Twisha Sharma

CSE 13S

Dr. Darrell Long

February 5th, 2022

Assignment 5 Design Document

Introduction:

Description of the program:

In this assignment, we will have to create 3 programs : one that is a key generator (keygen), one that is an encryptor (encrypt), and one that is a decryptor (decrypt). We will be processing a text file in chunks. In those chunks, we will convert each to a number and then encrypt that number using c = m^e (mod (n)). Then we will write out the result of the encryption (which is a number).

Files to be included in directory "asgn5":

1. decrypt.c
    a. This contains the implementation and main() function for the decrypt program
2. encrypt.c
    a. This contains the implementation and main() function for the encrypt program
3. keygen.c
    a. This contains the implementation and main() function for the keygen program.
4. numtheory.c
    a. This contains the implementations of the number theory functions
5. numtheory.h
    a. This specifies the interface for the number theory functions

6. randstate.c
    a. This contains the implementation of the random state interface for the RSA library and number theory functions.
7. randstate.h
    a. This specifies the interface for initializing and clearing the random state.
8. rsa.c
    a. This contains the implementation of the RSA library.
9. rsa.h
    a. This specifies the interface for the RSA library
10. Makefile
    a. Clang must be specified
    b. CFLAGS must be specified
    c. Make must build the life executable as should make all and make life
    d. Make clean must remove all files that are compiler generated
    e. Make format should format all the source code, including the header files
11. README.md
    a. Describes how to use the program and Makefile.
12. DESIGN.pdf
    a. Describes the design and design process. (This document)

Randstate:

In the rsa keygen algorithm, there are things that are pseudo randomly generated - p, q, and e. Therefore it is necessary to implement a pseudo random number generator. Gmp requires the use of "randstate".

Numtheory.c:

Power Mod Function:

Pow_mod is a function that computes the modular exponentiation given a "base", "exponent", and "modulus". The result of the computation is placed into an "out" variable. For this function, it is extremely important to clone the "base" and "exponent" variables as we don't want to change the original values in case we need to access them later on in the program. By cloning the variables we can easily do the calculations and store the result in another variable that can then be set to "out". Be sure to check that the exponent is a positive integer and an odd number.

PSEUDOCODE:

```
void pow_mod(out, base, exponent,modulus) {
    Set "var" to 1;
    Make a clone of "base", "p"
    Make a clone of "exponent", "exp"
    Check that the "exponent" is greater than 0 {
        If the "exp" is odd {
               Multiply "var" with the "p" and mod with
               "modulus"
        }
        Multiply the "p" by itself and mod with "modulus"
        Take the floor of the "exp" divided by 2
    }
    Set the result "v" to "out"
    Return "out"
}
```

Is Prime Function:

Is_primes function is a function that checks if the given number is a prime number. Like the last function, be sure to make a clone of the mpz_t n parameter for this function as we do not want to edit the original value of n. This function checks if n is a prime number by first checking if the number fed into the function is odd. This is because all prime numbers are odd with the exception of 2. After checking that the number is odd, it's necessary to calculate the values of "s" and "r". You find "r" by dividing n-1 by 2 until "r" is odd and the result can no longer be divided by 2. "S" is simply the number of times that r is divided by 2. After computing s and r, you have to generate a number "a" in the range of 2 to n-1. For this you may have to add the offset back in as you are implementing all these functions using the gmp library and mps functions. After generating the random number, compute the power mod of "a", "r", and "n" and set the result to "y". Once that is done, check that y doesn't equal 1 or n-1. Set "j" equal to 1 and subtract 1 from s and set it to s_1. After that it's necessary to check that "j" is not greater than "s_1" and double check that y does not equal n-1. Then compute the power mod of y, 2, and n and set that resulting value back to y. This loop will continue until y = n-1 and then it will quit, add another loop inside the current one to return false if y = 1. If y doesn't equal 1, add 1 to j. Outside these 2 loops, make one more check that will return false if y doesn't equal n-1. At the end of the first loop that checks if n is odd, return true. For the case where n is even, return false at the very end of the function. It will also be necessary to hardcode for cases where n = 0, 1, 2, or 3, as these values would not hold up.

