CSE 13s Assignment 5 Design

caowen

February 2023

1 Description of Program

This assignment asks for an implementation of 3 separate programs. These programs together will allow the user to create public and private SS keys and use those keys to encrypt and decrypt data. The *keygen* program is used to generate the public/private SS key pairs. The *encrypt* program uses the public key to encrypt the data, and the *decrypt* program uses the paired private key to decrypt the encrypted data. Each program has their own supplemental functions and libraries which must be used or implemented in order for all of the programs to work fully in conjunction with one another.

2 Key Generation

Given a specified number of bits for the public modulus n, values p and q, both extremely large prime numbers are generated randomly, such that the product p*p*q has at least *nbits* bits. This product becomes n, the public modulus, otherwise known as the public key.

The private key is generated next, using p and q. The private key is computed as the modular inverse of n (p*p*q) and lcm(p-1, q-1).

3 Encryption

The encryption is completed by splitting the input into blocks and encrypting each block individually. The block size, in bytes, is computed as floor((log$_2$($\sqrt{n}$) - 1) / 8). Each block is then encrypted using the public key n, and the block message converted into a numerical value, m. This is computed as the power modulus of m, n, and n. Each block is then written to a specified output file with a newline trailing each encrypted block.

4 Decryption

The decryption is completed similarly to the encryption. The message must be decrypted in blocks the same way as the encryption. However without knowing

the value of n, the public key, we cannot calculate k, the number of bytes to allocate to the buffer block. Instead, we estimate using pq, calculating k as floor(($\log_2$(pq) - 1) / 8). We then read each line from the encrypted file, decrypyting it as the power modulus of c, d, and pq. Each decrypted block is then written to the specified output file.

5 Files to be included in directory

- decrypt.c
- encrypt.c
- keygen.c
- numtheory.c
- numtheory.h
- randstate.c
- randstate.h
- ss.c
- ss.h

6 Pseudocode

- randstate.c functions
  - void randstate_init(uint64_t seed):
    ```
    gmp_randinit_mt(state)
    gmp_randseed_ui(state, seed)
    srand(seed)
    ```
  - void randstate_clear(void):
    ```
    gmp_randclear(state)
    ```
- numtheory.c functions
  - void pow_mod(mpz_t o, mpz_t a, mpz_t d, mpz_t n):
    ```
    v = 1
    p = a
    while(d > 0):
        if(d % 2 != 0):
            v = (v*p) % n
        p = (p*p) % n
        d = floor(d / 2)
    o = v
    ```
  - bool is_prime(mpz_t n, uint64_t iters):
    ```
    r = n - 1
    s = 0
    i = 1
    while(r % 2 == 0):
        r /= 2
        s += 1
    for i in range(iters):
        a = random(2, n-2)
        y = 0
        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, 2, n)
                if(y == 1):
                    return false
                j++
            if(y != n-1):
                return false
    return true
    ```

- void make_prime(mpz_t p, uint64_t bits, uint64_t iters):
  do:
    p = random with max bits + 1 bits
  while(!is_prime(p, iters) or p bits <= bits):

- void gcd(mpz_t g, mpz_t a, mpz_t b):
  t = 0
  while(b != 0):
    t = b
    b = a % b
    a = t
  g = a

- void mod_inverse(mpz_t o, mpz_t a, mpz_t n):
  r = n
  _r = a
  t = 0
  _t = 1
  q = 0
  temp = 0
  while(_r != 0):
    q = floor(r / _r)
    temp = r
    r = _r
    _r = temp - q * _r
    temp = t
    t = _t
    _t = temp - q * _t
  if(r > 1):
    o = 0
    return
  if(t < 0):
    t = t + n
  o = t

