Design Document - Assignment 5

Ria Chockalingam - Rchockal - 11/6/2022

Assignment Overview:

In this assignment, we are generating RSA private and public key pairs with Keygen.c, encrypting files using the public key with Encrypt.c, and decrypting files using the private key with Decrypt.c. We will also implement two libraries, RSA and Numtheory. Numtheory will contain five mathematical functions so we can implement the random state interface of the RSA library. After creating Numtheory, we will create the RSA library, which contains twelve functions that generate keys, encrypt and decrypt files, signs messages, verifies signatures, etc. Encrypt.c will parse through various commands and allow for users to input files to encrypt and define a file to output the encryption in. We will use the various RSA library functions we implemented to do this encryption. Decrypt.c will act in a similar manner, but deals with decryption. We will also implement randstate.c which generates pseudorandom numbers with the Mersenne Twister. This will be used in our numtheory functions.

Encrypt.c:

Overview:

Implements the encrypt program and includes the main() function. Parses through various commands to encrypt files. Inputs public key - remember that public key is used to encrypt.

Pseudo:

Use getopt() to parse through commands:

-i: specifies the input file to encrypt
-o: inputs the output file to encrypt
-n: inputs file containing the public key
-v: enables boolean variable verbose
-h: prints program manual and returns 0. If the help is printed to stderr in any way, return a 0.

If failures occur, print helpful messages and exit with non-zero value.
Open the public key file using fopen().

Check if the verbose variable is true, and if it is, print the username, signature, public modulus n, and public exponent e. Add the respective bits in parentheses, with mpz_sizeinbase().

Convert username that was read into an mpz_t. This is the expected value of a verified signature.
Verify it with rsa_verify, and print an error message and exit program if not verifiable.
Encrypt the inputted file using rsa_encrypt_file() and input the -i value and -o value (to be written to).
Use fclose() to close the -n public key file and mpz_clear() to clear any mpz_t variables.
In the event of failure, print an error message and exit the program.

Decrypt.c:

Overview:

Implements the decrypt program and includes the main() function. Parses through various commands to decrypt files. Main difference between encrypt.c is that it inputs the private key in order to decrypt the message.

Pseudo:

Use getopt() to parse through commands:

-i: specifies the input file to decrypt
-o: inputs the output file to decrypt
-n: inputs file containing the private key
-v: enables boolean variable verbose
-h: prints program manual and returns 0. If the help is printed to stderr in any way, return a 0.

If failures occur, print helpful messages and exit with non-zero value.
Open the private key file using fopen().

Check if the verbose variable is true, and if it is, print the public modulus, new line, and then the private key e. Add the respective bits in parentheses, with mpz_sizeinbase()
Decrypt the inputted file using rsa_decrypt_file() and input the -i value and -o value (to be written to).

Use fclose() to close the -n private key file and mpz_clear() to clear any mpz_t variables.
In the event of failure, print an error message and exit the program.

Keygen.c:

Overview:

Generates the necessary keys with a main function. Takes in various files and arguments to generate specific versions.

Pseudo:

Use get opt to loop through the following command options:

-b: indicates min bits needed for the public modulus (default: 1024) (max is 50,4096)
-i: indicates the number of miller-tabin iters for testing primes (default: 50) (max 1500)
-n pbfile: indicates public key file (default: rsa.pub)
-d pvfile: indicates private key file (default: rsa.priv)
-s: indicates random seed for the random state initialization (default: the seconds since the UNIX epoch, given by time(NULL)).
-v: enables verbose variable to be true or false
-h: prints program manual and returns 0. If the help is printed to stderr in any way, return a 0.

Print errors to standard error not to standard output.
If failures occur, print helpful messages and exit with non-zero value.

Now open public and private key files with fopen().
Use fchmod and fileno to ensure private key files are set to 0600 - read and write perms for ONLY the user. You will need to use the fchmod combinations S_IRUSR and S_IWUSR, and use bitwise operations to and these together.

