Isaac Flores
CruzID: **isgflore**
Prof. Miller
CSE13S
November 6, 2022

Final Design for ASGN 5: Public Key Cryptography

Overview: This assignment creates a public and private key to encrypt and decrypt a text file.

General Idea:

numtheory.c: Includes many functions that compute different number theory equations to be used when generating the public and private keys for encryption and decryption.

gcd(d, a, b): Get the greatest common denominator of a and b and return it as d. Do this by reducing a by getting the remainder of a divided by b. Set this value to b and set a to what b equaled to before. Once b is 0, a will be the greatest common denominator of a and b.

pow_mod(o, a, d, n): Implement modular exponentiation by repeatedly squaring a and reducing the value by dividing it by n and setting a to the remainder. Do this by creating a return value and starting it off as 1. If the exponent d is odd set v to the remainder of itself times p divided by n. Then set p to the remainder of its squared value divided by n. Then cut the exponent d in half rounding down. Once the exponent d is not greater than 0 return what v is.

mod_inverse(o, a, n): Implement modular inverse by setting values r and r' to n and a. Also set t and t' to 0 and 1. While r' isn't 0, set q to r divided by r' rounding down. Swap r with r' and set r' as the old value of r minus q times r'. Then swap t with t' and set t' as the old value of t minus q times t'. Once r' equals 0, return the fact that there is no inverse if r is greater than 1. If not check if t is negative and if it is add n to it. Return the value t.

is_prime(n, iters): Use the Miller-Rabin test to check if n is prime iters amount of times. Do this by finding values s and r such that r is odd and n minus 1 is equal to 2^s times r. Also, find a positive integer named a that is less than n minus 2. If a^r modulus n is not plus or minus 1 and a^(2^jr) modulus n isn't plus or minus 1 for all integers j between 0 and s, then n isn't prime. If not then n may be prime with a 25% chance that it isn't prime. The more times you run this formula then the higher the probability that n is prime.

make_prime(p, bits, iters): Generate a prime number as p and make sure it is bits amount of bits long. Make sure p is prime by running the Miller_rabin test on it iters amount of times. Running it 100 times makes sure that p is prime with a 2^{-200} probability that it isn't prime. This is a very low chance that it isn't prime.

randstate.c: Initializes the random state that will be used to generate random numbers to be used to generate the public and private keys.

randstate_init(seed): Initialize the seed for the random number generator algorithm, the global random state for the Mersenne Twister algorithm, and the seed value into the random global state.

randstate_clear(): Clear and free the memory set into the random global state.

rsa.c: Contains the code to create, read, and write a public and private key into a file. It contains the code to encrypt and decrypt a file. Also contains code to RSA sign and verify with the private key and public modulus.

rsa_make_pub(p, q, n, e, nbits, iters): Creates random prime numbers p and q with the make_prime() function. P should have a random number between nbits/4 to 3*nbits/4. The rest of the bits go to q. Then compute random prime numbers for p and q such that p*q will have nbits. Compute the public modulus n = p*q, totient p, totient q, and totient n. Then compute lambda n which is totient n / gcd(totient p, totient q). Use lambda n to find the public exponent e which should be coprime with lambda n meaning the greatest common denominator of e and lambda n is 1. This function finds the large prime integers p and q, the public modulus n, and the public exponent e which will be used for encryption and decryption.

rsa_write_pub(n, e, s, username[], pbfile): Writes the public modulus n, public exponent e, the signature s, and the username into the pbfile as hex strings except for the username. This will be used to store the data for encryption.

rsa_read_pub(n, e, s, username[], pbfile): Reads the public modulus n, public exponent e, the signature s, and the username from the pbfile. This will be used to read the data for encryption.

rsa_make_priv(d, e, p, q): Compute the private key d by setting it to the inverse mod of public exponent e modulo lambda n. The private key is used to decrypt encrypted files.

rsa_write_priv(n, d, pvfile): Writes the public modulus n and private key d into the pvfile as hex strings. This will be used to store the data for decryption.

