asgn6: Public Key Cryptography

Student name: *Serjo Barron*

Course: *CSE13S* – Professor: *Prof. Long*
Due date: *11/11/21*

1 Program

This program will focus on public key cryptography, but will specifically implement the usage of RSA encryption and decryption. And with security and privacy being a large and important aspect of the internet, being able to send information online while keeping its contents secret is imperative to make the internet a slightly safer place. RSA, the earliest known public-key cryptography algorithm was invented by Ronald Rivest, Adi Shamir, and Leonard Adleman all the way back in 1978. This is also where the acronym RSA derives from, the last name's of its inventors.
Now what exactly is public key cryptography, and what does it entail for us? Key cryptography makes the usage of large prime factors and the fact that the larger they are, the harder they are to factorize. As such, RSA takes advantage of this by taking astronomically large prime numbers and multiplying them and doing other forms of mathematics, which is all to generate a public key, and its corresponding private key. A public key is whats available to everyone, hence the reason its called a "public key", while the private key should only be available to you and to whoever you share it to.
With a public key and private key, one can encrypt any message and send that encrypted message to a person. Although, in order to decrypt that message, its receiver must have the public keys corresponding private key. And this in brief is the idea of public key cryptography.

2 Required Files

1. keygen.c
    - This file will generate pairs of public and private keys required for encryption and decryption.
2. encrypt.c
    - This file will include the implementation of encrypting files given a file to encrypt and a public key (rsa.pub) to encrypt that file.
3. decrypt.c
    - This file will include the implementation of decrypting files given a file to decrypt and a private key (rsa.priv) to decrypt that file.

4. numtheory.h
    - This header file will include the function prototypes of numtheory.c.
5. numtheory.c
    - This file will include the implementation of various theoretic mathematic functions (the numtheory ADT).
6. randstate.h
    - This header file will include the function prototypes of randstate.c and the extern (global) variable state.
7. randstate.c
    - This file will include the means to set the global state and clear the state (the randstate ADT).
8. rsa.h
    - This header file will include the function prototypes of rsa.c.
9. rsa.c
    - This file will include the implementation of the rsa ADT.
10. Makefile
    - This file will link all the required ADT's to keygen, encrypt, and decrypt, and will produce their corresponding executables/binaries.
11. README.md
    - A file in markdown syntax which contains a program description and the instructions on how to use and run the program.
12. DESIGN.pdf
    - The design process of the program (this document).

3 Psuedocode and Function Descriptions

3.1 Number Theory

This function will find the gcd (greatest common divisor) between two numbers a and b. The method for finding the gcd, d in this case, will be done using the Euclidean algorithm. Once, the gcd is found, its value will be passed (and stored) in the parameter d.

```
gcd(d,a,b):
mpz_inits()
a1 <- a // placeholders vars (a,b)
b1 <- b
while b1 != 0:
        t <- b1
        b1 <- a1 % b1
        a1 <- t
d <- a1
mpz_clears()
```

This function computes the modular inverse of i using modulo n. Calculating the inverse of a modulo will be done using an extended version of the Euclidian algorithm, known as Bezout's identity which asserts that $ns+at=1$. Though as seen in the pseudocode below, in order to account for parallel assignments in C, we must make use of temporary variables to hold original values as to not mess up later on calculations.

```
mod_inverse(i,a,n):
mpz_inits()
r <- n
r_1 <- a
t <- 0
t_1 <- 1
while r_1!=0:
        q <- floor(r,r_1)
        tmp_r <- r
        tmp_r1 <-r_1

        r <- tmp_r1
        qr1 <- q * tmp_r1
        r_1 <- tmp_r - qr1

        tmp_t <- t
        tmp_t1 <- t_1

        t <- tmp_t1
        qt1 <- q * tmp_t1
        t_1 <- tmp_t1 - qt1
if r > 1:
        inv_exists = false
        i <- 0 (no inverse, 0)
if t < 0:
        d <- add(t,n)
        t <- d
if inv_exists:
        i <- t (inverse is t)
```

```
inv_exists = true // reset bool
mpz_clears()
```

This function will perform exponentiation on the value *base* to the exponent of *exponent*, but instead this will be done using a modular approach which is why there is also a parameter *modulus*. Calculating exponentiation of a number with modular exponentiation will drastically speed up the process as opposed to a approach of something like $n^5 = n \times n \times n \times n \times n$. The result will then be passed and stored into the parameter *out*.

```
pow_mod(out, base, exponent, modulus):
mpz_inits()
v <- 1
p <- base
tmp_e <- exponent
while tmp_e > 1:
        if tmp_e is odd:
                vp <- mult(v,p)
                vp_mod <- vp % modulus
                v <- vp_mod
        pp <- mult(p,p)
        pp_mod <- pp % modulus
        p <- pp_mod
        f_e <- floor(tmp_e,2)
        tmp_e <- f_e
out <- v
mpz_clears()
```

