Keaton Shawhan
cruzID: kshawhan
Assignment 5

Assignment goal:

The first goal of this assignment is to develop an algorithm that can generate RSA public and private key pairs. The second task is to use the public key generated from the RSA algorithm to encrypt a message. Lastly, we must be able to use the private key generated from the RSA algorithm to decrypt the encrypted message from earlier. To complete this lab, we must also use the GNU multiple precision arithmetic library to handle larger numbers of size 256 bits.

numtheory.c:

Set the state for the random number generation with ‘extern’ to use in all functions

create the gcd function, passing in the output variable d, and 2 numbers a and b

- initialize a temporary variable
- Loop while b does not equal 0
    - Set the temporary variable to b
    - Calculate a modulo b, setting to a
    - Set a to the temporary variable
- Clear the temporary variable
- Set the output variable (d) to a

create the power_mod function, passing in the output variable out, a base, an exponent, and a modulus

- set a variable v to 1
- set a variable p to the base value passed in
- Loop while the exponent value passed in is positive
    - Check if the exponent value passed in is odd
        - Multiply v and p
        - Modulo the value of (v*p) by the modulus value passed in
    - Multiply p by p
    - Modulo the value of (p*p) by the modulus value passed in
- set the out variable to the value of v

create the is_prime function, passing in the number n and a number of iterations to check if n is prime or not (return a boolean)

Include edge cases

Initialize the temporary variables

Create mpz_t variables for constant values used throughout the function

Loop while r is even, as we want it to be odd

Calculate (2^s) using mpz_mul_2exp

Divide n-1 by the calculated value of 2^s

Increment s by 1

Subtract from n before random number generation to change the bounds of the gmp function's default 0-n-1

Loop from 1 to the number of iterations passed in originally

Generate the random number, changing upper bound

Call the power modulus function on the random number that was just generated as the base, r as the exponent, and the original n as the modulus

Check if y doesn't equal 1 and n-1

If so, set a new temporary variable 1 to compare in future statements

Loop while j is less than or equal to s-1 and while y does not equal n-1

Call the power modulus function on the earlier power modulus return value as the base, 2 as the exponent, and the original number n as the modulus

Check if y equals 1

If so, return false and clear temp variables

Add 1 to j

Check if y does not equal n-1

If so, return false and clear temp variables

After the while loop finishes, clear the temp variables and return true

create the mod_inverse function, calculating the inverse modulus i of a modulo n

Initialize temp variables

Make a copy of the value of n passed in

Set r' to the value of a passed in

Set a variable t to 0 and t' to 1

Loop while r' isn't 0

Find the value of q by dividing r by r'

Set a temporary variable for r and t to keep their original value in other calculations

Set r to r'

Calculate q*r' and then divide the r temporary variable (r original value) by it

Set t to t'

Calculate q*t' and then divide the t temporary variable (t original value) by it

Check if r is greater than 1

If so, set the output variable to 0, clear temp variables, and return nothing because the return type of mod_inverse is void

Check if t is less than 0

If so, add n to t and then set the output variable to t

Clear temp variables

Return nothing because the return type of mod_inverse is void

Create the make_prime function, passing in an output variable p, a specific amount of bits for the random prime number to be, and the number of iterations

Initialize a temporary variable to store the random value generated from the is_prime called within

Set the state for the random number generation within

Create a boolean variable for the output of is_prime, set to false

Loop until a break statement is hit

Generate the random number, using state and the number of bits passed in to make sure the number is big enough

Update the boolean variable by calling is_prime

Check if the boolean variable is true and the size of the random number generated is big enough (compare to bits)

If so, we know we have a big enough random prime number to end the loop

Set the output variable p to the random number generated

Break or return

rsa.c:

Create rsa_make_pub function, passing in prime numbers p/q, their product n, the public exponent e, nbits to represent the number of bits, and iters to specify the number of iterations desired

Find a random number of bits to give p in the range of nbits/4 to 3*nbits/4, giving the rest of the leftover bits to q (use srandom())

use make_prime() function created in numtheory.c to generate p and q variables passed in, using the bits calculated above for each

Find n by multiplying p and q together to use while calculating the totient of n for the least common multiple calculation

Use the n - (p+q) - 1 to calculate the totient of n to use in the least common multiple calculation

Find the absolute value of totient of n and then divide it by the greatest common denominator of p-1 and q-1 to get the Carmichael’s function of n

Loop while the greatest common denominator of the randomly generated number and the Carmichael’s function of n does not equal 1 (meaning they’re co-prime)

Break once you've found a co-prime number to Carmichael's function of n, set to the public exponent e

