Lynelle Goh
ljgoh
Assignment 5

General Idea

- We will be creating three programs for this assignment.
    - A key generator, an encryptor, and a decryptor.
    - Our key generator will produce RSA public and private key pairs. We will encrypt the files using a public key and use our decrypting program to decrypt the encrypted files using the private key for that.
        - For the RSA algorithm, we must choose two large random primes *p* and *q*. We will compute $n = pq$. Compute totient of n.
- We cannot use any GMP-implemented number theoretic functions or exponentiation functions. We are supposed to implement those ourselves.
- We are given pseudocode for the repeated squaring and modular reduction at each step.
- We are expected to implement the following functions:
    - `void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus)`
    - `bool is_prime(mpz_t n, uint64_t iters)`
    - `void make_prime(mpz_t p, uint64_t bits, uint64_t iters)`
    - `void gcd(mpz_t d, mpz_t a, mpz_t b)`
    - `void mod_inverse(mpz_t i, mpz_t a, mpz_t n)`
    - `void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters)`
    - `void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)`
    - `void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)`
    - `void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q)`
    - `void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile)`
    - `rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)`
    - `void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile)`
    - `void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)`
    - `void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e)`
    - `void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n)`
    - `void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d)`
    - `void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n)`
    - `bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n)`
- We need to accept command-line options for our key generator program, our encrypt, and our decrypt.
- All help messages and error messages for all three programs must be printed to standard error
- (stderr), not to standard output (stdout).
- Tip: reduce the algorithms we are asking you to implement to finding the quotient and the remainder of different numbers.

- ➔ We will have to use fopen, fclose, fread, fwrite.
- ➔ For really large numbers in C, mpz_t type in gmp library.
    - ◆ mpz_tdiv_dr
- ➔ -lgmp for the library.

<u>Pseudocode</u>

- ➔ Key Generator
    - ◆ Parse the command-line options and handle them accordingly.
        - -b: specifies the minimum bits needed for the public modulus n(default: 1024).
        - -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 theUN I X epoch, given by time(NULL)).
        - -v: enables verbose output.
        - -h: displays program synopsis and usage.
    - ◆ Open the public and private key files.
        - If there's an error, print out the error and exit.
        - Also print out an error if the iterations and bits are too much or too little.
    - ◆ Make sure the key file permissions are set to 0600.
    - ◆ Initialize the random state.
    - ◆ Make the public key public and the private key private.
    - ◆ Get the user's name as a string. Make sure to allocate memory for this.
    - ◆ Convert the username into a set string, also specifying the base as 62. Then use a function to compute the signature of the username.
    - ◆ Write the computed keys to their respective files.
    - ◆ Print out the following to standard error, each with a trailing newline.
        - 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
    - ◆ Clear and close variables and files.
- ➔ Encryptor
    - ◆ Parse the command-line options and handle them accordingly.
        - -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.
    - ◆ Open the public file.
        - If there's an error, print out the error and exit.

- Read the public key from the opened public key file.
- If verbose output is enabled, print the following:
  - username
  - the signature s
  - the public modulus n
  - the public exponent e
- Convert the username that was read. Verify the signature. If there's an error, report the error and exit the program.
- Encrypt the file using our function for encryption.
- Close the public key file and clear the variables used.

➔ Decryptor
- Parse the command-line options and handle them accordingly.
  - -i: specifies the input file to decrypt (default: stdin).
  - -o: specifies the output file to decrypt (default: stdout).
  - -n: specifies the file containing the private key (default: rsa.priv).
  - -v: enables verbose output.
  - -h: displays program synopsis and usage.
- Open the file. Print a helpful error and exit the program in case of failure.
- Read the private key from the opened private key file.
- If verbose output is enabled print the following, each with a trailing newline, in order:
  - (a) the public modulus n
  - (b) the private key d
- Close the public key file and clear the variables used.

➔ numtheory.c
- power_mod (a,d,n)
  - Initiate v as 1.
  - Initiate p as a.
  - Create a while-loop that terminates when d is less than 0.
    - If d is odd…
      - Set v as v times p modulus n.
    - Set p as p times p modulus n.
    - Set d as d divided by 2.
  - Return v.
- is_prime(n,k)
  - Put in two if-statements that check if the smaller prime numbers are the value being given to us.
    - Return true or false depending on what prime number it is.
  - Set a variable to equal n minus 1.
    - Create a while-loop to see if that variable is even and if it doesn't equal zero.
      - Divide that variable by 2.
      - Use a variable to count the number of times we go through the while-loop.

- Create a for-loop where it starts at i = 0, and it terminates when i equal to k minus 1.
  - Choose a random number from {2, 3, …, n-2}.
  - The variable y will equal power_mod(a,r,n).
  - If y does not equal 1 and y does not equal n -1…
    - Variable j will equal 1.
    - While j is less than or equal to s - 1 and y is not equal to n minus 1…
      - Variable y will equal power_mod(y,2,n).
      - If y does equal 1, return false.
      - j will equal j + 1.
    - If y does not equal n - 1, return false.
- Return true at the end.

