HW3 - Implementation of RSA
CNS Course Sapienza

Sultan Umarbaev, Matricola: 1954544

20/11/2020

1 Goal

The goal of this homework is to implement RSA. It is an asymmetric cryptographic algorithm which is widely used for secure data transmission. Asymmetric cryptography is a cryptographic system that uses pairs of keys: *public keys*(can be given to anyone) and *private keys*(must be kept private). The generation of such keys depends on algorithms based on mathematical problems to produce one-way functions, which means that by encrypting a message with one key(e.g. public key), it can not be decrypted using the same key, the decryption is performed with the other key(e.g. private key).

2 Implementation

The programming language used for the implementation is *C++*. Main library for working with large numbers is *The GNU Multiple Precision Arithmetic Library* (GMP) wrapper for *C++*. It is a libraray for **arbitrary-precision arithmetic** with basic interface for *C* programming language and wrappers for other languages like *C++*, *C#*, *Python*, *R*, etc. The main applications of GMP involve fields such as cryptography, Internet Security and computer algebra systems(CAS) [1].
The RSA algorithm consists of four steps [2]:

- Key generation
- Key distribution
- Encryption
- Decryption

Key distribution step out of the scope of the homework.

2.1 Key generation

The initial step in key generation process is to select 2 distinct large prime numbers p and q. For security purposes, p and q should be chosen at random and kept secret.
After choosing prime numbers, they are multiplied to give a very large number with 2 prime factors: $p * q = n$. It is used as the modulus for public and private keys. Its length in bits is the *key length*. It is a part of public key.
Next step is the calculation of the **totient** of n, which is the number of positive integers smaller than n that are coprime to n. For any prime number, $\phi(p) = p - 1$, hence for the modulus n with

2 prime factors $\phi(n) = (p-1)(q-1)$. It should be kept secret.
Subsequently, e and d are generated. Where e is an integer that satisfies 2 conditions: $1 < e < \phi(n)$ and $gcd(e, \phi(n)) = 1$; i.e. e and $\phi(n)$ are coprime. For more efficient encryption, e is short bit-length and the most commonly chosen value for e is $2^{16} + 1 = 65537$. An integer d should satisfy the congruence relation $de \equiv 1 \pmod{\phi(n)}$, i.e. d is the modular multiplicative inverse of e modulo $\phi(n)$. It must be kept secret. Public(encryption) key (e, n) and private(decryption) key (d, n) are generated.
In the current implementation, prime numbers p and q are generated with the help of GMP library *random number generation* and *prime number test* functions. According to the library manual [3], random number generation function used in the current implementation is based on **Mersenne Twister algorithm** and considered to be fast and have good randomness properties. The primality test consists of 2 testing algorithms **Baillie-PSW probable prime test** and **Miller-Rabin probabilistic primality tests** [4].
The modular inverse of e is calculated using the **extended Euclidean algorithm** [5], which is an extension to the Euclidean algorithm. In addition to the greatest common divisor (gcd) of integers a and b, it also computes the coefficients of **Bézout's identity**, which are integers **x** and **y**:

$$ax + by = \gcd(a, b) \tag{1}$$

In the case of RSA, the equation looks like this:

$$ed + \phi(n)y = \gcd(e, \phi(n)) = 1 \tag{2}$$

Then, if $\mod \phi(n)$ is taken of both sides, the $\phi(n)y$ disappears, and the equation becomes:

$$ed \equiv 1 \mod \phi(n) \tag{3}$$

Hence, coefficient x of **Bézout's identity** is the multiplicative inverse of the e (d generation function in Figure 8 in Appendix A).