Initialize random state
Generate keys with rsa_make_pub and rsa_make_priv by passing in respective variables. You will need to define all the mpz_t variables and pass them in along with bits and iters.
Use getenv() to put current users name as a string. Use “USER” not “USERNAME”.
Convert username into mpz_t with mpz_Set_str and specify a base of 62.
Use rsa_sign to compute the signature
Write public and private key to files with write_pub and write_priv

If the verbose variable is true, print the following in order:
-username (bits: ?);
-the signature s (bits: ?);
-the first large prime p (bits: ?);
-the second large prime q (bits: ?);
-the public modulus n (bits: ?);
-the public exponent e (bits: ?);
-the private key d (bits: ?);

Print with information about num of bits, and respective values in decimal.
Close public and private key files
Clear random state with randstate_clear() and use gmp_clear() to clear any possible vars.

Numtheory.c:

Overview:

Implements number theory functions gcd(), mod_inverse, pow_mod(), is_prime(), and make_prime() to be used to generate the keys. Test the accuracy of these functions by using the respective gmp library functions.

Gcd Function:

Overview:

Calculates the greatest common divisor of two numbers a and b and outputs result in d.

Pseudo:

```
Pass in mpz_t variables d: output, a: first num, b: second num.
Use a while loop and the mpz_cmp_ui function to continue the loop if b does not equal 0. We are
essentially looping until a % b equals zero, AKA a and b are divisible.
        Inside, initialize a temp variable. with mpz_init()
        Use mpz_set to assign b to t.
        Find mod b of a by using mpz_mod() and store value in b.
        Use mpz_set to assign the temp variable to a.
        (We do this in separate steps since mpz does not allow for multiple arithmetic operations
in one line)
```

Outside of loop use mpz_set to assign a to d.
Clear all variables.

Mod_inverse Function:

Overview:

Calculates the modulo inverse of X modulo Y.

Pseudo:
Pass in mpz_t variables o (output), a (), and n (modulo).
Use mpz_init() to initialize variables r, r’, t, t’, q and two temporary variables that will store r and t in the loop. Use mpz_set() to assign n (passed in value) to r. Assign a to r’, 0 to t, and 1 to t’. Remember to use mpz_set_ui() for assigning numbers to variables.

Create a while loop that will use mpz_cmp_ui() to continue the loop if r’ is not equal to 0. Use mpz_fdiv_q to divide r by r’and assign value to q. Since c cannot handle parallel assignments, we need to split the below two lines into separated assignments.

$$(r, r') \leftarrow (r', r - q \times r')$$
$$(t, t') \leftarrow (t', t - q \times t')$$

Use mpz_set to assign r to temp variable r and r’ to r. Create a temp variable with init() that will store the output of q*r’. Use mpz_mul(). Then subtract the temporary r value by the variable mentioned in the previous sentence, with mpz_sub(). Then assign that variable to r’ with mpz_set. Repeat these steps with t and t’, and make sure to create another temp variable to store multiplication.

Outside of while loop, check if r > 1 with mpz_cmp_ui(), and if it is true, do nothing and terminate.

Also check if t < 0 with mpz_cmp_ui() and if so, add n to t with mpz_add and then set output to t with mpz_set().
Exit if statement and set o to t1.
Clear all variables.

Pow_mod Function:

Overview:

Calculates the modulo inverse of X modulo Y.

Pseudo:

Pass in variables o: output to be written to, a: base, d: exponent, n: modulus.
Initialize variables v and p with mpz_init(). Use mpz_set_ui to assign 1 to v and mpz_set to assign a to p.

Create a while loop that will use mpz_cmp_ui to keep the loop running if exponent d is greater than 0.
Assign d to a temp variable d2.

Inside the loop, check if d2 is odd with mpz_odd_p function. If it is, multiply p*v with mpz_mul, store in v, and do v % n with mpz_mod, and store in v.

$v \leftarrow (v \times p) \mod n$ Since we cannot do this in one line with mpz, we must split into two lines as explained above.

