Assignment 6 Design

Michael Coe

November 2021

Provided Macro from professor Long in vertices.h : `VERTICES 26`

randstate.c

Goal: This program creates and initializes random states to use in key generation. Global Variable: `gmp_randstate_t state;` -global external variable to be used inside and outside the function.

- `void randstate_init(uint64_t seed)`
  This function initializes `state` using gmp_randinit_mt() and then seeds the random state using `gmp_randseed_ui(state, seed)`.
- `void randstate_clear(void)`
  This function frees the memory initialized in `state` by calling `gmp_randclear(state)`.

numtheory.c

Goal: To implement number theory functions used to implement kegen.c

- `void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus)`

  This function implements modular exponentiation, starting by setting `out` to 1, then while `exponent` is greater than 0, if `base` is odd, then set `out` using `mpz_mul( out, out, base)` and `mpz_mod(out, out, modulus)`. After the if statement, set `base` using `mpz_mul ( base, base, base)` and `mpz_mod (base, base, modulus)`. Then set `exponent` using `mpz_fdiv_q_ui (exponent, exponent, 2)`

- `bool is_prime(mpz_t n, uint64_t s_t iters)`

  This program implements a Miller-Rabin primality test. If n is 2 return true, else if n is even, then return false. Otherwise start by creating `mpz_t r` and `uint64_t s = 1`. set `r` to `(n-1)/2`. Then while `r` is even, divide `r` by 2 and increment `s`. After the while loop, create a for loop that iterates from 1 to `iters`. Inside the loop, create random variable `a` using `mpz_urandomm ( a, state, n-3)` and then adding 2 to the value to keep it in the bounds of $a \in \{2, 3, ..., n-2\}$. Then create variable `y` and set it using `pow_mod(y,a,r,n)`. Then if `y` is not 1 or `n-1`, then create variable `j` set to 1 and create a nested while loop that runs while `j <= s-1` and `y` is not `n-1`. Inside the while loop, set `y` using `pow_mod(y,y,2,n)`, then if `y ==1` return false. Otherwise, increment j and continue the loop. After the while loop, if `y` doesn't equal `n-1` then return false. Once the for loop is complete return true.

- `void make_prime(mpz_t p, uint64_t bits, uint64_t iters)`

  This function creates a random prime number and `bits` long. start by setting `p` using `mpz_rrandomb (p, state, bits)`, then while not `is_prime(p, iters)`, reset `p` using `mpz_rrandomb (p, state, bits)`.

- `void gcd(mpz_t d, mpz_t a, mpz_t b)`

  This function returns the greatest common divisor of values `a` and `b`. While `b` is not 0, set `d` to b, then set b using `mpz_mod(b,a, b)`, then set `a` to `d`.

- `void mod_inverse(mpz_t i, mpz_t a, mpz_t n)`

  This function computes the modular inverse of `a` mod `n`. Starting by creating 4 variables: `r = n, r_not = a, t = 0, t_not = 1`. While `r` is not 0, create variable q set using `mpz_fdiv_q(q, r, r_not)`. Then create a variable `temp` set to r, and set `r` to `r_not` . Then set `r_not` to `mpz_mul( r_not, q, r_not)`, and set it once more to `mpz_sub(r_not, temp, r_not)`. Next, follow the same steps with `t` to `t_not` . After the while loop, if `r > 1`, set `i` to 0 and return the function, otherwise, set t using `mpz_add(t, t, n)` if `t` is less than 0 and set `i` to `t`.

rsa.c

Goal: To create RSA encryption and decryption algorithms.

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

  This functions creates a public rsa key. Start by creating `uint64_t pbits` This will be set to `(rand() % (nbits /2)) + nbits /4`, then set `p` using `make_prime(p, pbits, iters)`, then set `q` using `make_prime(q, nbits - pbits, iters)`. Next create a `temp` variable set with `mpz_sub_ui (temp, q, 1)`. Then set `n` using `mpz_sub_ui (n, p, 1)` and `mpz_mul ( n, n, temp)`. Next, while the value of `temp` is not 1, set e using `mpz_rrandomb (e, state, nbits)`, Then set temp by calling `gcd(temp, e, n)`. After the loop is complete set `n` to `mpz_mul ( n, q, p)` and return.

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

  This function prints out `n, e, s` to `pbfile` in that order using `gmp_fprintf()` with the format `"%Zd\n"`. It then prints `username` using standard `fprintf()` with the format `"%s\n"`.

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

  This function reads out and stores `n, e, s` from `pbfile` in that order using `gmp_fscanf()` with the format `"%Zd\n"`. It then stores `username` using standard `fscanf()` with the format `"%s\n"`.

- `void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q)`

  This function generates a private key. First create variable `n` and `temp` set with `mpz_sub_ui (temp, q, 1)`, `mpz_sub_ui (n, p, 1)` and `mpz_mul ( n, n, temp)`. Then set d using `mod_inverse(d, e, n)`.

- `void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile)`

  This function prints out `n, d` to `pvfile` in that order using `gmp_fprintf()` with the format `"%Zd\n"`.

