Program Task Description:

The purpose of this program is to take a dive into the world of cryptography, and learn the implementation of public and private key cryptography, whilst making use of the RSA algorithm. The program has three tasks, creating a key generator, an encryptor, and a decryptor as well. The key generator creates a public key file and a private key file, which are used by encrypt and decrypt accordingly.

Pseudocode and/or Explanations of the various program parts:

GNU Multiple Precision Arithmetic

randstate_init(uint64_t seed)
    Use Mersenne Twister algorithm to initialize the global random state
        Use the given seed parameter for the function as the random seed
    Call gmp_randinit_mt(state) and gm_randseed_ui(state, seed)

randstate_clear(void)
    Call gmp_randclear() on the global variable *state* to clear and free all the memory associated with it

Number Theoretic Functions

Modular Exponentiation
pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus)
    Initialize mpz_t variables and temp variables
    While (exponent > 0)
        If exponent is odd // *call mpz_odd_p() to check if odd*
            Multiply v and p and store in v
            mod(v, v, mod)
        Multiply p*p and store in p_squared
        mod(p, p_squared, mod)
        Floor division of exponent / 2 // *call mpz_fdiv_q_ui()*
    Set out equal to v // *use mpz_set()*

Primality Testing
is_prime(mpz_t n, uint64_t iters)
    Initialize mpz_t variables

```
Base cases:
If n = 0 or n = 1
        mpz_clear(all mpz variables)
        Return false
If n = 2 or n =3
        mpz_clear(all mpz variables)
        Return true
If n = even
        mpz_clear(all mpz variables)
        Return false

While (r is still even) // While r i still even, keep dividing r by 2
        Divide r / 2 // use mpz_fdiv_q_ui()
        Add one to s
For loop that runs i <= iters number of iterations
        Call mpz_urandomm with a, state, n - 3 as parameters
        Add 2 to a // This sets a to range [2, n-2]
        Call pow_mod() with y, a, r, n as parameters

        If ((y != 1) && (y!= n-1))
                Set j = 1
                While ((j <= s-1) && y != n-1) // use mpz_cmp
                        Call pow_mod() with y, y, temp_exponent, and n as parameters
                        If (y is 1)
                                mpz_clear(all mpz variables)
                                Return false
                        Add 1 to j
                If y != n-1
                        mpz_clear(all mpz_variables)
                        Return false
mpz_clear(all mpz_variables)
Return true
```

make_prime(mpz_t p, uint64_t bits, uint64_t iters)

Set temp_p equal to the generation of a random number // *use mpz_urandomb() with state and bits as parameters as well*

While (!isprime() or temp_p is not at least bits amount of bits long)
Generate random temp_p // *using mpz_urandomb(temp_p, state, bits)*

Set temp_p to p
Clear temp_p

Modular Inverses

gcd(mpz_t d, mpz_t a, mpz_t b)
// *Implement the pseudocode given in asgn6.pdf*
While b not equal to 0 // *use mpz_cmp_ui to compare b and 0*
Store b in a temp variable // *mpz_set(t, b_temp)*
Set b = a mod b // *call mpz_mod on b_temp, a_temp, and t*
Store the temp variable into a // *mpz_set t to a_temp*
Store a_temp in d // *use mpz_set()*
mpz_clear(all mpz variables)

Mod_inverse(mpz_t i, mpz_t a, mpz_t n)
While (r' != 0) // *use mpz_cmp_ui*
Floor divide (r / r') and store in q // *use mpz_fdiv_q()*
Set x to r
Set r to r'
Multiply q * r' and store in temp // *use mpz_mul()*
Set y to t
Set t to t'
Multiply q * t' and store in temp_two
Subtract y - temp_two and store in t'
If no inverse is found // *use mpz_cmp_ui(r, 1) and check if its greater than 0*
Set i to 0
Mpz clear all variables and return
If t < 0 // *use mpz_cmp_ui*
Add t+n and store in t
Set i to t

RSA Library

rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters)

Set p_bits = (random() % (nbits / 2)) + (nbits / 4) and add 1 to p_bits
Set q_bits = nbits - p_bits and add 1

Make two primes, p and q // *use make_prime()*

Subtract p-1 and store in p_minus_one // *Use mpz_sub_ui*
Subtract q-1 and store in q_minus_one // *Use mpz_sub_ui*

Multiply p_minus_one and q_minus_one and store in totient vairable
Multiply p*q and store in n

