CSE 13S: Assignment 5 Design Doc

Johnny Li

February 26, 2023

1 Introduction

Cryptography has become an integral part of our daily lives, extending far beyond the boundaries of government, spies, and the military. Nearly all web pages use SSL for protection, and SSH connections are similarly safeguarded. The mechanism responsible for this protection is a combination of public key and symmetric key cryptography.

The earliest known public-key cryptography algorithm was RSA, named after its three inventors Ronald Rivest, Adi Shamir, and Leonard Adleman. However, the concept of public key encryption was proposed three years earlier in 1970 by J.H. Ellis, and Ralph Merkle proposed a similar idea for public key distribution. These ideas eventually led to RSA. Although RSA gave rise to numerous related algorithms like the Schmidt-Samoa algorithm, which is the primary focus of this assignment, it has not gained widespread adoption.

Public-key cryptography or asymmetric cryptography utilizes pairs of keys: public keys known to others and private keys known only by the owner. These key pairs are generated using cryptographic algorithms that are based on mathematical objects called one-way functions. The private key must be kept secure while the public key can be shared publicly. A sender can encrypt a message with the intended recipient's public key, but only the recipient can decrypt it using their private key. This approach enables a server to create a cryptographic key for suitable symmetric-key cryptography, use a client's openly shared public key to encrypt the newly generated symmetric key, and then securely transfer it to the client.

Symmetric-key algorithms employ the same cryptographic keys for both the encryption of plaintext and decryption of ciphertext. These keys represent a shared secret between two or more parties. The disadvantage of symmetric-key encryption is that both parties must have access to the secret key. This is a stark contrast to public-key encryption, which allows for secure communication without pre-sharing keys. While symmetric-key cryptography is faster, the need to pre-share keys reduces its overall security compared to public-key encryption.

2 Task

For Assignment 5, we're to implement three individual programs:

1. A key generator: keygen
2. An encryptor: encrypt
3. A decryptor: decrypt

The task of generating SS public and private key pairs will be assigned to the keygen program. On the other hand, the encrypt program will utilize the public key to encrypt files, while the decrypt program will use the corresponding private key to decrypt the now encrypted files.

We'll need to implement two libraries and a random state module that will be utilized in each of the programs. One libraries will hold functions related to the mathematics behind the Schmidt-Samoa (SS) Algorithm, while the other library itself will contain implementations of routines for SS. Additionally, we'll need to learn how to use the GNU multiple precision arithmetic library since the C programming language doesn't natively support arbitrary precision integers and SS heavily relies on large integers. Additionally, various packages and their dependencies will need to be installed with the new library.

2.1 GNU Multiple Precision Arithmetic

GNU Multiple Precision Arithmetic (GMP) is a versatile software library that offers a vast range of arithmetic operations for signed and unsigned integers, rational numbers, and floating-point numbers, all with high precision. It is designed to be efficient and easily portable across various platforms, and it is widely utilized in cryptography, computer algebra systems, and other fields that need high-precision computations.

GMP features many arithmetic operations, such as addition, subtraction, multiplication, division, exponentiation, and modular arithmetic. It also comprises functions for comparing and converting numbers, as well as functions for producing random numbers and performing bit-level operations.

One of GMP's most significant characteristics is its ability to perform arithmetic operations with large numbers, which can have thousands or millions of digits. This feature makes it highly valuable in cryptographic applications, where large numbers are frequently used for encryption and digital signatures. GMP is crucial to our implementation of SS since it allows us to handle with ease the large integer operations required by the algorithm, such as modular exponentiation, modular multiplication, and modular inversion. These operations are computationally intensive and can consume a significant amount of time and memory, especially for large prime numbers used in SS. GMP is optimized to handle such computations efficiently, allowing faster computation times and more precise results.

```
void randstate_init(uint64_t seed) {
    // initialize the global random state with Mersenne Twister algorithm
    gmp_randinit_mt(state);

    // seed the random state with the given seed
    gmp_randseed_ui(state, seed);

    // set the seed for the m random number generator
    srandom(seed);
}
```

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

```
void randstate_clear(void) {
    // clear and free memory used by the random state
    gmp_randclear(state);
}
```

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

2.2 An SS Library

```
Function ss_make_pub(p, q, n, nbits, iters):
    # Determine the number of bits for primes p and q
    p_bits = random(nbits/5, (2*nbits)/5)
    q_bits = nbits - (2 * p_bits)

    # Generate the prime numbers p and q
    p = make_prime(p_bits, iters)
    q = make_prime(q_bits, iters)

    # Calculate n as p^2 * q
    mpz_mul(n, p, p)
    mpz_mul(n, n, q)

    # Check that p - q^-1 and q - p^-1
    mpz_invert(q, q, p)
    mpz_sub(q, p, q)
    mpz_invert(p, p, q)
    mpz_sub(p, q, p)

    # Return the generated primes p and q, and n
```

```
    Return p, q, n
End Function
```

