Assignment 5 DESIGN.pdf

Victor Nguyen

February 17, 2022

1 Description of Programs:

In this assignment we are tasked to create three programs: an encryption program, decryption program, and the key generator program.

2 Files to be included in directory "asgn5":

1. decrypt.c
    - Source file that contains the implementation and main function for the decryption.
2. encrypt.c
    - Source file that contains the implementation and main function for the encryption.
3. keygen.c
    - Source file that contains the implementation and main function for the key generator.
4. numtheory.c
    - Source file that contains the implementations for the number theories.
    - Will be used in rsa.c.
5. numtheory.h
    - Header file thats used to link with other files if necessary.
6. randstate.c
    - Source file that contains the implementations for creating the random state for the RSA library and the number theory functions.
7. randstate.h
    - Header file thats used to link with other files if necessary.
8. rsa.c
    - Source file that contains the implementations of the RSA library.

9. rsa.h
    - Header file thats used to link with other files if necessary.
10. Makefile
    - File that formats the program into clang-format.
    - File also compiles the above .c programs.
    - Cleans files that were generated during the compilation process.
    - File also contains scan-build and valgrind to check for any errors/memory leaks.
11. README.md
    - A text file that's in Markdown format that describes how you would build/run the program.
12. DESIGN.pdf
    - Describes the design and thought process of the main program.

3 Structure and Pseudocode for randstate.c:

```
Declare a global variable.

Create a function to initialize a random number.
    Initialize state.
    Set the seed for the random number generator for both gmp and srandom.

Clear the random number using a function.
    Free up the heap space for the state variable.
```

3.1 Note about the pseudocode:

- We need to use the gmp library to generate numbers bigger than at least 64 bits. In c, we can't do that.

4 Structure and Pseudocode for numtheory.c:

Power-Mod Function

```
Create a function that computes the power mod, parameters that it should take
in is the base (a), the exponent (d) , and the modulus value (n).
    Initialize variable v to be equal to 1.
    Initialize variable p to be equal to the a.
    While the d is greater than 0:
        Check if d is an odd number:
            If it is, set v to be equal to v times p modulus n.
        Set p to be equal to p squared modulus n.
```

```
        Divide d by 2 and set that to be the new d.
    Return v after the while loop terminates.
```

Prime Identifier Function

```
Create a function that checks whether or not a value is a prime or not.
Function takes in a value n, and the number of iterations k, to perform.
    I need to check the previous value of n and write it in a way such that
    it equals (2^s)r, where r is odd.
    Starting at 1, iterate to k:
        Store a random number in some variable a.
        Perform a power-mod using the random number a, odd number r, and n.
        Store the value obtained from the power-mod in a variable y.
        Check if y is not equal to 1 and y is not equal to the previous n val.
            If so, set a variable j to be equal to 1.
            While j is less than or equal to the previous s value and that y
            is not equal to the previous n.
                Perform a power-mod using y, 2, and n, store the value into y.
                Check if y is equal to 1.
                    Return false.
                Add 1 to j and store it in j.
            Check if y is not equal to the previous n value.
                Return false.
    Return true.
```

Generate a prime number

```
Create a function that makes a prime number and store it in some variable p.
The function will take in an output p, number of bits minimal for the prime
number (bits), and the number of iterations.
    Generate a random number using the randstate function. Then pass the
    number into our prime identifier function to test if it's a prime.
    If the number is prime, we return the value.
    Keep generating numbers until we get a prime number.
```

Gcd function

```
Create a function that finds the gcd of two numbers.
The function should take in two numbers, a and b and store the result in a
variable d.
    Initialize a temporary variable t.
    While b is not equal to zero:
        Let t be equal to b.
        Do a mod b and store the value in b.
        Let a be equal to t.
    Return d after storing the a value inside.
```

