RSA Encryption and Decryption Design

Moore Macauley
University of California, Santa Cruz

November 4, 2022

1 Introduction

For this assignment, I was tasked with the reimplementation of the RSA Encryption Algorithm, including the generation of public and private key pairs, and encryption and decryption using those pairs. The RSA Algorithm works off a fairly simple idea. Every person has a unique public and private key, where the public key is known by everyone, and the private key only by the person who owns it. A message encrypted with the public key can be decrypted with the private key, and a message encrypted with the private key can be decrypted with the public key. So, person a wishing to send person b a message would first encrypt their message and a signature with b's public key, and then encrypt a copy of the signature with their private key. When person b receives the message and the signature, they can decrypt both with their private key, and are the only ones capable of doing so. They then decrypt the signature encrypted with a's private key with a's public key, and if it matches the signature that they decrypted earlier, they know that person a sent the message, as only person a is capable of creating a signature that could be decrypted by their private key.

In order to do this, I will create a number of libraries. The first, randstate.h, will simply be used to provide random numbers for later functions. The second, numtheory.h, will contain the functions that do all of the complicated math required by RSA encryption. rsa.h will contain the actual implementation of RSA encryption, dealing with the creation of seeds, encryption and decryption of files, ect. Finally, I will create three c programs, encrypt.c, decrypt.c, and keygen.c, which will parse command line arguments and use the rsa.h library to perform RSA encryption and decryption on various files.

2 randstate.h

This one will be fairly simple. It will consist of two functions and a global variable. The global variable will be called state, and will be a new type added by GMP, gmp_randstate_t. The first function, randstate_init will simply accept a seed, use the function gmp_randinit_mt to initialize state for it to use a Mersenne Twister algorithm, and then call gmp_randseed_ui to set state to the passed seed. It then calls srandom(seed) to set the seed for the standard random function as well. The second function, randstate_clear, will simply use gmp_randclear to free all memory allocated for state.

```
Pseudocode:

create the global variable state;
```

```
randstate_init(unsigned 64 bit int seed):
    Initialize state
    Set the seed of state to seed
    Set the seed of the library random function to seed

randstate_clear()
    gmp_randclear(state)
```

3 numtheory.h

3.1 Power Modulo

The function pow_mod will accept four integers, with the first being a return pointer, the second the base b, the exponent e, and the modulus m. The program will do b^e mod m, and place the resulting value into the return pointer. To do this efficiently, we remember that any integer can be represented by a polynomial via the formula

$$n = c_m 2^m + c_{m-1} 2^{m-1} + ... + c_1 2^1 + c_0 2^0 = \sum_{i=0}^{m} c_i 2^i$$

If we apply that formula to e in b^e and then simplify it, we find that we can represent any b^e simply by squaring b repeatedly. While we could simply calculate an exponent then find the mod of the result, exponents get very big very fast, so we apply mod on intermediary steps to ensure the end result stays relatively small.

Pseudocode:

```
pow_mod(int out, int base, int exponent, int modulus);
    out = 1;
    While exponent is greater than 0
        If exponent is odd
            out = out * base mod modulus
        base = base * base mod modulus
        exponent = exponent/2
```

3.2 Finding Primes

Unfortunately, a substantial part of RSA encryption involves large prime numbers. As such, we will need a fast way of finding them. In order to do this, I will create a function that implements the Miller-Rabin primality test. The test works very simply, when given the value n, it chooses a random number a, where a is less than n. It then defines $2^s d = n$, where d must be odd. If $a^d -/+ 1$ and $a^{2^r d} -/+ 1$ cannot be divided by n for all values of r between 0 and s-1, then n is not prime, and a is a witness of that fact. Otherwise, n might be prime, with us being wrong roughly a quarter of the time. However, we can simply choose

a different a value and try again, with the odds of the value not being prime decreasing to 2^{-200} by the time 100 trials have been performed. As such, while there is a minute chance of this value not being a real prime number, the chance is so small it is barely worth considering.

