Ranvir Singh (rsingh63) CSE13S Assignment 5 Design Document:

Objective:

decrypt.c, encrypt.c, and keygen.c utilize numtheory.c and rsa.c in order to create public and private key pairs that can be used for decryption and encryption.

File Contents:

numtheory.c lays out several mathematical functions utilized in RSA encryption

rsa.c contains the implementation of the RSA library,

randstate.c utilizes GMP random functions and srandom() to establish states for random numbers.

decrypt.c contains a main() function for decrypting a message using a private RSA key

encrypt.c contains a main() function for encrypting a message using a public RSA key

keygen.c generates a public and private RSA key pair

Randstate.c

- **Description:**
    - Creates a random state “state” and contains functions to initialize the state and clear it
- **randstate_init()** takes an integer seed, and initializes a random state for use by GMP library random functions
    - Pseudocode:
    - Initialize the random state “state” to use Mersenne’s Twister
    - Provide “seed” to state

    - Provide “seed” to srandom

- **randstate_clear()** clears the random state initialized by randstate_init()
    - Pseudocode:
    - Clear the random state “state”

numtheory.c

- **Description:**
    - Contains several mathematical functions for use within RSA encryption
- **pow_mod()** performs modular exponentiation
- follows exactly the pseudocode provided in asgn5.pdf

```
void pow_mod takes int o, int a, int d, int n
      v = 1
      p = a
      while d > 0
            if d is odd
                  v = (v * p) % n
            p = p^2 % n
            d /= 2
      o = v
```

- **is_prime()** tests for the primality of a number
- follows the Miller-Rabin test pseudocode provided in asgn5.pdf
- The while loop at the beginning of the function takes care of the first line of the pseudocode, finding values s and r in $n - 1 = 2^s r$ by iterating a counter each time n - 1 can be divided by 2, and storing the value currently being divided as n_minus

```
bool is_prime takes int n and int k
      r = n - 1
      While true
            If (n - 1) is even
                  r = r/2
                  s += 1
            else
                  BREAK
      END WHILE
      for all ints i from 1 to k
            a = random number in range [2, n - 2]
            y = power_mod(a,r,n)
            if y =/= 1 or n - 1
                  j = 1
                  while j ≤ s - 1 and y =/= n - 1
                        y = power_mod(y,2,n)
```

```
                        if y == 1
                              return False
                        j = j + 1
                  END INNER WHILE
                  if y =/= n - 1
                  END OUTER WHILE
                        return False
      END OUTER FOR
      return True
```

- **gcd()** computes the greatest common denominator of integers b and d, then stores the result in a
- follows pseudocode provided in asgn5.pdf

```
void gcd takes int a, int b, int d
      while b != 0
            temp = b
            b = a % b
            a = temp
      d = a
```

- **lcm()** computes the least common denominator of integer b and d, storing the result in integer a

void lcm takes int a, int b, int d

```
      a = absolute value(a * b) / gcd(a,b))
```

- **make_prime()** utilizes mpz_urandomb to generate a random binary number, add 2^{n+1}, and tests it with is_prime
- A while loop is used to continually test the randomly generated numbers for primality until one is prime

```
void make_prime takes int p, int bits, int iters
      mask = 2^bits
      while true
            p = (random number up to 2^bits) + 2^bits
            if (is_prime(p, iters))
                  BREAK
```

- **mod_inverse()** performs the inverse modulo arithmetic operation, following the asgn5.pdf pseudocode

```
void mod_inverse takes integers o, a, n
      r = n, r' = a
      t = 0, t' = 1
      while r' != 0
            q = r/r'
            temp = r'
            r' = r - q * r
            r = temp
            temp = t'
            t' = t - q * t
      END WHILE
      if r > 1
            return NULL
      if t < 0
            t += n
      o = t
```

rsa.c:

- **rsa_make_pub** finds the public exponent and stores it in integer e

```
void rsa_make_pub takes ints p, q, n, e, nbits, iters
      low = nbits / 4
      high = 3 * nbits / 4
      pbits = large random number % (1 + high - low) + low
      qbits = nbits - pbits
      make_prime with pbits, store in p
      make_prime with qbits, store in q
      n = p * q
      lambda(n) = lcm((p - 1), (q - 1))
      while (e_gcd != 1)
            e = random number up to 2^nbits
            e_gcd = gcd(e, lambda)
```

- **rsa_write_pub** writes the public RSA key to pbfile, separating each piece of information with a new line

```
rsa_write_pub takes ints n, e, s, string username, and file pbfile
      print n, e, s, as hexstrings to pbfile separated by new lines
      print username as is to pbfile with a new line
```

- **rsa_read_pub** reads in the public RSA key from pbfile

```
rsa_read_pub takes ints n, e, and s, string username, and file pbfile
      scan n, e, s, back into decimal integers from pbfile separated by new
lines
      scan username as is from pbfile with a new line
```

-

- **rsa_make_priv** creates a private RSA key and stores it in integer d

```
rsa_make_priv takes ints d, e, p, and q
         -  d = mod_inverse(d, e, lcm((p - 1), (q - 1)))
```

- **rsa_write_priv** writes the private RSA key to pbfile

```
rsa_write_priv takes ints n, d, and file pbfile
         -  Convert n, and d to hexstrings, then print them to pbfile with
            newline characters at the end
```

