Assignment 5 Design Document

Program Requirements

In this program, we need to implement the RSA public key encryption algorithm. RSA is an asymmetric key algorithm that uses two mathematically related keys to encrypt/decrypt messages. The public key, as the name suggests, is public, while the private key is known only to

the user. This algorithm resolves the key exchange problem, since the communication key can be encrypted using the public key of the recipient. Anyone else will not be able to decrypt and use the key, since the private key is required to decrypt the message.

The algorithm is assumed to be secure, since it is based on a problem that is considered to be hard to solve, namely factorization of large numbers. The program generates large prime numbers (potentially hundreds or thousands of bits long), and generates public and private keys by performing mathematical operations on them.

To achieve this, we use the gnu libgmp library. The standard data types in C don't help here, since the long data type is only 64 bits long. The mpz_t data type in this library allows us to specify positive integers of arbitrary size.

We will need to create the following files

- numtheory.c: Contains the mathematical functions that will be used in the RSA algorithm.
- rsa.c: Implements the different parts of the RSA algorithm, e.g., generation of keys, encrypting a plaintext to ciphertext and decrypting the ciphertext back to plaintext, signing a message, and verifying it, etc.
- keygen.c: Contains the main function for key generation. It will generate the public and private keys used by the algorithm.
- encrypt.c: Contains the main function for the encryption phase. It converts a plaintext to ciphertext.

- decrypt.c: Contains the main function for the decryption phase. It converts a ciphertext to plaintext.

numtheory.c Pseudocode

#include “numtheory.h”

#include <gmp.h>

1. Calculate the gcd of a and b using Euler's recursive algorithm

- Input parameters:
    - d: mpz_t: The gcd is stored here
    - a: mpz_t
    - b: mpz_t
- Returns: void

void gcd(mpz_t d, mpz_t a, mpz_t b) {

    Declare and initialize mpz_t t with mpz_init();

    while (mpz_cmp_ui(b, 0)) { // while b != 0

        Set t = b using mpz_set;

        Set b = a mod b using mpz_mod;

        Set a = t using mpz_set;

```
        }

        mpz_set(d, a);
        mpz_clear(t);
}
```

2. Compute the inverse i of a modulo n

- Input parameters:
    - i: mpz_t
    - a: mpz_t
    - n: mpz_t
- Returns: void

```
void mod_inverse(mpz_t i, mpz_t a, mpz_t n) {
        // Declare and initialize variables

        // Use mpz_set and mpz_set_ui to set the below variables
        r1 = n
        r2 = a
        t1 = 0
        t2 = 1

        // Use while to loop till r2 != 0
```

```
while (mpz_cmp_ui(r2, 0)) {
        Use mpz_fdiv_q to save quotient of r1, r2 in q
        Multiply q and r2. Save result in tmp1
        Subtract tmp1 from r1. Save in tmp2
        mpz_set(r1, r2);
        mpz_set(r2, tmp2);

        Repeat the above steps for t1 and t2, instead of r1 and r2
}

if (mpz_cmp_ui(r1, 1) > 0) { // If r1 > 1
        Clean up
        mpz_set_ui(i, 0); // Setting mod inverse to 0
}

if (mpz_cmp_ui(t1, 0) < 0) { // If t1 < 0
        Add n to t1. Save result in t1
}

Save t1 in i
Clean up.

return;
```

}

3. Perform fast modular exponentiation, computing base raised to the exponent power modulo modulus, and storing the computed result in out.

- Input parameters:
    - out: mpz_t: Stores the results
    - base: mpz_t:
    - base: exponent:
    - base: modulus:
- Returns: void

```
void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus) {
    Declare variables.
    v = 1, p = base

    while (exponent > 0) {
        if (mpz_odd_p(exponent)) {
            Multiply v and p
            Calculate mod of the above product with modulus
        }
        Multiply p and p
        Calculate mod of the above product with modulus
        exponent = floor(exponent/2)
    }
```

```
    mpz_set(out, v);
    mpz_clears(v, p, rop, NULL);
    return;
}
```

4. Conduct the Miller-Rabin primality test to indicate whether or not n is prime using iters number of Miller-Rabin iterations.

