Public Key Cryptography - CSE13S Assignment 6

Nicholas Reis

November 9, 2021

Program Description

In this assignment, I implemented three programs: one that generates RSA key pairs, one that encrypts an input file when given a public key, and one that decrpyts an input file that was encrypted with a public key when given the corresponding private key. This required me to create functions in other files that provide specific algorithms and equations for things such as finding the greatest common divisor, modular inverse, checking if a number is prime, and so on. Many of the functions I wrote for this assignment utilized the GNU GMP library in their implementation.

Required Files

- encrypt.c
- decrypt.c
- keygen.c
- numtheory.h
- numtheory.c
- randstate.h
- randstate.c
- rsa.h
- rsa.c
- Makefile
- README.md
- DESIGN.pdf

Number Theory

Within numtheory.c, I wrote the implementation for functions that provided useful algorithms for this assignment. These include: finding the greatest common divisor, finding the modular inverse, performing modular exponentiation, checking if a number is prime, and generating a prime number. Pseudocode for most of these functions was provided in the assignment 6 pdf.

Pseudocode

```
include necessary header files

void gcd(d, a, b):
    while (b is not 0):
        temp = b
        b = a mod b
```

```
        a = temp
    d = a

void mod_inverse(o, a, n):
    r = n
    r' = a
    t = 0
    t' = 1
    while (r' is not 0):
        q = floor(r / r')
        temp = r
        r = r'
        r' = temp - q * r'

        temp = t
        t = t'
        t' = temp - q * t'
    if (r > 1):
        o = 0
        return;
    if (t < 0):
        t = t + n
        o = t

void pow_mod(o, a, d, n):
    v = 1
    p = a
    while (d > 0):
        if (d is odd):
            v = (v * p) mod n
        p = (p * p) mod n
        d = d / 2
    o = v

bool is_prime(n, iters):
    if (n is 2 or 3):
        return true
    if (n <= 1 or n is even):
        return false
    r = n - 1
    s = 0
    while (r is even):
        floor divide r by 2
        increment s by 1
    for (i = 1 to k):
        choose random a between 2 and n-2
        pow_mod(y, a, r, n)
        if (y is not 1 or n-1):
            j = 1
            while (j < s and y is not n-1):
                pow_mod(y, y, 2, n)
                if (y == 1):
                    return false
                increment j by 1
```

```
            if (y is not n-1):
                return false
    return true

void make_prime(p, bits, iters):
    offset = 2 ^ bits - 1
    do:
        mpz_urandomb(p, state, bits - 1)
        p = p + offset
    while (p is not prime)
```

---

1. The gcd() function computes the greatest common divisor of a and b and returns the value in d. This is the implementation of the Euclidean algorithm, which states that the gcd of two numbers will stay the same if the difference replaces the large number with the small number. The while loop will make it so the numbers will become smaller until it reaches the gcd.
2. The mod_inverse() function computes modular inverses, using concepts from the extended Euclidean algorithm and Bézout's identity.
3. The pow_mod() function performs modular exponentiation, which computes in only O(log(n)) steps.
4. The is_prime() function implements the Miller-Rabin primality test, which is a strong pseudoprime test that does not guarantee primality but means it is probably prime. We use this test because it still has a high probability and is much faster than using a deterministic primality algorithm.
5. The make_prime() function generates random numbers that are a specified number of bits long (minimum), and continues to loop until that generated number is prime (checking with my is_prime() function).

Random State

This assignment requires us to use GMP, which means in order to use the random integer functions, we must initialize a random state variable. randstate.c contains the implementation for these functions that initialize and clear the global random state variable we create.

Pseudocode

```
include neccesary header files

gmp_randstate_t state

void randstate_init(seed):
    gmp_randinit_mt(state)
    gmp_randseed_ui(state, seed)

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

---

1. The randstate_init() function initializes 'state' using a Mersenne Twister algorithm, and the 'seed' argument is used for the random seed.
2. The randstate_clear() function clears and frees any memory used by 'state'.

RSA

The rsa.c file implements a variety of functions that are used to create, read, and write public and private keys, along with encrypting and decrypting messages and files.

Pseudocode

