Bruce Bai, Cse13s

Design Document: Assignment 6

Purpose of Programs

keygen.c: Implements to produce the public and private key files

encrypt.c: Implements to encrypt infile contents into the outfile

decrypt.c: Implements to decrypt infile contents into the outfile

randstate.c: Initializes the seed needed for the gmp library's random generators

randstate.h: Declares the functions for randstate.c

numtheory.c: Implement all the needed math functions for RSA

numtheory.h: Declares all the functions for numtheory.c

rsa.c: Implements the functions needed to deal with the key files

rsa.h: Declares all functions for rsa.c

Layout/Structure of Programs

keygen.c

- Getopt
- File handling
- Encrypt
- Public key handling
- Write to files

encrypt.c

- Getopt
- File handling
- Read public key
- Handle public key information
- Encrypt file

decrypt.c

- Getopt
- File handling
- Read private key
- Private key information handling
- Decrypt file

randstate.[ch]

- extern global variable: state
- randstate_init function (of type void)
- randstate_clear function (of type void)

numtheory.[ch]

- pow_mod function (of type void)
- is_prime function (of type bool)
- make_prime function (of type void)
- gcd function (of type void)
- mod_inverse function (of type void)

rsa.[ch]

- rsa_make_pub (of type void)
- rsa_write_pub (of type void)
- rsa_read_pub (of type void)
- rsa_make_priv (of type void)
- rsa_write_priv (of type void)
- rsa_read_priv (of type void)
- rsa_encrypt (of type void)
- rsa_encrypt file (of type void)
- rsa_decrypt (of type void)
- rsa_decrypt_file (of type void)
- rsa_sign (of type void)
- rsa_verify (of type bool)

Description of Individual Parts of Programs

keygen.c

Getopt: Should parse through user arguments, rejecting invalid ones
File handling: Should end the program if files not properly open
Encrypt: Should call a few rsa functions
Public key handling: Should deal with the individual public key information
Should print out information if verbose is specified
Write to files: should call a few rsa functions

encrypt.c

Getopt: Should parse through user arguments, rejecting invalid ones
File handling: Should end the program if files not properly open
Read public key: Should call an rsa function
Handle public key information: Should print out key information if verbose is specified
Should also verify if signature is valid
Encrypt file: Should call an rsa function

decrypt.c

Getopt: Should parse through user arguments, rejecting invalid ones
File handling: Should end the program if files not properly open
Read private key: Should call an rsa function
Private key information handling: Should print out key information if verbose is specified
Should also verify if signature is valid
Decrypt file: Should call an rsa function

randstate.c

randstate_init: Should initialize the gmp library seed
randstate_clear: Should free memory allocated by randstate_init

numtheory.c

pow_mod: Should perform efficient/fast modular exponentiation
is_prime: Should test whether 'n' is a prime number using a specific amount of iterations
make_prime: should make a prime number at least 'bits' bits long + test using is_prime
gcd: should find the greatest common divisor of two numbers and return value
mod_inverse: finds the inverse of a modulo operation

rsa.c

rsa_make_pub: Should randomly generate a public key using prime numbers to generate
rsa_write_pub: Should write the public key to the outfile
rsa_read_pub: Should read the public key from the infile
rsa_make_priv: Should make private key using totient concept
rsa_write_priv: Should write private key to outfile
rsa_read_priv: Should read private key from infile
rsa_encrypt: Should encrypt a message using power modulus
rsa_encrypt file: Should write encrypted contents into outfile
rsa_decrypt: Should decrypt a message using power modulus
rsa_decrypt_file: Should write decrypted contents into outfile
rsa_sign: Should create a signature using power modulus
rsa_verify: Should test to see if signature is valid