2.2 Encryption and Decryption

Public key (e, n) is released by one person and anyone with it can encrypt message. Message M is turned into an integer m such that $0 \leq m < n$ and ciphertext c is computed corresponding to: $m^e \equiv c \pmod{n}$. Ciphertext c is then sent to the one who generated public key.
Decryption is performed using private key (d, n) given ciphertext c: $c^d \equiv (m^e)^d \equiv m \pmod{n}$, hence recovering the original message M.
Since m and n are large numbers, the performance of exponentiation operation is slow. It can be resolved using modular exponentiation. **Right-to-left binary method** [6] is a combination of the **Binary Exponentiation algorithm** and modulo arithmetic. It is memory-efficient and reduces the number of operations to perform modular exponentiation. The idea of binary exponentiation is to use the binary representation of the exponent, Equation 4.

$$e = \sum_{i=0}^{n-1} a_i 2^i \tag{4}$$

Where a_i can be 0 or 1. As a result, b^e can be rewritten to Equation 5.

$$b^e = b^{\left(\sum_{i=0}^{n-1} a_i 2^i\right)} = \prod_{i=0}^{n-1} b^{a_i 2^i} \tag{5}$$

For example, $b^{13} = b^{1101_2} = b^{1*2^3} b^{1*2^2} b^{0*2^2} b^{1*2^1} = b^8 b^4 b^0 b^1$. Since the modulo operator doesn't interfere with multiplications: $ab \equiv (a \mod m)(b \mod m) \mod m$, therefore:

$$c \equiv \prod_{i=0}^{n-1} b^{a_i 2^i} \pmod m \tag{6}$$

Algorithm 1: Right-to-left binary algorithm pseudocode [6]

```
result ← 1
base ← base mod modulus
while exponent > 0 do
  if exponent mod 2 == 1 then
    result ← (result * base) mod modulus
  end if
  exponent ← exponent >> 1
  base ← (base * base) mod modulus
end while
return result
```

The running time of this algorithm is O(log_2 *exponent*), while time of naive approach is O(*exponent*). For instance, consider *exponent* = 4294967296, this algorithm would require 32 ($2^{32} = 4294967296$) steps to perform an exponentiation instead of 4294967296 steps [6].
The exponentiation operation for encryption and decryption processes of the current implementation of RSA utilizes the described algorithm (see Figure 9 in Appendix A).

2.3 Limitations and weaknesses

The main weakness of the current implementation is the absence of **Padding Scheme**. The absence of padding makes RSA vulnerable to the number of attacks [7, 8] considering different cases:

- size of the message and exponent are too small
- same message is encrypted using same exponent, but with different modulus
- same message is encrypted using coprime exponent, but with same modulus

In addition, RSA without padding is not **semantically secure** which means that information about the message can be feasibly extracted from the ciphertext. This allows an attacker to perform **chosen-ciphertext attack** [7].
Practical and real-world RSA implementations are built with padding schemes. Randomized padding is added into the message before encrypting it resulting in less predictable message structure. It must be carefully designed to avoid and prevent mentioned attacks [7].

3 Results

The current implementation of RSA is capable of performing 3 out 4 steps of RSA algorithm: key generation, encryption and decryption. *Key generation* operation can generate keys of size up to 4096 bit-length. However, it does not provide *key management*, as a result keys are stored in raw format as a plaintext, Figure 1. Public key consists of key size, e, n, Key size is embedded in public key to avoid situation where data size $> n$. Private key involves d and n.

Figure 1: Public and private keys of RSA implementation.

The encrypted message is not encoded and stored as a big number in the file, see Figure 2. The size of the encrypted message depends on the size of the key, n, thus the message size has to be $n/8$ bytes size, considering the absence of padding. However it was detected and noted that the *encryption* and *decryption* operations of the current implementation can only work with the sizes slightly lower than expected ones. For instance:

- with the key size 4096 bits, the message size has to be less than 512 bytes, and current implementation can not properly encrypt and decrypt data more than 410 bytes.
- with the key size 2048 bits, the requirement is $m < 256$ bytes, however the actual threshold

is approximately 200.

This issue hasn't been resolved yet and is on the list of tasks, alongside with padding scheme and encrypted message encoding, for the future improvements.