Don't return anything because the function type is void

Create rsa_write_pub function, passing in n (p*q), the public exponent e, the signature s, the username, and a file to write to

use mpz_fprintf() function (ability to write the RSA key to pbfile), passing in the file name given and printing the hex strings in the format: n, e, s, and username. Each of these hex strings should be followed by a trailing new line to separate the lines

Create rsa_read_pub function, passing in n (p*q), the public exponent e, the signature s, the username, and a file to read from

use mpz_fscanf() to read the hex strings from the file name passed in, passing in a specifier to find every trailing new line to separate each variable n, public exponent e, signature s, and the username

Convert the hex string back to mpz_t for further calculations using mpz_set_str()

Create rsa_make_priv function, passing in the private key d, each large prime number p and q, and the public exponent e

Find n by multiplying p and q together to use while calculating the totient of n for the least common multiple calculation

Use the n - (p+q) - 1 to calculate the totient of n to use in the least common multiple calculation

Find the absolute value of totient of n and then divide it by the greatest common denominator of p-1 and q-1 to get the Carmichael's function of n

use inverse_mod() passing in the public exponent e and the carmichael's function of n to calculate e % carmichael's function of n

Set the private key d to the output

Create the rsa_write_priv function, passing in the private key d, n (which is p*q), and the file name which to write the private key to

use mpz_fprintf() function, passing in the file name given and printing the hex strings in the format: n then d. Each of these hex strings should be followed by a trailing new line to separate the lines

Create rsa_read_priv function, passing in n (p*q), the private exponent d, and a filename to read from

use mpz_fscanf() to read the hex strings from the file name passed in, passing in a specifier to find every trailing new line to separate the variable n and the private key d

Convert the hex string back to mpz_t for further calculations using mpz_set_str()

Create the rsa_encrypt function, passing in the output variable c (for ciphertext), the message m, the public exponent e, and the n

Use the pow_mod function to calculate the ciphertext c, calculating the message m^the public key e, modulus n

Create the rsa_encrypt_file function, passing in the file to be read *infile, the file to print the output to *outfile, n, and the public exponent e

calculate the block size k with the mpz_sizeinbase() function ($\lfloor(\log2(n)-1)/8\rfloor$)

(do-while loop probably)

Loop while the filled array from the fread() function has the specified amount of elements

Use calloc or malloc to allocate memory for an array the size of the block (k)

Set the 0th index of the allocated array to 0xFF for the workaround byte

Use the fread() function to fill the array of block size k with the file information, setting a variable ‘j’ to the amount of bits actually read

Use mpz_import to convert the array of bytes to mpz_t variable type once the array has been filled

Use the rsa_encrypt() function to encrypt the array that was just filled from the fread() function, turning the mpz_t back into hex string and writing it into the outfile with trailing new line

Create the rsa_decrypt function, passing in the for ciphertext, the out message m in plaintext, the private exponent d, and the n

Use the pow_mod function to calculate the plaintext m, calculating the ciphertext c^the private key d, modulus n

Create the rsa_decrypt_file function, passing in the file to be read *infile, the file to print the output to *outfile, n, and the private exponent d

calculate the block size k with the mpz_sizeinbase() function ($\lfloor(\log2(n)-1)/8\rfloor$)
(do-while loop probably)
Loop while the filled array from the fread() function has the specified amount of elements
Use calloc or malloc to allocate memory for an array the size of the block (k)
Use the fread() function to fill the array of block size k with the file information, setting a variable 'j' to the amount of bits actually read
Use the rsa_decrypt() function to decrypt the array that was just filled from the fread() function to the message
Use mpz_export() to convert the message to bytes in the allocated block size, j will be the amount of bytes
Write out j−1 bytes starting from index 1 of the block to outfile

Create the rsa_sign function, passing in the output signature s, the message m, the private exponent d, and the public modulus n

Calculate the signature by using the pow_mod function, calculating the message m^the private key d, modulus n

Create the rsa_verify function, returning a boolean based on if signature is verified, passing in the message m, the signature, the public exponent e, and the public modulus n

Verify the signature by using pow_mod to calculate the signature s^the public exponent e, modulus the public modulus n
Return true if the signature is the message is the same as the signature, false otherwise

keygen.c:

include libraries/header files

define options

initialize the global variables from the header file with values
initialize the iters, public file name, private file name, and random seed all to default values if the command line options aren't called

parse through each command line character
initialize opt and other variables

loop, checking which option was used and if it's in the list of options
switch
Case for b:
Set a variable to the minimum bits needed for the public modulus n using optarg
Case for i:
Set an iters variable to the number of Miller-Rabin iterations for testing primes (default: 50)
Case for n:
Set a variable to specify the public key (e) file (default: rsa.pub)
Case for d:
Set a variable to specify the private key (d) file (default: rsa.priv)
Case for s:
Set the random number generation state to the input (optarg), if this option isn't used, use time(NULL) to obtain the seconds since the UNIX epoch
Case for v:
Set a boolean for verbose output when the file's are filled with their respective information
Case for h:
Create a synopsis and usage page to help the user use the algorithms

