Assignment 5 DESIGN.pdf

Roman Luo

February 21, 2023

1 Description of Program

- A key generator: `keygen`
- An encryptor: `encrypt`
- A decryptor: `decrypt`

The `keygen` program will be in charge of key generation, producing SS public and private key pairs. The `encrypt` program will encrypt files using a public key, and the `decrypt` program will decrypt the encrypted files using the corresponding private key.

You will need to implement two libraries and a random state module that will be used in each of your programs. One of the libraries will hold functions relating to the mathematics behind SS, and the other library itself will contain implementations of routines for SS. You also need to learn to use a library: the GNU multiple precision arithmetic library.

2 Files to be included in directory "asgn5":

- Major files
  1. `decrypt.c`: This contains the implementation and `main()` function for the `decrypt` program.
  2. `encrypt.c`: This contains the implementation and `main()` function for the `encrypt` program.
  3. `keygen.c`: This contains the implementation and `main()` function for the `keygen` program.
  4. `numtheory.c`: This contains the implementations of the number theory functions.
  5. `numtheory.h`: This specifies the interface for the number theory functions.
  6. `randstate.c`: This contains the implementation of the random state interface for the SS library and number theory functions.
  7. `randstate.h`: This specifies the interface for initializing and clearing the random state.
  8. `ss.c`: This contains the implementation of the SS library.
  9. `ss.h`: This specifies the interface for the SS library.
- Minor files
  1. Makefile
  2. README.md
  3. DESIGN.pdf
  4. WRITEUP.pdf

3 Pseudocode / numtheory.c:

3.1 Power Mod

```
function POWER-MOD(a, d, n):
    use mpz initialize
    use mpz set
    v ← 1
    p ← a
    while mpz_cmp d > 0 do
        if d % 2 == 1 then
            v ← (v × p) mod n
        end if
        p ← (p × p) mod n
        d ← d / 2
    end while
    return v
    clear all mpzs
end function
```

3.2 MILLER-RABIN/ is prime

```
    MILLER-RABIN(n, k)
    use mpz initialize
    use mpz set

    Write n-1 = 2^s * r such that r is odd

    For i from 1 to k do:
        Choose a random integer a from the range {2, 3, ..., n-2}
        Compute y = POWER-MOD(a,r,n)

        If y is not equal to 1 and y is not equal to n-1, then do:
            Set j to 1
            While j is less than or equal to s-1 and y is not equal to n-1, do:
                Set y to POWER-MOD(y,2,n)
                If y equals 1, then return FALSE
                Set j to j+1
            If y is not equal to n-1, then return FALSE
    Return TRUE
    clear all the mpzs
```

3.3 GCD

```
    use mpz initialize
    use mpz set
    GCD(a, b) {
    while b is not 0
        set t to b
        b to a mod b
        a to t
    return a
    clear all mpzs
    }
```

3.4 mod inverse

```
Initialize all mpz variables
Set (r, r0) to (n, a)
Set (t, t0) to (0, 1)

while r0 is not equal to 0
    Set q to r divided by r0
    Set (r, r0) to (r0, r - q * r0)
    Set (t, t0) to (t0, t - q * t0)
    if r is greater than 1
        Return no inverse
    if t is less than 0
        Add n to t
Clear all mpz variables
Return t
```

4 Pseudocode / randstate:

```
iniialize state

//
// Initializes the random state needed for SS key generation operations.
// Must be called before any key generation or number theory operations are used.
//
// seed: the seed to seed the random state with.
//
void randstate_init(uint64_t seed) {
    //call gmp functions to initialize state
    //use srandom to initialize seed
    //use gmp function to set seed and state
}

//
// Frees any memory used by the initialized random state.
// Must be called after all key generation or number theory operations are used.
//
void randstate_clear(void) {
   use gmp function to clear state
}
```

5 Pseudocode / ss.c:

5.1 ss make pub

```
    Initialize the variables p_value, q_value, p_minus_1, and q_minus_1 as mpz_t types.
    Generate two flags, p_flag and q_flag, and initialize both to true.
    Calculate the number of bits for p (p_bits)
    by generating a random number between nbits/5 and 2*nbits/5.
    Calculate the number of bits for q (q_bits)
    by subtracting twice the number of bits in p from nbits.
    While p_flag is true or q_flag is true, do the following:
        a. Generate two prime numbers, p_value and q_value,
        both with the number of bits specified by p_bits and q_bits, respectively.
        b. Check if p divides (q-1). If it does, set p_flag to true and
        continue with the next iteration of the loop.
        c. Check if q divides (p-1). If it does, set q_flag to true and
        continue with the next iteration of the loop.
        d. If both checks pass, set p_flag and q_flag to false to exit the loop.
        Calculate the value of n by setting n equal to the product of p_value squared and q_value.
        Copy the values of p_value and q_value to the output parameters p and q.
        Clear the variables p_value, q_value, p_minus_1, and q_minus_1.
```

5.2 ss make priv

```
// Initialize variables
n, p_minus_1, q_minus_1, phi_pq, gcd_pq, lamda_n = 0

// Calculate pq
pq = p * q

// Calculate p - 1 and q - 1
p_minus_1 = p - 1
q_minus_1 = q - 1

// Calculate phi(pq)
phi_pq = p_minus_1 * q_minus_1

// Calculate gcd(p-1, q-1)
gcd_pq = gcd(p_minus_1, q_minus_1)

// Calculate lamda(n)
lamda_n = phi_pq / gcd_pq

// Calculate n using pq
n = p * pq

// Calculate d as the inverse of n modulo lamda_n
d = mod_inverse(n, lamda_n)

// Clean up variables
clear(n, p_minus_1, q_minus_1, phi_pq, gcd_pq, lamda_n)
```