PSEUDOCODE:
```
    bool is_prime(n, iters) {
        Make a clone of n, "ncopy"
        Get the value of n - 1, "n_1"
        Clone "nminus" as a comparison var that is untouched,
        "nminus"
```

```
     Initialize “r”
     Initialize “s”
     If n is 2 or 3 {
          return true;
     }
     If n is 0 or 1 {
         return true;
     }
     If n is odd {
          Loop while ncopy is even, quit once odd
               n_1 = ncopy / 2
               update the ncopy value
          }
          Set end result to r
          For (i = 1; i <= iters; i ++) {
               Generate a random number “a” in the range of
               2 to   n - 2
               Compute power mod of a, r, and n and store
               in y
               If (y doesn't equal 1 and does not equal
               “nminus”) {
                    Set j to 1
                    Set “s_1” to s - 1
                    While (j is less than or equal to (s -
                    1) and y doesn't equal “nminus”) {
                         Compute power mod of y, 2, n and
                         set to y
                         If (y equals 1) {
                              return false;
                         }
                         Increment j by 1;
                    }
                If (y does not equal nminus) {
                    return false;
                }
            }
        }
        return true;
    }
    return false;
}
```

Make Prime Function:

This function generates a prime number. The generated prime number must be of a certain size. Implement the is_primes functions to check if the generated number is prime. After making clones of the values, calculate the power mod of 2 and the number of bits the function is being fed and set the result to a variable “v”. Once that is done, initialize 2 more variables, “v1” and “v2” to 0. In a loop, generate a random number of “bits size” and set it to “v2”. Then add v and v2 and set it to v1. The conditionally for the loop will check if “v1” is actually a prime number. Outside the loop, meaning once the generated number + the powermod result is a prime number, set the number (held in “v1”) to p, as that is the parameterized variable that holds the result of the function.

```
PSEUDOCODE:
void make_prime(p, bits, iters) {
     Initialize t, v , v1, and v2
     Set t equal to 2;
     Calculate the powermod of t and bits and set it to v
     Set v1 and v2 to 0;
     While (is_prime(v1, iters) == false) {
          Generate a random number of "bits" length and set it
          to v2
          V1 equals v2 plus v
     }
     P equals v1
     return;
}
```

Mod Inverse Function:

Mod_inverse is a function that calculates the modular inverse of n and stores the result in i. This function requires parallel assignment, where the values of 2 variables are basically swapped. Because c does not do this we will have to swap them manually using temporary variables. After making clones of all the parameter variables so the original values are not edited, initialize a variable “t” to 0, and “tprime” to 1. Then in a loop, take “rprime” (the clone of a) and loop as long as prime does not equal 0. Calculate the floor of “r” and “rprime” and set the result to “q”. Then start the parallel assignment, setting “r” to “rprime” and “rprime” to “r” - “q” * “rprime”. Do the same computations and assignments for t and tprime. If r is greater than 1, return no inverse. If t is less than 0, add “t” and “n” together and set it to t. At the very end, set t to the out parameter variable “p”.

PSEUDOCODE:

```
     void mod_inverse(i, a, n) {
          mpz_t r, rprime, t, tprime, q, rcopy, tcopy, t_n;
          mpz_inits(r, rprime, t, tprime, q, rcopy, tcopy, t_n,
          NULL);
          Initialize r, rprime, t, tprime, q
          Make r a clone of n
          Make rprime a clone of a
          Set t to 0
          Set t prime to 1
          While(rprime does not equal 0) {
               Calculate the floor of r and rprime and set the
               result to q
               Set r to rprime
               Set rprime to r - q times rprime
               Set t to tprime
               Set tprime to t - q times tprime
          }
          If (r is greater than 1) {
               Set i to 0
               Return;
          }
          if(t is less than 0) {
             T equals t plus n
          }
          Set i to t
          return;
     }
```

GCD Function:

Gcd is a function that computes the greatest common divisor of variables a and b and stores the result in a variable d. Be sure to clone the parameters a and b as we do not want to edit the original values.

