Assignment 5: Public Key Cryptography Design Document

Shirin Rokni
shrokni

November 3, 2022

Citations

This PDF was generated using a LaTeXtemplate made by Jessie Srinivas (CSE13s tutor).
All pseudocode was provided by the assignment pdf, course staff (section leaders, tutors, TAs), or the GNU manuel.

Purpose

This assignment creates three programs called keygen, encrypt and decrypt. Keygen is a program that is in charge of key generation, producing RSA public and private key pairs. Security requires keeping the private key private, and the public key can be distributed widely. Any person can encrypt a message using the intended receiver's public key, but that encrypted message can only be decrypted with receivers private key. The encrypt program will encrypt files using a public key. The decrypt program will decrypt the encrypted files using the corresponding private key which consists of the public modulus and public exponent.

---

1 Test Harness

1.1 Building

to building the program, type the following in your terminal

```
make
```

to remove any compiler-generated files, type the following in your terminal

```
make clean
```

1.2 keygen command-line options

when running keygen, you can use the following flag options

```
./keygen [-vh] [-b bits] [-i iterations] [-n public key file] [-d private key file] [-s seed]
```

these are what the flags do

```
-b: specifies the minimum bits needed for public modulus n
-i: specifies the number of Miller-Rabin iterations for testing primes (default: 50)
-n pbfile: specifies the public key file (default: rsa.pub)
-d pvfile: specifies the private key file (default: rsa.priv)
-s: specifies the random seed for the random state initialization
-v: enables verbose output
-h: displays program synopsis and usage
```

1.3 encrypt command-line options

when running encrypt, you can use the following flag options

```
./encrypt [-vh] [-i input file] [-o output file]
```

these are what the flags do

```
-i: specifies the input file to encrypt (default: stdin)
-o: specifies the output file to encrypt (default: stdout)
-n: specifies the file containing the public key (dfeault:rsa.pub)
-v: enables the verbose output
-h: displays program synopsis and usage
```

1.4 decrypt command-line options

when running decrypt, you can use the following flag options

```
./decrypt [-vh] [-i input file] [-o output file]
```

these are what the flags do

```
-i: specifies the input file to decrypt (default: stdin)
-o: specifies the output file to decrypt (default: stdou)
-n: specifies the file containing the private key (default:rsa.priv)
-v: enables the verbose output
-h: displays program synopsis and usage
```

---

2 Randstate

2.1 randstate.h

The header file for randstate.c, it allows the main programs (keygen, encrypt, and decrypt) to access the functions within the randstate source code. It uses the standard library functions gmp.h and stdint.h. gmp.h is used so that all declarations needed to use GMP is able to be used. stdint.h is used to define integer types, limiting other types and macros for integer constant expressions. Essentially, it's incredibly useful for the portability of the code.

2.2 randstate.c

purpose: The function of randstate.c is the contain the implementation of the random state interface for the RSA library and number theory functions.

The standard library functions that this file uses are randstate.h and stdlib.h. randstate.h is so that the source code can be properly accessed. The stdlib.h header is used for being able to implement the srandom() function.

2.2.1 randstate_init function

purpose: initializes the random state needed for RSA key generation operations

First, you initialize the state for a Mersenne Twister algorithm. This can be done with the gmp_randint_mt GMP function. We then want to set an initial seed value into state. This can be done with the gmp_randseed_ui function since the seed value is an unsigned long integer. Finally, we want to set the seed value as the seed for a new sequence of pseudo-random integers to be returned as a randomly generated number. We can do this with the srandom() function.

2.2.2 randstate_clear function

purpose: frees any memory used by the initialized random state

For this function, all that is needed is use the gmp_randclear function to free all memory occupied by the state.

---

3 mpz context

The reason why this assignment uses the mpz variable types over the standard ones is because the security of RSA relies on large integers, so using variables that allow for multiple precision integers is very important for the functionality of the programs. Thus, some of the build-in C functions like addition and subtraction are replaced with the GNU replacements such as mpz_add and mpz_sub_ui.

Before any mpz variable is used, it must be initialized. It must also be freed at the end of each function. Thus, when mentioning any variables, be aware that it has been initialized and freed in each function.

---

4 Numtheory

4.1 numtheory.h

The header file for numtheory.c, it allows the main programs (keygen, encrypt, and decrypt) to access the functions within the numtheory source code. The standard library functions that this file uses are stdbool.h, stdint.h, stdio.h, and gmp.h. It uses the stdbool.h header to permit the usage of booleans, which is the type of one of the written functions. Another library function is stdint.h, which is necessary to make the code portable. Then, the stdio.h library function to print strings, integers and characters on the output screen. Finally, the gmp.h library function is essential so that all declarations needed to use GMP are able to be used.

4.2 numtheory.c

purpose: This file contains the implementation of the number theory functions. Such functions are gcd, mod_inverse, pow_mod, is_prime, and make_prime.