Creates parts of a new SS public key: two large primes p and q, and n computed as $p \cdot p \cdot q$. 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 $\log_2(n) \geq nbits$. 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 of Miller-Rabin iterations is specified by *iters*. You should obtain this random number using `random()` and check that $p - q^{-1}$ and $q - p^{-1}$.

```
void ss_write_pubfunction ss_write_pub(n, username, pbfile):
    // Write public SS key to pbfile
    n_hexstring = mpz_get_str(NULL, 16, n)  // convert n to hexstring
    fprintf(pbfile, "%s\n%s\n", n_hexstring, username)
// write n and username to pbfile
end function
```

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. See the GMP functions for formatted output for help with writing hexstrings.

```
void ss_read_pub(mpz_t n, char username[], FILE *pbfile) {
    char buf[256];

    // read n from pbfile as a hexstring and set it to n
    fgets(buf, sizeof(buf), pbfile);
    mpz_set_str(n, buf, 16);

    // read username from pbfile and remove the trailing newline
    fgets(username, 255, pbfile);
    username[strcspn(username, "\n")] = '\0';
}
```

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.

```
void ss_make_priv(mpz_t d, mpz_t pq, mpz_t p, mpz_t q)
    Compute lambda = lcm(p-1, q-1)
    Compute d = n^(-1) mod lambda using the GMP function mpz_invert()
```

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 $\lambda(pq) = \text{lcm}(p-1, q-1)$.

```
void ss_write_priv(mpz_t pq, mpz_t d, FILE *pvfile):
```

```
write hexstring(pq) to pvfile
write newline to pvfile
write hexstring(d) to pvfile
write newline to pvfile
```

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 will be written as hexstrings.

```
function ss_read_priv(pq, d, pvfile):
    buf = array of size 256 bytes

    fgets(buf, sizeof(buf), pvfile)
    mpz_set_str(pq, buf, 16)

    fgets(buf, sizeof(buf), pvfile)
    mpz_set_str(d, buf, 16)
```

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 will have been written as hexstrings.

```
void ss_encrypt(mpz_t c, const mpz_t m, const mpz_t n) {
    pow_mod(c, m, n, n);
}
```

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

```
void ss_encrypt_file(FILE *infile, FILE *outfile, mpz_t n) {
    function ss_encrypt_file(infile, outfile, n):
k = ceil(log2(sqrt(n)) - 1) / 8
block = allocate(k bytes)
block[0] = 0xFF

while not end of file(infile):
    j = read(infile, block + 1, k - 1)
    m = mpz_import(0, j + 1, 1, 1, 0, 0, block)
    if m == 0 or m == 1:
        continue
    c = ss_encrypt(m, n)
    write(outfile, c)
}
```

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

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 $k = \lfloor b(\log_2(p/n) - 1)/8 \rfloor$.
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.

```
void ss_decrypt(mpz_t m, const mpz_t c, const mpz_t d, const mpz_t pq) {
    pow_mod(m, c, d, pq);
}
```

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 $D(c) = m = c^d \pmod{pq}$.

```
void ss_decrypt_file(FILE *infile, FILE *outfile, mpz_t pq, mpz_t d) {
    function ss_decrypt_file(infile, outfile, p, q):
    k = ceil(b * log2(p*q - 1) / 8)
    block = allocate(k bytes)

    for each line in infile:
        c = read_hexstring(line)
        m = ss_decrypt(c, p, q)
        j = mpz_export(block + 1, 0, 1, 1, 0, 0, m)
```

```
        write(outfile, block[1:j])
}
```

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 $k = b(\log_2(pq) - 1)/8$ 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 encryption process. In this context, we don't know the value of n, but we can overestimate the number of bits in pn using pq.

$$\log_2(pn) = \log_2(qp^2 \times q) \quad = \log_2(p \times \frac{p}{q} \times q^2) \ < \log_2(pq) \quad (1)$$

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.

3 Command-Line Options

Each of our three programs will support a variety of command-line options. These options are non-exclusive within the each individual program but are exclusive to their respective program. As such, an option used for one program will not directly impact another.

keygen.c

- -b : specifies the minimum bits needed for the public modulus n.
- -i : specifies the number of Miller-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.

encrypt.c

- -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.

decrypt.c

- -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.

4 Sources

- The C Programming Language
- Psuedo-code provided by TA John Yu
- Various Stack Overflow posts to aid with debugging