Stanley Chen

CSE 13S

Assignment 6 PDF

Purpose of program:

This program will have multiple .c files, and will perform RSA public key encryption. Numtheory.c contains the math functions that RSA encryption requires and rsa.c contains various functions used to perform the encryption and decryption. Keygen.c is used to generate public and private keys for encryption and decryption, encrypt.c encrypts files with RSA encryption, and decrypt.c will decrypt files encrypted with RSA encryption.

Deliverables for assignment 6:

- decrypt.c
    - decrypt.c contains the program for decryption
- encrypt.c
    - encrypt.c contains the program for encryption
- keygen.c
    - keygen.c contains the program for generating a key
- numtheory.c
    - numtheory.c contains the number theory functions
- numtheory.h
    - numtheory.h is the header file for numtheory.c

- randstate.c
    - randstate.c contains the random state interface for the RSA library and number theory functions
- randstate.h
    - randstate.h is the header file for randstate.c
- rsa.c
    - rsa.c contains the RSA library
- rsa.h
    - rsa.h is the header file for rsa.c
- Makefile
    - CC = clang must be specified
    - CFLAGS = -Wall -Wextra -Werror -Wpedantic must be specified
    - make must build encrypt, decrypt, and keygen as should make all
    - make encrypt should just build the encrypter
    - make decrypt should just build the decrypter
    - make keygen should just build the keygen
    - make clean must remove all files that are compiler generated
    - make format should format all your source code, including the header files

- README.md
    - README.md is a text files that uses markdown formatting that includes:
        - the title of the program
        - the purpose and usage of the program
            - a description of the program and how it is used
        - how to build the program
            - the commands needed to run to produce the program
        - any known bugs
- DESIGN.pdf
    - DESIGN.pdf is a pdf file that describes the design and design process of the program, with information like pseudocode and diagrams

Pseudocode for assignment 6:

randstate.c:

Used to initialize a random state variable for any GMP random integer functions.

- randstate_init(seed)
    - Initialize global variable "state" with a Mersenne Twister algorithm
        - Use gmp_randseed_ui() for seed and gmp_randinit_mt() to initialize state

- randstate_clear()
    - Clear and free all memory used by global variable “state”
        - Use the gmp_randclear() function to clear

numtheory.c

Used to hold all number theory functions for en/decryption and the RSA library.

- pow_mod(out, base, exponent, modulus)
    - Performs modular exponentiation, raising the “base” to the “exponent” power modulo “modulus” and saving result to “output”
        - Set v to 1
        - Set p to base
        - Create a loop that while “exponent” > 0
            - If the “exponent” is odd
                - Set v to v * p mod “modulus”
            - Set p to p^2 mod “modulus”
            - Set d to floor(d/2)
        - Once loop then free/clear everything and set “out” to v
- is_prime(n, iters)

- Performs the Miller-Rabin primality test to see if "n" is prime or not using "iters" number of iterations in Miller-Rabin
    - Write n - 1 = 2^s * r such that r is odd
    - Set i to 1
    - Create a loop for i to k
        - Choose a random number a (2, 3, …, n-2)
        - Call power mod and save to variable "y"
            - pow_mod(y, a , r, n)
        - If y is not equal to 1 and y not equal to n - 1
            - Set j to 1
            - Create a loop while j less than or equal to s - 1 and y not equal to n - 1
                - Set y to pow_mod(y,y,2,n)
                - If y is 1
                    - Return false
                - Set j to j + 1
            - If y not equal to n - 1
                - Return false
    - Return true

- make_prime(p, bits, iters)
    - Generates a new prime number “bits” number of bits long stored in “p”, and tested with is_prime() with “iters” number of iterations
        - Use GMP library to get a random number with “bits” number of bits
        - check primality with is_prime()
            - If true then save prime
            - If false then get new random number
- gcd(d, a, b)
    - Computed the greatest common divisor of “a”, “b”, and stores the divisor to d
        - Create a loop while b is not equal to 0
            - Set t to b
            - Set b to a mod b
            - Set a to t
        - Set d to value of a
- mod_inverse(i, a ,n)
    - Computes the inverse i of a modulo n. If inverse cannot be found, set i to 0

- Set (r,r') to (n,a)
- Set (t,t') to (0,1)
- Create a loop while r' not equal to 0
    - Set q to floor(r/r')
    - Set (r,r') to (r',r - q * r')
    - Set (t,t') to (t',t - q * t')
- If r is greater than 1
    - Set i to 0 (no inverse)
- If t is less than 0
    - Set t to t + n
- Set i to value of t

rsa.c

Used to create the RSA library for all aspects of RSA cryptography

- rsa_make_pub(p, q, n, e, nbits, iters)
    - Create primes p and q be calling make_prime(), having the "bits" for p be in the range of (nbits/4, (3 * nbits)/4), and remainder will be the "bits" for q
    - Compute the totient of (n) = (p-1)(q-1)
    - Get a public exponent e

        - Create a loop while number is not coprime with totient
            - Generate random number around nbits
                - Call mpz_urandomb()
            - Compute gcd() of the random number an totient
    - Save number to e
- rsa_write_pub(n, e, s, username[], *pbfile)
    - Writes a public RSA key to pbfile
        - Format is n, e, s, username, all with a trailing new line
            - n, e, s are all hex strings
- rsa_read_pub(n, e, s, username[], pbfile)
    - Reads a public RSA key from pbfile
        - Format is n, e, s, username, all with a trailing new line
            - n, e, s are all hex strings
- rsa_make_priv(d, e, p, q)
    - Creates a new private RSA key given primes p and q with public exponent e
        - d is computed with the inverse of e modulo totient of (p-1)(q-1)
            - Call mod_inverse()
- rsa_write_priv(n, d, pvfile)