Open both of the key files (either default names or the ones obtained from -n and -d options)
Check if there is an error when opening the file,
If so print an error message

Use fchmod() and fileno() functions to ensure that each respective key file has the proper permissions to be read and written into for the user.

Set the seed with either the time(NULL) default value or the argument passed in by the -s option from the user

Use rsa_make_pub() and rsa_make_priv() functions to generate the RSA key pair

Use the getenv() function to take the user's name as an input to generate the signature in later lines

Convert the username to an mpz_t (use mpz_set_str()), then using rsa_sign() to actually compute the signature with the username that was inputted

Write the public key (e) and the private key (d) to print them each to their own respective files (default or passed in through a command-line argument)

Check if the verbose (-v option) boolean is true
    If so, print the username, signature, first large prime p, second large prime q, public modulus n, and each key (e and d)
    Make sure to add a trailing new line between each piece of information
    Include the number of bits for each mpz_t variable with their decimal values next to them

Close the public and private files (default or passed in)
Use Clears() to clear all of the mpz_t variables and the random state for the random number generation

encrypt.c:

include libraries/header files

define options

initialize the global variables from the header file with values
initialize the input file name to encrypt, the output file name to encrypt, and the file name with the public key all to default values if the command line options aren't called

parse through each command line character
    initialize opt and other variables
    loop, checking which option was used and if it's in the list of options
        Switch
            Case for i:
                Set an input encrypt variable to the file name passed in (default: stdin)
            Case for o:
                Set an output encrypt variable to the file name passed in (default: stdout)
            Case for n:
                Set a variable to specify the public key (e) file (default: rsa.pub)
            Case for v:
                Set a boolean for verbose output when the file's are filled with their respective information
            Case for h:
                Create a synopsis and usage page to help the user use the algorithms

Open the public key file (either default name or the one obtained from the -n option)
Check if there is an error when opening the file,

If so print an error message

Use the rsa_read_pub function to read the public key from the file

Check if the verbose output boolean is true

If so, print out the username, signature s, public modulus n, and the public exponent (key) e, all with the trailing new line after each

Include the number of bits for each mpz_t variable with their decimal values next to them

Convert the hex string username to a mpz_t using mpz_import, verifying the signature with rsa_verify()

Check if the signature couldn't be verified (False return value)

If so, print an error message

Use the rsa_encrypt_file() to encrypt the file

Close the public file key

Use the Clears() function to clear all the memory allocated for the mpz_t variables

decrypt.c:

include libraries/header files

define options

initialize the global variables from the header file with values

initialize the input file name to decrypt, the output file name to decrypt, and the file name with the private key all to default values if the command line options aren't called

parse through each command line character

initialize opt and other variables

loop, checking which option was used and if it's in the list of options

Switch

Case for i:

Set an input decrypt variable to the file name passed in (default: stdin)

Case for o:

Set an output decrypt variable to the file name passed in (default: stdout)

Case for n:

Set a variable to specify the private key (d) file (default: rsa.priv).

Case for v:
Set a boolean for verbose output when the file's are filled with their respective information
Case for h:
Create a synopsis and usage page to help the user use the algorithms

Open the private key file (either default name or the one obtained from the -n option)
Check if there is an error when opening the file,
If so print an error message

Use the rsa_read_priv function to read the private key from the file

Check if the verbose output boolean is true
If so, print out the public modulus n and the private exponent (key) d, all with the trailing new line after each
Include the number of bits for each mpz_t variable with their decimal values next to them

Use the rsa_decrypt_file() to decrypt the file

Close the private file key
Use the Clears() function to clear all the memory allocated for the mpz_t variables

randstate.c

use extern gmp_randstate_t state to initialize the random state seed globally for reference throughout other files

Create the randstate_init() function, passing in the desired seed
Pass the desired seed variable into the srandom() function
Pass the externally declared state variable into the gmp_randinit_mt() function to initialize the state
Pass the initialized state and the desired seed into the gmp_randseed_ui() function to set the seed

Create the randstate_clear() function, passing in no arguments (void)
Use the gmp_randclear() function to clear and free all memory used by the initialized global random state named state