Assignment 5 Public Key Cryptography

Design Document

Derrick Ko - Winter 2023

1 Overview

2 randstate.c

2.1 void randstate_init(uint64_t seed)

Initializes the global random state named state with a Mersenne Twister algorithm, using seed as the random seed. You should call srandom() using this seed as well. This function will also entail calls to gmp_randinit_mt() and to gmp_randseed_ui().

- Initialize the random state with the Mersenne Twister algorithm.
- Set the random seed using the provided seed value.
- Initialize the random seed for srandom() as well.

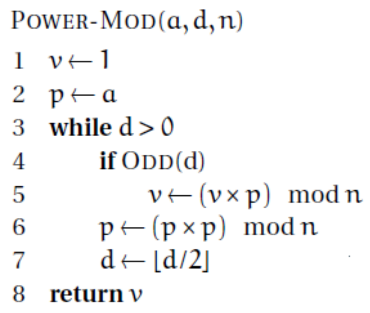
2.2 void randstate_clear(void)

Clears and frees all memory used by the initialized global random state named state. This should just be a single call to gmp_randclear().

3 numtheory.c

3.1 void pow_mod()

Performs fast modular exponentiation, computing base raised to the exponent power modulo modulus, and storing the computed result in out.



```
POWER-MOD($a, d, n$)
$v \leftarrow 1$
$p \leftarrow a$
while $d > 0$
    if ODD($d$)
        $v \leftarrow (v \times p) \bmod n$
    $p \leftarrow (p \times p) \bmod n$
    $d \leftarrow \lfloor d/2 \rfloor$
return $v$
```

3.2 bool is_prime()

Conducts the Miller-Rabin primality test to indicate whether or not n is prime using iters number of Miller-Rabin iterations. This function is needed when creating the two large primes p and q in SS, verifyingif a large integer is a prime.

```
MILLER-RABIN($n, k$)
write $n - 1 = 2^s r$ such that $r$ is odd
for $i \leftarrow 1$ to $k$
    choose random $a \in \{2, 3, \ldots, n-2\}$
    $y = $ POWER-MOD($a, r, n$)
    if $y \neq 1$ and $y \neq n - 1$
        $j \leftarrow 1$
        while $j \leq s - 1$ and $y \neq n - 1$
            $y \leftarrow$ POWER-MOD($y, 2, n$)
            if $y == 1$
                return FALSE
            $j \leftarrow j + 1$
        if $y \neq n - 1$
            return FALSE
return TRUE
```

3.3 void make_prime()

Generates a new prime number stored in p. The generated prime should be at least bits number of bits long. The primality of the generated number should be tested using is_prime() using iters number of iterations.

3.4 void gcd()

Computes the greatest common divisor of a and b, storing the value of the computed divisor in d.

```
GCD($a, b$)
while $b \neq 0$
    $t \leftarrow b$
    $b \leftarrow a \bmod b$
    $a \leftarrow t$
return $a$
```

3.5 void mod_inverse()

Computes the inverse i of a modulo n. In the event that a modular inverse cannot be found, set i to 0. Note that this pseudocode uses parallel assignments, which

C does not support. Thus, you will need to use auxiliary variables to fake the parallel assignments.

```
MOD-INVERSE($a, n$)
1  $(r, r') \leftarrow (n, a)$
2  $(t, t') \leftarrow (0, 1)$
3  while $r' \neq 0$
4      $q \leftarrow \lfloor r/r' \rfloor$
5      $(r, r') \leftarrow (r', r - q \times r')$
6      $(t, t') \leftarrow (t', t - q \times t')$
7  if $r > 1$
8      return no inverse
9  if $t < 0$
10     $t \leftarrow t + n$
11 return $t$
```

4 ss.c

4.1 void ss_make_pub()

Creates parts of a new SS public key: two large primes p and q, and n computed as $p \times p \times q$.

