APP NAME: PUBLIC KEY CRYPTOGRAPHY

Design Specification

SOFIA PETROVA, UC Santa Cruz, CSE 13S – Fall 2021
Nov. 11. 2021

Document Version: 2.0
Application Version: 2.0

Accepted by:

Darrell Long

NAME Date

TITLE

NAME Date

TITLE

NAME Date

TITLE

Table of Contents

1. Amendment History

Date	Doc. Version	Amendment Description
11/11/2021	1.0	Initial draft, Sofia Petrova
11/18/2021	2.0	Final draft, Sofia Petrova

2. References

Information contained in this document was based on the following documents:

Assignment 6

Public Key Cryptography

Prof. Darrell Long

CSE 13S – Fall 2021

3. Acknowledgments

The information presented in this document was drawn from various discussions with the following individuals:

- Darrell Long

4. Included files

- decrypt.c- This contains the implementation and main() function for the decrypt program.
- encrypt.c- This contains the implementation and main() function for the encrypt program.
- keygen.c- This contains the implementation and main() function for the keygen program.
- numtheory.c- This contains the implementation of the number theory functions.
- numtheory.h- This specifies the interface for the number theory functions.
- randstate.c- This contains the implementation of the random state interface for the RSA library and number theory functions.
- randstate.h- This specifies the interface for initializing and clearing the random state
- rsa.c- This contains the implementation of the RSA library.
- rsa.h- This specifies the interface for the RSA library.
- README.md contains instructions for compiling and running the code
- DESIGN.pdf contains the design and design process for the program, describing it well enough that the program should be replicable to someone who has not seen the code. This is the current document
- Makefile compiles, formats, and runs the code, as well as cleaning output files

NOTE: Many file descriptions are taken directly from the Assignment 6 document written by Darrel Long for CSE13S Fall as they best describe each file.

5. Design Overview

5.1. Purpose of Document

This program contains key generator for RSA encryption, an encryptor, and a decryptor. The key generator finds two large random primes (using the GMP library to generate large enough numbers) and uses them to generate a public key- represented by integers n and e- and a private key- represented by integer d. The encryptor uses the public key to encrypt a file. The decryptor uses a private key to decrypt the file using the reverse process.

5.2. System Architecture

5.2.1. Overall Structure

This program contains three mini libraries to perform key generation, encryption, and decryption. These libraries are the RSA functions, the random state interface, and the number theory functions. The random state interface merely initializes and clears the random state used in the GMP library. The number theory functions build the mathematical functions needed in key generation and throughout the rest of the program. The RSA library contains the actual implementation for key generation (both public and private), encryption, and decryption.

The number theory library contains modular exponentiation and functions to make and determine prime numbers using the Miller Rabin Test. It also contains a function to calculate the greatest common denominator of two numbers and an inverse modular operation.

The RSA library, meanwhile, contains functions (using the number theoretical functions) to generate public and private keys as well as reading from and writing them into files. Key generation consists of generating two large primes (p and q), finding (p-1)*(q-1), and generating (e*d) such that (e*d) mod (p-1)*(q-1) =1. The public key is (e,p*q) The private key is (d,p*q). This module also contains the functions to encrypt and decrypt files..

The three main() functions in this program are the key generation, encryption, and decryption functions. Each main function takes arguments from the command line while running the executables, and depending on the arguments given by the user, the program will generate keys, encrypt, or decrypt files or messages.

keygen.c test harness arguments:

-h Display program help and usage.
-v Display verbose program output.
-b bits Minimum bits needed for public key n.
-c confidence Miller-Rabin iterations for testing primes (default: 50).
-n pbfile Public key file (default: rsa.pub).
-d pvfile Private key file (default: rsa.priv).
-s seed Random seed for testing.

encrypt.c test harness arguments:

-h Display program help and usage.
-v Display verbose program output.
-i infile Input file of data to encrypt (default: stdin).
-o outfile Output file for encrypted data (default: stdout).
-n pbfile Public key file (default: rsa.pub).

decrypt.c test harness arguments:

-h Display program help and usage.
-v Display verbose program output.
-i infile Input file of data to decrypt (default: stdin).
-o outfile Output file for decrypted data (default: stdout).
-d pvfile Private key file (default: rsa.priv).

5.2.2. ERROR HANDLING

- For inputting an invalid argument on the command line
  - Output the help screen normally outputted by -h
- For inputting an invalid file (failure to open an input or output file)
  - Default to stdin or stdout
- For memory allocation failures
  - Print error message and exit with error code 1
- For writing or reading text or files that are too small
  - Print error message and exit with error code 1

6. Pseudocode Implementation

6.1 Number Theoretic Functions

Miller-Rabin Prime Test(n, n-1, x, r)

