Luca Schram
Assignment 5
lschram

General Assignment Overview / Goal:

The general goal of this assignment is to write a program that performs RSA encryption on a message, with functions for encryption, decryption, and key generation. We will be able to generate an RSA public key and private key and use those as means to encrypt messages and files (with the public key) and decrypt said messages and files with the private key. The other goal of this assignment is to familiarize ourselves with the GNU Multiprecision Library, a library of functions that allow for the creation and manipulation of arbitrary precision numbers, which C does not support natively. We will be using this library (also known as GMP) to be able to deal with very large numbers so that we can compute sufficiently secure key pairs.

Files and included Functions:

1) **numtheory.c**
    a) gcd
    First initialize a *temp* value using GMP. Then, within a while loop, compare the passed in GMP variable *b* with 0 to check that it is not 0. Then, set the *temp* variable = to *b* and take *a* mod *b* and set that value to *b*. Then, set *a* equal to the *temp* value. This will loop continuously until *b* is 0. Then, set the output *d* equal to *a* and return void.
    b) pow_mod
        i) First, set the *out* mpz variable equal to 1. Then, initialize a temp variable *p* and set it equal to *base* (passed argument). Then, check that the *exponent* variable (passed argument) is greater than 0. If the exponent variable is odd, multiply *out* * *p* and take the resulting variable mod *modulus* variable (passed argument) and set *out* equal to that value. Outside of that if, multiply *p* * *p* and then take the modulo of *p* by the *modulus* variable and set *p* equal to that. Finally, set the *exponent* variable equal to *exponent* // 2 (using gmp function). This will repeat until the *exponent* variable is less than or equal to 0. Then return void.
    c) is_prime
        i) First, initialize the necessary variables, including *s, a, y*, and *j*. Then, set *s* to 0 to begin with and set *r* to *n*-1. Next, initialize a while loop that continually loops while *r* is even. Within the while loop, continually divide r by two until r becomes odd. Then, set a for loop that loops *iters* (passed argument) number of times. Within this for loop, first generate a random number between two and *n*-2. Then, call pow-mod of (*a, r, n*) and set the output var to *y*. Then, check if *y* != 1 and *y* != *n-1*, if both are true, set j to one. Then, initialize a while loop that loops as long as *j* <= *s* - 1 and *y* != *n*-1. Within that loop, set the *y* equal to the output of pow_mod(*y*, 2, *n*). Then, check if *y* == 1. If it does, clear all temp variables and return *false*. Otherwise, increment *j* by one. Then, outside of the while loop, check if *y*

!= *n* - 1. If true, clear all temp variables and return *false*. If all these conditions pass, after the for loop, clear all variables and return *true*.

d) mod_inverse
   i) First, initialize necessary variables, *r, t, r', t', q*, and two temp variables to fake parallel assignment. Set *r* equal to *n* and *r'* equal to *a*. Then, set *t'* equal to 0 and *t'* = 1. Then, initialize a while loop that loops while *r'* != 0. Within this loop, first floor div *r* // *r'* and set output to *q*. Then, set a temp variable equal to *r*. Then, set *r* equal to *r'*. Then, set *r'* = *temp* - (*q* * *r')*. Repeat this same process with the *t, t'*, and another *temp* var. Outside of the loop, check if *r* > 1, if true set *i* (passed argument) to 0, clear variables, and return. Then, check if *t* < 0, and increment *t* by *n* if true. Finally, clear all variables, set *i* equal to *t*, and return.

e) make_prime
   i) First, initialize an mpz variable *r_num* to hold a random number. Then, initialize a boolean flag for whether a number is prime. Then, initialize a while loop that loops while the size of the random number is < *bits* and the prime flag is false. Within the loop, generate a random number using mpz_urandomb and check if it's prime by setting the boolean flag = to the output of is_prime. This will loop until a suitable prime number is found for the designated bit count. Finally, set *p* equal to *r_num*, clear *r_num*, and return void.

2) randstate.c

a) extern gmp_randstate_t state
   i) Declares the randstate variable *state* as a global variable so it can be referenced across the program without having to seed it multiple times or redeclare it.

b) randstate_init
   i) Takes in a uint64_t var *seed* and initializes the random number generator using gmp_randinit_mt() and seeds the Mersenne Twister algorithm with gmp_randseed_ui(*seed*). Also seeds C random number generator using srandom(*seed*).

c) randstate_clear
   i) Simply clears the global randstate variable using gmp_randclear(*state*).

3) rsa.c

a) rsa_make_pub
   i) First, select a random number in the range of *nbits* / 4 to (3 * *nbits*) / 4. Generate that number using random(). That number will serve as *p*'s bit count. Then, subtract *p*'s bitcount from *nbits* to find *q*'s bitcount. Then call make_prime twice, once for *p* with *p*'s bitcount and *iters* (passed argument) and a second time for *q* with *q*'s bitcount and *iters*. Then, compute *n* = (*p* * *q)*. Then, compute Carmichael's function of (n), which can be written as (*n* - (*p* + *q*) + 1) / gcd(*p* - 1, *q* - 1). Then, compute *e*, which will be made by looping mpz_urandomb in the range of (*nbits*, 2^{nbits} - 1) until the generated number is coprime with