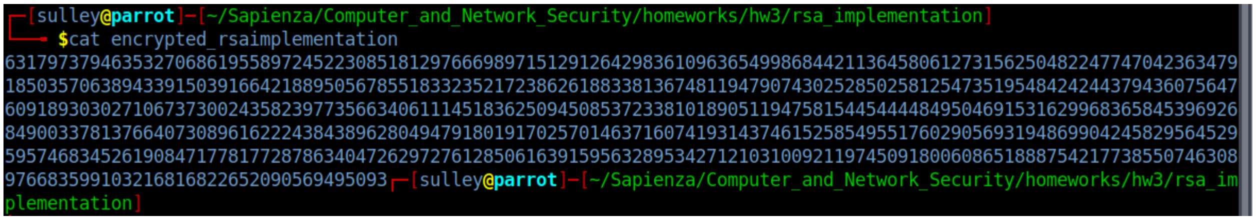

Figure 2: Encrypted data of RSA implementation.

4 Experimental comparison

Considering the comparison, it was experimented with OpenSSL implementations of RSA and AES. The experiment consists of encryption and decryption performance speed, see Figures 3, 4, 5. The file size for experiment is 199 bytes. The key size is chosen to be 2048 bits long. Based on the results, it can be concluded that the speed of encryption process is approximately identical, if not. However, the decryption speed ranks them in the following order: OpenSSL AES-256 with 0.002 seconds, OpenSSL RSA with 0.003 seconds, RSA implementation with 0.006 seconds.

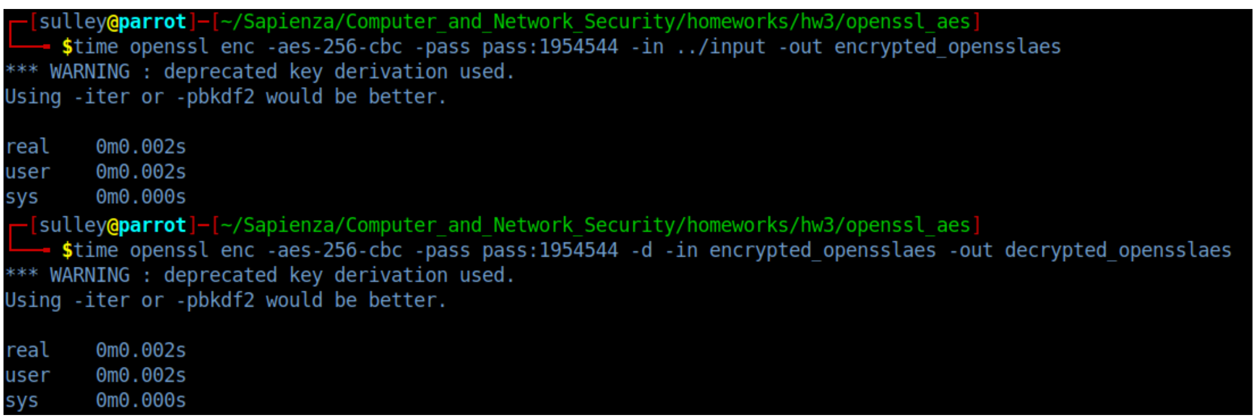

Figure 3: OpenSSL AES-256 encryption and decryption speed results.