While (gcd != 1)

Generate random number // *use mpz_urandomb*
Find the gcd of that number and totient // *use gcd()*

Set the random number from the last while loop equal to e
Mpz_clear(all the mpz variables) and return

rsa_write_pub (mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)

Use gmp_fprintf(pbfile, "%Zx\n", variable) format to print out n, e, and s to pbfile
Use fprintf() with the "%s\n" format specifier to print out the username

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

Use gmp_fscanf(pbfile, "%Zx\n", variable) format to scan in n, e, and s from pbfile
Use fscanf() with "%s\n" format specifier to scan in the username

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

Subtract p - 1 and store in p_minus_one
Subtract q - 1 and store in q_minus_one
Multiply p_minus_one and q_minus_one and store in totient

Use mod_inverse() with d, e, and totient as parameters
Mpz clear all the mpz variables and return

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

Use gmp_fprintf(pvfile, "%Zx\n", variable) to print n and d to pvfile as hexstring

return

rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile)
Use gmp_fscanf(pvfile, "%Zx\n", variable) to scan in n and d from pvfile as hexstring
return

Rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)
// *Compute the formula m^e (mod n)*
Call pow_mod() with c, m, e, and n as parameters

rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e)
Set k = log(2(n) -1 / 8 // *use mpz_sizeinbase(n, 2) for the log(2) part*
Set j = 0
Create a uint8_t buffer using calloc (k, sizeof(uint8_t)
Set the 0th index of the buffer array to 0xFF // *buffer[0] = 0xFF*

While (j = fread() > 0) // *meaning there's still unprocessed bytes*
mpz_import(m, j+1, 1, sizeof uint8_t, 1, 0, buffer)
rsa_encrypt() with c, m, e, and n as parameters
gmp_fprintf() c to outfile using hexstring format specifier // *"%Zx\n"*
Free (buffer)
Mpz_clear all mpz_t variables and return

rsa_decrpyt(mpz_t m, mpz_t c, mpz_t d, mpz_t n)
// Compute the formula c^d (mod n)
Call pow_mod() with m, c, d, and n as parameters and return

rsa_decrpyt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d)
Set k = log(2(n) -1 / 8 // *use mpz_sizeinbase(n, 2) for the log(2) part*
Set j = 0

Create a uint8_t buffer using calloc (k, sizeof(uint8_t)

While (gmp_fscanf(infile, hex format specifier, c) // *meaning there's still unprocessed bytes, format specifier = "%Zx\n"*
rsa_decrypt(m, c, d, n)
mpz_export(buffer, &j, 1, sizeof uint8_t, 1, 0, m
fwrite(&bufer[1], sizeof uint8_t, j-1, outfile // *Writes out j-1 bytes to outfile*
Free (buffer)
Mpz_clear all mpz_t variables and return

rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n)
    Set s = m^d (mod n) // *call pow_mod() with s, m, d, and n as parameters*

Rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n)
    Set t = s^e (mod n) // *Call pow_mod() with t, s, e, and n as parameters*
    If (t is the same as m) // *Use mpz_cmp(t, m) and check if its 0*
        Clear mpz variables
        Return true
    Clear mpz variables
    Return false

Key Generator

Create a help_message function that prints out the specified help message

Int main(argc, arv)
    Opt = 0
    Nbits = 256, iters = 50
    Char *public_path = "rsa.pub" // *as specified by asgn6.pdf*
    Char *private_path = "rsa.priv" // *as specified by asgn6.pdf*
    SEED = time(NULL) // *also specified in asgn6.pdf*
    Bool verbose as false
    File *pbfile and File *pvfile
    Char *user = "USER" and char *username

    Get opt while loop
        Switch (opt) // *Note: all case statements should have break statements*
        Case 'h'
            Call help message function
            Return -1
        Case 'b'
            Set nbits = atoi(optarg)
        Case 'i'
            Set iters = atoi(optarg)
        Case 'n'
            Set public_path = optarg
        Case 'd'
            Set private_path = optarg

Case ‘s’
Set SEED = atoi (optarg)
Case ‘v’
Set verbose = true

Set pbfile = fopen(pbfile_path, “w”) // *“w” since we want to write out to this file later*
Set pvfile = fopen(pvfile_path, “w”) // *give the path, default if user didn’t specify*

if the files didn’t open correctly // *Meaning if pbfile and/or pvfile == NULL*
printf() an error message // *“Error opening file.\n”*
Return -1