PSEUDOCODE:
```
     void gcd(d, a, b) {
          Initialize and make clones of a and b, "acopy" and
     "bcopy"
          Initialize t
          While(bcopy does not equal 0) {
             Set t to bcopy
             Set bcopy to acopy mod bcopy
             Set a to t
          }
          Set d to acopy
          return ;
     }
```

Rsa.c:

Rsa Make Pub Function:

This function creates parts of a new RSA public key. If implemented this function will generate 2 large prime numbers, p and q, and their product n, along with the public exponent e. Start by initializing all the necessary variables. Be sure to make clones of the parameter variables as we dont want to edit the original values of the variables fed into the function. In order to generate the prime numbers p and q, it is necessary to determine the number of bits each prime number will be generated with. For this you must generate a random bit number in the range of "nbits" / 4 to 3* "nbits"/ 4. This bit number will be the number of bits for p. Subtract pbits from nbits to find the number of bits that will be used to generate q. After creating the primes, take their product and set it to a variable n. Once that is done, we must calculate e. This is done by first taking the gcd of p-1 and q-1 and storing that value in a variable gcd_out. Then multiply p-1 and q-1 and store that in another variable, p1_q1. Finally take the floor of the gcd_out variable and p1_q1 variable. The result from the floor division calculation is “lambda n”. Have a loop that checks if the exponent e is a coprime of lambda n. While it isn't a coprime, generate a random number e, and find the gcd of the generated e and lambda n. Once the loop quits, an e that fits the necessary parameters has been found.

PSEUDOCODE:

```
     void rsa_make_pub(p, q, n, e, nbits, iters) {
          generate pbits random bit number;
          qbits = nbits - pbits;
          Make a prime number p using pbits
          Make a prime number q using qbits
          Multiply p and q to find n
          Find p-1
          Find q-1
          Compute the gcd of p-1 and q-1
          Multiply p-1 and q-1
          Compute the floor of the gcd of p-1 and q-1 and p-1
          times q-1 (this is lambda n)
          Set gcd_e to 0
          While gcd_e does not equal 1 { // while e is not a
          coprime of lambda n
               Generate a random number e
               Gcd_e is the gcd of e and lambda n
          }
         // the value e holds should be a coprime of lambda n ->
     its now the public exponent
         return;
     }
```

Rsa Write Pub Function:

This function will write out the public rsa key to a “pbfile” using the gmp function gmp_fprintf. The key will be printed in the format of n, e, s, username all with a newline after them. All the values should be written out in the form of hex strings

Rsa Read Pub Function:

This function will read the public rsa key from "pbfile" using the gmp function gmp_fscanf. The key will be read in the format of n, e, s, username, all of which should have been written with a trailing newline after them.

Rsa Make Priv Function:

This function creates a new RSA private key "d" given primes p and q and a public exponent e. To do this begin by initializing all the necessary variables and cloning all parameter variables. This is because we don't want to edit the original values of the parameterized variables. Once that is done, calculate p-1 and q-1. Then compute the gcd of p-1 and q-1 and store it into a variable. After that, multiply p-1 and q-1. Once done take the floor of the 2 previous computations. The resulting computation is lambda n. You can take the inverse of e and lambda n and store the result into d.

PSEUDOCODE:

```
    void rsa_make_priv(d, e, p,q) {
        Compute p-1
        Compute q-1
        Compute the gcd of p-1 and q-1
        Multiply p-1 and q-1
        Lambda n equals the floor of the gcd of p-1 and q-1
        and p-1 times q-1
        Take the modular inverse of e and lambda n and store
        in d
        return;
    }
```

Rsa Write Priv Function:

This function will write out the private rsa key to a "pvfile" using the gmp function gmp_fprintf. The key will be printed in the format of n, d, both with a newline after them. All the values should be written out in the form of hex strings

Rsa Read Priv Function:

This function will read the private rsa key from “pvfile” using the gmp function gmp_fscanf. The key will be read in the format of n, d, both of which should have been written with a trailing newline after them.