- Input parameters:
    - n: mpz_t: Number to check for primality
    - iters: uint64_t: Number of Miller-Rabin iterations
- Returns: bool: True if prime. False otherwise

```
bool is_prime(mpz_t n, uint64_t iters) {
    Declare and initialize variables

    // First step of the algo requires us to identify an r such that
    // we write (n-1 = 2^s r), where r is odd.
    r = (n - 1)/2
    s = 1;

    while (r is not odd) {
        s++;
        r = r/2;
    }
```

```
for (uint64_t i = 0; i < iters; i++) {
        // choose random a ∈ {2,3,...,n − 2}

        // To achieve this, we call mpz_urandomm(). However, this generates
        // random numbers from 0,...,n-1. Consequently, we first initialize
        // a temporary variable to n-4, generate the random number, and
        // add 2 to it.
        a = n - 4;
        mpz_urandomm(a, state, a);
        a = a + 2;

        pow_mod(y, a, r, n);

        If (y != 1 and y != n-1) {
                uint64_t j = 1;

                while (j < s and y != n-1) {
                        pow_mod(y, y, two_mpz, n);

                        if y == 1 {
                                cleanup
                                return false;
```

```
                    }
                    j++;
                }

                if y != n-1 {
                    cleanup
                    return false;
                }
            }
        }
        cleanup
        return true;
}
```

5. Generate a mersenne prime (of the form (2^n)-1 where n >= bits
    a. It uses is_prime to check for primality with the given number of iterations.

- Input parameters:
    - p: mpz_t: Prime number is stored here
    - bits: uint64_t: Minimum number of bits in the generated number
    - iters: uint64_t: Number of iterations to validate primarily
- Returns: void

```
void make_prime(mpz_t p, uint64_t bits, uint64_t iters) {
    Declare and initialize variables
```

```
        // Create an mpz_t variable with value 2 and raise it to the desired power
        Set p = (2^bits - 1)

        // Keep incrementing the bits till we get a prime number
        while (! is_prime(p, iters)) {
                bits++;
                p = (2^bits - 1)
        }

        Cleanup
        return;
}
```

rsa.c Pseudocode

```
#includes
```

1. Create an RSA public key. Two large prime numbers p and q, their product n, and the public exponent e.

- Input parameters:

    - p: mpz_t: Prime number to be generated
    - q: mpz_t: Prime number to be generated
    - n: mpz_t: n = pq
    - e: mpz_t: Exponent
    - nbits: uint64_t: Minimum number of bits for n
    - iters: uint64_t: Number of iterations to be used for primality test
- Returns: void

```
void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters) {
    Declare and initialize variables

    // Use a number in the interval [nbits/4, 3*nbits/4] as bit length for p,
    // and the rest for q.
    uint64_t p_len = nbits/4 + (random()%nbits)/2;
    uint64_t q_len = nbits - p_len;

    make_prime(p, p_len, iters);
    make_prime(q, q_len, iters);

    Set tmp1 to p-1 and tmp2 to q-1

    // Calculate gcd of tmp1 and tmp2. Save in tmp3
    gcd(tmp3, tmp1, tmp2);
```

```
        // Multiply tmp1 and tmp2. Save in tmp4
        mpz_mul(tmp4, tmp1, tmp2);

        Calculate lambda = lcm(p-1,q-1) = product/gcd

        Generate random numbers of length >= nbits

        // Loop till a random number is found that's coprime with lambda.
        // This number is the exponent.
        while (! mpz_cmp_ui(tmp2, 1)) {
                mpz_urandomb(tmp1, state, nbits);
                gcd(tmp2, tmp1, lambda);
        }
        e = tmp1

        n = pq

        Cleanup
        return;
}
```