Call fchmod() and fileno() // *fchmod(fileno(pvfile), 0600)*

Set seed to randstate_init()

Call rsa_make_pub() and rsa_make_priv() to make the public and private keys // *rsa_make_pub takes p, q, n, e, nbits, iters as parameters and rsa_make_priv takes in d, e, p, q as parameters*

Set username = getenv(user)

Convert the username that we got into mpz_t using mpz_set_str() // *mpz_set_str(m, username, 62)*

Use rsa_sign() to compute signature // *Giving s, m, d, and n as parameters*

Write out the public and private keys to the specified files // *Use rsa_write_pub(), giving n, e, s, username, and pbfile as parameters. Also call rsa_write_priv(), giving n, d, and pvfile as paramters*

If the verbose flag was triggered // *Meaning if verbose was set to true*
Use printf() with format specifier “%s\n” to print out username
Use gmp_printf(“variable (%d bits) = %Zd\n”, mpz_sizeinbase(variable, 2), variable) as the format for the variables,

Close public and private files
Call randstate_clear() and mpz_clear all the mpz_t variables that were used throughout the execution of the program

Encryptor

Create a help_message function that prints out the specified help message

```
Int main(argc, arv)
    Opt = 0
    FILE *infile = stdin // as specified in asgn6.pdf
    FILE *outfile = stdout
    FILE *pbfile
    Char *pbfile_path
    Bool verbose = false // only true if user does -v when running program
    Bool verify
    char username[32] // Initialize username array to clal rsa_read_pub later

    Get opt while loop
        Switch (opt) // Note: all case statements should have break statements
        Case ‘h’
            Call help message function
            Return -1
        Case ‘i’
            Infile = fopen(optarg) // give it “r” as parameter as well
        Case ‘o’
            outfile = fopen(optarg) // give it “w” since we’ll write out to it later
        Case ‘n’
            Set pbfile_path = optarg
        Case ‘v’
            Set verbose = true

Set pbfile = fopen(pbfile_path, “r”) // “r” because we just want to read the public key from the specified file, no need to write, path will be the default unless specified otherwise

If pbfile == NULL
    Print error message // “Error opening pbfile.\n”
    Return -1

Call rsa_read_pub(n, e, s, username, pbfile) // Reads/Scans in n, e, s, and username from pbfile



If verbose = true
    Call printf() to print out the username, giving “%s\n” as format specifier
```

Call gmp_printf(variable (%d bits) = %Zd\n", mpz_sizeinbase(variable, 2), variable) for all the variables that need to be printed out, in this case, s, n, and e.

Call mpz_set_str(m, username, 62) to convert the username
Set verify = rsa_verify() where m, s, e, and n are parameters

If verify == false
    printf() an error message // *"Error while verifying signature.\n"*
    Return -1

Call rsa_encrypt_file(infile, outfile, n, e)

Close all the opened files and clear all the mpz_t variables

Decryptor

Create a help_message function that prints out the specified help message

```
Int main(argc, arv)
    Opt = 0
    FILE *infile = stdin // as specified in asgn6.pdf
    FILE *outfile = stdout
    FILE *pvfile
    Char *pvfile_path
    Bool verbose = false // only true if user does -v when running program

    Get opt while loop
        Switch (opt) // Note: all case statements should have break statements
        Case 'h'
            Call help message function
            Return -1
        Case 'i'
            Infile = fopen(optarg) // give it "r" as parameter as well
        Case 'o'
            outfile = fopen(optarg) // give it "w" since we'll write out to it later

        Case 'n'
            Set pvfile_path = optarg
```

Case ‘v’

Set verbose = true

Set pvfile = fopen(pvfile_path, “r”) // *“r” because we just want to read the private key from the specified file, no need to write, path will be the default unless specified otherwise*

If pvfile == NULL

Print error message // *“Error opening pvfile.\n”*

Return -1

Call rsa_read_priv(n, d, pvfile) // *Reads/Scans in n and d from pbfile*

If verbose = true

Call gmp_printf(variable (%d bits) = %Zd\n”, mpz_sizeinbase(variable, 2), variable) for all the variables that need to be printed out, in this case, n and d.

Call rsa_decrypt_file(infile, outfile, n, d) // *Decrypt infile and write it to outfile*

Close all the opened files and clear all the mpz_t variables