rsa_read_priv(n, d, pvfile): Reads the public modulus n and private key d from the pvfile. This will be used to read the data for decryption.

rsa_encrypt(c, m, e, n): Perform RSA encryption by setting the ciphertext c as the power mod of base message m, public exponent e, public modulus n. This will be used to encrypt blocks of data.

rsa_encrypt_file(infile, outfile, n, e): Encrypt the file infile into outfile by encrypting blocks of data of size k = ((number of bits in n - 1) // 8). Each block will be stored in a dynamically allocated array comprised of integers. The 0th element of each block will be padded with a byte of 0xFF which will allow each block to be less than public modulo n. The function encrypts at most k - 1 bytes of data starting after the padding. The blocks will be converted to integers, then encrypted, then written into the outfile as a hex string followed by a newline. This function will encrypt the entire file that the user wants encrypted.

rsa_decrypt(m, c, d, n): Perform RSA decryption by setting the message m to the power mod of base ciphertext c, exponent private key d, and public modulus n. This function will be used to decrypt a block of encrypted data.

rsa_decrypt_file(infile, outfile, n, d): Decrypt the file infile into outfile by decrypting blocks of data of size k = ((number of bits in n - 1) // 8). Each block will be stored in a dynamically allocated array comprised of integers. The function scans a hex string of data separated by a new line. It then decrypts the hex string into a message and changes the message back into bytes while keeping track of the number of bytes that were converted. This data is stored in a block. The data is written into outfile starting at index 1 since there is a byte of padding. This function is used to decrypt an entire encrypted file.

rsa_sign(s, m, d, n): RSA signs a signature s by setting s to the power mod of base message m, exponent private key d, public modulus n. This will be used to ensure that only the user is accessing the files.

rsa_verify(m, s, e, n): RSA verify the signature by returning true if t, calculated with the power mod of base signature s, public exponent e, and public modulus n, is equal to the message m. If t does not equal to message m then return false. This function will be used to verify if the user is accessing the files.

keygen.c: Generates the random public and private keys to be used when encrypting and decrypting the files.

main(): Generates the public and private keys to be used for data encryption and decryption. It then writes the public and private key data into their respective files. Takes options for the minimum bits set into public modulus n, the Miller-Rabin test iterations, storage files for the public and private keys, set seed, verbose output, and help messages. If no options were given for an option, it sets the option to the default.

encrypt.c: Encrypts the file the user wants to be encrypted with the data in the public key file.

main(): Encrypts the data from the file the user wants to be encrypted into an output file. The program reads the data from the public key storage file and uses it to encrypt the file. Takes the options for the input and output files for encryption, the file with the public key data, verbose output, and help messages. If no options were given for an option, it sets the option to the default. If the user wants to encrypt from standard input, store that data in a temporary file and encrypt that file. If the user wants the encrypted data to be printed to standard output then print the hexstrings line for line.

decrypt.c: Decrypts the file the user wants to be decrypted with the data in the private key file.

main(): Decrypts the data from the file the user wants to be decrypted into an output file. The program reads the data from the private key storage file and uses it to decrypt the file. Takes the options for the input and output files for decryption, the file with the private key data, verbose output, and help messages. If no options were given for an option, it sets the option to the default. If the user wants to decrypt from standard input, store the hexstring data line for line and decrypt that file. If the user wants the decrypted data to be printed to standard output then print the decrypted data.

Pseudocode:

numtheory.c:

gcd(d, a, b):

```
# while b is not 0
    # set a temporary variable to b
    # set b to the remainder of a divided by b
    # set a to the temporary variable
# set d as a and return d
# clear all variables made inside the function
# return
```

pow_mod(o, a, d, n):

```
# set v equal to 1
# while d is greater than 0
    # if d is odd
        # set v to the remainder of (v times p) divided by n
    # set p to the remainder of $p^2$ divided by n
    # set d to itself divided by 2
# set o to v and return o
# clear all variables made inside the function
# return
```