2. Write a public RSA key to pbfile. n, e, and s are written as hexstrings in that order.

- Input parameters:

    - Constituents of the public key
        - n: mpz_t
        - e: mpz_t
        - s: mpz_t
    - username: char[]
    - pbfile: FILE *: File pointer to the public key file

```
void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile) {
        gmp_fprintf(pbfile, "%Zd\n", n);
        gmp_fprintf(pbfile, "%Zd\n", e);
        gmp_fprintf(pbfile, "%Zd\n", s);
        gmp_fprintf(pbfile, "%s\n", username);
}
```

3. Read the public key file pbfile to obtain the public key constituents.

- Input parameters:
    - Constituents of the public key read as hex strings
        - n: mpz_t
        - e: mpz_t
        - s: mpz_t
    - username: char[]
    - pbfile: FILE *: File pointer to the public key file

```
void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile) {
```

Read the file using fscanf, and store contents in the parameter variables

}

4. Given primes p and q and public exponent e, create an RSA private key d.

- Input parameters:
  - Primes
    - p: mpz_t
    - q: mpz_t
  - e: mpz_t: Exponent
  - d: mpz_t: Generated private key is saved here
- Returns: void

void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q) {

Compute the inverse of e modulo lambda(n) = lcm(p-1,q-1)

}

5. Write a private RSA key to pvfile. n and d are written as hexstrings in that order.

- Input parameters:
  - Constituents of the private key
    - n: mpz_t
    - d: mpz_t
  - pvfile: FILE *: File pointer to the private key file
- Returns: void

void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile) {

```
    gmp_fprintf(pbfile, "%Zx\n", n);
    gmp_fprintf(pbfile, "%Zx\n", d);
}
```

6. Read a private RSA key from pvfile.

- Input parameters:
  - Components of the private key stored as hex and read in that order
    - n: mpz_t
    - d: mpz_t
  - pvfile: FILE *: File pointer of the file to be read
- Returns: void

```
void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile) {
    Read contents of pvfile using gmp_fscanf()
}
```

7. Performs RSA encryption, computing ciphertext c by encrypting message m using public exponent e and modulus n.

- Input parameters:
  - c: mpz_t: Generated ciphertext
  - m: mpz_t: Original plaintext message
  - e: mpz_t: Public exponent
  - n: mpz_t: Modulus
- Returns: void

```
void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n) {
    Generate encrypted ciphertext c = m^e (mod n)
}
```

8. Encrypts the contents of infile, writing the encrypted contents to outfile.

- Input parameters:
    - infile: FILE *: Input file to be encrypted
    - outfile: FILE *: Encrypted output file
    - n: mpz_t: Modulus
    - e: mpz_t: Exponent
- Returns: void

```
void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e) {
    Calculate the block size k = floor(log_2(n)-1/8)
    Allocate an array to hold k bytes. Typecast it to uint8_t *.
    Set the 0th byte of the block to 0xFF

    while(! feof) {
        Read at most k-1 bytes from the infile
        j = number of bytes actually read
        Copy the bytes in the allocated blocks starting from position 1.
        Use mpz_import to convert bytes to mpz_t m
        Encrypt m using rsa_encrypt
        Write encrypted number to outfile as hex followed by newline
```

}

Cleanup the memory allocated

}

9. Performs RSA decryption, computing message m by decrypting ciphertext c

- Input parameters:
    - m: mpz_t: Decrypted message
    - c: mpz_t: Ciphertext to be decrypted.
    - d: mpz_t: Private key
    - n: mpz_t: Public modulus
- Returns: void

void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n) {

m = c^d (mod n)

}

10. Decrypts the contents of infile, writing the decrypted contents to outfile.

- Input parameters:
    - infile: FILE *: Input file containing the ciphertext
    - outfile: FILE *: Output file that will contain the plain text
    - n: mpz_t: Modulus
    - d: mpz_t: Private key
- Returns: void

void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d) {

Calculate the block size k = floor(log_2(n)-1/8)

Allocate an array to hold k bytes. Typecast it to uint8_t *.