Rsa Encrypt Function:

This function calls the pow_mod from our numtheory.c. It computes the power mod of m, e, and n and stores the result into c.

Rsa Encrypt File Function:

This function takes in an in and out file, along with n, and e, and will encrypt the contents of infile and write the encrypted contents to the outfile. It does this by first calculating the block size of k, which can be computed with the floor of log base 2 of n - 1 over 8. Once calculated, it's necessary to dynamically allocate an array that can hold the k number of bytes just calculated. Then set the zeroth byte of the array to 0xFF. While there are bytes in the file that are still unprocessed, use fread to read the array byte. Then use mpz_import to turn the bytes into numbers, use rsa_encrypt to encrypt the message. Finally, use gmp_fprintf to print out the message to the outfile. Be sure to clear your variables and free your allocated array once you are done processing all the bytes.

Rsa Decrypt Function:

This function calls the pow_mod from our numtheory.c. It computes the power mod of c, d, and n and stores the result into m.

Rsa Decrypt File Function:

This function takes in an in and out file, along with n, and d, and will decrypt the contents of infile and write the decrypted contents to the outfile. It does this by first calculating the block size of k, which can be computed with the floor of log base 2 of n - 1 over 8. Once calculated, it's necessary to dynamically allocate an array that can hold the k number of bytes just calculated. Then set the zeroth byte of the array to 0xFF. While there are bytes in the file that are still unprocessed, use fscanf to scan the infile. Then decrypt the file using rsa_decrypt. After that, use mpz_export to turn the numbers back into bytes, then use fwrite to write the decrypted message to the outfile. Be sure to clear your variables and free your allocated array once you are done processing all the bytes.

Rsa Sign Function:

This function calls the pow_mod from our numtheory.c. It computes the power mod of m, d, and n and stores the result into s

Rsa Verify Function:

This function checks that the signature s is verified. If it is verified it will return true, otherwise it will return false. Compute the power modulus of s (the signature), e, and n. If the result of that computation is equal to m then the signature is verified.

Keygen.c :

This file holds the main for a key generator program. In this program, it is necessary to implement a get opt loop to parse through the different command line arguments that the user may specify. Use the get opt loop to correctly set values based on the command line arguments. The key generator program opens public and private key files (rsa.pub and rsa.priv) if not specified by the user. After opening all the necessary files and setting up the get opt loop, make small error loops for if the private and public files fail. Once that is done, set the permissions of the private file to only ve readable and writable by the user and no one else. Implement this using fchmod() and fileno(). After that, set the random state using the randstate_init functions from randstate.c. Then generate the public and private keys using rsa make_pub and make_priv respectively. Then create a character pointer for username, and using getenv() determine the current user's username. Then convert the username into an mpz_t type. Then use rsa_sign to generate the signature of the user. Use that signature to write out the public key to "pbfile" using rsa_write_pub. Additionally write out the private key to "pvfile" rsa_write_priv. Make sure to close all open files and clear all initialized variables.

Encrypt.c :

This file holds the main code to run the encryption program. It implements a get opt loop to parse through command line arguments. Those arguments are taken into consideration and set in the correct spots to run the program accurately. Be sure to open and close all the necessary files as well as initialize and clear all the necessary variables. After formatting the get opt loop, and opening the necessary files, write a small loop that will exit the program in the event of a failure.

After that initialize a character pointer for the username using “getenv()”. Feed that username into rsa_read_pub from rsa.c along with the public file to read the public key file. Convert the username into an mpz_t type, then use rsa_verify, to verify the signature “s”. If the signature cant be verified, print an error message, close the files, clear the variable, and return 1. Assuming the signature can be verified, the next step will be to encrypt the file using rsa_encrypt_file. Then close all the files, clear all the variables, and return a 0.