The standard library functions that this file uses are stdbool.h, stdint.h, stdio.h, numtheory.h and randstate.h. stdbool.h is a necessary library function because of a function that outputs a boolean type. stdint.h is necessary in order to make the code portable. The library function stdio.h is necessary in order to use various variable types and function types. The numtheory.h and randstate.h use quotations when they are being called so that the compiler can search the current directory. These header files are being used so that the current file can be accessed by its header file and so that random values can be initialized.
At the top of the file, it is necessary to declare the pseudo-random numbers in GMP.

4.2.1 gcd

purpose: computes the greatest common divisor of two values a and b, storing the value of the computed divisor d.

The arguments of the gcd function are d, a, b. First, the function needs to make copies of a and b in order to manipulate them. This is because the variable type mpz_t behaves like a pointer, and thus changing them in a separate function is manipulating their values directly. Thus, copies of a and b are necessary, which will be called A and B. So, the function defines A and B as mpz_ variable types and then initializes them with the command mpz_inits.

Then, the function checks for if B is equal to 0 or not, so it uses the mpz_cmp_ui command in order to

```
GCD(a, b)
1  while b ≠ 0
2        t ← b
3        b ← a mod b
4        a ← t
5  return a
```

Figure 1: This pseudo code was given by Ethan L. Miller and is used as a basis for this function

compare B to a unsigned integer. Inside of the while loop, the function defines a temporary variable with mpz_t in order for a swap of variable values to succesfully take place. Now, the swapping occurs. Using the command mpz_set, temp now has the value of B. Then, by using the command mpz_mod, B is redefined with A mod B. Next, A has it's value swapped with temp.

At the end of the while loop, the temp variable is cleared with mpz_clear. Outside of the while loop, A's value is set to d's value with mpz_set to act as the return value, and the A and B variable values are then cleared.

4.2.2 mod_inverse

purpose: computes the inverse i of a modulo n. In the event that a modular inverse cannot be found, set i to 0.

```
MOD-INVERSE(a, n)
 1  (r, r') ← (n, a)
 2  (t, t') ← (0, 1)
 3  while r' ≠ 0
 4        q ← ⌊r/r'⌋
 5        (r, r') ← (r', r − q × r')
 6        (t, t') ← (t', t − q × t')
 7  if r > 1
 8        return no inverse
 9  if t < 0
10        t ← t + n
11  return t
```

Figure 2: This pseudo code was given by Ethan L. Miller and is used as a basis for this function

The arguments that this function accepts are o, a and n. Since none of the arguments are being manipulated, there is no need to make a copy of them. In looking at the pseudo code, it becomes obvious that this function needs to initialize R, R_prime, T, and T_copy. However, it must also define R_copy and T_copy as the pseudo code does not take into account how C cannot do parallel assignments.

Then, the function enters a while loop which checks for if R_prime is not equal to 0. In the loop, it first defines a variable Q with the floor division of R and R_prime, and this is done through the command mpz_fdiv_q. Then, it sets R to R_prime and R_prime to R with the mpz_set command. Next, it multiplies Q and R_prime and sets the output to R_prime with mpz_mult. It then subtracts R_prime from R_copy and sets the output to R_prime with the mpz_sub command. Next, it sets T to T_copy and T_prime to T with the mpz_set command. And then it sets T_prime to the multiplication of Q and T_prime and the subtraction of T_prime from T_copy with mpz_mult and mpz_sub.

The while loop terminates and then does a check to see if R is greater than 1 with the mpz_cmp_ui command, with the expression that then terminates being that T is set to 0 as that is the failure of a modular

inverse signal. Then, it checks for if T is less than 0, which if so would then increment T.

Finally, T is set to the output variable o with mpz_set and all of the variables are cleared and with mpz_clears command.

4.2.3 pow_mod

purpose: Performs fast modular exponentiation, computing base raised to the exponent power modulo modulus, and storing the compound result in the out argument.

```
POWER-MOD(a, d, n)
1  v ← 1
2  p ← a
3  while d > 0
4      if ODD(d)
5          v ← (v × p) mod n
6      p ← (p × p) mod n
7      d ← ⌊d/2⌋
8  return v
```

Figure 3: This pseudo code was given by Ethan L. Miller and is used as a basis for this function

The arguments that this function accepts are o (out), a (base), d (exponent), and n (modulus). Since the only argument being manipulated is d, it will be the only one that needs to be made a copy of. Based off of the pseudo code, variables also need to be made for v and p which will be manipulated throughout the function.

First, the unsigned integer 1 needs to be set to the v variable. Then, the a variable is set to the p variable. These steps are both done with the mpz_set command.

The while loop is initialized with the condition that d is to be greater than 0, so the mpz_cmp_ui command is necessary here.

Inside of the while loop, there is an if condition which checks for if d is an odd number. This can be checked with the mpz_cmp_ui command as it directly checks for if an mpz variable is odd. Inside of the if statement, it redefines v with the multiplication of itself and p, as well as itself mod n.

Outside of the if statement, p is redefined as the multiple of itself, as well as itself mod n. Then, d is redefined as the floor division of itself and 2. This can be achieved with the mpz_fdiv_q_ui command.

Outside of the while loop, v must be set to o, as o represents the output. Finally, all of the variables are cleared.

4.2.4 totient

purpose: To compute the totient and ensure that the r variable is odd, and the s variable such that s equals the exponent of 2 multiplied by a large odd value. The totient denotes the number of positive integers up on a given integer n that are relatively prime to n.

