Asgn 6 Design Doc

Elliott Jensen

November 2021

1 Introduction

This program uses the RSA algorithm to generate public and private keys and to encrypt and decrypt files. It uses three main files: keygen.c encrypt.c and decrypt.c which include calls to main, and then three library files: rsa.c, numtheory.c and randstate.c. This program heavily relies on the gmp library in order to store arbitrarily long integers that make up the public and private key.

2 Keygen: keygen.c

Keygen.c relies on numtheory.c, rsa.c and randstate.c in order to generate random public and private key pairs. First, keygen.c establishes the seed and state variables through calls to randstate like this

```
seed = time(NULL);
if 's':
    seed = optarg
rand_state_init(seed)
randstate_clear()
```

Once the seed has been created keygen then creates the public and private key like so:

```
rsa_make_pub(t,q,n,e,nbits,iters)
rsa_make_priv(t,d,e,p,q)
rsa_write_pub(t,n,e,s,username,pbfile)
rsa_write_priv(n,d,pvfile)
```

3 Randstate.c

This library uses seeds to generate a random state. It contains two functions: randstate_init and randstate_clear. It relies on the gmp library to generate large numbers based on the seed. It looks like this:

```
def randstate_init(seed):
    gmp_randinit_mt(state)
    gmp_randseed_ui(state,seed)

def randstate_clear():
    gmp_randclear(state)
```

4 Number Theory Functions

Here I implement a library of arithmetic functions. Included are: power mod, mod inverse, greatest common denominator, is prime and make prime. In all of these functions it is important that the only argument that is changed is the first argument which acts as a return value. To do this I duplicate all other arguments and create temporary variables as seen here:

```
def gcd(mpz_tg, mpz_t a, mpz_t b):
    a1 = a
    b1 = b
    while b != 0:
        temp = b1
        b1 = a1 % b1
        a1 = temp
    d = a1

def pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t n):
    v=1
    p=a
    exp = exponent
    while d >= 0:
        if mpz_mod(i,2) == 1:
            v= v*p %n
        p = p * p % n
        d = d / 2
    out = v
```

The mod inverse function is tricky because it contains so many parallel assignment operations. To address this I use temporary variables to hold the original values of the variables and then perform the assignments sequentially like so:

```
def mod_inverse(mpz_t i, mpz_t a, mpz_t n):
    r = n
    r_prime = a
    while (r_prime != 0):
        q = r / r_prime
        temp = r
        r_prime = temp - q * r_prime
        temp = t
        t = t_prime
        t_prime = t = q * t_prime
    if r > 1:
        return
    if t < 0:
        t = t + n
    return t
```

Also included in numtheory.c is the is_prime function and make_prime function. is_prime relies on the Miller-Rabin Primality test in order to probablistically generate prime numbers. To do so I rely on the assignment doc sudo code:

```
MILLER-RABIN(n,k)
write n1 = 2sr such that r is odd
for i ←1 to k
    choose random a {2,3,...,n2}
    y = POWERMOD(a,r,n)
    if y 6= 1 and y 6= n1
    j ←1
    while j s1 and y 6= n1
        y ←POW E R-MO D(y,2,n)
        if y == 1
            return FALSE
        j ←j+1
        if y 6= n1
            return FALSE
return TRUE
```

In make_prime random numbers at least nbits long are generated. Many random numbers are generated and once a prime number with the correct number of bits are found that number is returned. To generate the random numbers I use the function mpz_urandommb, which accepts an argument to the the maximum of the random number. Since we want to set a minimum rather than a maximum I add 2 ̂(n-1). This puts a 1 in the left most bit, which causes the nbits in the arguments to be treated as a minimum rather than a maximum. The code looks like this:

```
    firstbit = 2 ^ (n - 1)

    while(true):
        generate random number with range equal to nbits
        randomnum += firstbit
        if prime(randomnum):
            break
```

5 RSA Libraray: rsa.c

This library contains the primary functions needed to generate public/private key pairs. It computes the components of the key: p,q,n and e which make up the public and private keys. To do this the function relies on the numtheory.c library in order to generate prime numbers. This file also contains the read and write functions for both the public and private key. Most of these functions are one liners that can be put together by copying the sudo code in the assignment doc, but three of them, rsa_make_pub, encrypt_file, and decrypt_file are more complex. This is what rsa_make_pub looks like:

```
p_range = [nbits/4,(3×nbits)/4)
q_range = n - p_range
phi = (p - 1) * (q - 1)

while true:
    e = random number
    if gcd(e,phi) == 1:
        break
```

rsa_encrypt_file loads bytes from the input file into an array, encrypts blocks of that array and then writes the encrypted bytes onto an output file. It looks like this:

```
create array
array[0] == 0xFF
while (not at EOF):
    read from infile in blocks
    import text from the infile into the array
    encrypt plaintext and output ciphertext
    write the encrypted bytes to the outfile
```

rsa_decrypt_file is like encrypt but in reverse:

```
create array
array[0] = 0xFF
while (not at EOF):
    scan cipher text from the input file
    export cipher text into the array
    decrypt ciphertext and output plain text
    write plain text to to the output file
```

6 Encryption: encrypt.c

This file encrypts files. First the file parses through command line options in order to establish the input file, output file and whether or not the user wants verbose printing. Second, the program reads from the public key and if verbose printing is enabled, it prints out the username, signature, public modulus and public exponent. The file is then encrypted with a call to rsa_encrypt_file().

```
>> parse through command line options <<
rsa_read_pub(n,e,s,pbfie)
mpz_set_str(m,username,62)
if !rsa_verify:
    print error
    exit
rsa_encrypt_file(infile,outfile,n,e)
```

7 Decryption: decrypt.c

Similarly, but in reverse, decrypt parses command line options and reads the private key. If verbose printing is enabled, the program prints the public modulus n and the private key e. It then decryps the file with a call to rsa_decrypt_file().

```
>> parse through command line options <<
rsa_read_priv(n, d, pvfile)
rsa_decrypt_file(infile,outfile,n,d)
```