Using this principle, my function will accept two arguments, the value to be checked n, and the number of iterations that I should do, k. It will start by dividing n by 2 until n is odd, incrementing s every time this is necessary, then setting n to d. After this, we perform the above steps to n k times, returning false if one of the above statements occurs, and true if it does not.

```
Pseudocode:

is_prime(int n, int iterations)

Create int s and r, set s to 0, and r to n

While r is even, determined with modulus
    Divide r by 2 and increment s

For int i between 0 and k
    Create the int a, set it to a random number between 2 and n-2 using
    the gnu implementation of the Mersenne Twister
    Set int y to a^r % n by running pow_mod
    If y is not equal to 1 or n-1
        Set the int j to 1
        While j is less than or equal to s-1 and y is not equal to n-1
            Set y to y^2 % n by running pow_mod
            If y == 1
                Return False
            j += 1
        if y is not equal to n - 1
            Return False

Return True
```

Then, using that, I will write a second function, make prime, which will generate a new prime number by testing its prime-ness with is_prime, given the number of bits long (nbits) it should be and the number of iterations (iters) that should be done by the above function. To do this, I will simply generate a random number that is at least nbits long and make it odd if it is not. I will then continuously check to see if it is prime with the above function, and if it is not, then I will add two to it. Otherwise, I return the prime value.

```
Pseudocode:

make_prime(int p, int bits, int iters)
```

```
Left shift a 1 bits bits over to the left to determine the
value represented by the bit at bits, save that as min.

Randomize the remaining bits in min by creating a random number
with mpz_urandomb using bits, then adding that to min, before
saving that value as p.

If p is even, add 1 to it

While p is not prime
    Add 2 to p
```

3.3 Modular Inverses

We will also need to find the greatest common divisor of two numbers. To do this, we can simply use the provided Euclid's Algorithm to do so.

```
Pseudocode:

gcd(int out, int a, int b)

while b is not equal to 0
    set temp to b
    set b to a mod b
    set a to temp
```

Using this algorithm, we can then go on to calculate the inverse modulo of two numbers

```
Pseudocode:

mod_inverse(int out, int a, int n)

int r = n, int r prime = a;
int t = 0, int t prime = 1;
While r prime is not equal to 0
    int q = r/r prime
    int temp r = r
    Set r to r prime
    Set r prime to temp r - q * r prime

    int temp t = t
```

```
    Set temp t to t prime
    Set t prime to temp t - q * t prime

if r is greater than 1
    Set out to 0
if t is less than 1
    Add n to t

Set out to t
```

4 rsa.h

4.1 Public Key Making

To begin with, to do RSA encryption properly, we need to be able to generate a public key. To do so, we create a function that, given the number of bits the public key should be and then number of Miller-Rabin iterations that should be used for prime checking, creates the two primes p and q, their product n, and the public exponent e. To do this, we first generate p and q with make_prime as it was defined above, distributing the number bits used for p and q randomly. We then compute $\lambda(n) = \lambda(p \times q)$, which because p and q are prime can be expanded to

$$\lambda(n) = \frac{|(p-1) \times (q-1)|}{gcd(p-1, q-1)}$$

We then choose a random value between 0 and $2^{numberofbits} - 1$ and check to see if the gcd of that random number and $\lambda(n)$ is 1. If it is not, we continue the loop until we find a value that is.

```
Pseudocode:

Create a random number between the number of bits/4 and (the number of bits * 3) / 4
using random between the number of bits/2, then adding the number of bits/4

Save the above value into p bits, and save q bits as the number of bits - p bits
Make the primes p and q using p bits and q bits respectively with my make prime

Divide p-1 * q-1 by the gcd of p-1 and q-1, and set that to lam_n
Set the variable e to a random number generated around the number of bits with
mpz_urandomb
While the gcd of e and lam_n is not 1
    Generate a new e around the number of bits with mpz_urandomb

Return p, q, e, and p*q
```

We will also need to be able to write these values to a file. This should be fairly simple with the function gmp_fprintf, which will print a formatted string to a file.

Similarly, we need to be able to read these values from a file, and for this, we will use the corresponding function gmp_fscanf.