The arguments of this function are r, s, and n, which represent the odd value, the exponent, and the given integer n. Two new arguments need to be initialized, one that needs to act as a copy for the argument n (as it is being manipulated), and the other to act as a variable to the arithmetic functions on. This temp_mod variable is used to check if n_copy is even, and will thus use the mpz_even_p command.

A while loop is initialized that has the condition where the temp_mod variable is equal to 0. Within the loop, it increments the s variable. It then sets n_copy to the floor division of itself and 2. Then, temp_mod is redefined as n_copy modulus 2.

Outside of the while loop, n_copy is set to r as the manipulation done onto n represents the creation of an odd integer r. Finally, all of the mpz integers are cleared.

4.2.5 is_prime

purpose: Conducts the Miller-Rabin primality test to indicate whether or not n is prime using iters number of Miller-Rabin iterations. This function is needed when creating two large primes p and q in RSA, verifying if a large integer is prime.

```
MILLER-RABIN(n, k)
 1  write n − 1 = 2^s r such that r is odd
 2  for i ← 1 to k
 3      choose random a ∈ {2, 3, ..., n − 2}
 4      y = POWER-MOD(a, r, n)
 5      if y ≠ 1 and y ≠ n − 1
 6          j ← 1
 7          while j ≤ s − 1 and y ≠ n − 1
 8              y ← POWER-MOD(y, 2, n)
 9              if y == 1
10                  return FALSE
11              j ← j + 1
12          if y ≠ n − 1
13              return FALSE
14  return TRUE
```

Figure 4: This pseudo code was given by Ethan L. Miller and is used as a basis for this function

The arguments that this function accepts are n (a inputted variable) and iters (the number of iterations). Based off of the pseudo code, some necessary variables are a, s, r, y and j. Some less obvious but still necessary variables to use are n_minus_3, two and n_minus_1. The purpose and usage of these variables will be explained shortly.

Before we initialize and assign any variables, there needs to be a few base cases and catches in order to optimize the code. The base cases will be for the values 1, 2 and 3. The function will return false if the inputted value is 1 because 1 is not a prime vale. It will return true if the inputted value is 2 or 3 because they are prime values. This comparison can be achieved with the mpz_cmp_ui command. Then, the function needs to look for even values and return false because all even numbers are not prime. This can cut the time significantly because half of all values are even, and immediately returning false will optimize the run time.

First, the variables n_minus_3 and n_minus_1 need to be defined, so with the mpz_sub_ui command, they will be defined with n minus 3 and n minus 1. The reason for making separate variables for them, is because the totient function requires that the input be a variable - 1, and later on inside the Miller-Rabin portion of the function, a random variable a needs to be generated with a specific range in mind, with the upper range being n minus 3.

Then, the totient function is called with the inputs r, s, and n_minus_1. After the totient function finishes running, r and s will be defined with different values.

Two more steps need to take place before the Miller-Rabin portion can begin, and that is to define the two variable with the value 2, and to decrement s by 1. These steps are necessary because 2 is an argument in one of the function calls, and the decrement of s is a while condition.

A for-loop condition is initialized with the initialization statement being the variable i, the condition being for as long as i is less than the iters variable, and the update statement being that i is incremented by 1. Inside the for-loop, a needs to be defined as a random value between a specific range. So, the command mpz_urandomm is helpful as it generates a uniformly distributed random integer through a specified range. The range is from the state to n_minus_3 because this range is two decrements below the range given in the pseudo code. The reason for doing this is so that the inputted range is well within the boundaries of the urandomm range. Then, a must be incremented by 2 to take into consideration the decrement.

Then, y must be redefined as the power-pod of the arguments a, r, and n.

Next, an if statement is initialized that checks for if y is not equal to 1 as well as checks for if y is not equal to n minus 1. Thus, the mpz_cmp_ui command is useful, as well as the n_minus_1 variable.

Inside of the if statement, j is set to the unsigned integer 1. Next, a while loop is initialized with the condition that j be less than or equal to the decrement of s, and the y not be equal to n minus 1. Inside of the while loop, the power-mod function is called again where y must be redefined as the power mod of the arguments y, 2, and n. This is where the two variable is used.

Then, an if condition is presented where it checks for if y is equal to 1, then it returns false. However, since we are using mpz variables, they must be cleared before the false bool is returned.

Outside of the if statement, the j variable is to be incremented by 1 using the mpz_add_ui command.

Then, another if condition checks for if y does not equal n minus 1, which then also clears all of the mpz variables and returns false if true.

Outside of all of the conditions and statements, all of the variables are cleared and the function returns true.

4.2.6 make_prime

purpose: This function generates a new prime number stored in p. The generated prime should be at least bits number of bits long. The primality of the generated number should be tested using is_prime() using iters numbers of iterations.

The arguments that this function accepts are p (the output prime number), bits, and iters (iterations).