Modulus Inverse function

```
Create a function the calculates the inverse of a modulus function.
This function should take in a value a and the modulus n.
    Create variables r, r', t, t', and q.
    Set r to be n, r' to a, t to 0, and t' to 1.
    While r' is not equal to 0:
        Set q to be equal to r divided by r'.
        Perform a swap on r and r'.
        The value of r should be r', and r' to be r minus q times r'.
        Perform a swap on t and t'.
        The val of t should be t' and t' to be t minus q times t'.
    Check to see if r is greater than 1:
        Return the value 1 to signify that no modular inverse was found.
    Check to see if t is less than 0:
        Set t to be equal to t + n;
    Return t.
```

4.1 Note about the pseudocode:

- I'm primarily following the provided pseudocode in the asgn5 pdf.
- In the gcd function, we are performing a type of swap between the variables a and b. We need to create a temporary variable to do so.
- The modulus inverse function uses a lot of parallel assignment, temporary variables will be necessary.
- In the is prime function, note that no even numbers are ever prime. Check for the smaller numbers to such as 0, 1, 2, and 3. Write conditions accordingly for them before we even attempt to waste any time on these numbers.
- Our make prime function should also check to see if $\log_2(n)$ >= nbits.
- Remember to use temporary variables to hold some values that were stored in gmp types. This is because gmp variables are call by references. Use temporary variables where ever necessary.

5 Structure and Pseudocode for rsa.c

Compute the least common multiple

```
Create a function that finds the lcm of two numbers.
Function should take in two numbers a and b.
    Compute the lcm of a and b.
    Store the value into another variable and return it.
```

Make a public rsa key

```
Create a function that makes a rsa public key.
    Use the prime function generator to create two large prime numbers.
```

```
    With the two prime numbers, compute the least common divisor of p -1 and
    q - 1.

    Generate random numbers around the number of bits until we found a
    suitable exponent.
        When a coprime is found, use that as the public exponent and return
        that value.
```

Write the public key into a file

```
Make a function that writes the public rsa key into some file.
    The format of the file should have the n value be first, e second,
    s third, and the username.
```

Read the rsa public key

```
Given a file, read the file that contains the rsa public key. The order if
which the file reads each line is that it should expect n first, e second,
s third, and username last.
```

Make a private rsa key

```
Create a function that creates a private rsa key using two prime numbers,
and a public exponent e.
    The private key should be generated via performing an inverse modulus on
    the least common divisor of the two prime numbers that were generated from
    the public key generator.
```

Write the private key into a file

```
Similar to writing a public key into a file, we do the same for the
private key.
```

Reading the private key from a file

```
Similar to reading the public key, we do the same to a private key.
```

Encrypting a message

```
Given a file, encrypt it using the the encryption formula given in the
assignment pdf. This is simple, as we just perform a power mod on the
message. Return this value.
```

Encrypting a file

```
Given a file, we will encrypt the file in blocks.
Create an array and allocate the 0th index to be the hex number 0xFF.
Calculate the block size and store it into k.
While there are still unprocessed bytes:
    Read in k - 1 bytes and store them in the array starting at the 1st index.
    Translate the bytes of data into mpz using mpz_import.
    Encrypt the message using the rsa_encrypt function we made earlier.
    Put the encrypted message into the outfile.
```

Decrypting a message

```
Similar to encryption, we need to decrypt the message so we can read it again.
```

Decrypting a file

```
Similar to encrypting a file, we need to decrypt the file and store its
contents to another specified outfile.

Create an array.
Calculate the block size and store it into k.
While there are still unprocessed bytes:
    Scan in a line.
    Perform decryption on the message.
    Export the message and translate it into bytes and store onto the array.
    Write the array onto the outfile.
```

RSA Signing

```
Perform an rsa sign on some value. This can be done using a specific formula.
```

RSA Verification

```
Create a function that verifies the RSA signature.
    Use the formula to perform the verification on the arguments that are
    taken in.
    Return true if the signature was verified, false if otherwise.
```