Supporting Pseudocode

keygen.c:

```
Initialize default values
While still more user input to handle:
    If bits was specified, input exists and is right datatype, and not less than four:
        Convert input string to a number and set bits
    Else end program
    If iterations was specified, input exists and is right datatype
        Convert string to a number and set iterations
    Else end program

    If public key file was specified:
        Set input name
    If private key file was specified:
        Set input name
    If seed was specified, input exists and is not zero, and is the right datatype:
        Set seed
    If verbose was specified:
        Set a verbose boolean variable to true
    If help was specified:
        Print out help manual then end program
    Default:
        Print out help manual then end program
Open public file
If file is not valid:
    End program
Open Private file:
If file is not valid:
    Close public file then end program
Make sure private file is only accessible to user
Initialize all thing gmp library based
Get username of user
Convert username to signature and see if it's valid
If not valid:
    End program
If verbose was specified:
    Print out key information in terms of bits and decimals
free/clear all things gmp library based
Close all files
```

encrypt.c:

Initialize Files
While still more User Input to read in:
    If infile is specified:
        Open the file
    If outfile is specified:
        Open the file
    If public key file is specified and input exists:
        Open file
    If verbose was specified:
        Set a verbose boolean value to true
    If help was specified:
        Print out Help manual message and end program
    Default:
        Close any files that are potentially opened
        Print Help manual message and end program
If any file is not able to be opened:
    Close the needed/proper files and end program
Read the public key and get username
If verbose was specified:
    Print out the public key Information in terms of bits and decimals
Get signature from the Username
If signature is not valid:
    End program
Encrypt infile into outfile

decrypt.c:

Initialize Files
While still more user inputs to read in:
    If infile was specified:
        Open file
    If outfile was specified:
        Open file
    If private key file was specified:
        Set file name
    If verbose printing was specified:
        Set a verbose boolean variable to true
    If help was specified:
        Print help manual and end program
    Default:
        Close any files potentially opened
        Print help manual and end program
If Infile or outfile files are not openable:
    Close the proper files and end program
Open the private key file
If the private key file cannot be opened:
    Close both infile and outfile and then end program
Read contents of private key from the private key file
If verbose was specified:
    Print out private key contents in terms of bits and decimals
Decrypt the infile into outfile

randstate.c:

Declare ‘state’ (gmp seed variable) globally

randstate_init:

Use Mersenne Twister Algorithm to initialize state
Set the passed-in seed to be ‘state’

randstate_clear:

Clear the state to free any memory allocated

numtheory.c:

pow_mod: [Credits for pseudocode to documentation]

While the exponent is more zero
    If the exponent is odd:
        Multiply ‘out’ by the base and modulo that by ‘n’
    Square the base and modulo that by ‘n’
    Divide the exponent by two
Return the outcome

is_prime: [Credits for pseudocode to documentation]

If the passed in number is not two, and is even, or is less than two:
    It is not prime
If the number is two or three:
    It is prime
While r is not odd:
    Compute 2^s * r
    Add one to the result (n-1 vs n)
    If the result is equal to n and r is odd:
        Break from the loop
    Add one to s to counter how many twos have been used to divide r
    Divide r by two
From 1 to the specified iterations ‘iters’ inclusive:
    Choose a random number from [0, n-4]
    Add two to the result so the true range is: [2, n-2]
    Call power_mod function for ‘a^r modulo n’ and assign to y
    If ‘y’ is not 1 nor n - 1:
        Initialize a variable to ‘1’ ‘j’
        While j is less than or equal to the exponent ‘s’ -1 and y is not n - 1
            call pow_mod again for ‘y^2 modulo n’ and assign to y
            If y is 1:
                The number is not prime, end the function
            Otherwise increment j by one
        If y is not n - 1:
            The number is not prime, end the function here
The number is most likely prime at this point, so return true