1. Begin by creating primes p and q using make_prime(). We first need to decide the number of bits that go to each prime respectively such that log2(n) ¸ nbits. Let the number of bits for p be a random number in the range $nbits/5(2 \times nbits)/5)$. Recall that $n = p^2 \times q$: the bits from p will be contributed to n twice, the remaining bits will go to q. The number ofMiller-Rabin iterations is specified by iters. You should obtain this random number using random() and check that $p \nmid q-1$ and $q \nmid p-1$.
2. Next, compute $\lambda(n) = lcm(p-1, q-1)$. You will need to compute $gcd(p-1, q-1)$ for this.

4.2 void ss_write_pub()

Writes a public SS key to pbfile. The format of a public key should be n, then the username, each of which are written with a trailing newline. The value n should be written as a hexstring.

4.3 void ss_read_pub()

Reads a public SS key from pbfile. The format of a public key should be n, then the username, each of which should have been written with a trailing newline. The value n should have been written as a hexstring.

4.4 void ss_make_priv()

Creates a new SS private key d given primes p and q and the public key n. To compute d, simply compute the inverse of n modulo $\lambda(pq) = lcm(p-1, q-1)$.

4.5 void ss_write_priv()

Writes a private SS key to pvfile. The format of a private key should be pq then d, both of which are written with a trailing newline. Both these values should be written as hexstrings.

4.6 void SS_read_priv()

Reads a private SS key from pvfile. The format of a private key should be pq then d, both of which should have been written with a trailing newline. Both these values should have been written as hexstrings.

4.7 void ss_encrypt()

Performs SS encryption, computing the ciphertext c by encrypting message m using the public key n. Remember, encryption with SS is defined as $E(m) = c = m^n (modn)$.

4.8 void ss_encrypt_file()

Encrypts the contents of infile, writing the encrypted contents to outfile. The data in infile should be in encrypted in blocks. Why not encrypt the entire file? Because of n. We are working modulo n, which means that the value of the block of data we are encrypting must be strictly less than n. We have two additional restrictions on the values of the blocks we encrypt:

1. The value of a block cannot be 0: $E(0) \equiv 0 \equiv 0^n (modn)$.
2. The value of a block cannot be 1. $E(1) \equiv 1 \equiv 1^n (modn)$.

A solution to these additional restrictions is to simply prepend a single byte to the front of the block we want to encrypt. The value of the prepended byte will be 0xFF. This solution is not unlike the padding schemes such as PKCS and OAEP used in modern constructions of RSA. To encrypt a file, follow these steps:

1. Calculate the block size k. This should be $k = \lfloor (\log_2 \sqrt{n} - 1)/8 \rfloor$.
2. Dynamically allocate an array that can hold k bytes. This array should be of type (uint8_t *) and will serve as the block.
3. Set the zeroth byte of the block to 0xFF. This effectively prepends the workaround byte that we need.

4. While there are still unprocessed bytes in infile:

    (a) Read at most k-1 bytes in from infile, and let j be the number of bytes actually read. Place the read bytes into the allocated block starting from index 1 so as to not overwrite the 0xFF.

    (b) Using mpz_import(), convert the read bytes, including the prepended 0xFF into an mpz_t m. You will want to set the order parameter of mpz_import() to 1 for most significant word first, 1 for the endian parameter, and 0 for the nails parameter.

    (c) Encrypt m with ss_encrypt(), then write the encrypted number to outfile as a hexstring followed by a trailing newline.

4.9 void ss_decrypt()

Performs SS decryption, computing message m by decrypting ciphertext c using private key d and public modulus n. Remember, decryption with SS is defined as $D(c) = m = c^d (mod pq)$.

4.10 void ss_decrypt_file()

Decrypts the contents of infile, writing the decrypted contents to outfile. The data in infile should be decrypted in blocks to mirror how ss_encrypt_file() encrypts in blocks. To decrypt a file, follow these steps:

1. Dynamically allocate an array that can hold $k = \lfloor (\log_2 \sqrt{n}(pq) - 1)/8 \rfloor$ bytes. This array should be of type (uint8_t *) and will serve as the block.