Since this function generates a new number, the mpz_urandomb command must be used in order to generate a uniformly distributed random integer. The arguments of this command will be rand_var, state, and bits - 1. This is so that the function can generate values from 0 to $2\hat{}$(bits - 1). Then, the function sets the 0th bits to 1 to ensure that the number will be odd at all times. This can be done with the mpz_setbit command with the arguments p and 0. Finally, we want the number generator to have at least bit number of bits, so we can set p to bits - 1. This adds $2\hat{}$(bits - 1) to the current number and can be done with the mpz_setbit command. In order to have the function generate random numbers until the generated value is successfully prime, this will all be under a while loop checking for if p is not prime. This will be carried out with the is_prime function from earlier with the arguments p and iters. Thus, once the prime value is generated, the function terminates.

---

5 rsa

5.1 rsa.h

The header file for rsa.h. It allows the main programs (keygen, encrypt, and decrypt) to access the functions within the rsa source code. The standard library function that this file uses are stdbool.h, stdint.h, stdio.h and gmp.h. It uses the stdbool.h header to permit the usage booleans, which is the type of one of the used functions as well as one of the written functions. Another library function is stdint.h, which makes the code portable. The stdio.h library function is used to print stringers, integers and characters on the output screen. Finally, the gmp.h library function is essential so that all declarations needed to use GMP are able to be used.

5.2 rsa.c

purpose: The file contains the implementation of the RSA library. Such functions are rsa_make_pub, rsa_write_pub, rsa_read_pub, rsa_make_priv, rsa_read_priv, rsa_encrypt, rsa_encrypt_file, rsa_decrypt, rsa_decrypt_file, rsa_sign and rsa_verify.

5.2.1 rsa_make_pub

purpose: This function creates parts of a new RSA public key: two large primes p and q, their product n, and the public exponent e.

First, the function needs to decide which number of bits to go into both p and q. Since the number of bits for q is the remainder of how many bits p used, most of the calculations will occur in obtaining p. The number of bits for p will be a manipulation of a random number, and that number will be generated with the random() command. In order to manipulate it, we will take random() modulus n_bits divided by 2.

This is because when the given range is subtracted by n_bits, the upper range is n_bits divided by 2. Then, p_bits must add n_bits / 2 to account for the offset of changing the range. Thus, q_bits will simply become the remainder of n_bits subtracted by p_bits.

Then, the function will create primes p and q using the function make_prime(), using the parameters p and q (as the output), p_bits and q_bits (as the bits) and iters (as the iterations).

Since one of the arguments of this function, n, stores the product of p and q, the function will then use the mpz_mul command with the arguments n, p and q to define n as the product of p and q.

Next, the function will compute the totient using the least common multiple. This can be achieved by finding the product of (p - 1) and (q - 1). Thus, using mpz_sub_ui, p and q will both be subtracted by 1 and then be multiplied with the mpz_mult command. The output will be set to a variable called totient. Since the arguments p and q were manipulated, we must bring them back to their original value, so they will both be incremented back using mpz_add_ui.

Now, the function must find a suitable public exponent e. This can be achieved with a while loop that iterates until the greatest common denominator of a random value and the totient is 1. This condition will represent finding a number coprime with the totient.

So, the while loop will have a condition of while a variable exit is equal to 0. Inside of the loop, a random number will be generated using mpz_urandomb with the arguments exp (a random value that acts as the public exponent e), state and n_bits. Then, the greatest common denominator of exp and totient will be put into a new variable greatest. If the greatest variable equals 1, then that means that the co prime has been successfully found and exit can now equal 1 and the while loop will terminate.

Finally, e will be set with exp (ultimately finding the suitable public exponent e) and all of the variables will be cleared.

5.2.2 rsa_write_pub

purpose: This function writes a public RSA key to pbfile. The format of public key should be n, e, s, then the username, each of which are written with a trailing newline. The values of n, e, and s should be written as hex strings.

The arguments of this function is n (the public modulus), e (the public exponent), s (the signature of the username), username (the username that was signed as s), and pbfile (the file that the the public key contents are written to).

In order to write mpz variables to a file, they must use a specialized command. This command is gmp_fprintf. The arguments of this function are pbfile (the file to be written to), 'Zx\n' (the hex string type the values should be written as), and the value. These values will be n, e, and s in their own respective gmp_pfrintf statements. At the end of these, will be a fprintf statement, as username is a character, not a mpz_t variable type. It will also use a 's\n' which is the format specifier for strings. The reason for doing \n at the end of each format specifier is to insert the newline character.

5.2.3 rsa_read_pub

purpose: This function reads a public RSA key from pbfile. The format of public key should be n, e, s, then the username, each of which should have been written with a trailing newline. The values n, e, and s should have been written as hex strings.

In order to read mpz variables to a file, they must use a specialized command. This command is gmp_fscanf. The arguments of this function are pbfile (the file to be read from), 'Zx\n' (the hex string type the values should be read from), and the value. These values will be n, e, and s in their own respective gmp_fscanf statements. At the end of these, will be a fscanf statement, as username is a character, not a mpz_t variable type. It will also use a 's\n' which is the format specifier for strings. The reason for doing \n at the end of each format specifier is to insert the newline character.

5.2.4 rsa_make_priv

purpose: This function creates a new RSA private key d given primes p and q and public exponent e.