```
include necessary header files

gmp_randstate_t state

uint32_t log_base2(n):
    k = 0
    n = absolute value of n
    while (n > 0):
        floor divide n by 2
        increment k
    return k

void rsa_make_pub(p, q, n, e, nbits, iters):
    random number of bits between [nbits/4 -> 3*nbits/4] for p
    rest of bits for q

    make_prime() p and q

    mpz_mul() such that n = p * q

    mpz_mul() such that totient = (p-1)*(q-1)

    do:
        mpz_urandomb() to get a random number rand
        gcd(divisor, totient, rand)
    while: (divisor != 1)

    set e = rand
    mpz_clears(rand, divisor)

void rsa_write_pub(n, e, s, username[], pbfile):
    gmp_fprintf() such that n, e, and s are hexstrings and everything is separated by newlines

void rsa_read_pub(n, e, s, username[], pbfile):
    gmp_fscanf() assuming that n, e, and s are hexstrings and everything separated by newlines

void rsa_make_priv(d, e, p, q):
    create mpz_t totient, p_minus1, q_minus1
    totient = (p_minus1)*(q_minus1)

    mod_inverse(d, e, totient)

    mpz_clears(totient, p_minus1, q_minus1)

void rsa_write_priv(n, d, pvfile):
    gmp_fprintf() such that n and d are hexstrings and everything is separated by newlines

void rsa_read_priv(n, d, pvfile):
```

```
    gmp_fscanf() assuming that n and d are hexstrings and everything separated by newlines

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

void rsa_encrypt_file(infile, outfile, n, e):
    k = (log_base2(n) - 1) / 8

    calloc a uint8_t *block
    block[0] = 1

    do:
        fread() k-1 bytes from infile
        mpz_import() to convert the read bytes to mpz_t
        rsa_encrypt() the read bytes
        if (bytes read is more than 0):
            gmp_fprintf() the message to outfile
    while (bytes read is more than 0)
    free(block)

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

void rsa_decrypt_file(infile, outfile, n, d):
    k = (log_base2(n) - 1) / 8

    calloc a uint8_t *block

    do:
        gmp_fscanf() the message from infile
        rsa_decrypt() the scanned bytes
        mpz_export() to convert the read mpz_t to bytes
        fwrite() bytes to outfile
    while (feof(infile) is false)
    free(block)

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

bool rsa_verify(m, s, e, n):
    initialize mpz_t t
    pow_mod(t, s, e, n)

    if (t and m are the same):
        return true
    else:
        return false
```

---

1. rsa_make_pub() makes a public key when called. This is done by first making random large primes p and q, before calculating n = (p-1)*(q-1), and then generating random numbers until one of them is coprime with n (which happens once their greatest common denominator is 1). Set e = random number and that is the created public key.

2. rsa_write_pub() prints out a public key to an output FILE *pbfile with the mpz_t variables printed as hexstrings and each argument separated by a newline.
3. rsa_read_pub() reads a public key from an input FILE *pbfile with the mpz_t variables printed as hexstrings and each argument separated by a newline.
4. rsa_make_priv() makes a private key when called. This is done by calculating the modular inverse of e modulo n and storing the result in d.
5. rsa_write_priv() prints out a private key to an output FILE *pvfile with the mpz_t variables printed as hexstrings and each argument separated by a newline.
6. rsa_read_priv() reads a private key from an input FILE *pvfile with the mpz_t variables printed as hexstrings and each argument separated by a newline.
7. rsa_encrypt, decrypt, and sign only require a call to pow_mod() to calculate the modular exponentiation of the provided variables.
8. rsa_encrypt and decrypt file perform inverse processes on the given files. rsa_encrypt_file reads in bytes from infile, converts them to mpz_t, encrypts the mpz_t's, and then prints the encrypted messages to outfile. rsa_decrypt_file scans hexstrings from infile, decrypts them, converts the mpz_t to bytes, and then writes the bytes to outfile.
9. rsa_verify returns true if the pow_mod() of the given variables outputs a message that is the same as the expected message m.

Keygen

A public key and private key pair are necessary for this program, and the implementation for creating this pair is in keygen.c.

Pseudocode

```
include necessary header files
define OPTIONS "b:i:n:d:s:vh"

void print_usage():
    helper function I made to print out help and usage information to stderr

void print_verbose():
    print out username, signature, prime p, prime q, modulus n, exponent e, and private key d

main:
    set the default public key file and private key file to be rsa.pub and rsa.priv
    set the default seed to be time(NULL)
    create booleans for help and verbose

    while loop that uses getopt to access the options:
    switch:
    case 'h' '?' and default:
        set help boolean to true
    case 'v':
        set verbose boolean to true
    case 'n':
        open the given public key file
    case 'd':
```