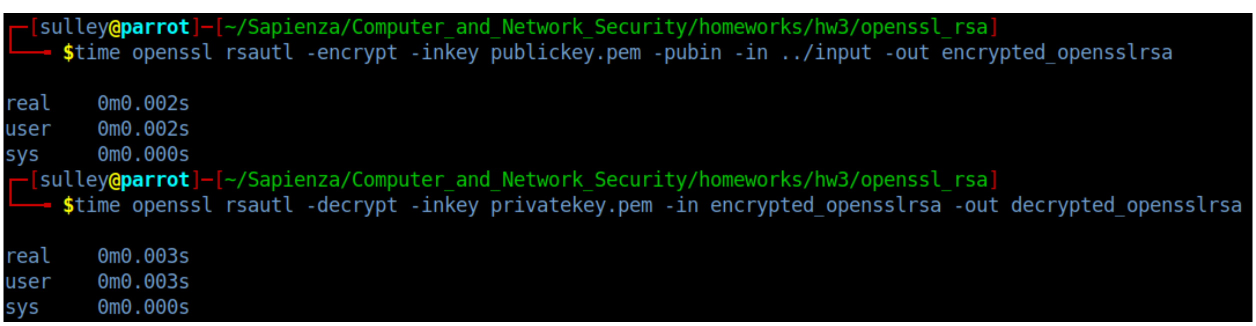

Figure 4: OpenSSL RSA encryption and decryption speed results.

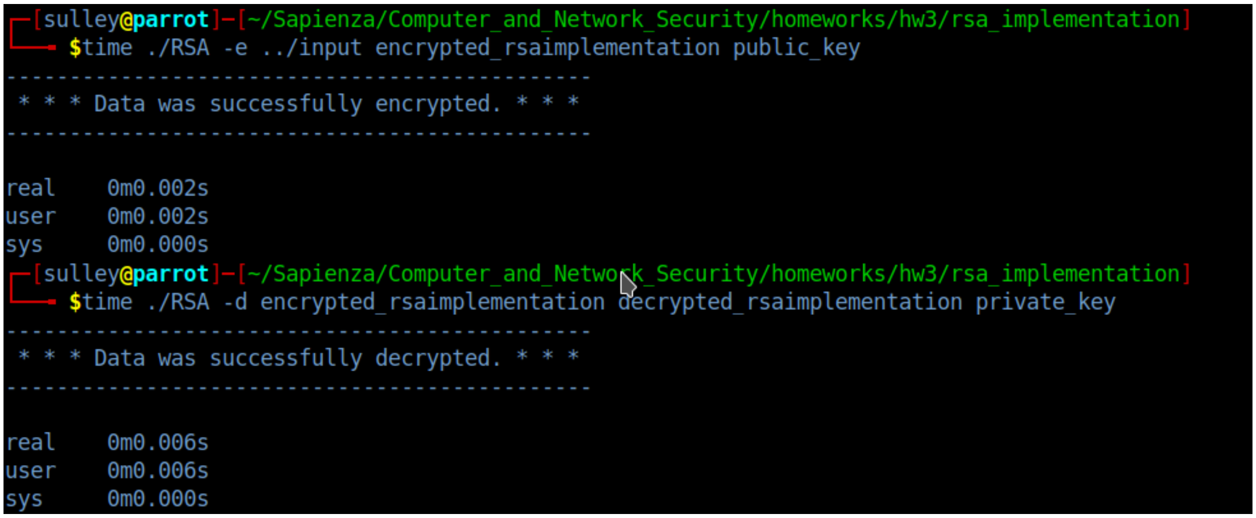

Figure 5: RSA implementation encryption and decryption speed results.

In addition, the performance speed of *key generation* step of RSA implementations were compared, see Figures 6, 7. The speed results of the RSA implementation seem to be better, however it is all due to the absence of *key management* process.