5.1 Note about the pseudocode

- The lcm of two numbers can be found using the formula $\frac{|x*y|}{gcd(x,y)}$.
- The numbers in the public rsa key function should be generated randomly.
- The number of bits that should be allocated to the first prime number should fall into the range of nbits/4 (inclusive) to (3*nbits)/4 (non inclusive). The second prime number should be allocated the rest of the number of bits.
- The lcd will require the gcd in order to properly work.
- When writing the public key, we should add trailing new lines for each line.
- The private key file should only be accessible and writable by the specified users only.
- Encrypt the file in blocks, since we specified that some modulo n.
- We need to prepend a single byte to the front of the array so we can avoid problems if the block were ever to be the value 0 or 1.
- The block size can be calculated via $\lfloor(\log_2(n)-1)/8\rfloor$

- Make sure to only write k-1 bytes for rsa_decrypt_file to be mindful of the 0xFF that was appended to the beginning of the block.
- The RSA sign formula is $s = m^d \, mod \, n$
- The RSA verification formula is $t = s^e \, mod \, n$

6 Pseudocode for keygen.c

```
I want to implement a usage page that will display if the user inputs invalid
arguments. To do this, I'll create a function called usage.

Get the user inputs and parse it through getopt.
If the user specifies the number of bits that the prime numbers should be,
store it for later use.
You should also do the same if the user specifies the number of iterations,
the public file/private file names, the seed for the random number generator,
and verbose.

Open the specified public file and private files.
If either one of these files were unable to be opened print out an error
message.

Set the perms for the private file to only be read and writable by the user.
This can be done with fileno and fchmod.

Initialize the state.
Make the public and private keys.

Get the username of the user via getenv.
Perform an RSA sign and store it in a variable.

Write the public key and private key data into their respective files.

Check to see if verbose is true:
    Print out the signature, the prime numbers, the product of the prime
    numbers, the public e exponent, and the private key d.

Close the public and private files.
Clear the state.
Return 0.
```

6.1 Notes about pseudocode

- If verbose was enabled, the format should be as followed and also have the number of bits of the number for the mpz_t values:

  - username
  - signature s
  - prime p
  - prime q
  - the public modulus n
  - the public exponent e
  - the private key d
- The default public/private files are rsa.pub/rsa.priv.
- The default number of bits is 256.
- Default seed is time(NULL)
- Default number of iterations is 50.

7 Pseudocode for encrypt.c

```
I want to implement a usage page that will display if the user inputs invalid
arguments. To do this, I'll create a function called usage.

Get the user inputs and parse it through getopt.
If the user specifies the input file or output file, store this information
for later usage.
Do the same if the user specifies the default public key location n,
and if v is set to true/false.

Open the specified public file. If the file failed to open, print out a
message saying so and close the program.

Read the public file.

Close the public file.

Check to see if verbose is true:
    Print out the username, the signature, n, and the public e exponent.

Get the username of the user and convert it to mpz.
Perform an RSA verification to see if this user has the correct perms.
Check to see if verification failed:
    Print out a message saying we were unable to verify user.
    Close the program.

If the user specified an input/output file, open the files up.
Perform an RSA encryption.
```

```
Close the input/output files.
Return 0.
```

7.1 Notes about pseudocode

- If verbose was enabled, the format should be as followed and also have the number of bits of the number for the mpz_t values:
  - username
  - signature s
  - the public modulus n
  - the public exponent e
- The default public file is rsa.pub.
- The default input file is stdin.
- The default output file is stdout.

8 Pseudocode for decrypt.c

```
I want to implement a usage page that will display if the user inputs invalid
arguments. To do this, I'll create a function called usage.

Get the user inputs and parse it through getopt.
If the user specifies the input file or output file, store this information
for later usage.
Do the same if the user specifies the default private key location n,
and if v is set to true/false.

Open the specified private file. If the file failed to open, print out a
message saying so and close the program.

Read the private file.

Close the private file.

Check to see if verbose is true:
    Print out the n, and private d key.

If the user specified an input/output file, open the files up.
Perform an RSA decryption..

Close the input/output files.
Return 0.
```

8.1 Notes about pseudocode

- If verbose was enabled, the format should be as followed and also have the number of bits of the number for the mpz_t values:
  - the public modulus n
  - the private key d
- The default private file is rsa.priv.
- The default input file is stdin.
- The default output file is stdout.