<u>Decrypt.c :</u>

This file holds the main code to run the decryption program. It implements a get opt loop to parse through command line arguments. Those arguments are taken into consideration and set in the correct spots to run the program accurately. Be sure to open and close all the necessary files as well as initialize and clear all the necessary variables. After formatting the get opt loop, and opening the necessary files, write a small loop that will exit the program in the event of a failure. After that initialize a character pointer for the username using “getenv()”. Feed that username into rsa_read_pub from rsa.c along with the private file to read the private key file. Next, decrypt the file using rsa_encrypt_file. Then close all the files, clear all the variables, and return a 0.

To decrypt that message, you take the resulting d, and raise it to the with prower and multiply by the mod(n). To summarize, this function takes the ciphertext and decrypts with the private key.

<u>General Notes and Comments:</u>

Error Handling:

- One error I ran into was with my Is_Primes function. When calculating the remainder r, you divide n-1 by 2 multiple times. In this part of the code, I created a clone on n, and subtracted 1 from it to do the necessary calculations. The function also requires a check that will quit looping if the power mod calculation y is equal to n-1. When I implemented this check I created a while loop that ran while y was not equal to the n-1 variable I had cloned and calculated. I didn't realize until later on that the n-1 variable now held the value of the remainder and wasn't truly holding the n-1 value. I found this out using print statements to first make sure that the code was entering all the necessary loops, and then using gmp_printf to see the values of each variable. Once I realized the error it was a simple fix of making another clone of n and subtracting it by 1, and placing that variable in the check.

- Another error I came across was with my gcd function where the result calculated was incorrect when I was testing my rsa_make_pub function. After some gmp_printf statements, I realized that I didn't clone my a and b variables and therefore I was editing the parameter variables rather than cloning, calculating, and updating the result. Because I didn't clone, when I tried to access my a and b parameter variables, the values had been changed and were not correct.
- In addition, while testing my rsa_decrypt_file function, I ran into an error where my code was not decrypted correctly. Brian pointed out that 2 of my variables were in the wrong place. He explained that when calling mpz_export, you must feed in the decrypted message rather than the scanned message, and that while the file may be an outfile from encrypt_file, it is still an infile in decrypt_file and should be called as such in gmp_fscaf.

Credits:

- Eugene Section 2/4/22 - Eugene's section helped me understand the assignment and how to go about starting each of the three programs.
- Audrey Office Hours 2/9/22 - Audrey helped me debug my randstate.c file.

- Omar Office Hours 2/10/22 - Omar's office hours helped me understand numtheory and also showed me how to reference mpz functions. He also showed me how to reference the gmp library, so I could look up the specifics of each function. He also showed how to write a more streamlined makefile for this program.
- Ben's Tutoring Section 2/10/22 - As I was writing my code in simple C and also writing it in mpz on the side, Ben checked my functions to make sure that I was on the right track and that my syntax wouldnt generate any errors.
- Brian Tutoring Session 2/14/22 - Brian helped correct my errors with generating pbits for my rsa.c. He showed me that the range in which I generated my random pbit number was incorrect, and explained how to fix it. In addition he also explained rsa_write_pub to me so I could understand what to do within that function.
- Miles Tutoring Session 2/15/22 - Miles helped explain how to write the while loop for the rsa_encrypt_file function. He explained that the while loop would first have to read in the file using gmp_freadf. After that it's necessary to turn the bytes into a number using mpz_import. After that we need to encrypt the message using the rsa_encrypt function previously written, and finally print the result to the outfile using gmp_fprintf.
- Brian Tutoring Session 2/16/22 - Brian helped me with debugging my rsa_decrypt_file. He pointed out that when scanning the file in the decrypt file function, its necessary to take in the outfile of the encrypt, but within the function itself the file would still be an "infile" not an "outfile"