Set the 0th byte of the block to 0xFF

while(! feof) {

Scan in a hexstring to a variable c (for ciphertex)

Use mpz_export to convert c back into bytes

let j = number of bytes actually converted

Write j-1 bytes starting from index 1 to outfile

}

Cleanup the memory allocated

}

11. Performs RSA signing

- Input parameters:
    - s: mpz_t: Signature that's produced
    - m: mpz_t: Message to be signed
    - d: mpz_t: Private key
    - n: mpz_t: Public modulus
- Components of the private key stored as hex and read in that orderReturns: void

void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n) {

s = m^d (mod n)

}

12. Performs RSA verification

- Input parameters:
    - m: mpz_t: Expected message
    - s: mpz_t: Signature to be verified
    - e: mpz_t: Exponent
    - n: mpz_t: Modulus
- Returns: bool: True if signature is verified. False otherwise

```
bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n) {
    Generate t = s^e (mod n)
    Compare t and m using mpz_cmp()
    Return true if the above function returns 0. False, otherwise.

}
```

keygen.c Pseudocode

```
#includes
```

1. Usage Function

- Input parameters:

    - exec_name: char *: Name of the program
- Returns: void

```
void usage(char *exec_name) {
        printf("USAGE: %s [-b <num_bits>][-i <num_iters>][-n <pub_key_file>][-d
<priv_key_file>][-s <seed>][-vh]\n", exec_name);
        printf("-b <num_bits>: Minimum number of bits needed for public modulus n\n");
        printf("-i <num_iters>: Number of Miller-Rabin iterations for testing primes\n");
        printf("-n <pub_key_file>: File containing the public key. Default is rsa.pub\n");
        printf("-d <pub_key_file>: File containing the private key. Default is rsa.priv\n");
        printf("-s <seed>: Seed for random state initialization\n");
        printf("-v: Turn on verbose mode\n");
        printf("-h: Print this message\n");
        return;
}
```

2. The main function

- Input parameters:
    - argc: int: Number of input arguments
    - argv: char **: The input arguments
- Returns: int: 0 in case of success, non-zero for failure

```
int main(int argc, char **argv) {
        int opt;
        uint32_t mr_iters = 50;
```

```
char *pbfile = "rsa.pub";
char *pvfile = "rsa.priv";
FILE *pbfp, *pvfp;
time_t seed = time(NULL);
bool verbose = false;
char *user_name;

// Parse the input options. Store the sorting options in a set.
while ((opt = getopt(argc, argv, "bvi:n:d:s:h")) != -1) {
        switch (opt) {
                Handle different options
        }
}

// Open the public key and private key files for writing
// Error and exit in case of failure
Open public key file for writing
Open private key file for writing
Print error message if fails
else
        // Ensure that private key file permissions are set to 0600.
        Obtain file descriptor using fileno
        Set permissions using fchmod
```

```
// Initialize the random state with the seed
randstate_init(seed);
// Make public and private keys by calling functions from rsa library
rsa_make_pub();
rsa_make_priv();

// Get the user name using getenv.
user_name = getenv("USER");

// Compute the signature of the user name by calling rsa_sign
rsa_sign();

// Write the public and private keys
rsa_write_pub();
rsa_write_priv();

if (verbose == true) {
        printf("Username\n");
        printf("Signature s\n");
        printf("First large prime p\n");
        printf("Second large prime q\n");
        printf("Public modulus n\n");
```

```
        printf("Public exponent e\n");
        printf("Private key d\n");
    }

    Close the files
    Clear all mpz_t variables
    // Clear the random state
    randstate_clear();
    return 0;
}
```