mod_inverse(o, a, n):

```
# set r and r’ to n and a respectively
# set t and t’ to 0 and 1 respectively
# while r’ doesn’t equal 0
    # set q to r divided by r’
    # set r to r’ and r’ to r minus q times r’
    # set t to t’ and t’ to t minus q times t’
# if r is greater than 1 meaning there is no inverse
    # set o to 0, clear all variables made inside the function, and return
# if t is negative
    # set t to t plus n
# set o to t
# clear all variables made inside the function
# return
```

is_prime(n, iters):

```
# set s and r such that n minus 1 is equal to $2^s$ times r and r must be odd
# for i to k with i starting at 1
    # choose an integer for a that between 2 and n-2 inclusive
    # set y to the power mod of base a, exponent r, and modulus n
    # if y isn't 1 and y is not equal to n minus 1
        # set j to 1
        # while j is less than or equal to s minus 1 and y isn't n minus 1
            # set y to the power mod of base y, exponent 2, and modulus n
            # if y is 1
                # clear all variables made inside the function and return false
            # add 1 to j
        # if y isn't equal to n minus 1
            # clear all variables made inside the function and return false
# clear all variables made inside the function
# return true
```

make_prime(p, bits, iters):

```
# generate a random prime number into p and make sure p is at least bits amount of bits long
# check if p is prime iters amount of times with the Miller-Rabin test function created
# if p isn't prime generate a new random number and continue until p is prime
# clear all variables made inside the function
```

randstate.c:

randstate_init(seed):

```
# initialze the seed for the random number generator
# initialize the global state for the Mersenne Twister algorithm
# set the seed value into the global state
```

randstate_clear():

```
# clear all the memory stored in the global state
```

rsa.c:

rsa_make_pub(p, q, n, e, nbits, iters):

```
# while the size of n is less than nbits
    # get the number of bits p will be by using random(), modulus this by 2*nbits/4 so the
random number will be between 0 and 2*bits/4. Add nbits/4 to this number so the number of bits
will be between nbits/4 and 3*nbits/4.
    # get the number of bits for q by subtracting nbits from the bits in p
```

        # generate prime numbers into p and q with each being their respective amount of bits long using make_prime() iters amount of time
        # calculate n by multiplying p and q and setting the value into n
# calculate φ(p) and φ(q) by setting them to p-1 and q-1 respectively
# calculate totient n by multiplying φ(p) and φ(q) and setting the value into φ(n)
# calculate λ(n) by setting it to φ(n) / gcd(φ(p), φ(q))
# calculate a random value for e that is nbits large. Check if e is coprime with λ(n) by checking if gcd(e, λ(n)) is 1. If it isn't generate a new value for e until it is coprime with λ(n).
# clear all variables made inside the function
# return

rsa_write_pub(n, e, s, username[], pbfile):
# write the integers public modulus n, public exponent e, and signature s into pbfile as hex strings with each hex string ending with a trailing new line. Also, add the username to the line below s with a trailing new line.

rsa_read_pub(n, e, s, username[], pbfile):
# scan the values for public modulus n, public exponent e, signature s, and the username from the pbfile. Remember that each value has a trailing newline after it.

rsa_make_priv(d, e, p, q):
# calculate totient p and totient q by equalling them to p-1 and q-1 respectively.
# calculate totient n by equalling it to totient p times totient q.
# calculate lambda n by setting it equal to (totient n / gcd(totient p, totient q)).
# set the private key d to the inverse mod of e modulus lambda n.
# clear all the variables made inside the function.

rsa_write_priv(n, d, pvfile):
# write the integers for public modulus n and the private key d into pvfile as hex strings with each hex string ending with a trailing new line.

rsa_read_priv(n, d, pvfile):
# scan the hex strings for public modulus n and private key d from the pvfile. Remember that each value has a trailing newline after it.

rsa_encrypt(c, m, e, n):
# set ciphertext c as power mod of base message m, public exponent e, and public modulus n.