Since the arguments p and q are being manipulated in this function, copies will be made to represent them. Similar to the rsa_make_pub function, the totient can be found through multiplying (p - 1) and (q - 1) as that is another way of representing the least common multiply. This output of their product is set to a variable totient.

Then, this function must compute the inverse of e modulo totient, and this can be done with the command mod_inverse with the arguments d (will store the private key), e, and totient. Finally, all of the mpz variables are to be cleared.

5.2.5 rsa_write_priv

purpose: This function writes a private RSA key to pvfile. The format of a private key should be n then d, both of which are written with a trailing newline. Both these values should be written as hexstrings.

In order to write mpz variables to a file, they must use a specialized command. This command is gmp_fprintf. The arguments of this function are pvile (the file to be written to), 'Zx\n' (the hex string type the values should be written as), and the value. These values will be n and d in their own respective gmp_pfrintf statements. The reason for doing \n at the end of each format specifier is to insert the newline character.

5.2.6 rsa_read_priv

purpose: This function reads a public RSA key from pvfile. The format of a private key should be n then d, both of which should have been written with a trailing newline. Both of these values should be written as hexstrings.

In order to read mpz variables from a file, they must use a specialized command. This command is gmp_fscanf. The arguments of this function are pvile (the file to be read from), 'Zx\n' (the hex string type the values should be read as), and the value. These values will be n and d in their own respective gmp_pfrintf statements. The reason for doing \n at the end of each format specifier is to insert the newline character.

5.2.7 rsa_encrypt

purpose: This function performs RSA encryption, computing ciphertext c by encrypting message m using public exponent e and modulus n.

Since the definition of encryption with RSA is $E(m) = c = m^e (mod\ n)$, it is clear that the pow_mod function is necessary. Thus, this function only needs to call pow_mod with the arguments as c (ciphertext), m (message), e (public exponent), and n (public modulus)

5.2.8 rsa_encrypt_file

purpose: This function encrypts the contents of infile, writing the encrypted contents to outfile. The data in infile should be encrypted in blocks.

The arguments that this function has is infile (the input file to encrypt), outfile (the output file to write the encrypted input to), n (the public modulus), and e (the public exponent).

So first, the function needs to determine the size of the blocks, and it can do this with the command mpz_sizeinbase with base 2. This can achieve the same functionality as $log2$. Then, it will manipulate this variable k by decrementing it by 1 and dividing it by 8 to achieve the final block size.

Next, the function needs to dynamically allocate an array that can hold k bytes. To do this, we will be using calloc since we want to explicitly identify the number of blocks to be allocated as well as the size of each block. thus, the pointer variable block as the variable type uint8_t will be defined as the array type

uint8_t with the arguments of k, and sizeof(uint8_t) inside of the calloc.

Next, the function sets the zeroth byte of the block to 0x0FF to prepend the workaround byte that we need.

Before we process in the infile, the function initializes a variable 'process' to act as the check for the while loop. If it is 0, then the while loop continues, if it is 1, then it terminates.

The while loop then initializes with the condition of 'process' equaling 0. Inside of it, the function first sets a variable 'j' to the number of items succesefully read. This can be done with the fread() function, with its arguments being the allocated block starting from index 1, the size which is the sizeof uint8_t, the number of items being k - 1, and the pointed stream is infile.

If all of the items in the file has been read, then j would be equal to 0, so an if condition is put inside of the while loop which checks for that case. The expression inside of the if statement being that process is redefined to equal 1, terminating the while-loop.

Otherwise, that means that there are still unprocessed bytes in the infile. Thus, the function uses mpz_import to convert the read bytes into the argument n. This can be done by calling mpz_import with the arguments m (the returned operand), j + 1 (the count as it needs to be incremented each iteration), 1 (the order), size of uint8_t (the size), 1 (the endian), 0 (the nail), and block (the pointer).

Then, the function needs to encrypt m with rsa_encrypt(), the arguments being encrypted (the ciphertext), m (the message), e (the public exponent) and n (the public modulus).

Next, the function needs to write the encrypted number to the outfile. This can be done with the gmp_fprintf command with the intended file to be written to as outfile and the variable being written as the ciphertext. Since the encrypted number is intended to be written as a hex string, the format specifier will be followed with 'Zx'.

Finally, the block will have to be freed using the free() command and all the mpz variables will need to be cleared.

5.2.9 rsa_decrypt

purpose: This function performs RSA decryption, computing message m by decrypting ciphertext c using private key d and public modulus n

Since the definition of encryption with RSA is $D(c) = m = c^d (mod\ n)$, it is clear that the pow_mod function is necessary. Thus, this function only needs to call pow_mod with the arguments as m (the storage of the decrypted message), c (the ciphertext to decrypt), d (the private key), and n (the public modulus).

5.2.10 rsa_decrypt_file

purpose: Decrypts the contents of infile, writing the decrypted contents to outfile. The data in infile should be decrypted in blocks.

The arguments that this function has is infile (the input file to decrypt), outfile (the output file to write the decrypted input to), n (the public modulus) and d (the private key).