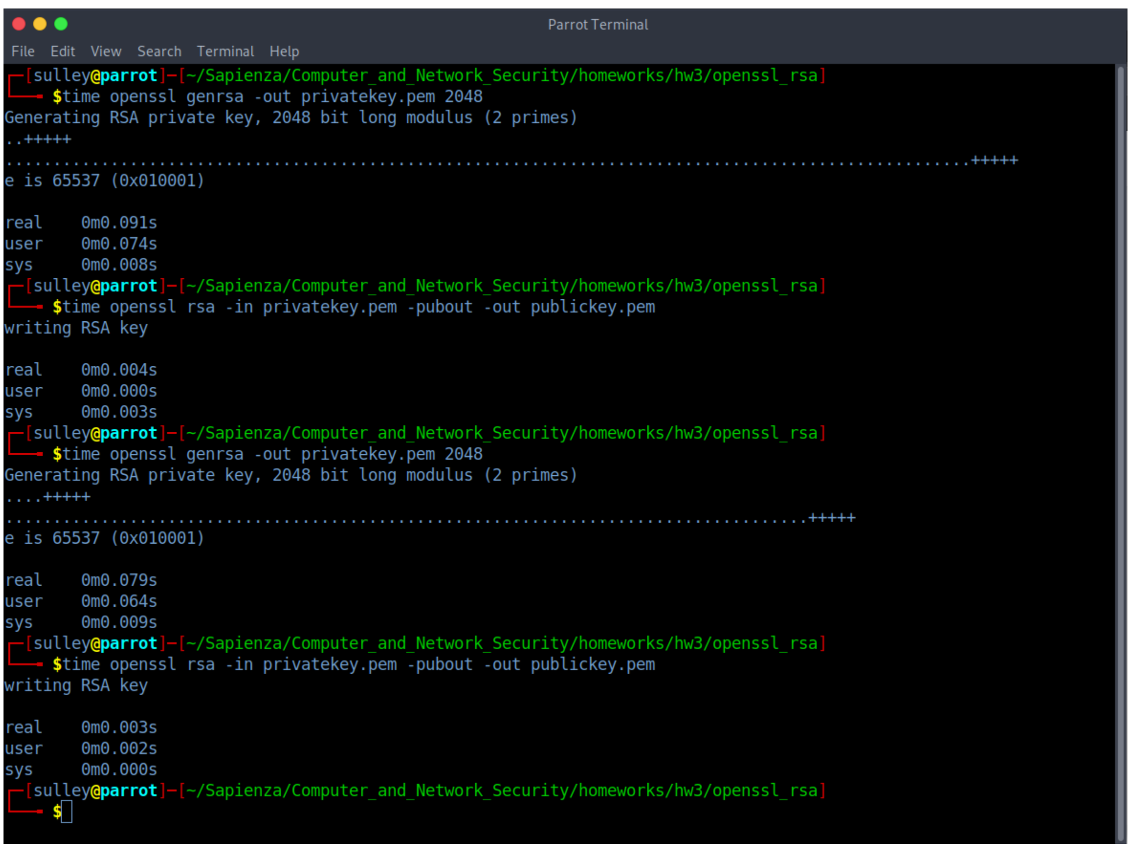

Figure 6: OpenSSL RSA key generation speed results.