    - Writes a private RSA key to pvfile
        - Format is n, d all with a trailing new line
            - n, d are all hex strings
- rsa_read_priv(n, d, pvfile)
    - Reads a private RSA key to pvfile
        - Format is n, d all with a trailing new line
            - n, d are all hex strings
- rsa_encrypt(c, m, e, n)
    - Performs RSA encryption, encrypting message m using the exponent e and modulus n into ciphertext c
        - Use the equation E(m) = c = m^e (mod n)
- rsa_encrypt_file(infile, outfile, n, e)
    - Encrypts the contents of infile and writes that into outfile
        - Calculate the block size k using equation k = floor(log2(n - 1)/8)
        - Dynamically allocate an array of (uint8_t *) that is k bytes long
        - Set 0th byte of array to 0xFF
        - Create a loop while infile is not fully read yet
            - Read at most k-1 bytes from infile, and j is the number of bytes read, start at index 1

        - Use mpz_import() to convert read bytes, including oth byte 0xFF into mpz_t m, and also set order parameter to 1 for most significant word first, 1 for endian parameter, and 0 for the nails parameter in mpz_import()
        - Encrypt message (m) by calling rsa_encrypt() into ciphertext (c)
        - Write c to outfile as a hex string followed by a trailing new line
    - Clear all mpz_t variables and free array
- rsa_decrypt(m, c, d, n)
    - Performs RSA decryption, decrypting ciphertext c using the exponent e and modulus n into message m
        - Use the equation D(c) = m = c^d (mod n)
- rsa_decrypt_file(infile, outfile, n, e)
    - Decrypts the contents of infile and writes that into outfile
        - Calculate the block size k using equation k = floor(log2(n - 1)/8)
        - Dynamically allocate an array of (uint8_t *) that is k bytes long
        - Create a loop while infile is not fully read yet

            - Scan in a hexstring, save to mpz_t c, save fscanf() return to variable j
            - Call rsa_decrypt() to decrypt ciphertext (c) to message
            - Use mpz_export() to convert message back into bytes and save to the array
            - Write out j-1 bytes starting with index 1 from outfile, skipping index 0 (0xFF)
        - Clear all mpz_t variables and free array
- rsa_sign(s, m, d, n)
    - Performs RSA signing, signing message m using private key d and modulus n to get signature s
- rsa_verify(m, s, e, n)
    - Performs RSA verification, returning true if signature s is verified and false if not verified
        - Call pow_mod() to calculate temp variable (t), using parameters s, e, and n
        - Compare t to m
            - If same
                - Clear mpz_t variables

        - Return true
    - Else
        - Clear mpz_t variables
        - return false

keygen.c

Creates public and private keys for RSA cryptography

- Parse through the command line options with getopt()
    - 'b':
        - Number of bits needed for the public modulus
    - 'i':
        - Number of iterations for Miller-Rabin for testing primes
    - 'n':
        - Name of public key file
    - 'd':
        - Name of private key file
    - 's':
        - The random seed for random state initialization, the default is (time(NULL))
    - 'v':

        - Enables verbose printing
    - 'h':
        - Print program usage and help
- Use fopen() to open the public and private key files
- Use fchmod() and fileno() to set private key file permissions to 0600
- Call randstate_init() using the seed to initialize the randstate
- Call rsa_make_public() and rsa_make private() to make public/private keys
- Use getenv() to get username
- Convert username to mpz_t with mpz_set_str(), with base being 62
- Use rsa_sign() to compute signature of username
- Write the computed public and private keys to files
- If verbose printing is enabled, print the following, each with a trailing newline, in this order (mpz_t values should have have info on the # of bits they have):
    - Username
    - Signature s
    - First large prime p
    - Second large prime q
    - Public modulus n

    - Public exponent e
    - Private key d
- Close the public and private files, call randstate_clear() to clear randstate, and clear any other mpz_t variables.

encrypt.c

Encrypts a message using RSA encryption.

- Parse through the command line options with getopt()
    - 'i':
        - Input file to encrypt (default: stdin)
    - 'o':
        - Output file to encrypt (default: stdout)
    - 'n':
        - File containing public key (default: rsa.pub)
    - 'v':
        - Enables verbose printing
    - 'h':
        - Print program usage and help
- Open the public key file with fopen()
    - Return with error if cannot open file

- Read public key from file
- If verbose printing is enabled, print the following, each with a trailing newline, in this order (mpz_t values should have have info on the # of bits they have):
    - Username
    - Signature s
    - Public modulus n
    - Public exponent e
- Convert username into a mpz_t
- Verify signature by calling rsa_verify()
    - Return with error if cannot verify signature
- Encrypt file with rsa_encrypt_file()
- Close public key file and clear mpz_t variables

decrypt.c

Decrypts a message using RSA decryption.

- Parse through the command line options with getopt()
    - 'i':
        - Input file to decrypt (default: stdin)
    - 'o':

        - Output file to decrypt (default: stdout)
    - 'n':
        - File containing private key (default: rsa.priv)
    - 'v':
        - Enables verbose printing
    - 'h':
        - Print program usage and help
- Open the private key file with fopen()
    - Return with error if error occurs
- Read private key from file
- If verbose printing is enabled, print the following, each with a trailing newline, in this order (mpz_t values should have have info on the # of bits they have):
    - Public modulus n
    - Private key e
- Decrypt file with rsa_decrypt_file()
- Close private key file and clear mpz_t variables