Outside of the if statement, multiply p by p with mpz_mul and store in p. Then modulo p by n with mpz_mod. Now use mpz_fdiv_q_ui to divide d2 by 2. Notice we use ui to divide by an unsigned integer and f to floor it.

Outside of the while loop, use mpz_set to set o as v.
Clear all variables.

Is_prime Function:

Overview:

Uses the Miller-Rabin primality test to output if variable n is prime using variable "iters" number of Miller-Rabin iterations.

Pseudo:

Passes in mpz_t variables "n", and "iters" (iterations).

Create a for loop that continues for "iters" number of iterations.

- Inside, create a variable that contains a randomly generated number from 2 to n-2. To do this, call random() and mod it by (n - 2 - 2 + 1) and add 2 to the result. This is because to generate the random value we need to do (upper-lower+1)+lower to obtain this range.
- Then call pow_mod() and pass in output var, randomly generated variable as base, variable "r" as exponent, and value "n"(passed in) as the modulus.
- Check (mpz_cmp) if the output variable of pow_mod() is 1 and if it is equal to n-1. If they are not: initialize variable j with mpz_init() and assign 1 to var j with mpz_set_ui().
    - Inside if statement, create a while loop that continues when j is less than or equal to s-1 and when the output of pow_mod does not equal n - 1. Use mpz_cmp().

```
            - Call pow_mod again and pass in output, the original output result from the first
            pow_mod, 2, and n.
            -Check if this new output is equal to one with mpz_cmp(), and if it is, return
            False
            - Outside of if statement, assign j+1 to j with mpz_set()
        - Check if output from pow_mod does not equal n -1 , and if so, return False
Otherwise return true.
```

Clear all variables.

<u>Make_prime Function:</u>

<u>Overview:</u>

Passes in p (prime), bits (num of bits), and num of iters. Generates a new prime number which gets stored in "p". This number should be at least "bits" long. This number must be tested for its primality using is_prime() with "iters" number of iterations.

<u>Pseudo:</u>

```
Takes in n and iters
Create a loop that loops until found prime num.
Generates random number with num "bits". Use formula random() % (upper-lower+1)+lower to
set this number of bits.
Call is_prime(number,iters), stop loop once true is returned.
Clear all variables.
```

Randstate.c:

<u>Overview:</u>

Implement random state interface for RSA library and number theory functions.

<u>Randstate_init Function:</u>

<u>Overview:</u>

Initializes the seed and state in order to generate random variables.

Pseudo:

Pass in the seed as an unsigned 64 bit integer. Initialize seed with srandom(seed). Then use gmp_rand_init_mt() to initialize the state. Initialize the seed once again with gmp_randseed_ui().

Randstate_clear Function:

Overview:

Clears and frees all memory used by initialized global random state.

Pseudo:

Call gmp_randclear().

Rsa.c

Overview:

Implements the RSA library with 12 different functions.

rsa_make_pub Function:

Overview:

Creates parts of a new RSA public key by creating two large primes p and q, their product, and the public exponent e.

Pseudo:

- Generates primes p and q using make_prime() generated values.
- Use random() to generate a random number within the inputted number of bits. Use formula “nbits”/4 to (3* “nbits”)/4). This generates the number of bits for p, and the remaining bits go to q.
- Call makeprime with the corresponding p, p_bits and iters. For q as well.
- To generate e: Use formula $\lambda(n) = \varphi(n)/gcd(p-1,q-1)$ where $\varphi(n) = (p-1)(q-1)$
- We will use mpz_sub and mpz_mul to conduct this arithmetic.

- Create while loop.
    - Now generate random nums amount n bits. Use mpz_randomb().
    - Calculate gcd(random number, λ(n)). If this is coprime, AKA if gcd() output var == 1, (use mpz_cmp) that is the public exponent. Also make sure e is between 2 and n.

Clear all variables.

rsa_write_pub Function:

Overview:

Writes a public RSA key to pbfile.