So first, the function needs to determine the size of the blocks, and it can do this with the command mpz_sizeinbase with base 2. This can achieve the same functionality as $log2$. Then, it will manipulate this variable k by decrementing it by 1 and dividing it by 8 to achieve the final block size.

Next, the function needs to dynamically allocate an array that can hold k bytes. To do this, we will be using calloc since we want to explicitly identify the number of blocks to be allocated as well as the size of each block. thus, the pointer variable block as the variable type uint8_t will be defined as the array type uint8_t with the arguments of k, and sizeof(uint8_t) inside of the calloc.

Before we process in the infile, the function initializes a variable 'process' to act as the check for the while loop. If it is 0, then the while loop continues, if it is 1, then it terminates.

The while loop then initializes with the condition of 'process' equaling 0. Inside of it, the function first scans in a hexstring, saving the hexstring as mpz_t c. This can be done with the gmp_fscanf command, the arguments being infile (the file it is scanning) and c (the variable it is saved as). Since the decryped block is intended to be written as a hex string, the format specifier will be followed with 'Zx'.

If all of the items in the file have been read, then the 'done' variable would equal -1. So, there is an if-statement that checks for this possibility, and then redefines process to equal 1, termianting the while loop.

Otherwise, that still means that there are unprocessed bytes in the infile. Then the function decrypts the message using rsa_decrypt with the arguments being the message, c (the output file to write decrypted input to), d (the private key), and n (the public modulus). This function essentially decripts the input file and stores it in the variable 'message'.

Then, using mpz_export(), the function converts c back into bytes, storing them into the allocated block. The arguments would be block (the returned operand), the size_t size, 1 (the order), sizeof uint8_t size, 1 (the endian), 0 (the nails), and message (the pointer).

Next, the function writes out j - 1 bytes. Thus, we use the fwrite() function with the arguments block[1] (starting from index 1 of block), size of uint8_t as the size, j - 1 (the number of bytes to write out), and outfile (the streamed file).

Outside of the while-loop, the allocated memory is freed using free() and all of the mpz variables are cleared.

5.2.11 rsa_sign

purpose: Performs RSA signing, producing signature s by signing message m using private key d and public modulus n.

Since the definition of signing with RSA is $S(m) = s = m^d (mod\ n)$, it is clear that the pow_mod function is necessary. Thus, this function only needs to call pow_mod with the arguments as s (the storage of the signed message), m (the message to sign), d (the private key), and n (the public modulus).

5.2.12 rsa_verify

purpose: Performs RSA verification, returning true if signature s is verified and false otherwise. The signature is verified if and only if t is the same as the expected message m.

The arguments this function accepts are m (the expected message), s (the signature to verify), e (the public exponent), and n (the public modulus).

Since the definition of t is $V(s) = s^e (mod\ n)$, it is clear the the pow_mod function is necessary. Thus, after initializing the variable t, the function calls pow_mod with the arguments t, s, e, and n.

The signature is only verified if t is equivalent to m, so there is an if-statement that checks if they are equivalent using the command mpz_cmp. Inside of the if-statement, it clears all of the mpz variables and returns true. If the two variables are not equivalent, then the function returns false.

---

6 Keygen

6.1 keygen.c

purpose: This file contains the implementation and main() function for the keygen program. The keygen program will be in charge of key generation, producing RSA public key pairs, and producing RSA private key pairs.

The standard library function this file includes are numtheory.h, rsa.h, and randstate.h. The numtheory.h, rsa.h and randstate.h headers use quotations when they are being called so that the compiler can search the current directory. These header files are being used so that the current file can be accessed by its header file and so that random values can be initialized. Another standard library function used is unistd.h, which is needed in order to use the getopt function. Some other library functions are time.h (to use the time() function), sys/stat.h (to use fchmod), and stdlib.h (to use strtoul, genv, and other functions). This file also defines the variable OPTIONS with all of the command line flag options. At the top of the file, there is also a line initializing randstate in order to use the random number generation features.

First we need to define a few default variables so that if the user doesn't enter in any command-line options, the program can still run on default values. The character type file name 'public' that will be opened to access the public key will be set to "rsa.pub", and the character type file name 'private' that will be opened to access the private key will be set to "rsa.priv". The minimum bits 'min_bits' needed to open the public modulus will be set to 1024 and the number of Miller-Rabin iterations 'iters' will be set to 50.

Then the main function initializes accepting the arguments argc and argv. Within the next few lines will be a few more variable definitions, as these ones don't need to be outside of the main function. The public key 'public_key' and private key 'private_key' are then initializes as FILE types which are necessary to use the fopen() function. Then, opt and verbose are both defined as int variables and equaled to 0. The 'opt' variable is necessary for using getopt(), and the 'verbose' variable is necessary in calling the verbose. The seed variable is also initialized as the seconds since UNIX epoch, and is thus defined as time(NULL).