This function will determine whether a number n is likely to be prime using the Miller-Rabin primality test. First, we must find a suitable r in the formula $n-1 = 2^s \times r$, such that r is odd. After the fact, the test will be run a certain amount of times as specified by iters, and if none of the false cases trigger, the number n has a high likelihood to be prime. In reality, this test does not necessarily check whether a number is prime, it checks whether that number is composite, and if its not, then the number is "likely" to be prime, hence the reason why the test is run an iters amount of times.

```
is_prime(n, iters):
        mpz_inits()
        if n is 2 or 3:
                mpz_clears()
                return true
        if n <= 1 or n is even:
                mpz_clears()
                return false

        s <- 0
        r <- n - 1
        two <- 2 // used for pow_mod
```

```
        while r is even:
                s <- s + 1
                r <- floor(r, 2)

        n_1 <- n - 1
        s_1 <- s - 1
        n_3 <- n - 3
        for i in range(i <- 1, k):
                // choose random 'a' in set {2,...,n - 2}
                mpz_urandomm(rand, state, n_3) // range is {0,...,n - 4}
                a <- rand + 2 // range is now {2,...,n - 2}
                pow_mod(y,a,r,n)
                if y != 1 and y != n - 1:
                        j <- 1
                        while j < s and y != n - 1:
                                pow_mod(y,y,two,n)
                                if y == 1:
                                        mpz_clears()
                                        return false
                                j <- j + 1
                        if y != n - 1:
                                mpz_clears()
                                return false
        mpz_clears()
        return true
```

This function will generate a random prime number around $nbits$ length. This means the generated random numbers are within the range of 2^{n-1} and $2^n - 1$ bits. For each randomly generated number using $mpz\_rrandomb()$, we will use the previous function $is\_prime()$ to check that numbers primality. And if the generated number passes the $is\_prime()$ function (meaning it is likely to be prime), then we assign the parameter p the randomly generated number. Until a prime number is found, this function will continuously loop.

```
make_prime(p,bits,iters):
        mpz_inits()
        min <- 1 * 2^(bits - 1)
        while not prime_found:
                // generate a number of range [0, 2^(bits - 1) - 1]
                mpz_urandomb(rand,state,bits - 1)
                // add offset to fix range as [2^(n - 1), (2^n) - 1]
                rand <- rand + min
                if rand is prime:
                        p <- rand
                        prime_found = true
        prime_found = false
```

```
        mpz_clears()
```

3.2 Randstate

This function will initiate a global (extern) random state with the Mersenne Twister algorithm, using the specified seed in the function parameter.

```
randstate_init(seed):
        gmp_randinit_mt(state) // set state of extern var "state"
        gmp_randseed_ui(state,seed) // set seed
```

This function will clear and free memory used by the function $randstate_init()$.

```
randstate_clear():
        gmp_randclear(state) // clear the (extern) state
```

3.3 RSA

This is a helper function meant to compute the log_2 of a number n, where the result will be stored into the exponent e. The algorithm will calculate the log of the number n, by dividing it in half while simultaneously incrementing the exponent while the value of n is larger than 0. We subtract 1 at the end to account for the extra exponent.

```
log(n,e):
        mpz_inits()
        tmp_n <- |n|
        tmp_e <- 0
        while n > 0:
                tmp_e += 1
                tmp_n // 2
        e <- tmp_e - 1
        mpz_clears()
```

This function will create 2 large primes p and q, their product n, and a public exponent n, all of which will serve as the RSA public key. First, the specified number of nbits will be split up such that the nbits allocated to p plus the nbits allocated to q equals nbits. p's number of bits will be chosen first randomly in the range $(nbits/4, (3*nbits)/4)$, then q will be given the remaining bits.
Afterwards, the totient of n will be calculated using $\varphi(n) = (p-1)(q-1)$. We will then loop until we find a random number of nbits length coprime to the totient n. Once that random coprime number is found, it will be assigned to the public exponent e.

```
rsa_make_pub(p,q,n,e,nbits,iters):
        min = nbits / 4
        max = 2 * min + 1
        // range of [nbits / 4, (3 *nbits) / 4) for p bits
        // always add 1 bit to p and q to ensure n is at least nbits long
```

```
bits_for_p = 1 + (random() % max) + min
bits_for_q = 1 + nbits - bits_for_p // remaining bits go to q
make_prime(p,bits_for_p,iters)
make_prime(q,bits_for_q,iters)

compute φ(n) = (p - 1)(q - 1)

do:
        generate rand number rand (mpz_urandomb)
        gcd(g,rand,φ(n)) // get gcd of rand number and φ(n)
while (g != 1)
e <- rand // found number to be coprime with φ(n)
```