The Miller–Rabin primality test is used to test if a number is prime or not. Given an integer n, choose some positive integer a < n. a will be the base. With s and d as positive integers and d being odd, n can be written as 2^s * d + 1; or 2^s * d = n-1. n is considered a possible prime to base a if a^d = 1 mod n OR a^(2^r)*d = -1 (mod n) for some value of r less than s.

```
Set mpz variables y and big N
Set y to user given x
Perform power mod with y,r,n and store into y

If y is 1 or n-1
     Clear variables
     Return true
For range of n-1
     Set big N to 2
     Compete power mod of big N and n
     If y = n-1
          Clear variables
          Return true
     Else
          Clear variables
```

```
            Return false
    Clear variables
    Return false
```

Greatest common denominator function(a, b)

The GCD function is used to find the greatest common denominator of two numbers, a and b.

```
    While b isn't 0
        Assign temp variable as b
        Assign b as a mod b
        Assign temp as a
        return a
```

Mod inverse function (result, number, modulo)

This function computes the inverse result of number % modulo and stores it into the result, with result, number, and modulo being mpz variables.

```
    Initialize mod_n mpz variable
    Set mod_n to modulo
    Initialize number mpz variable
    Set number to argument number
    Initialize t as mpz and set to 0
    Initialize q as mpz and set to 0
    Initialize r as mpz and set to 1
    Initialize s1 as mpz and set to 1
    Initialize s2 as mpz and set to 0
```

```
Initialize s3 as mpz and set to 1

Initialize t1 as mpz and set to 0

Initialize t2 as mpz and set to 1

Initialize t3 as mpz and set to 0

Initialize temporary variable



While r is greater than 0

      Set q to the floor of mod_n divided by number

      Multiply q and number and move into temporary variable

      Subtract temporary variable from mod_n

      Multiply q and s2 and store in temporary variable

      Subtract temporary variable from s1 and store into s3

      Multiply q and t2 and store into temporary variable

      Subtract t1 from temporary variable and store into t3

If r is greater than 1

      Set mod_n to number

      Set number to mod_n

      Set s1 to s2

      Set s2 to s3

      Set t1 to t2

      Set t2 to t3

Set t to t2



Store the absolute of number to temporary variable



If temporary variable equals 1
```

```
            Add t and modulo and store into result
            Perform result mod modulo and store into result
            Clear all temporary variables
      Return
```

Generate prime number(p, bits, iterations)

This function generates a large prime number using the Miller Rabbins test with the number of iterations given and the size of the bits given. A large amount of numbers are tested for primality, and when found is returned in the output, p.

```
      Initialize random number
      For i in 10,000 (large number but reasonable)
            Generate random number with bits specified
            If prime is found
                  Move found prime into p
                  Break loop and return
```

Check if number is prime(n, iterations)

This functions checks whether or not a given number is prime using the Miller Rabbins test with the amount of iterations given.

```
      If number is negative, 0, or 1
            Return false
      If number is one of first 100 primes
            Return true
      If number is even
            Return false
```

```
For i in iterations given
    Calculate x
    Perform MR test
    If prime isn't found with MR test
        break
```

Compute powder mod(o output, a base, d power, n modulo)

Computes A to the power of D modulo N and puts the output into O.

```
Set output to 1 initially
While d is greater than 0
    If d isn't even (Even makes a number automatically composite)
        v = v times p mod n
    Set p to p^2 mod n
    Set d to floor division of d/2
Set output to v
```

6.2 RSA Functions

Make public key

Creates public key to be used in RSA encryption. The public key is the pair(e, p*q).

```
Get seed
Generate random number in the range of nbits/4,(3×nbits)/4
Make prime to generate 2 prime numbers
```

Compute public mod n by multiplying p and q
Compute (p-1)*(q-1)
Generate (e*d) such that (e*d) mod (p-1)(q-1) = 1
Public key is (e,p*q)

Make private key

Creates private key to be used in RSA decryption. The private key is the mod inverse of public exponent and (p-1)(q-1).

Calculate (p-1)(q-1)
Mod inverse public exponent and (p-1)(q-1), store into private key

Sign(signature, plaintext, private key, public mod n)
Power mod function with plaintext, private key, public mod n. Store into signature

Write public key to file

This function writes the public key line by line to the input file specified by the user.

Output key parts as hex strings into file
Output new line into file
Output username into file
Output new line into file

Read public key

This function reads the private key and puts them into mpz variables.

Read public mod n and put into mpz variable
Read public exponent and put into mpz variable
Read signature and put into mpz variable
Read username and put into mpz variable

Write private key to file

This function writes the private key into the specified input file.

Output public mod n as hex string into file
Output private key as hex string into file
Output new line into file

Read private key

This function reads the private key and puts them into mpz variables.

Read public mod n and put into mpz variable
Read private key and put into mpz variable

RSA sign

This function encrypts the username with the private key to verify the identity of the sender.

Raise the username to the power of private key and modulo by n (power mod function)

RSA verify

This function decrypts the username with the public key to verify the identity of the sender and the message being received

Raise the username to the power of private key and modulo by n (power mod function)

RSA encrypt

This functions performs encryption on plaintext converted to an mpz.

Raise the message to the power of e and modulo by n (power mod function)

RSA decrypt

This functions performs decryption on plaintext converted to an mpz.