Now, there is a while-loop which checks for if there are any command-line inputs the user wishes to enter. If the 'b' flag is used with proceeding values, then the program will set the 'min_bits' variable to the user's input using the optarg command. However, if the user enters in a value less than 50 or greater than 4096, the program will terminate with a helpful error. If the user uses the 'i' flag, then the program will set the 'iters' variable to the value the user input after the flag using the optarg command. If the user uses the 'n' flag with a value after it, then the user can specify the public key file. If the user uses the 'd' flag with a value proceeding it, then theu ser can specify the private key file. If the user uses the 's' flag with a value proceeding it, then the 'seed' variable will be replaced with their input. If the user simply calls the 'v' flag, then it will enable the program to print a verbose output. If the user calls the 'h' flag, then the usage statement will print with a nonzero exit code. Each of these case statements within the switch(opt) statement end with breaks to ensure that multiple flags do not run at once.

Outside of the while loop, the file opens the public key for reading and writing. This means that the 'public_key' will be equal to fopen() with the arguments 'public' and the mode 'w'. This was the reason for making 'public' a character type variable, it is to be used as the filename for the fopen() function. In the case where there is no public_key, the program will print a helpful error and then exit the program.

Then, the file opens the private key for reading and writing. This means that 'private_key' will be equal to fopen() with the arguments 'private' and the mode 'w'. This was the reason for making 'private' a character with a character type variable, it is to be used as the filename for fopen(). In the case where there is no private_key, the program will print a helpful error and then exit the program.

Then, the file sets the permission for the user, and denies the permission for anyone else. This means using the fchmod() statement with the first argument being the output of fileno(private_key) and the second argument being 0600. The fileno() function examines the argument private_key and return the integer file descriptor used to implement the stream.

Next, the file initializes the random state using the seed 'seed', which means that it uses the randstate_init function from the rsa.h library and uses 'seed' as the argument.

Then the program needs to make the public and private keys. So, it defines and initializes all of the various variables that can be taken as arguments from the rsa_make_pub and rsa_make_priv functions. Then, it calls those functions with their designated parameters.

Then the file needs to get the current user's name as a string, which means that it needs to define a new character type variable 'user' with the output of getenv("USER"). The getenv() function searches for the environment variable, so when the argument is "USER", it searches for the user's name. Next, it converts the username into a mpz_t type variable, which means using the mpz_set_str command, specifying the base to 62. To compute the signature of the username, the file uses rsa_sign with the arguments sign, username, d and n. The 'sign' variable will then have the signature of the username.

Now, the file needs to write the computed public and private key to their respective files. This can be done with the rsa_write_pub and rsa_write_priv functions. The public key function will have the arguments n, e, sign, user and public_key and the private key function will have the arguments n, d, and private_key.

In this portion of the file, there should be an if statement that checks for if verbose is equivalent to 1 or 0, and if it is equal to 1, then it prints information about the mpz_t values and their number of bits. To print the values, all that needs to be done is to use gmp_printf with the format specifier 'Zd\n'. To get the number of bits, the file uses mpz_sizeinbase with the base size of 2 and the first argument being whichever variable is to be printed.

Then, the file needs to close the public and private keys, which means useing the functions fclose() on the

'public_key' and 'private_key' files. We also must clear the random state with the randstate_clear() function. Finally, all of the mpz variables are cleared and the file returns with a zero. Opt and verbose are both defined as int variables and equaled to 0.

---

7 Encrypt

purpose: This file contains the implementation and main() function for the encrypt program. The encrypt program will be in charge of encrypting files using the a public key.

The standard library function this file includes are numtheory.h, rsa.h, and randstate.h. The numtheory.h, rsa.h and randstate.h headers use quotations when they are being called so that the compiler can search the current directory. These header files are being used so that the current file can be accessed by its header file and so that random values can be initialized. Another standard library function used is unistd.h, which is needed in order to use the getopt function. This file also defines the variable OPTIONS with all of the command line flag options. At the top of the file, there is also a line initializing randstate in order to use the random number generation features.

First, we need to define a few variables to act as their default setting before the user has the possibility to change them or before they get called further down in the main function. The 'opt' variable is necessary for using getopt(), and the 'verbose' variable is necessary in calling the verbose. Then, the character type variable is used for a 'user' which has any array value, but must be set to an array of 0 values. Then, the public key file character type 'public' is defined as being set to 'rsa.pub'. The reason for needing this variable type to be a character, is so that optarg allows the user to change it if they want the file destination to change. Next, some FILE type variables are set: 'input_file' which is set to stdin, 'output_file' which is set to stdout, and 'public_key' which is necessary for the fopen() function.

Now, there is a while-loop which checks for if there are any command-line inputs the user wishes to enter. The 'i' flag with proceeding values specifies the input file to encrypt and will allow the user to specify which file they want encrypted, otherwise it would be a command-line input. This can be done through redefining 'input_file' with the output of fopen() with the arguments optarg and the mode 'r'. The 'o' flag with proceeding values specifies the output file to encrypt and allows the user to specify which file they want the output file to be. This can be done through redefining 'output_file' with the output of fopen() with the arguments of optarg and the mode 'w'. If the user calls the 'n' flag with proceeding values, they will specify the file containing the public key and this is done through redefining 'public' with optarg. If the user calls the 'v' flag, it will enable the verbose output and redefine the 'verbose' variable to 1. If the user calls the 'h' flag, it will enable the synopsis and usage output and redefine the 'usage' variable to 1. Each of these case statements within the switch(opt) statement end with breaks to ensure that multiple flags do not run at once.