encrypt.c Pseudocode

#includes

1. Usage Function

- Input parameters:
    - exec_name: char *: Name of the program
- Returns: void

```
void usage(char *exec_name) {
```

```
    printf("USAGE: %s [-i <input_file>][-o <output_file>][-n <priv_key_file>][-vh]\n",
exec_name);
    printf("-i <input_file>: Input file to decrypt. Default is stdin\n");
    printf("-o <output_file>: Output file to decrypt. Default is stdout\n");
    printf("-n <pub_key_file>: File containing the public key. Default is rsa.pub\n");
    printf("-v: Turn on verbose mode\n");
    printf("-h: Print this message\n");
    return;
}
```

2. The main function

- Input parameters:
    - argc: int: Number of input arguments
    - argv: char **: The input arguments
- Returns: int: 0 in case of success, non-zero for failure

```
int main(int argc, char **argv) {
    int opt;
    char *infile = NULL;
    char *outfile = NULL;
    char *pub_key_file = "rsa.pub";
    FILE *ifp, *ofp, *pkfp;
    bool verbose = false;
```

```
// Parse the input options. Store the sorting options in a set.
while ((opt = getopt(argc, argv, "bvi:n:d:s:h")) != -1) {
        switch (opt) {
                Handle different options
        }
}

// Open the public key file for reading. Exit with message if error
if ((pkfp = fopen(pub_key_file, "r")) == NULL) {
        printf("The public key file is invalid. Please provide a valid input file\n");
        exit(EXIT_FAILURE);
}
// Read the file using rsa_read_pub
rsa_read_pub();

if (verbose == true) {
        // printf("%s\n", username);
        printf("Print the signature s\n");
        printf("Print the public modulus n\n");
        printf("Print the public exponent e\n");
        printf("Include info on number of bits along with their decimal value\n");
}
```

Convert the username to an mpz_t variable

Verify the signature using rsa_verify()

Exit in case of failure

Open the input file for reading. Exit if there is an error

Open the output file for writing. Exit if there is an error

Encrypt the contents of the input file using rsa_encrypt_file

rsa_encrypt_file();

// Close the files and clear any mpz_t variables

return 0;

}

decrypt.c Pseudocode

#includes

1. Usage Function

- Input parameters:
    - exec_name: char *: Name of the program

- Returns: void

```
void usage(char *exec_name) {
        printf("USAGE: %s [-i <input_file>][-o <output_file>][-n <priv_key_file>][-vh]\n",
exec_name);
        printf("-i <input_file>: Input file to decrypt. Default is stdin\n");
        printf("-o <output_file>: Output file to decrypt. Default is stdout\n");
        printf("-n <priv_key_file>: File containing the private key. Default is rsa.priv\n");
        printf("-v: Turn on verbose mode\n");
        printf(

}
```

2. The main function

- Input parameters:
    - argc: int: Number of input arguments
    - argv: char **: The input arguments
- Returns: int: 0 in case of success, non-zero for failure

```
int main(int argc, char **argv) {
        int opt;
        char *infile = NULL;
        char *outfile = NULL;
        char *pub_key_file = "rsa.pub";
        FILE *ifp, *ofp, *pkfp;
```

```
bool verbose = false;

// Parse the input options. Store the sorting options in a set.
while ((opt = getopt(argc, argv, "bvi:n:d:s:h")) != -1) {
        switch (opt) {
                Handle different options
        }
}

// Open the private key file for reading
rsa_read_priv();

if (verbose == true) {
        printf("Print the public modulus n\n");
        printf("Print the private key\n");
        printf("Include info on number of bits along with their decimal value\n");
}

// Open the input file for reading and output for writing

// Decrypt the file using rsa_decrypt_file
rsa_decrypt_file();
```

```
        // Close the private key file
        fclose(pkfp);
        // Clear any mpz_t variables

        return 0;
}
```