- ss.c functions
  - void ss_make_pub(mpz_t p, mpz_t q, mpz_t n, uint64_t nbits, uint64_t iters):
    p_bits = random(nbits/5, (2*nbits)/5)
    q_bits = nbits - (2*p_bits)
    do:
      p = make_prime(p, p_bits, iters)
      q = make_prime(q, q_bits, iters)
      n = p*p*q
    while(p % q - 1 == 0 or q % p - 1 == 0 or n bits < nbits)
  - void ss_write_pub(mpz_t n, char username[], FILE *pbfile):
    gmp_fprintf(pbfile, "%Zx\n%s\n", n, username)
  - void ss_read_pub(mpz_t n, char username[], FILE *pbfile):
    gmp_fscanf(pbfile, "%Zx\n%s\n", n, username)
  - void ss_make_priv(mpz_t d, mpz_t pq, mpz_t p, mpz_t q):
    pq = p * q
    n = pq*p
    lcm = ((p-1) / gcd(p-1, q-1)) * (q-1)
    mod_inverse(d, n, lcm)
  - void ss_write_priv(mpz_t pq, mpz_t d, FILE *pvfile):
    gmp_fprintf(pvfile, "%Zx\n%Zx\n", pq, d);
  - void ss_read_priv(mpz_t pq, mpz_t d, FILE *pvfile):
    gmp_fscanf(pvfile, "%Zx\n%Zx\n", pq, d);
  - void ss_encrypt(mpz_t c, mpz_t m, mpz_t n):
    pow_mod(c, m, n, n)
  - void ss_encrypt_file(FILE *infile, FILE *outfile, mpz_t n):
    next_pow = 0
    k = 0
    do:
      k += 1
      next_pow = 2^k
    while (next_pow < $\sqrt{n}$)
    k -= 2
    k /= 8
    allocate k bytes to uint8_t pointer block
    block[0] = 0xFF

j = 0
while((j = fread(&block[1], sizeof(char), k - 1, infile)) > 0):
mpz_import(m, j+1, 1, sizeof(char), 1, 0, block)
ss_encrypt(m, m, n)
gmp_fprintf(outfile, "%Zx\n", m)

- void ss_decrypt(mpz_t m, mpz_t c, mpz_t d, mpt_t pq):
  pow_mod(m, c, d, pq)

- void ss_decrypt_file(FILE *infile, FILe *outfile, mpz_t pq, mpz_t d):
  next_pow = 0
  k = 0
  do:
  k += 1
  next_pow = 2^k
  while(next_pow <= pq)
  k -= 2
  k /= 8
  allocate k bytes to uint8_t pointer block
  while(gmp_fscanf(infile, "%Zx\n", c) > 0):
  ss_decrypt(m, c, d, pq)
  j = 0
  mpz_export(block, &j, 1, sizeof(char), 1, 0, m)
  fwrite(&block[1], sizeof(char), j-1, outfile)

- keygen.c functions
  - void usage(char *exec):
    print synopsis and usage of keygen.c
  - int main(int argc, char **argv):
    seed = time(NULL)
    username = getlogin()
    pbfile_path = "ss.pub"
    pvfile_path = "ss.priv"
    bool verbose = false
    nbits = 256
    iters = 50
    iterate through argv:
    b: nbits = optarg
    i: iters = optarg
    n: pbfile_path = optarg
    d: pvfile_path = optarg
    s: seed = optarg
    v: verbose = true

default: usage(argv[0])
randstate_init(seed)
ss_make_pub(p, q, n, nbits, iters)
ss_make_priv(d, pq, p, q)
if(verbose):
print verbose statements on p, q, n, pq, d
pbfile = fopen(pbfile_path, "w")
ss_write_pub(n, username, pbfile)
pvfile = fopen(pvfile_path, "w")
ss_write_priv(pq, d, pvfile)

- encrypt.c functions
  - void usage(char *exec):
    print synopsis and usage of encrypt.c
  - int main(int argc, char **argv):
    n = 0
    allocate 100 bytes to username
    pbfile = fopen("ss.pub", "r")
    infile = stdin
    outfile = stdout
    verbose = false
    iterate through argv:
    i: infile = fopen(optarg, "r")
    o: outfile = fopen(optarg, "w")
    n: pbfile = fopen(optarg, "r")
    v: verbose = true
    default: usage(argv[0])
    ss_read_pub(n, username, pbfile)
    if(verbose):
    print verbose statements on username and n
    ss_encrypt_file(infile, outfile, n)

- decrypt.c functions
  - void usage(char *exec):
    print synopsis and usage of decrypt.c
  - int main(int argc, char **argv):
    pq, d = 0
    pvfile = fopen("ss.priv", "r")
    infile = stdin
    outfile = stdout
    verbose = false

```
iterate through argv:
    i: infile = fopen(optarg, "r")
    o: outfile = fopen(optarg, "w")
    n: pvfile = fopen(optarg, "r")
    v: verbose = true
    default: usage(argv[0])
ss_read_priv(pq, d, pvfile)
if(verbose):
    print verbose statements on pq and d
ss_decrypt_file(infile, outfile, pq, d)
```