This function will write a public RSA key to the file pbfile. The format will be n, e, s, followed by the username last. Each of these values or strings will be written on their own line.

```
rsa_write_pub(n,e,s,user,pbfile):
        // use the specifier %Zx for hex values of mpz_t type
        gmp_fprintf(pbfile, hex value of n)
        gmp_fprintf(pbfile, hex value of e)
        gmp_fprintf(pbfile, hex value of s)
        // username is just a string, hence type %s
        fprintf(pbfile, username)
```

This function will read the contents of a public RSA key in the file pbfile, and save those contents by passing them back to their appropriate variables.

```
rsa_read_pub(n,e,s,user,pbfile)
        // use the specifier %Zx for hex values of mpz_t type
        gmp_fscanf(pbfile, read hex value of n, n)
        gmp_fscanf(pbfile, read hex value of e, e)
        gmp_fscanf(pbfile, read hex value of s, s)
        // username is just a string, hence type %s
        fscanf(pbfile, read username, user)
```

This function will create a private RSA key given two large primes p and q along with its public exponent e. The private RSA key will be d, and will be calculated by taking the inverse of e % $\varphi(n) = (p-1)(q-1)$.

```
rsa_make_priv(d,e,p,q):
        compute φ(n) = (p - 1)(q - 1)
        mod_inverse(d,e,n) // mod inverse of e % φ(n)
```

This function will write a private RSA key to the file pvfile. The format will be n, then d last. Each of these values or strings will be written on their own line.

```
rsa_write_priv(n,d,pvfile):
        // use the specifier %Zx for hex values of mpz_t type
        gmp_fprintf(pvfile, hex value of n)
        gmp_fprintf(pvfile, hex value of d)
```

This function will read the contents of a private RSA key in the file pvfile, and save those contents by passing them back to their appropriate variables.

```
rsa_read_priv(n,d,pvfile):
        // use the specifier %Zx for hex values of mpz_t type
        gmp_fscanf(pvfile, read hex value of n, n)
        gmp_fscanf(pvfile, read hex value of e, d)
```

This function performs RSA encryption on a message m using public exponent e and modulus m. This will produce c, which is the ciphered text. Encryption is defined as $E(m) = c = m^e (mod n)$.

```
rsa_encrypt(c,m,e,n):
        pow_mod(c,m,e,n)
```

This function will encrypt the contents on infile by reading bytes into a buffer (BLOCK), this will be done until there are no more bytes to be read. And for each instance of data being read, the contents of the BLOCK will be converted to m which in essence will concatenate strings using their ASCII value to a large number m. m will then be encrypted to c then be written to outfile as a hexstring.

```
rsa_encrypt_file(infile,outfile,n,e):
        lg(n,e) // e = log2(n)
        log2_1 <- e - 1
        k = floor(log2_1,8)
        dynamically allocate BLOCK of length k (type uint8_t)
        BLOCK[0] = 0xFF // prepend 0xFF to start of BLOCK

        nbytes = k - 1
        do:
                j = fread(BLOCK[1],1,nbytes,infile) // start buffer index 1
                mpz_import(m, j + 1,1,1,1,0,BLOCK) // convert BLOCK to m
                rsa_encrypt(c,m,e,n)
                if j > 0:
                        gmp_fprintf(outfile, write c as hexstring)
        while (j > 0)
        mpz_clears()
        free buffer
```

This function performs RSA decryption on c the ciphered text using the private key d and public modulus n. This will produce m which was the original message before encryption. Decryption is defined as $D(c) = m = c^d (mod n)$.

```
rsa_decrypt(m,c,d,n):
        pow_mod(m,c,d,n)
```

This function will decrypt the contents of infile and write the decrypted contents to outfile. This will be done by scanning in the data of infile which are hexstrings that are separated by newlines. For each hexstring read, it will be saved into c, then c will be converted back into bytes using $mpz_export()$. The converted hexstring stored in BLOCK will then be writen to outfile starting at index 1 to account for the prepended 0xFF from $rsa_encrypt_file()$. This process will be repeated until there are no more hexstrings to be scanned in from infile.

```
rsa_decrypt_file(infile,outfile,n,d):
        lg(n,e) // e = log2(n)
        log2_1 <- e - 1
        k = floor(log2_1,8)
        dynamically allocate BLOCK of length k (type uint8_t)

        do:
                gmp_fscanf(infile, scan in hexstring as c)
                rsa_decrypt(m,c,d,n)
                mpz_export(BLOCK,j,1,1,1,0,m) // convert m to bytes in BLOCK
                fwrite(BLOCK[1],1,j - 1,outfile)
        while not end of file
        mpz_clears()
        free buffer
```