- mod_inverse(a,n)
  - Set variable r to n. Set variable r prime to a.
  - Set variable t to 0. Set variable t prime 1.
  - Create a while-loop where it goes through if r prime does not equal 0.
    - Set variable q to r divided by r prime.
    - Set variable r to r prime.
    - Set variable r prime to r minus q times r prime.
    - Set variable t to t prime.
    - Set t prime to t minus q times t prime.
  - Create an if-statement if r is greater than 1.
    - Return t equals 0.
  - Create an if-statement if t is less than 0.
    - Set t to t plus n.
  - Return t.
- GCD(a,b)
  - Create a while-loop that compares the b value and 0. It must not be 0.
    - Do modulus for our b variable using a and b.
    - Set the variable for a.
  - Once we're out of the loop, we can set our answer variable to a's value.
- make_prime()
  - Generate a prime number stored in p.
  - We will use the generated number and test it using the is_prime() function.
  - Create a while loop that ends when bits is higher than our created prime number and its bits.
    - Inside the whole-loop, we'll call the function is_prime, then check if it is. If it is, exit with a true boolean.
- rsa_make_pub()
  - Use our function to make two primes, p and q. To gain iters, use the random(). Let the number of bits for p be a random number in the range [nbits/4,(3×nbits)/4). The remaining bits will go to q.
  - Compute $\lambda(n) = \mathrm{lcm}(p-1,q-1)$. To do this, we need $(p-1,q-1)$.

- To generate a suitable public exponent e. We will create a loop that generates random numbers around nbits using mpz_urandomb(). Compute the gcd() of each random number and the computed λ(n). Stop the loop you have found a number coprime with λ(n)

◆ rsa_write_pub()
  - Write a public RSA key to pbfile. Some of the values will be written as hexstrings. Refer to our key generator program.

◆ rsa_read_pub()
  - Scan each line and set the correct variable to the readings.

◆ rsa_make_priv()
  - Create a RSA private key given the primes p and q and the exponent e. Refer to our key generator program. Use our modulus inverse function to get the answer. We need to follow the formula given to us.

◆ rsa_write_priv()
  - Write a private RSA key to pvfile and make both values that are to be written as hexstrings.

◆ rsa_read_priv()
  - Reads a private RSA key from pvfile. Both values will be written as hexstrings.

◆ rsa_encrypt()
  - Performs RSA encryption (refer to our encryption program). Use pow_mod.

◆ rsa_encrypt_file()
  - We will encrypt the contents of the infile. Refer to our encryption program.
    - Calculate the block size k. k= ⌊(log2(n)−1)/8⌋.
    - Dynamically allocate memory for an array that holds k bytes. This array should be of type (uint8_t *).
    - Set the zeroth byte of the block to 0xFF. This effectively prepends the workaround byte that we need.
    - Read at most k - 1 bytes from the file. We will use j as the number of bytes actually read.
  - The data will be encrypted in blocks.
  - Prepend a single byte to the front of the block we want to encrypt.
  - Create an if-statement to check if there's unprocessed bytes in the file.
    - Read at most k - 1 bytes. Use variable j to be the number of bytes actually read. Place all read bytes into the allocated block starting from index 1.
    - Using mpz_import(), convert the read bytes, including the prepended 0xFF into an mpz_tm. set the order parameter of mpz_import() to 1 for most significant word first, 1 for the endian parameter, and 0 for the nails parameter
    - Encrypt m with rsa_encrypt(). Write the encrypted number to outfile as a hexstring.

◆ rsa_decrypt()
  - Decrypt the contents of the encrypted file. Computing message m by decrypting ciphertext c using private key d and public modulus n. Use pow_mod.

- rsa_decrypt_file()
    - Calculate the block size k. $k = \lfloor(\log 2(n) - 1)/8\rfloor$.
    - Dynamically allocate an array that can hold k bytes. This array should be of type (uint8_t *) and will serve as the block.
    - Create an if-loop to check if there's unprocessed bytes in the file.
        - Scan in a hexstring, saving it as mpz_t c.
        - Decrypt c into mpz_t m by calling rsa_decrypt().
        - Convert m back into bytes, storing them in the allocated memory block. Let j be a variable of the number of bytes actually converted.
        - Write out $j-1$ bytes starting from index 1 of the block to outfile. Recall that the first byte will be the 0xFF that you prepended when you encrypted the data; you don't want to write it out.
- rsa_sign()
    - Use our pow_mod function along with S(m) = s= md (mod n) to perform RSA signing.
- rsa_verify()
    - Using t= V(s) = se (mod n), the signature is verified, and will return true, if the expected message m is the same. If not, it will return false.