4.2 Private Key Making

Similarly, we need to be able to generate a private key, given the primes used to generate the public key and the public exponent. To do this, we follow the steps detailed above to calculate $\lambda(n)$, then use the inverse modulo function created above to calculate the inverse of e mod $\lambda(n)$.

```
Pseudocode:

Divide p-1 * q-1 by the gcd of p-1 and q-1 (calculated with my gcd function), and set
that to lam_n
Set d to the inverse e modulo lam_n using my inverse mod function

Return d
```

4.3 Encrypting

To encrypt files, we will create two functions. One will convert blocks of text into a number, and the second will then encrypt that number. The first function will calculate the size of the blocks that will be encrypted by taking $log_2(n)/8$, and save this as k. It will then create an array that can hold k characters, and take k-1 characters from the file to be encrypted and save them in that array. The 0th element of the array will be set to 0xFF, to prevent errors in our encryption method when the whole block is equal to 0 or to 1. We then convert the whole array to a number, run that int through the second method, defined below, before saving the int as a hex string in a file with a trailing newline.

```
Pseudocode:

k = log_2(n)/8
Create the array arr with length k using calloc and size of char
Read at most k-1 bytes into arr, starting at arr[0] with fread, save the number of
bytes actually saved into num

While num equals k-1
    Set arr[0] to 0xFF
    Convert arr to the int, m
    Encrypt m
    Write m'\n' to the outfile with gmp_fprintf
    Read at most k-1 bytes into arr, starting at arr[0] with fread, save the
    number of bytes actually saved into num
```

The second function, used to encrypt the message m, will be very simple. All it does is convert m to an int, then use pow_mod to calculate m^e mod(n), where e is the public exponent, and n is the product of the large primes.

4.4 Decrypting

Decryption will simply be encryption in reverse. We start by calculating k the same way, and making an array arr of length k. We get the encrypted message (saved in c) from the file by reading characters until we encounter a trailing newline. We then run the decrypt function on c, before converting c back into an array of characters saved in arr. We then iterate over arr, writing every character except the 0th character (which contains the 0xFF) to the outfile.

```
Pseudocode:

k = log_2(n)/8
Create the array arr with length k
Create the int c
Create the int converted

While there are unprocessed characters in the file
    Using gmp_fscanf, read the next hex string into c.
    Decrypt c with the rsa_decrypt, defined below
    Convert c into an array, storing the result in r, and the number of characters
    converted into converted.
    Iterate over the characters that were converted, writing each one to the
    output file
```

The second function, used to decrypt the message c, will be very simple. All it does is convert c to an int, then use pow_mod to calculate c^d mod(n), where d is the private exponent, and n is the product of the large primes.

4.5 Signatures

To create a signature, all the function must be able to do is given the message m, the public modulus n, and the private key d, compute and return m^d mod(n) with my pow_mod function. To verify a signature, given the signature s, the expected message m, the public exponent e, and the public modulus n, we compute s^e mod(n) with my pow_mod function, and save that in a. If a and m are equal, we return true, otherwise return false.

5 Main Functions

5.1 Key Generator

The role of the key generator function is very simple. It generates a public and private key based on a few arguments passed from the command line, namely the number of bits needed for the public modulus n, then saves the keys into a file specified by the user. It should accept the arguments -b bits, which specifies the number of bits n should be, -i iterations (default 50), which is the number of iterations used for prime testing, -n file (default rsa.pub) specifies the location the public key should be saved to, -d file (default rsa.priv) specifies the file that the private key should be saved to, -s seed (default seconds since UNIX epoch) specifies the seed to be used by the various random number generators used, -v enables verbose output, and finally, -h displays a help message.