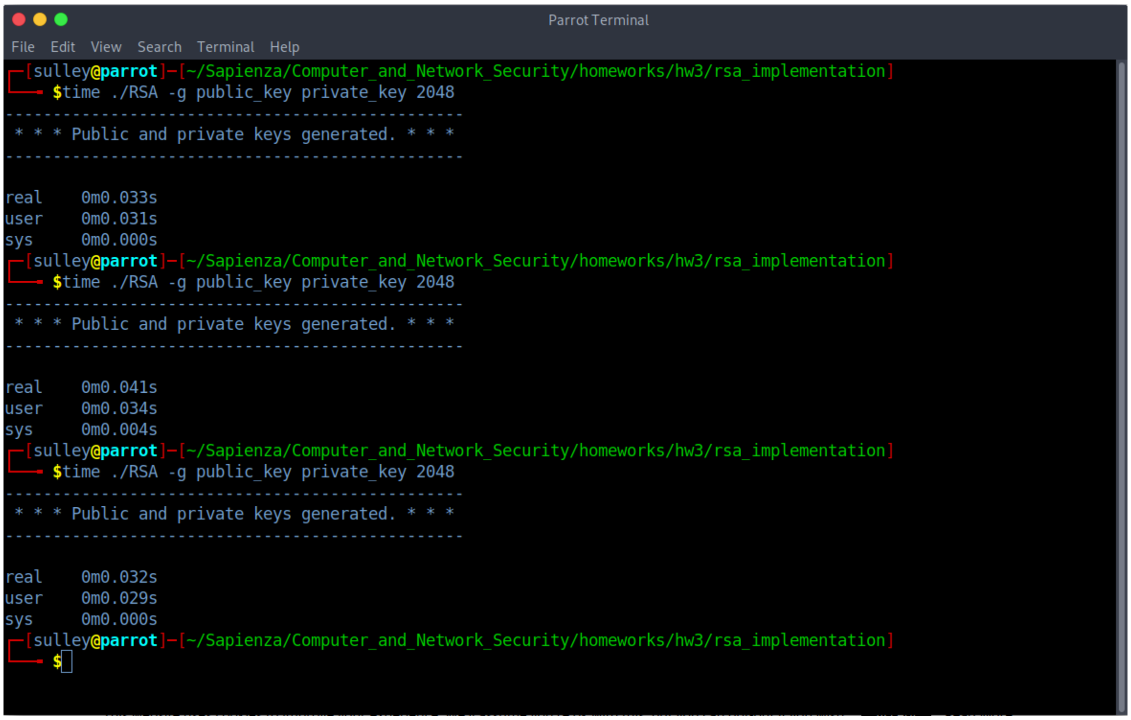

Figure 7: RSA implementation key generation speed results.

5 Conclusion

The goal of implementing RSA algorithm was achieved, however it requires improvements and is not recommended for practical use. RSA is a relatively slow algorithm and is not practical to use with large data comparing to AES. As a result, it is not commonly used for user data encryption and decryption. Practically, RSA is used to transmit shared keys for symmetric key cryptography, which are then used to encrypt and/or decrypt data.

References

[1] GNU Multiple Precision Arithmetic Library, URL: `https://en.wikipedia.org/wiki/GNU_Multiple_Precision_Arithmetic_Library`

[2] RSA (cryptosystem) operation, URL: `https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Operation`

[3] Random State Initialization, URL: `https://gmplib.org/manual/Random-State-Initialization#Random-State-Initialization`

[4] Number Theoretic Functions, URL: `https://gmplib.org/manual/Number-Theoretic-Functions`

[5] Modular multiplicative inverse, URL: `https://en.wikipedia.org/wiki/Modular_multiplicative_inverse#Extended_Euclidean_algorithm`

[6] Right-to-left binary method, URL: `https://en.wikipedia.org/wiki/Modular_exponentiation#Right-to-left_binary_method`

[7] RSA (cryptosystem) padding wiki, URL: https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Padding

[8] Asymmetric Ciphers II, URL: https://piazza.com/class_profile/get_resource/kf2apkmqhrq4qw/kgrqkx1l5wt3ux

Appendix A Functions from source code

```cpp

/*
 * Recursive Extended Euclidean Algorithm for generating decryption key: d
 */
void eea(const mpz_class &a, const mpz_class &mod, mpz_class &x, mpz_class &y)
{
    if (a == 0)
    {
        x = 0, y = 1;
        return;
    }
    mpz_class x1, y1;
    eea(mod%a, a, x1, y1);
    x = y1 - x1*(mod/a);
    y = x1;
}

/*
 * Find the Modular Inverse of e with respect to phi
 * using Extended Euclidean Algorithm(eea)
 */
mpz_class generate_d(const mpz_class &e, const mpz_class &phi)
{
    mpz_class s, t;
    eea(e, phi, t, s);
    return (t%phi + phi) % phi;
}

```

Figure 8: Multiplicative inverse calculation.

```
/*
 * Modular exponentiation operation(right-to-left binary exponentiation method)
 */
mpz_class modular_exponentiation(mpz_class base, mpz_class exponent, const mpz_class &modulo)
{
    mpz_class result = 1;
    base = base % modulo;

    while (exponent > 0)
    {
        if (exponent % 2 == 1)
        {
            result = (result*base) % modulo;
        }
        base = (base*base) % modulo;
        exponent = exponent / 2;
    }

    return result;
}
```

Figure 9: Modular exponentiation calculation.