Carmichael's(n), also written as $\lambda(n)$. This can be determined by computing gcd($\lambda(n)$, e) until the gcd is 1. Once this is done, all parts of the public key have been determined. This will be implemented in roughly the same way that make_prime generates random prime numbers, except the check conditional is when gcd($\lambda(n)$, e) == 1.

b) rsa_write_pub
    i) Use the gmp_fprintf() function to write the public RSA key to a public file by writing *n, e, s*, and *username* as with trailing newlines after each. *n, e, and s* will be hexstrings.

c) rsa_read_pub
    i) Use the gmp_fscanf() function to read the hexstrings from the public file the key was written to. Need to read the hexstring values for *n, e, s,* and string value of *username*. Convert *n, e,* and *s* variables into mpz_t values using gmp_set_str(). Write the *username* into the passed in *username* char array (string).

d) rsa_make_priv
    i) Compute *d* by computing inverse_mod(*e*, $\lambda(n)$). Compute $\lambda(n)$ in the same way it was computed in rsa_make_pub.

e) rsa_write_priv
    i) Use gmp_fprintf() function to write the private key to a private file. Write *n* and *d* to the file with trailing newlines after each.

f) rsa_read_priv
    i) Use the gmp_fscanf() function to read the hexstrings from the private file the private key was written to. Read *n* and *d* hexstrings and convert them to mpz_t vars.

g) rsa_encrypt
    i) Performs the encryption by taking the hex representation of the provided message *m* and computing pow_mod(*m, e, n)*. This will give the output ciphertext value *c*.

h) rsa_encrypt_file
    i) First, calculate the block size k by taking the value of floor div ($\log_2(n)$ - 1) / 8. Then, do while the number of elements read from fread() is equal to the block size to make sure you stop when there is no more unprocessed data. Within this loop, allocate an array of size k with the type (uint8_t *), which will serve as the block of bytes. Then, set the zeroth index of the array to 0xFF for the workaround byte. Then, read k - 1 bytes in from the infile and place each of the read bytes into the allocated block starting from index 1 to keep the workaround 0xFF. Then, use mpz_import() to convert the read bytes into an mpz_t var *m*. Then, encrypt *m* with rsa_encrypt() and write the encrypted number to outfile as a hexstring with a trailing newline.

i) rsa_decrypt
    i) Performs the decryption by taking the ciphertext *c* and computing pow_mod(*c, d, n*). This will yield the decrypted message *m*.

j) rsa_decrypt_file
    i) Loop in the same way rsa_encrypt_file looped with fread() and calculate block size/ allocate the array for each block while there are still unprocessed bytes. Then, scan in a hexstring and save it as mpz_t var *c*. Then, call rsa_decrypt() on *c* to decrypt it to *m*. Then, use mpz_export() to convert m into bytes and store them in the allocated block. *J* will be the number of bytes converted. Then, write *j* - 1 bytes from index 1 of the block to the final decrypted outfile.
k) rsa_sign
    i) Performs RSA signing by computing the signature *s* = pow_mod(*m, d, n*).
l) rsa_verify
    i) Verifies the signature of a message. Computes the value *t* = pow_mod(*s, e, n*). If *t* is equivalent to the message *m*, the message is verified and the function returns true. Otherwise, return false.

4) keygen.c

a) This file will be the caller for all key generation functions created in rsa.c. It will effectively be a getopt loop that takes in command line arguments and specifies the information used to create the key pairs. First, initialize a getopt loop that parses command line options. The available options and their functions are listed below. After parsing the options and setting the necessary parameters to the inputted argument values, open the public and private key files. Then, set the private key file permissions to 0600. Next, initialize the random state variable and set its seed. Then, make public and private keys. Then, obtain the username of the user as a string and convert it into an mpz_t. Compute the signature using rsa_sign(). Write the public and private key info to their respective files. Finally, print out verbose output information, with both the numbers and the number of bits that constitute them. Finally, close the pub and priv files, clear randstate, and clear all mpz_t variables used.
    i) -b: specifies the minimum number of bits needed for public modulus n (default: 1024)
    ii) -i: specifies the number of Miller-Rabin iterations for testing primes (default: 50)
    iii) -n pbfile: specifies public key file (default: rsa.pub)
    iv) -d pvfile: specifies the private key file (default rsa.priv)
    v) -s: specifies the random seed for the random state initialization (default: seconds since unix epoch, aka time(NULL)).
    vi) -v: enables verbose output
    vii) -h: displays program synopsis and usage

5) encrypt.c

a) This file will support the encryption of files and messages using the previously generated rsa keys. This program will first parse command lines options, which are described below. Then, it will open the public key file and read the public key. If verbose output is enabled, print the *username*, signature *s*, public modulus *n*, and public exponent *e* with the number of bits they constitute. Then, convert the

username that was read to an mpz_t using mpz_set_str() and verify the signature, reporting an error and exiting the program if it is not verified. Finally, encrypt the file using rsa_encrypt_file() and close the public key file and clear all mpz_t variables.

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

6) decrypt.c

a) This file will support the decryption of files and messages using the previously generated private keys. The file will first parse command line arguments which are described below. Then, open the private key file and read the private key. If verbose output is available, print the public modulus *n* and the private key *d* with the number of bits that constitute them. Then, decrypt the file using rsa_decrypt_file(), close the private key file, and clear any mpz_t variables used.

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