- **rsa_read_priv** reads the private RSA key from pbfile

```
rsa_read_priv takes ints n, d, and file pbfile
         -  Reads n and d from pbfile, converting them from hexstrings back
            into the previous format
```

- **rsa_encrypt** performs the RSA encryption

```
rsa_encrypt takes given ints c, m, e, and n
      pow_mod(c, m, e, n)
```

- **rsa_encrypt_file** encrypts the contents of an entire file and writes the encrypted contents

  to another file

```
rsa_encrypt_file takes files infile and outfile, and ints n and e
         -  Encrypts contents of infile and writes them to outfile
               -  Encrypts in blocks of k =  (log2(n) - 1)/8
         -  Dynamically allocate an array of k bytes
         -  Set zeroth byte of block to 0xFF
         -  Read j < k - 1 bytes from infile, place them into the allocated
            array from index 1
         -  Use mpz_import to convert bytes from hex into mpz_tm
         -  rsa_encrypt() m, convert m to a hexstring and write it to outfile
            with a new line character at the end
```

- **rsa_decrypt** decrypts ciphertext and stores the decrypted message

```
rsa_decrypt takes ints m, c, d, and n
         -  Decrypts ciphertext c and stores the decrypted message in m
      pow_mod(m, c, d, n)
```

- **rsa_decrypt_file** decrypts the contents of an entire file and writes the decrypted contents

  to another file

```
rsa_decrypt_file takes files infile and outfile, ints n and d
         -  Decrypts contents of infile and writes them to outfile
               -  Decrypts in blocks of k =  (log2(n) - 1)/8
         -  Dynamically allocate an array of k bytes
         -  Scan and save a hexstring as c in mpz_t format
         -  mpz_export() c from a hexstring to bytes and store them in the
            allocated array
         -  Write j - 1 bytes (where j is the number of bytes converted) from
            index 1 of the block into outfile
```

- **rsa_sign** creates a signature using the public key and public modulus

```
rsa_sign takes ints s, m, d, n
      pow_mod(s, m, d, n)
```

- **rsa_verify** compares a given message to one created with the given signature, verifying the signature if the messages are the same

```
rsa_verify takes ints m, s, e, n
      pow_mod(t, s, e, n);
      if t != m, return false
      else, return true
```

keygen.c:

- **Description:**
    - Generates a public and private RSA key pair using functions from rsa.c

```
void help()
      print out help and usage message
default minimum_bits = 1024
default miller-rabin iterations = 50
default seed = time since epoch
verbose = false
string username = getenv(user)
Command line options:
      -b = minimum bits for public modulus n
            set minimum bits to argument
      -i = number of Miller-Rabin iterations
            set minimum iterations to argument
      -n [pbfile]: specifies public key file
            set public file string to argument
      -d [pbfile]: specifies private key file
            set private file string to argument
      -s = random seed for initializing random state
            set random seed to argument
      -v enables verbose output
            verbose = true
      -h displays help and usage
            help()
Open key files
Use fchmod() and fileno() to make sure that private key file permissions ==
0600
      if the files cannot be opened or modified, throw an error message
Use randstate_init() with the set seed to initialize a random state
Use rsa_make_pub() and rsa_make_priv() for a key pair
Use mpz_set_str() with base 62 to turn user's name into an mpz_t
Use rsa_sign() to compute signature of the username
use rsa_write_pub and rsa_write_priv to write keys to their respective files


if verbose:
      compute and store the number of bits of the user signature, both large
primes p and q, the public modulus, public exponent, and private exponent,
then print the values out with their bits


Close files, use randstate_clear() to clear random state, and clear mpz_t
variables
```

encrypt.c:

- **Description:**
    - Uses a public key to encrypt a message from a file

```
default input file = stdin
default output file = stdout
string pub = rsa.pub
verbose = false
empty string user holds 1024 characters


Command line options:
      -i specifies input file to encrypt
            input file = argument
      -o specifies output file to write encrypted message to
            output file = argument
      -n specifies file containing public key for encryption
            pub = argument
      -v enables verbose output
            verbose = true
      -h displays help and usage


Open public key file and read the public key with rsa_read_pub

if verbose:
      compute and store the number of bits of the user signature, the modulus,
and public exponent, then print the values out with their bits

Convert username read in to mpz_t, verify with rsa_verify() and exit if the
signature cannot be verified
Encrypt input file with rsa_encrypt_file(), print to output file
Close public key file, clear mpz_t variables
```

decrypt.c:

- **Description:**
    - Uses a private key to decrypt a message from a file

```
default input file = stdin
default output file = stdout
string priv = rsa.priv
verbose = false


Command line options:
      -i specifies input file to encrypt
            input file = argument
      -o specifies output file to write encrypted message to
            output file = argument
      -n specifies file containing public key for encryption
            priv = argument
      -v enables verbose output
            verbose = true
      -h displays help and usage


Open public key file and read the public key with rsa_read_pub

if verbose:
      compute and store the number of bits of the modulus and private
exponent, then print the values out with their bits

Convert username read in to mpz_t, verify with rsa_verify() and exit if the
signature cannot be verified
Encrypt input file with rsa_encrypt_file(), print to output file
Close public key file, clear mpz_t variables
```