```
    open the given private key file
    use fchmod() to set file permissions to 0600
case 'b':
    set number of bits to optarg
case 'i':
    set number of iters to optarg
case 's':
    set seed to optarg

if (help):
    print_help()

randstate_init(seed)

initialize mpz_t variables p, q, n, e, d
rsa_make_pub() using these variables
rsa_make_priv() using these variables

use getenv() to get username

convert username to mpz_t with mpz_set_str base 62
rsa_sign() with the mpz_t username

rsa_write_pub(..., pbfile)
rsa_write_priv(..., pvfile)

if (verbose):
    print_verbose()

fclose() pbfile and pvfile

randstate_clear()

clear mpz_t variables
```

---

1. The default public and private key files are set to rsa.pub and rsa.priv, however if the user enables options 'n' or 'd' followed by a file name, they will be created there instead. The private key file's permissions are set to 0600 so only the user has read an write permissions.
2. mpz_t variables are used in rsa_make_pub/priv(), so the program initializes them beforehand and pass them as arguments to both functions. nbits and iters are also used in rsa_make_pub.
3. getenv() provides functionality to get the current user's name, and then it is converted to mpz_t so the signature can be computed.
4. rsa_write_pub/priv() writes each of them to their respective files, with the public key being signed by the user.
5. After use, the files need to be closed, the randstate needs to be cleared, and the mpz_t variables need to be cleared.

Encrypt

The code that performs the encryption process is is encrypt.c. It utilizes the public key generated from keygen.c to encrypt a given input file and write the encrypted file to a given output file.

Pseudocode

```
include necessary header files
define OPTIONS "i:o:n:vh"

void print_usage():
    helper function I made to print out help and usage information to stderr

void print_verbose():
    print out username, signature, modulus n, and exponent e

main:
    set the default infile and outfile to be stdin and stdout
    set the default public key file to be rsa.pub
    create booleans for help, verbose, input file, and output file

    while loop that uses getopt to access the options:
    switch:
    case 'h' '?' and default:
        set help boolean to true
    case 'v':
        set verbose boolean to true
    case 'i':
        input file = true
        use optarg to get infile name
    case 'o':
        output file = true
        use optarg to get outfile name
    case 'n':
        use optarg to get public key file name

    if (help):
        print_help()

    if (input file):
        open the input file
    if (output file):
        open the output file
    open the public key file

    initialize mpz_t variables n, e, s, user
    create a buffer for username
    rsa_read_pub() using these variables

    if (verbose):
        print_verbose()

    mpz_init_set_str() to convert username buffer to mpz_t
    if (rsa_verify() is false):
        print an error message
```

```
        exit the program

    rsa_encrypt_file()

    fclose() the files
    clear mpz_t variables
```

---

1. print_verbose() prints out the stated variables - username, s, n, and e - with the latter three in hexstring format along with the number of bits that constitutes them.
2. I chose to open the files outside of the switch case, so I use char * to store the value of the file names (that are in optarg). The input file and public key file only require read permissions, while the output file needs write permissions.
3. A simple call to rsa_read_pub() is all that is needed to get the information in the public key, so long as the mpz_t variables n, e, and s, along with a username buffer are created before the function call.
4. mpz_init_set_str() with a base of 62 is used to convert the username buffer to an mpz_t, which is then used in rsa_verify() to verify the signature of the public key.
5. rsa_encrypt_file() takes in the input file and output file along with n and e as arguments. It will encrypt the input file and write the encrypted output to the output file.

Decrypt

The code that performs the decryption process is is decrypt.c. It utilizes the private key generated from keygen.c to decrypt an encrypted input file and write the decrypted file to a given output file.

Pseudocode

```
include necessary header files
define OPTIONS "i:o:n:vh"

void print_usage():
    helper function I made to print out help and usage information to stderr

void print_verbose():
    print out modulus n and private key d

main:
    set the default infile and outfile to be stdin and stdout
    set the default private key file to be rsa.priv
    create booleans for help, verbose, input file, and output file

    while loop that uses getopt to access the options:
    switch:
    case 'h' '?' and default:
        set help boolean to true
    case 'v':
        set verbose boolean to true
    case 'i':
        input file = true
        use optarg to get infile name
    case 'o':
```

```
    output file = true
    use optarg to get outfile name
case 'n':
    use optarg to get private key file name

if (help):
    print_help()

if (input file):
    open the input file
if (output file):
    open the output file
open the private key file

initialize mpz_t variables n and d
rsa_read_priv() using these variables

if (verbose):
    print_verbose()

rsa_decrypt_file()

fclose() the files
clear mpz_t variables
```

---

1. print_verbose() prints out the stated variables - n and d - in hexstring format along with the number of bits that constitutes them.
2. I chose to open the files outside of the switch case, so I use char * to store the value of the file names (that are in optarg). The input file and private key file only require read permissions, while the output file needs write permissions.
3. A simple call to rsa_read_priv() is all that is needed to get the information in the private key, so long as the mpz_t variables n and d are created before the function call.
4. rsa_decrypt_file() takes in the input file and output file along with n and d as arguments. It will decrypt the encrypted input file and write the decrypted output to the output file.