make_prime: (Note - the mathematical reasoning behind this is provided in my code)

```
Set a variable for ‘bits - 1’ (let’s call it bitsMinusOne)
Set ‘offset’ variable to 2^bitsMinusOne
Set an outcome variable ‘out’
While prime number not found yet:
        If bitsMinusOne is not 0:
                Call urandomb with bitsMinusOne passed in rather than bits
        Add offset to ‘out’
        If ‘out’ at this point is only one:
                Add one to ‘out’ to avoid infinite loop
        If ‘out’ is a prime number:
                Break out of loop
set ‘p’ to ‘out’ and end program here
```

mod_inverse: [Credits for pseudocode to documentation]

```
Set r to n and inverse_r to a
Set t to 0 and inverse_t to 1
While inverse_r is not 0
        q = r / inverse_r
        Save value of ‘r’ in ‘tmp’
        r is now inverse_r
        inverse_r = tmp - (q * inverse_r)
        Save value of ‘t’ in ‘tmp’
        t is now inverse_t
        inverse_t = tmp - (q * inverse_t)
If r is more than 1:
        There is no inverse, set i to 0 and then end program
If t is more than 0:
        There is an inverse, add t and n to t (t = t+n)
Set ‘i’ to ‘t’ and then end program here
```

gcd: [Credits for pseudocode to documentation]

```
While the second number ‘b’ is not 0:
        Assign b to another variable ‘t’
        ‘b’ is now the first number ‘a’ modulo by b
        Assign the variable t back to a
Return a
```

rsa.c:

rsa_make_pub:

While ‘n’ is not the right amount of bits (aka nbits):

Generate pbits from [0, 2*nbits / 4] then offset by nbits/4 for: [nbits/4, 3*nbits/4]

The remaining bits go to qbits: nbits - pbits

Make a prime number for p of pbits

Make a prime number for q of qbits

Multiply p and q to create n

If n is the right amount of bits (aka nbits):

Break out of loop

Calculate the totient: (p-1)(q-1)

While gcd of totient and random public key num is not 1:

Use urandomb to generate random number of nbits

If that random number and the totient have a gcd of 1:

Break out of loop

End of program

rsa_write_pub:

write the public key ‘n’ and ‘e’, the signature ‘s’, and username into the public key file

rsa_read_pub:

Get the public key ‘n’ and ‘e’, the signature ‘s’, and username from the public key file

rsa_make_private:

Calculate the totient: (p-1)(q-1)

Find the modular inverse of e^totient, the result is the private key

rsa_write_priv:

Write the private key ‘n’ and ‘d’ into the private key file

rsa_read_priv:

Get the private key ‘n’ and ‘d’ from the private key file

rsa_encrypt:

Store the result of ‘m^e modulo n’ into ‘c’

rsa_encrypt_file:

Set ‘k’ to be of size: (nbits / 2 - 1) / 8

Create an array/buffer through zero-initialized memory allocation

Set the zero-eth index of the array to 255 (0xFF)

Set variable to counter how many bytes read per round (‘b’), initializing as 1

While ‘b’ is more than 0:

Read contents from infile, storing them into the array

(While reading contents, track how many bytes have been read using ‘b’)

Convert the contents from the array into an mpz variable ‘m’

Call ‘rsa_encrypt’ with exact same arguments/parameters (stores result in ‘c’)

Store c into outfile

End of program

rsa_decrypt:

Store result of ‘c^d modulo n’ into ‘m’

rsa_decrypt_file:

Set ‘k’ to be of size: (nbits / 2 - 1) / 8

Create an array/buffer through zero-initialized memory allocation

Set variable to counter how many bytes read per round (‘b’), initializing as 0 this time

While ‘b’ is more than 0:

- Get ‘c’ from infile
- Call ‘rsa_decrypt’ with exact same arguments/parameters (stores result in ‘m’)
- Convert the mpz variable ‘m’ back into the original contents in array
- write contents from array into outfile

End of program

rsa_sign:

Store the result of ‘m^d modulo n’ into ‘s’

rsa_verify:

Store result of ‘s^e modulo n’ into ‘t’

If m is not equal to t:

- The signature is not valid, return false

Otherwise if it is:

- The signature is valid, return true