    - We need to ensure that our buffer is able to hold at least the number of bits that were used during the enryption process. In this context we don't know the value of n, but we can overestimate the number of bits in $\sqrt{n}$ using pq.

$$\log_2 \sqrt{p^2 \times q}$$

2. Iterating over the lines in infile:

    (a) Scan in a hexstring, saving the hexstring as a mpz_t c. Remember, each block is written as a hexstring with a trailing newline when encrypting a file.

    (b) First decrypt c back into its original value m. Then, using mpz_export(), convert m back into bytes, storing them in the allocated block. Let j be the number of bytes actually converted.You will want to set the order parameter of mpz_export() to 1 for most significant word first, 1 for the endian parameter, and 0 for the nails parameter.

    (c) Write out j-1 bytes starting from index 1 of the block to outfile. This is because index 0 must be prepended 0xFF. Do not output the 0xFF.

5 Keygen

Your key generator program should accept the following command-line options:

- -b : specifies the minimum bits needed for the public modulus n.
- -i : specifies the number ofMiller-Rabin iterations for testing primes (default: 50).
- -n pbfile : specifies the public key file (default: ss.pub).
- -d pvfile : specifies the private key file (default: ss.priv).
- -s : specifies the random seed for the random state initialization (default: the seconds since the UNIX epoch, given by time(NULL)).
- -v : enables verbose output.
- -h : displays program synopsis and usage.

The program should follow these steps:

1. Parse command-line options using getopt() and handle them accordingly.
2. Open the public and private key files using fopen(). Print a helpful error and exit the program in the event of failure.
3. Using fchmod() and fileno(), make sure that the private key file permissions are set to 0600, indicating read and write permissions for the user, and no permissions for anyone else.
4. Initialize the random state using randstate_init(), using the set seed.
5. Make the public and private keys with make_pub() and make_priv()
6. Get the current user's name as a string. You will want to use getenv().
7. Write the computed public and private key to their respective files.
8. If verbose output is enabled print the following, each with a trailing newline, in order:

(a) username

(b) the first large prime p

(c) the second large prime q

(d) the public key n

(e) the private exponent d

(f) the private modulus pq
9. Close the public and private key files, clear the random state with randstate_clear(), and clear any mpz_t variables you may have used.

All of the mpz_t values should be printed with information about the number of bits that constitute them, along with their respective values in decimal. See the reference key generator program for an example.

6 Encryptor

Your encryptor program should accept the following command-line options:

- -i : specifies the input file to encrypt (default: stdin).
- -o : specifies the output file to encrypt (default: stdout).
- -n : specifies the file containing the public key (default: ss.pub).
- -v : enables verbose output.
- -h : displays program synopsis and usage.

1. Parse command-line options using getopt() and handle them accordingly.
2. Open the public key file using fopen(). Print a helpful error and exit the program in the event of failure.
3. Read the public key from the opened public key file.
4. If verbose output is enabled print the following, each with a trailing newline, in order:
   (a) username
   (b) the public key n
   All of the mpz_t values should be printed with information about the number of bits that constitute them, along with their respective values in decimal. See the reference encryptor program for an example.
5. Encrypt the file using ss_encrypt_file().
6. Close the public key file and clear any mpz_t variables you have used.

7 Decryptor

Your Decryptor program should accept the following command-line options:

- -i : specifies the input file to decrypt (default: stdin).

- -o : specifies the output file to decrypt (default: stdout).
- -n : specifies the file containing the private key (default: ss.priv).
- -v : enables verbose output.
- -h : displays program synopsis and usage.

The program should follow these steps:

1. Parse command-line options using getopt() and handle them accordingly.
2. Open the private key file using fopen(). Print a helpful error and exit the program in the event of failure.
3. Read the private key from the opened private key file.
4. If verbose output is enabled print the following, each with a trailing newline, in order:
   (a) the private modulus pq
   (b) the private key d

   Both these values should be printed with information about the number of bits that constitute them, along with their respective values in decimal. See the reference decryptor program for an example.
5. Decrypt the file using ss_decrypt_file().
6. Close the private key file and clear any mpz_t variables you have used.