5.3 ss read

```
function ss_read_pub(n, username, pbfile):
    // Read n and username from pbfile as a hexstring
    gmp_fscanf(pbfile, "%ZX\n%s\n", n, username)
end function

function ss_read_priv(pq, d, pvfile):
    // Read pq and d from pvfile as hexstrings
    gmp_fscanf(pvfile, "%ZX\n%ZX\n", pq, d)
end function
```

5.4 ss write

```
function ss_write_pub(n, username, pbfile):
    // Write n as a hexstring along with username to pbfile
    gmp_fprintf(pbfile, "%ZX\n%s\n", n, username)
end function
```

```
function ss_write_priv(pq, d, pvfile):
    // Write pq and d as hexstrings to pvfile
    gmp_fprintf(pvfile, "%ZX\n%ZX\n", pq, d)
end function
```

5.5 ss encrypt

```
pow_mod(c, m, n, n);
```

5.6 ss decrypt

```
pow_mod(m, c, d, pq);
```

5.7 ss encrypt file

```
// Initialize variables
sqrt_n = 0
k = 0

// Calculate the block size k
mpz_init(sqrt_n)
mpz_sqrt(sqrt_n, n)
k = (mpz_sizeinbase(sqrt_n, 2) - 1) / 8

// Allocate a block of k bytes
block = (uint8_t *)malloc(k * sizeof(uint8_t))

// Prepend the workaround byte to the block
block[0] = 0xFF

// While there are still unprocessed bytes in infile
while not end of file:
    // Read at most k-1 bytes into the allocated block starting from index 1
    j = fread(&block[1], sizeof(uint8_t), k - 1, infile)

    // Convert the read bytes, including the prepended 0xFF, into an mpz_t m
    // Set the order parameter to 1 for most significant word first,
    // and set the endian and nails parameters to 1 and 0, respectively.
    m = 0
    mpz_init(m)
    mpz_import(m, j + 1, 1, sizeof(uint8_t), 1, 0, block)

    // Encrypt m using ss_encrypt()
    c = 0
    mpz_init(c)
    ss_encrypt(c, m, n)

    // Write the encrypted number to outfile as a hexstring, followed by a trailing newline
    gmp_fprintf(outfile, "%ZX\n", c)

    // Clean up variables
    mpz_clears(m, c, NULL)
end while

// Clean up
mpz_clear(sqrt_n)
```

```
free(block)
```

5.8 ss decrypt file

```
// Initialize variables
c = 0
m = 0
mpz_inits(c, m, NULL)
k = 0

// Calculate the block size k
k = (mpz_sizeinbase(pq, 2) - 1) / 8

// Allocate a block of k bytes
block = (uint8_t *)malloc(k * sizeof(uint8_t))

// Iterate over the lines in infile
while not end of file:
    // Scan in a hexstring from infile and save it as a mpz_t c
    gmp_fscanf(infile, "%ZX\n", c)

    // Decrypt c back into its original value m
    ss_decrypt(m, c, d, pq)

    // Convert m back into bytes using mpz_export()
    // Set the order parameter to 1 for most significant word first,
    // and set the endian and nails parameters to 1 and 0, respectively.
    j = 0
    mpz_export(&block[1], &j, 1, sizeof(uint8_t), 1, 0, m)

    // Write j-1 bytes starting from index 1 of the block to outfile
    // Do not output the 0xFF prepended to the block
    fwrite(&block[1], sizeof(uint8_t), j-1, outfile)
end while

// Clean up variables
mpz_clears(c, m, NULL)
free(block)
```

6 Pseudocode / keygen, encrypt, decrypt:

6.1 keygen

```
Parse command-line options using getopt() and handle them accordingly.
Open the public and private key files using fopen().
Using fchmod() and fileno(), make sure that the private key file permissions are set to 0600.
Initialize the random state using randstate_init(), using the set seed.
Make the public and private keys using ss_make_pub() and ss_make_priv(), respectively.
Get the current user's name as a string. You will want to use getenv().
Write the computed public and private key to their respective files.
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
```

6.2 encrypt

```
Parse command-line options using getopt():
a. Set default values for options
b. While there are more options to parse:
i. Check the option flag and update the corresponding
variable with the value provided by the user
ii. If the input or output files are provided,
open them with fopen() and handle any errors.

Open the public key file using fopen(). If the file cannot be opened,
display an error message and exit the program.

Read the public key from the opened public key file using ss_read_pub().

If verbose output is enabled, print the following in order:
    a. The username obtained from the public key file
    b. The public key n with its bit size and decimal value.

Encrypt the input file using ss_encrypt_file().

Close the public key file and clear any mpz_t variables used in the program.
Close the input and output files.
```

6.3 decrypt

```
Parse command-line options using getopt() and handle them accordingly.
Open the private key file using fopen(). Print a helpful error
and exit the program in the event of failure
Read the private key from the opened private key file.
If verbose output is enabled print the following, each with a trailing newline, in order:
    a. pq value
    b. d value
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.
Decrypt the file using ss_decrypt_file().
Close the private key file and clear any mpz_t variables you have used.
```

7 Pseudocode / Makefile:

7.1 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.