Outside of the while loop, the file opens the public key for reading. This means that the 'public_key' will be equal to fopen() with the arguments 'public' and the mode 'w'. This was the reason for making 'public' a character type variable, it is to be used as the filename for the fopen() function. In the case where there is no public_key, the program will print a helpful error and then exit the program.

Then, the file reads the public key. It does by defining and initializing all of the necessary parameters the rsa_read_pub takes, and then calling the function with those arguments.

In this portion of the file, there should be an if statement that checks for if verbose is equivalent to 1 or 0, and if it is equal to 1, then it prints information about the mpz_t values and their number of bits. To print the values, all that needs to be done is to use gmp_printf with the format specifier 'Zd\n'. To get the number of bits, the file uses mpz_sizeinbase with the base size of 2 and the first argument being whichever variable is to be printed.

Now, the file converts the username into a mpz_t type variable, which means that it uses the mpz_set_str command with the arguments username, user and 62 as the base. This means that the 'user' variable that was used as the username argument for rsa_read_pub will be reset to a mpz_t type under the name 'username'.

Here, the file does an RSA signature check by verifying that the signature is the same as the username. This can be done with the rsa_verify function from rsa.h. So, it checks for if that function returns a false bool, and if so, it will print a helpful message informing the user that they could not verify their signature, clear all of the mpz type variables, close all of the files, and then return a nonzero integer.

Here, we encrypt the file. This means that we use the rsa_encrypt_file function with the arguments input_file, output_file, n and e. This means that we are encrypting the input file into the output file.

Finally, we are cleaning up all of the loose ends by closing all of the files, clearing all of the mpz type variables, and returning a zero value.

---

8 Decrypt

8.1 decrypt.c

purpose: This file contains the implementation and main() function for the decrypt program. The decrypt program will be in charge of decrypting the encrypted files using the corresponding private key.

The standard library function this file includes are numtheory.h, rsa.h, and randstate.h. The numtheory.h, rsa.h and randstate.h headers use quotations when they are being called so that the compiler can search the current directory. These header files are being used so that the current file can be accessed by its header file and so that random values can be initialized. Another standard library function used is unistd.h, which is needed in order to use the getopt function. This file also defines the variable OPTIONS with all of the command line flag options. At the top of the file, there is also a line initializing randstate in order to use the random number generation features.

First, we need to define a few variables to act as their default setting before the user has the possibility to change them or before they get called further down in the main function. The 'opt' variable is necessary for using getopt(), and the 'verbose' variable is necessary in calling the verbose. Then, the private key file character type 'private' is defined as being set to 'rsa.priv'. The reason for needing this variable type to be a character, is so that optarg allows the user to change it if they want the file destination to change. Next, some FILE type variables are set: 'input_file' which is set to stdin, 'output_file' which is set to stdout, and 'private_key' which is necessary for the fopen() function.

Now, there is a while-loop which checks for if there are any command-line inputs the user wishes to enter. The 'i' flag with proceeding values specifies the input file to decrypt and will allow the user to specify which file they want decrypted, otherwise it would be a command-line input. This can be done through redefining 'input_file' with the output of fopen() with the arguments optarg and the mode 'r'. The 'o' flag with proceeding values specifies the output file to decrypt and allows the user to specify which file they want the output file to be. This can be done through redefining 'output_file' with the output of fopen() with the arguments of optarg and the mode 'w'. If the user calls the 'n' flag with proceeding values, they will specify the file containing the private key and this is done through redefining 'private' with optarg. If the user calls the 'v' flag, it will enable the verbose output and redefine the 'verbose' variable to 1. If the user calls the 'h' flag, it will enable the synopsis and usage output and redefine the 'usage' variable to 1. Each of these case statements within the switch(opt) statement end with breaks to ensure that multiple flags do not run at once.

Outside of the while loop, the file opens the private key for reading. This means that the 'private_key' will be equal to fopen() with the arguments 'private' and the mode 'w'. This was the reason for making 'private' a character type variable, it is to be used as the filename for the fopen() function. In the case where there is no private_key, the program will print a helpful error and then exit the program.

Then, the file reads the private key. It does by defining and initializing all of the necessary parameters the rsa_read_priv takes, and then calling the function with those arguments.

In this portion of the file, there should be an if statement that checks for if verbose is equivalent to 1 or 0, and if it is equal to 1, then it prints information about the mpz_t values and their number of bits. To print the values, all that needs to be done is to use gmp_printf with the format specifier 'Zd\n'. To get the number of bits, the file uses mpz_sizeinbase with the base size of 2 and the first argument being whichever

variable is to be printed.

Now, the file converts the username into a mpz_t type variable, which means that it uses the mpz_set_str command with the arguments username, user and 62 as the base. This means that the 'user' variable that was used as the username argument for rsa_read_priv will be reset to a mpz_t type under the name 'username'.

Here, we decrypt the file. This means that we use the rsa_decrtot_file function with the arguments input_file, output_file, n and d. This means that we are decrypting the input file into the output file.

Finally, we are cleaning up all of the loose ends by closing all of the files, clearing all of the mpz type variables, and returning a zero value.