rsa_encrypt_file(infile, outfile, n, e):
# calculate the block size k by setting it equal to ((number of bits in n - 1) // 8).

```
# dynamically allocate memory to a block array so that it can hold k bytes of an 8-bit integer.
# pad the 0th byte of the array to 0xFF or 11111111
# dynamically allocate memory to a buffer array so that it can hold k bytes of an 8-bit integer.
# while infile hasn't been fully looked through
    # read at most k - 1 bytes from the infile to the buffer array and set j to the number of bytes actually read in this iteration. Make sure each element stores a byte of data.
    # for index 1 until the index is greater than j
        # set the index element of the block array to the index - 1 element of the buffer array, then add 1 to index
    # if the number of bytes actually read is equal to k-1
        # use mpz_import() to import k  amount of data with each data value being 1 byte of data from the block array. Set the order to 1, the endian to 1, and the nails to 0. Set the output to message m.
    # if the number of bytes actually read is not equal to k-1
        # use mpz_import() to import j+1 amount of data with each data value being 1 byte of data from the block array. Set the order to 1, the endian to 1, and the nails to 0. Set the output to message m.
    # use rsa_encrypt() to encrypt the message m with public exponent e and public modulus n into an encrypted message m.
    # print the encrypted message m into the outfile as a hex string with a trailing new line
# free all the dynamically allocated arrays and variables used inside the function
```

rsa_decrypt(m, c, d, n):

```
# set message m as the power mod of base ciphertext c, private key d, and public modulus n.
```

rsa_decrypt_file(infile, outfile, n, d):

```
# calculate the block size k by setting it equal to ((number of bits in n - 1) // 8).
# dynamically allocate memory to a block array so that it can hold k bytes of an 8-bit integer.
# while infile hasn't been fully looked through
    # scan a hex string into c and remember that each block ends with a trailing newline.
    # decrypt c into m using rsa_decrypt() with the private key d and the public modulus n.
    # use mpz_export() to export m into the block array with each value stored being 1 byte long. Set the order to 1, the endian to 1, and the nails to 0. Set j to the number of bytes actually written into the block array.
    # write j - 1 bytes starting from the 1st element of the block into the outfile
# free all the dynamically allocated arrays and variables used inside the function
```

rsa_sign(s, m, d, n):

```
# set the signature s by setting it the power mod of base message m, exponent private key d, and public modulus n.
```

rsa_verify(m, s, e, n):
# set t to the power mod of base signature s, public exponent e, and public modulus n.
# if t is the same as the expected message m, then return true else return false

keygen.c:
main():
# take the options from the user
    # b is the minimum bits needed for the public modulus n with the default value being 1024. If b is less than 50 or greater than 4096, print an error message, print help, and return a non-zero error code. If not, set b to user input.
    # i is the number of iterations for the Miller-Rabin test with the default value being 50. If i is less than 1 or greater than 500, print an error message, print help, and return a non-zero error code. If not, set i to the user's input.
    # n is the public key file with the default file being rsa.pub. If n is given, set the public key file location to the user's input.
    # d is the private key file with the default file being rsa.priv. If d is given, set the private key file location to the user's input.
    # s is the specific random seed when initializing the global state with the default being the seconds since the UNIX epoch. If s is given, set the seed to the user's input.
    # v enables verbose output. Set a verbose variable to 1 if the v option is given. If given, print help and return 0.
    # h displays the program synopsis and helps with program usage. If given, print help and return 0.
# create and open the public and private key files with reading and writing permissions and return an error message if there was a failure.
# use fchmod() and fileno() to set the private key permissions to 0600 so that the file can only be read and written by the user.
# initialize the global state with randstate_init() using the seed value.
# make the public and private key data with rsa_make_pub() and rsa_make_priv().
# get the user's username as a string.
# use mpz_set_str() to change the username into an integer with a base of 62. Then compute the signature of the username.
# write the public and private key data into their assigned files.
# if the user wants verbose output, print the username, the signature, the first large prime, the second large prime, the public modulus, the public exponent, and the private key in this order with each being followed by a trailing new line. Each integer value should also have the number of bits that make them and their decimal equivalent.
# close the public and private key files and clear the random state plus any variables used in the file.