To do this, we will first create variables representing the default values used by the program. We then change these values based on the command line arguments. Following this, we open the files that the public and private key are to go into, and change the permissions on the private key file to 0600, removing the ability for anyone except the user to read and write from it. We then initialize the various random number generators, before calling rsa_make_pub() and rsa_make_priv() to make the public and private keys. After, we need to create a signature, using rsa_sign() to convert the user's username into a signature, before writing the public and private keys to their files. Finally, we print the results if verbose output was enabled, and then garbage collect by closing the files, clearing the random states, and freeing all the used data.

```
Pseudocode:

Create the variables for minimum bits, iterations, pub file name, priv file name,
the seed, verbose output, help, and error.

Process the command line inputs with getopt, changing the above variables based
on the user's inputs. Return 0 and print the help message if help is called,
return 1 and print the help message if one of the arguments is incorrect.

Open the files for public and private keys with fopen, returning 1 and printing
the help message if an error occurs

Set the permissions of the private key file to 600 with fchmod and fileno

Initialize random number generators with the function created above

Create the public and private keys with my private and public key functions

Get the username of the user, convert it to an integer base 62 with mpz_set_str,
and run it through the rsa sign function to make it a signature.

Use the rsa function to write the public and private keys to their files
```

```
If verbose is true
    Print out the username, the signature, the first large prime number, the
    second large prime number, the public mod n, the public exponent e, and
    the private key d.

Free everything, uninitialize the random number generators, and close the files
with free, my randstate_clear function above, and fclose respectively
```

5.2 Encryptor

The encryptor, shockingly, will encrypt a file based on arguments from the command line. It should accept the arguments, -i input file (default stdin), which is the file to encrypt, -o output file (default stdout), which is where the encrypted file is saved, -n public key file (default rsa.pub), which is the file where the public key is stored, -v to enable verbose output, and finally -h to display a help message.

To do this, same as the key generator, we create variables representing the default values before the command line arguments are processed. We then process these arguments, overwriting any defaults that appear, and printing the help message and returning 0 if the help message was called. After this, we open the public key file and read it, getting the user's username, the signature s, the public modulus n, and the public exponent e, all of which we print if verbose output is enabled. Convert the username to an int, and use rsa_verify() with that and the retrieved username to verify the signature. Print the help message and return 1 if it is not verified. Finally, we encrypt the file using rsa_encrypt_file(), before freeing everything and closing the files.

```
Pseudocode:

Create the variables for pub file name, input file, output file, verbose output,
help, and error.

Process the command line inputs with getopt, changing the above variables based
on the user's inputs. Return 0 and print the help message if help is called,
return 1 and print the help message if one of the arguments is incorrect.

Open the file for the public key with fopen, returning 1 and printing the help
message if an error occurs

Read the public key, public exponent, signature, and username from the public
key to corresponding variables using the previously created rsa function

If verbose is true
    Print out the username, the signature, the public mod, and the public
    exponent

Convert the username to an int, then plug it and the signature in to rsa_verify(),
printing an error message and returning 1 if the result is false
```

```
Use rsa_encyrpt_file to encrypt the file and send the ciphertext to the output

Free everything, uninitialize the random number generators, and close the files
with free, my randstate_clear function above, and fclose respectively
```

5.3 Decryptor

Finally, decrypt will decrypt a file based on arguments from the command line. It should accept the arguments, -i input file (default stdin), which is the file to decrypt, -o output file (default stdout), which is where plaintext is saved, -n private key file (default rsa.priv), which is the file where the private key is stored, -v to enable verbose output, and finally -h to display a help message.

To do this, same as the other main functions, we create variables representing the default values before the command line arguments are processed. We then process these arguments, overwriting any defaults that appear, and printing the help message and returning 0 if the help message was called. After this, we open the private key file and read it, getting the public modulus n and the private ke e, which we print if verbose output is enabled. We then decrypt the file using rsa_decrypt_file(), before freeing everything and closing the files.

```
Pseudocode:

Create the variables for priv file name, input file, output file, verbose output,
help, and error.

Process the command line inputs with getopt, changing the above variables based
on the user's inputs. Return 0 and print the help message if help is called,
return 1 and print the help message if one of the arguments is incorrect.

Open the file for the private key with fopen, returning 1 and printing the help
message if an error occurs

Read the private key, public modulus from the public key to corresponding
variables using the previously created rsa function

If verbose is true
    Print out the private key and public modulus

Use rsa_decrypt_file to decrypt the file and send the plaintext to the output

Free everything, uninitialize the random number generators, and close the files
with free, my randstate_clear function above, and fclose respectively
```