Raise message to the power of private key mod n (power mod function)

RSA encrypt file

This function encrypts InFile, writing the encrypted contents to the OutFile. The data in InFile is encrypted in blocks which are calculated based on the number of bits in the public key. The last block is taken care of separately as the number of bits of the last block may be shorter. GMP library functions are used.

```
Create big number variable plaintext;
Create big number variable cyphertext;
Assign N as public mod N


Calculate blocksize as blocksize = Floor[(LOG_2_N-1)/8];

Create block array of block size for cipher text
Set first index of cipher block array to 1
Create file size integer
Create byte buffer for plain text
While not end of file
      Increment file size
      Read lines from plain file
      Reallocate memory for plaintext buffer
```

Decrement filesize by 1 for indexing purposes (to not overwrite first index of cipher block array)

```
Calculate number of blocks in file
For number of blocks
      Pad plaintext

Convert read bytes to mpz_t using mpz_import function
```

Call RSA encrypt function using everything read and output into cipher text buffer

Print cipher text buffer into output file

Clear memory

RSA decrypt file

This function decrypts InFile, writing the decrypted contents to the OutFile. The data in InFile is decrypted in blocks which are calculated based on the number of bits in the private key. The last block is taken care of separately as the number of bits of the last block may be shorter. GMP library functions are used.

Assign private key to big number N;
Assign public exponent to big number PubExp;

Using mpz_get_d_2exp and N as operand:
Convert N to a double DI, truncating if necessary;
Return the exponent EXP separately as a signed integer;

Calculate LOG_2 of N, where LOG_2_N = LOG_2(N) = LOG_2(DI) + LOG_2(2) * EXP;

Calculate blocksize as blocksize = Floor[(LOG_2_N-1)/8];

Create array variable blockarray of blocksize size;

Create big number variable plaintext;
Create big number variable cyphertext;

Using gmp_fscanf, read InFile row by row until no more rows:

Scan InFile row and put result in cyphertext;
Use RSA Decrypt and N and PubExp to decrypt cyphertext and put result in plaintext;

```
            Use mpz_export to convert plaintext into  blockarray;
            Use mpz_export to get blocksize of blockarray;
            Skip first element of  blockarray;
            Write remaining blockarray up to  blocksize -1 into
OutFile;

      Clear temporary structures;
```

6.3 Key Generation

This program generates a public key and a private key to be used in RSA encryption and decryption. The keys are outputted into files specified by the user on the command line. The user can also specify the seed, number of bits to generate the key, number of iterations for Miller-Rabins test, and whether or not to output verbose statistics.

```
Initialize booleans for command line options (help, verbose, public
key, private key, n bits, seed, iterations)

Set default values for command line

Initialize file path name strings

While there are still arguments on the command line
      Case h: bool help is true
      Case v: bool verbose is true
      Case n: bool private key is true, take command line option
      Case d: bool public key is true, take command line option
      Case b: bool bits is true, take command line option
      Case s: bool seed is true, take command line option
      Case i: bool iterations is true, take command line option
```

```
If bool help is true
      Print help screen
      Free memory
      Exit program

Depending on if the user types out full file path, prepend file path
onto user argument

Initialize input, output, prime p, prime q, public exponent e, public
mod n, private key d

For 1000 iterations
      Make public and private keys
      If it's the right number of bits
            Break from for loop

Open public and private key files
Write public and private keys into files

If boolean verbose
      Print statistics

Close files
Free memory
```

6.4 Encryption

This program utilizes the function written to encrypt an input file and output the decrypted crypttext into an output file given by the user on the command line. The user can also use the command line to trigger verbose statistics.

Initialize booleans for command line options (help, verbose, public key, input file, output file)

Set default values for command line

Initialize file path name strings

While there are still arguments on the command line

    Case h: bool help is true

    Case v: bool verbose is true

    Case n: bool public key is true, take command line option

    Case i: bool input is true, take command line option

    Case o: bool output is true, take command line option

If bool help is true

    Print help screen

    Free memory

    Exit program

Depending on if the user types out full file path, prepend file path onto user argument

Open public key file

Open input and output files

Read and store public key

Verify signature

Encrypt file (using RSA encrypt function)

If boolean verbose is true

    Print statistics

Close files and free memory

6.5 Decryption

This program utilizes the function written to decrypt an input file and output the decrypted plaintext into an output file given by the user on the command line. The user can also use the command line to trigger verbose statistics.

Initialize booleans for command line options (help, verbose, private key, input file, output file)

Set default values for command line

Initialize file path name strings

While there are still arguments on the command line

    Case h: bool help is true

    Case v: bool verbose is true

    Case n: bool private key is true, take command line option

```
      Case i: bool input is true, take command line option
      Case o: bool output is true, take command line option

If bool help is true
      Print help screen
      Free memory
      Exit program

Depending on if the user types out full file path, prepend file path
onto user argument

Open private key file
Open input and output files
Read private key
Decrypt file (with RSA decrypt function)

If boolean verbose is true
      Print statistics

Close files and free memory
```