Helper Functions:
<u>verbose_print():</u>
# prints the username, the signature, the first large prime, the second large prime, the public modulus, the public exponent, and the private key in this order with each being followed by a trailing new line. Each integer value should have the number of bits that make them and their decimal equivalent.

<u>print_help():</u>
# prints the help message for the decrypt program when the user asks for help or when the user inputs an invalid amount of bits for the public modulus or the Miller-Rabin iterations

encrypt.c:
<u>main():</u>
# take the options from the user
    # i is the input file to encrypt with the default being stdin. If given, set the input file to encrypt to the user's input.
    # o is the output file to encrypt with the default being stdout. If given, set the output file to encrypt to the user's input.
    # n is the public key file with the default file being rsa.pub. If n is given, set the public key file location to the user input.
    # v enables verbose output. Set a verbose variable to 1 if the v option is given.
    # h displays the program synopsis and help with program usage. If given, print help and return 0.
# open the public key file. Return an error message and return a non-zero error code if there was a failure.
# if the input file to encrypt was left as stdin
    # open a new temporary file that will hold the input from the user with reading and writing permissions. Scan each character the user inputs and print it to the temporary file.
# if the input file to encrypt wasn't left as stdin
    # open the file the user wants to encrypt with reading permissions. If there was an error, print an error message and return a non-zero error code.
# if the output file to encrypt was left as stdout
    # open a new temporary file that will hold the output of the encryption with reading and writing permissions
# if the output file to encrypt wasn't left as stdout
    # open the new file the user wants the encryption output to be in with reading and writing permissions
# read the public key data from the public key file.

# if verbose output is enabled, print the username, the signature, the public modulus, and the public exponent. Each integer value should also have the number of bits that make them and their decimal equivalent.
# convert the read username into an integer which will be the expected message for the signature verification. Verify the signature and exit the program with a non-zero error code and an error report if the signature was not verified.
# encrypt the file or text the user wants encrypted with rsa_encrypt_file().
# if the output file to encrypt was left as stdout
    # scan each hexstring put into the outfile and print it as a hexstring
# close the public key file and clear any variables used in the file.

Helper Functions:
<u>verbose_print():</u>
# prints the username, the signature, the public modulus, and the public exponent. Each integer value should have the number of bits that make them and their decimal equivalent.

<u>print_help():</u>
# prints the help message for the decrypt program when the user asks for help

decrypt.c:
<u>main():</u>
# take the options from the user
    # i is the input file to decrypt with the default being stdin.
    # o is the output file to decrypt with the default being stdout.
    # n is the private key file with the default file being rsa.priv. If n is given, set the private key file location to the user input.
    # v enables verbose output.
    # h displays the program synopsis and help with program usage. If given, print help and return 0.
# open the private key file and return an error message with a non-zero error code if there was a failure.
# if the input file to decrypt was left as stdin
    # open a new temporary file that will hold the input from the user with reading and writing permissions. Scan each hexstring the user inputs and print it to the temporary file.
# if the input file to decrypt wasn't left as stdin
    # open the file the user wants to decrypt with reading permissions. If there was an error, print an error message and return a non-zero error code.
# if the output file to decrypt was left as stdout
    # open a new temporary file that will hold the output of the decryption with reading and writing permissions

```
# if the output file to decrypt wasn't left as stdout
    # open the new file the user wants the decryption output to be in with reading and writing permissions
# read the private key data from the private key file.
# if verbose output is enabled, print the public modulus and the private key. Both values should also have the number of bits that make them and their decimal equivalent.
# decrypt the file the user wants decrypted with rsa_decrypt_file().
# if the user left the output file as stdout
    # read each character from the temporary output file and print it
# close the private key file and clear any variables used in the file
```

Helper Functions:

verbose_print():

```
# prints the public modulus and the private key. Both values should also have the number of bits that make them and their decimal equivalent.
```

print_help():

```
# prints the help message for the decrypt program when the user asks for help
```