Pseudo:

- Takes in mpz vars n,e,s, (since public key involves n and e), a username, and a file pbfile to write to.
- Convert numbers to hexadecimal:
- Use gmp to format hex strings with **gmp_fprintf** and **%Zx**
- Write to file pbfile → n then new line, then exponent then s, and username.

rsa_read_pub Function:

Overview:

Reads a public RSA key from pbfile.

Pseudo:

- Use fseek to jump to beginning of file.
- Use fscanf() and to read contents of the rsa key
- Should be formatted as hex strings with %Zx

rsa_make_priv Function:

Overview:

Creates new RSA private key given primes p and q and public exponent e. We are finding d.

Pseudo:

- Find d by calling mod_inverse of e mod of totient(n).
- Do this by calculating (p-1)(q-1) and dividing by gcd (p-1,q-1)
- Then call mod_inverse on the resulting totient and e values.
- Store this in variable d.

rsa_write_priv Function:

Overview:

Writes private RSA key to inputted pvfile.

Pseudo:

- Take the d and n value obtained by make_priv and write it to a file containing n then d with trailing newlines in hexstring format.

rsa_read_priv Function:

Overview:

Reads the private RSA key from pvfile.

Pseudo:
- fseek to start reading at beginning of file
- Take pvfile and read it with a reading function, ex: fscanf()

rsa_encrypt Function:

Overview:

RSA encryption to compute ciphertext c by encrypting message m with e and n.

Pseudo:

Pass in ciphertext c, message m, exponent e , and modulus n into power mod.
Use formula m^e mod n to define the encryption.

rsa_encrypt_file Function:

Overview:

Encrypts the inputted var m and writes the encrypted content to var c. Encrypt in blocks.

Pseudo:

Value of block cannot be 0 or 1.
Calculate block size with ((log2(n) - 1)/8)
Use malloc to dynamically allocate an array that can hold k bytes. Array should be type uint8_t. This is essentially the block.
Prepend a single bytee 0xFF to the front of the block we want encrypted.
Set zeroth byte of block to 0xFF
When there are unprocessed bytes infile:

- read at most k-1 bytes from the infile and let a variable store number of bytes actually read. Put read bytes into allocated block beginning index 1 .
- Mpz_import() will convert read bytes into an mpz_tm. Set order parameter of mpz_import() to be message, block read + 1, 1, sizeof(uint8_t),1,0, and the block.
- Encrypt m with rsa_encrypt.
- Write encrypted num to outfile as a hex string with newline.

rsa_decrypt Function:

Overview:

RSA decryption to compute message m by decrypting ciphertext c. Use private key d and public mod n.

Pseudo:

Pass in modulus n, ciphertext c, message m, and key d into power mod.
Reference formula c^d mod n to calculate decryption.

rsa_decrypt_file Function:

Overview:

Decrypts the inputted file infile and writes the decrypted content to outfile. Decrypt in blocks.

Pseudo:

- To apply the block requirement: calculate block size by using mpz_sub, mpz_fdiv_q_ui, log(), to store block size from formula ((log2(n) - 1)/8)
- Use malloc to dynamically create an array that uses size k bits of type uint8_t. This is essentially the “block”.

- Create a while loop that will scan the infile until it reaches end of file.
- call decrypt for the message, cipher, d and n.
- create a value j that equals 0
- Use mpz_export to pass in the block, then address of j, then 1,1,1,0, and the message
- Use fwrite to pass in block+1, 1, j-1, output file
- Close the file
- And free the block

rsa_sign Function:

Overview:

Performs RSA signing - generating a signature by signing the message m using the private key d and public modulus n.

Pseudo:

Perform signature by referencing m^d modulus n
Call pow_mod()
So s should be verified if signed.

rsa_verify Function:

Overview:

Returns a boolean that indicates if the signature s is verified and false otherwise.

Pseudo:

checks to see if s is verified AKA if s^e modulus n = message m.
Call pow_mod on the signature, exponent, and n.
If the output is equal to message, clear output and return true.O
Otherwise, clear t and return false.