This function performs RSA signing, which will produce a signature s by signing message m using the private key d and public modulus n. Signing in RSA is defined as $S(m) = s = m^d(modn)$.

```
rsa_sign(s,m,d,n):
        pow_mod(s,m,d,n)
```

This function performs RSA verification, and will return true if the signature s is verified and false otherwise. Verification is the inverse of signing, and will be determined with $t = V(s) = s^e(modn)$. The signature will be verified if and only if t is equivalent (the same) as the message m.

```
rsa_verify(m,s,e,n):
        mpz_init()
        pow_mod(t,s,e,n)
        if t == m:
                mpz_clear()
                return true
        mpz_clear()
        return false
```

---

4 Keygen

This executable will provide the means necessary to generate a public and private key pair. Since this is RSA cryptography, this will be done by first generating 2 random (preferably very large) primes p and q. Multiply the two to get n, and then find a random number e coprime to $\varphi(n)$, which will serve as the public exponent. After that we can use e and the totient of n ($\varphi(n)$) to calculate d as the modular inverse of e and $\varphi(n)$, this will be the private key.
Once done these keys will simply be printed to rsa.pub and rsa.priv (by default), which will contain information pertaining to the public key and private key.

```
while getopt loop:
        case 'h':
                print header = true
        case 'v':
                print verbose stats = true
        case 'b':
                take user specified bits
        case 'c':
                take user specified iters
        case 'n':
                take user specified pbfile name
        case 'd':
                take user specified pvfile name
        case 's':
                take user seed
        default:
                print header = true

if specified:
        print header message
        end program

if using default seed:
        seed = time(NULL) // time since UNIX epoch

open pbfile and pvfile
if opening either of the files fail:
        print error message
        exit program
use fchmod to set pvfile perms to "0600"

initiate the random state

mpz_inits()

rsa_make_pub() // make keys
```

```
rsa_make_priv()

make buffer to read username in (kilobyte/1024 bytes)
username = getenv("USER") // store user in username
set username to mpz_t with base 62
rsa_sign()

rsa_write_pub() // write keys
rsa_write_priv()

if specified:
        print verbose statistics

close pbfile and pvfile
clear the state
mpz_clears()
end program
```

---

5 Encrypt

This executable will encrypt a file using a public key generated by keygen. Encrypting will be done using RSA encryption, which as defined earlier is $E(m) = c = m^e(mod n)$. m is the message being encrypted which will be retrieved by reading in bytes from infile, and converting those bytes into a singular mpz_ t m. This process will be repeated until the file (infile) is completely encrypted.

```
while getopt loop:
        case 'h':
                print header = true
        case 'v':
                print verbose stats = true
        case 'i':
                take user specified infile
        case 'o':
                take user specified infile
        case 'n':
                take user speicifed pbfile
        default:
                print header = true

if specified:
        print header message
        end program

open pbfile
```

```
if opening pbfile fails:
        print error message
        exit program

mpz_inits()
rsa_read_pub() // read pub key

if specified:
        print verbose stats

verify that the pbfile has the same user:
rsa_verify()

rsa_encrypt_file()

close infile, outfile, and pbfile
mpz_clears()
end program
```

---

6 Decrypt

This executable will decrypt a file using the corresponding private key of the public key used to encrypt a file. Decrypting will be done using RSA decryption, which as defined earlier is $D(c) = m = c^d (mod n)$. c is the ciphertext which was calculated previously in $E(m)$, and will be inversed to retrieve the original message m (still an mpz_ t). We will then take the mpz_ t m, and convert it back to its original bytes which can then be written to outfile. This process will be repeated until the encrypted file is completely decrypted.

```
while getopt loop:
        case 'h':
                print header = true
        case 'v':
                print verbose stats = true
        case 'i':
                take user specified infile
        case 'o':
                take user specified outfile
        case 'n':
                take user specified pvfile
        default:
                print header = true

if specified:
        print header message
        end program
```

```
open pvfile
if opening pvfile fails:
        print error message
        exit program

mpz_inits()
rsa_read_priv() // read priv key

if specified:
        print verbose stats

rsa_decrypt_file()

close infile, outfile, and pvfile
mpz_clears()
```

---

7 Error Handling

All executables include minor error handling regarding opening files, and making sure they have been successfuly opened (not null), before deciding to continue to run the program. However, encrypt includes rsa_ verify() which will compare the signature s retrieved from the public key file and compare that to the original message (the mpz_ t of the username of base 62). Once the user has been verified (the user and signature s are equivalent), then the encryption can continue.

8 Credit

1. All the involved mathematics and function designs have been based off of the provided assignment 6 specifications/document by Prof. Long.
2. The method of calculating $n-1 = 2^s \times r$, such that r is odd has also been inspired by Eugene Chou's explanations in sections.
3. The log function, lg(n,e), used in rsa.c is also based off of Prof. Long's (python) method of computing log base 2.