CSE 13S Fall 2021
James Gu
jjgu@ucsc.edu
11 November 2021

Assignment 6: Public Key Cryptography

Description of Program:

In computer science, cryptography was once only used for the government, spies, and the military, but now it is very commonly used in most websites using SSL. The earliest public key cryptography algorithm and the one we will be focusing on in this assignment is RSA. Public key cryptography uses a pair of keys: public keys and private keys. Someone can encrypt a message using the public key but it can only be decrypted with the private key. The three programs we will be creating are a key generator, an encryptor, and a decryptor. The keygen program will produce key pairs. The encryptor will encrypt files using the public key. And the decryptor will decrypt encrypted files using the private key.

Files:

decrypt.c:
This contains the implementation and main() function for the decrypt program

encrypt.c:
This contains the implementation and main() function for the encrypt program.

keygen.c:
This contains the implementation and main() function for the keygen program

numtheory.c:
This contains the implementations of the number theory functions

numtheory.h:
This specifies the interface for the number theory functions.

randstate.c:
This contains the implementation of the random state interface for the RSA library and number theory functions.

randstate.h:
This specifies the interface for initializing and clearing the random state.

rsa.c:
This contains the implementation of the RSA library.

rsa.h:
This specifies the interface for the RSA library.

Makefile:
• CC = clang must be specified.
• CFLAGS = -Wall -Wextra -Werror -Wpedantic must be specified.
• pkg-config to locate compilation and include flags for the GMP library must be used.
• make must build the encrypt, decrypt, and keygen executables, as should make all.
• make decrypt should build only the decrypt program.
• make encrypt should build only the encrypt program.
• make keygen should build only the keygen program.
• make clean must remove all files that are compiler generated.
• make format should format all your source code, including the header files.

README.md:
This must use proper Markdown syntax and describe how to use your program and Makefile. It should also list and explain any command-line options that your program accepts. Any false positives reported by scan-build should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.

DESIGN.pdf:
This document must be a proper PDF. This design document must describe your design and design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode.

Pseudocode:

randstate.c:
Initialize global random state named state.
Void randstate_init(uint64_t seed):
    Initialize state for the mersenne twister algorithm
    Set the seed

Void randstate_clear(void):
    Clear memory used by state

numtheory.c:
Initialize d
Void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t mod):
    Set out to 1
    Set p to base
    while(d is greater than 0):
        If d is odd:
            Set out to (out * p) % mod
        Set p to (p*p) % mod

```
            Set d to the floor of d/2

bool is_prime(mpz_t n, uint64_t iters):
    Write n - 1 = 2^s*r such that r is odd
    For iters times:
        Set base to a random number 2 to n-2
        Initialize out
        Out = pow_mod(out, base, r, n)
        If out does not equal 1 and out does not equal n-1:
            Set j to 1
            While j is less than or equal to s - 1 and out is not equal to n - 1:
                Set out to pow_mod(out, 2, n)
                If y is equal to 1:
                    Return false
                Set j to j +1
            If y does not equal n - 1:
                Return false
    Return true

Void make_prime(mpz_t p, uint64_t bits, uint64_t iters):
    while(1 == 1):
        Generate a random number into p that is at least bits long
        If is_prime(p, iters):
            Break

Void gcd(mpz_t d, mpz_t a, mpz_t b):
    While b is not equal to 0
        Set temp equal to b
        Set b to a % b
        Set a to temp
    Set d to a

Void mod_inverse(mpz_t i, mpz_t a, mpz_t n):
    Set r to n
    Set r_prime to a
    Set temp to 0
    Set temp_prime to 1
    while(r_prime is not 0):
        Set q to floor of r/r_prime
        Set temp_r to r
        Set r to r_prime
        Set r_prime to temp_r - q * r_prime
        Set temp_t to temp
        Set temp to temp_prime
```

Set temp_prime to temp_t - q * temp_prime
If r is greater than 1:
Set i to 0
If t is less than 0:
Set temp to temp + n
Set i to temp

rsa.c:
Void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters):
Set rand_nbits to a random number of bits in the range nbits/4 to (3*nbits)/4
make_prime(p, rand_nbits, iters)
make_prime(q, nbits - rand_nbits, iters)
Set totient to (p-1)*(q-1)
while(true):
Generate random numbers around nbits and call the current random number curr_rand
Initialize d
gcd(d, curr_rand, totient)
if(curr_rand and totient are coprime):
Break
Set e to curr_rand

Void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile):
Write n, e, s, username\n to pbfile

Void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile):
Read the public RSA key from the pbfile

Void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q):
Set totient to (p-1)*(q-1)
mod_inverse(e, p, q):
Set d to e % totient

Void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile):
Write n, d\n to pvfile

Void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile):
Read the private RSA key from the pvfile

Void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n):
Set c to m^e%n

Void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n):
Set m to c^d%n

Void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n):
    Set s to m^d%n

bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n):
    Set t to s^e%n
    If t is equal to m:
        Return true
    Return false

Void rsa_encrypt_file(FILE *infile, FILE, *outfile, mpz_t n, mpz_t e):
    Set k to floor($\log_2$(n) -1) / 8)
    Initialize array block
    Set first element of block to 0xFF
    While there are unprocessed bytes in infile:
        Block[curr_element] = what was read from infile (at most k-1 bytes)
        Convert the read bytes to m
        Write rsa_encrypt(m) to outfile

Void rsa_decrypt_file(FILE *infile, FILE, *outfile, mpz_t n, mpz_t d):
    Set k to floor($\log_2$(n) -1) / 8)
    Initialize array block
    While there are unprocessed bytes in infile:
        Set c to scanned hexstring
        Convert c back into bytes and store them in block[curr_element]
        Write the number of bytes - 1 to outfile

Keygen.c:
Parse command line options using flags.
Open the private key and public key files from the user input or rsa.priv or rsa.pub if nothing was given
Set private key file permissions to 0600.
Initialize the seed
rsa_make_pub()
rsa_make_priv()
Using the user's username compute the signature of the username using rsa_sign()
Write the computed public and private keys to their respective files
Close files

encrypt.c:
Parse command line options using flags.
Open the input file, output file, and public key files from the user input stdin, stdout, or rsa.pub if nothing was given
Read the public key from the opened public key file

Use the username to verify the signature rsa_verify()
Encrypt the file using rsa_encrypt_file()
rsa_make_priv()
Close files

decrypt.c:
Parse command line options using flags.
Open the input file, output file, and private key files from the user input stdin, stdout, or rsa.priv if nothing was given
Read the private key from the opened private key file
Decrypt the file using rsa_decrypt_file()
Close files