9 Error Handling:

Some problems and solutions I found while coding.

1. If the user inputs anything incorrectly, I will display the help page.
2. Specifying a small nbit number would originally break my code. I've made it so that you can use small prime numbers. Not very useful in the scope of this assignment, but I thought it was worth mentioning.

10 Credit:

- I've tried to follow the information provided in asgn5 to implement my programs.
- I attended Eugene's section on February 4th and 11th. He helped me understand the basics of the assignment. He gave me ideas on how to implement the functions. He also helped me a lot with understanding the encrypt file and decrypt file functions.
- Audrey's example design pdf helped me structure my own design pdf. This can be found on this quarters cse13s discord server in the asgn-1-faq-n-tips channel.
- I used my asgn4 sorting.c file as reference and inspiration to implementing my switch cases for my keygen.c/decrypt.c/encrypt.c.
- I've used the asgn5.pdf command line options as my synopsis help page for the keygen, encrypt, decrypt programs.
- The C Programming Language book may have influenced some of the code I've written.
- Ben's tutorial on how to compile and link multiple files together was really helpful. I also followed his scan-build guide on the hilalmorrar.com website.
- I've used my own Makefile from asgn4 as reference to build the makefile for this assignment.
- I took inspiration from the discord user Elmer#1515 on implementing fscanf. He also helped me find a way to make my random number generators be more efficient.

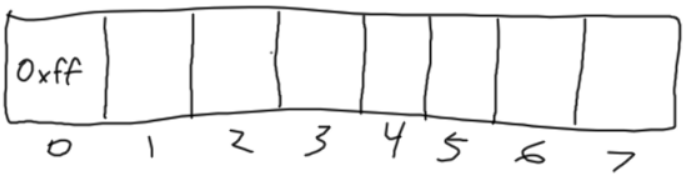
- 190n#1979 helped me understand how to use fileno/fchmod and how I could fopen some files.

(Here's a terrible picture that I drew on how to encrypt a file. It helped me a lot to understand whats happening when we are encrypting a file and thought that I should include it regardless if how it looked.)

encrypt

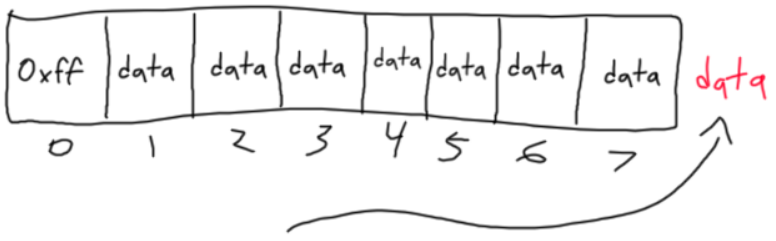
- let block size = 8
- make a block

- set 0th slot to 0xFF
(ask me why if your interested)

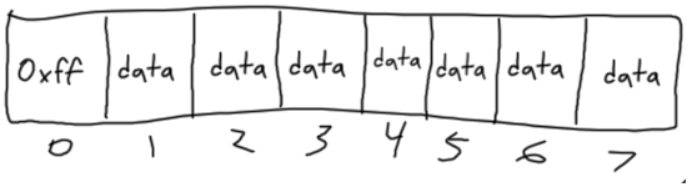


- read in from infile
(we want only blocksize-1)

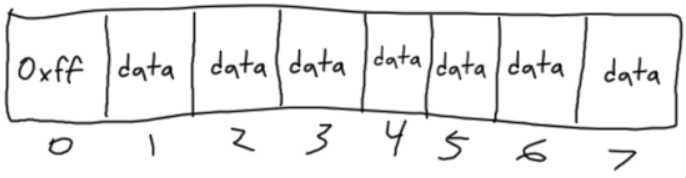
why? if we read in blocksize



we get a segfault since we ran out of space to put onto the block



- once its on the block, put the whole block into mpz



$= m$

- encrypt m

-> encryptor -> 

move to outfile as hex #