- `void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile)`

  This function reads and stores `n, d` from `pvfile` in that order using `gmp_fscanf()` with the format `"%Zd\n"`.

- `void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)`

  This function performs rsa encryption on `m`. Set `c` by calling `pow_mod(c, m, e, n)`.

- `void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e)`

  This function encrypts `infile` and writes the encryption `outfile` This function starts by creating integer `n_int` which is set to `mpz_get_ui(n)`. Then create integer `k = ((log(n_int)/log(2)) -1) / 8`. Use k to dynamically allocate `uint8_t *block` to be an array size `k` using `calloc` and set `block[0] = 0xFF`. Next, create integer `j` to track number of bytes read and set it to 1. Then while `j` is greater than 0, fill `block` using `fread(block+1, 1, k-1, infile)` and setting j to it's output. Still inside the loop, if `j` is greater than 0 create `mpz_t m` and set it using mpz_import (m, j, 1, 1, 1, 0, block). Create `mpz_t c` using `rsa_encrypt( c, m, e, n)`, then print it to create `outfile` using `gmp_fprintf()` with the format `"%Zd\n"`.

- `void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n)`
  This function performs rsa decryption on `c`. Set `m` by calling `pow_mod(m, c, d, n)`.

- `void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d)`
  This function decrypts `infile` and writes the decryption `outfile`. This function starts by creating integer `n_int` which is set to `mpz_get_ui(n)`. Then create integer `k = ((log(n)/log(2)) -1) / 8`. Use k to dynamically allocate `uint8_t *block` to be an array size `k` . Next Create a variable `c`,`m`, and `j = 1`. While `gmp_fscanf()` does not return EOF, use it to read and store a hex value to `c`, call `rsa_decrypt( m, c, d, n)`, then fill block with the values of `m` using `mpz_export(block,j,1,1,0,m)`. Next write `block` to `outfile` using `fwrite(block+1, 1, j-1, outfile)`. After the loop completes clear `m` and `c` and return.

- `void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n)`
  This function produces an RSA signature by setting `s` by calling `pow_mod(s, d, d, n)`.

- `bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n)`
  This function verifies an RSA signature. Start by setting `m` by calling

`pow_mod(m, s, e, n)`. Then if `mpz_cmp (t, m) == 0)` then return true, otherwise return false.

keygen.c

Goal: to generate a pair of RSA public and private Keys.

Using `getopt`, take parameters from the command line to perform actions:

- -b: specifies the minimum bits needed for the public modulus n (default 256).
- -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 (default: the seconds since the UNIX epoch, given by `time(NULL)`).
- -v: enables verbose output.
- -h: displays program synopsis and usage.

Once all the input commands are parsed, open the pbfile and pvfile and and use `fchmod` and `fileno()` to set their permisisons to 0600. Next initialize `state` using `randstate_init()`, using the set seed. Now create variables `char *user = getenv(USER)`, `mpz_t name` and `mpz_t sign`. Set name with `mpz_set_str (name, user, 62)` and the signature with . Next, make the public and private keys using `rsa_make_pub()` and `rsa_make_priv()`, and write them to their respecive files using `rsa_write_pub()` and `rsa_write_priv()`.

If verbose output is enabled print the following, each with a trailing newline, in order:

- username
- the signature s
- the first large prime p
- the second large prime q
- the public modulus n
- the public exponent e
- the private key d

Conclude by closing all files and clearing all `mpz_t` variables.

encrypt.c

Goal: to encrypt a file using RSA encryption.

Using `getopt`, take parameters from the command line to perform actions:

- -b: specifies the minimum bits needed for the public modulus n (default 256).
- -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 (default: rsa.pub).
- -v: enables verbose output.
- -h: displays program synopsis and usage.

  After the user inputs have been processed, open the public key and read it using `rsa_read_pub()`, then if verbose output is enabled print the following, each with a trailing newline, in order:

  - username
  - the signature s
  - the public modulus n
  - the public exponent e

  Next, convert the username read from the public key file into `mpz_t sign` set with `mpz_set_str (sign, user, 62)`, and verify it using `rsa_verify()`. If it is not verified, then print an error and exit the program. Otherwise, encrypt the file using `rsa_encrypt_file()`, close all files and clear all `mpz_t` variables.

decrypt.c

Goal: To decrypt an RSA encrypted file

Using `getopt`, take parameters from the command line to perform actions:

- -b: specifies the minimum bits needed for the public modulus n (default 256).
- -i: specifies the input file to encrypt (default: stdin).
- -o: specifies the output file to encrypt (default: stdout).
- -n: specifies the file containing the private key (default: rsa.priv).
- -v: enables verbose output.

- -h: displays program synopsis and usage.

After the user inputs have been processed, open the public key and read it using `rsa_read_priv()`, then if verbose output is enabled print the following, each with a trailing newline, in order:

- the public modulus n
- the private key e

Next, decrypt the file using `rsa_encrypt_file()`, close all files and clear all `mpz_t` variables.