the loop
```

void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)
Write to the public key
Make sure we are at the beginning of the file
Change n, e, s to hex strings
Write on pbfile n, e, s, and username with a trailing new line after each one

void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)
Read from the public key
Make sure we are at the beginning of the file
Get the value from pbfile by reading from it n,e,s, and username

void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q)
Creates a new RSA private key d
Set d to the LCM of (p-1, q-1) which is equal to the absolute value of the totient of n / (GCD of p-1 and q-1))

Calculate the inverse mod of d

void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile)
Write to the private key file
Make sure we are at the beginning of the file

Change n and d to hex strings
Write on pbfile n and d with a trailing new line after each one

void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile)
Read from the private key file
Make sure we are at the beginning of the file
Get the value from pvfile by reading it

void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)
Performs RSA encryption
Set c to m to the power of e (mod n)

void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e)
Encrypts the contents of infile, writing the encrypted contents to outfile. The data in the infile should be encrypted in blocks.

Set k to the floor of ($\log_2(n)$-1)/8 // k is the block size. Need to divide the message to be encrypted into blocks. This is used to make sure the message would fit the right number of bits for the encryption (which is less than n)

Dynamically allocate memory for an array using malloc because it does not matter if each element is initialized to zero // the encrypted message would be inputted into the new array
Set the first byte of the block to 0xFF

```
While there are more bytes:
    Read k-1 bytes from the infile and input them into the allocated block + 1
    Convert the bytes to mpz_t
    Encrypt the message using rsa_encrypt()
    Send the encrypted message to outfile
```

void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n)
Performs RSA decryption

Set m to c to the power of d (mod n)

void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d)
Decrypts the contents of infile, writing the decrypted contents to outfile.

Set k to the floor of ($\log_2(n)$-1)/8 // k is the block size

Dynamically allocate memory for an array using malloc because it does not matter if each element is initialized to zero

```
While there are more bytes:
    Read hexstring from the file and save it in c
    Store converted c bytes in the allocated array
    Set j to the number of converted bytes
    Write on outfile j-1 bytes starting with location one of the array
```

void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n)

Conducts RSA signing

```
Set s to m to the power of d mod n
```

bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n)

Conducts RSA verification

```
Set t to s to the power of e mod n
If m is equal to t:
    Return true
Else:
    Return false
```

Makefile

Used to compile the files and clear all files that were compiled during run time. Can be adjusted to run specific files, but by default, it makes keygen, decrypt, and encrypt.