

# 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 library 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  $y$ :

$$ax + by = \gcd(a, b) \quad (1)$$

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

$$ed + \phi(n)y = \gcd(e, \phi(n)) = 1 \quad (2)$$

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

$$ed \equiv 1 \pmod{\phi(n)} \quad (3)$$

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.

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

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

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

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

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

---

**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  $O(\log_2 \text{exponent})$ , while time of naive approach is  $O(\text{exponent})$ . For instance, consider  $\text{exponent} = 4294967296$ , 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 of 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  $n$ .

```

└─[sulley@parrot]-[~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_im
plementation]
└─$ cat public_key
2048-65537|193622085381478196350718651304641959171948732968916106736492690145990
80269524683221487817360164746866669162394967813821140966916931491665368398357341
80277934062621241144270482863348556608955337839288450489330011534905173618612656
90131678539927604028520772776858772002909600077352260280560863802691956279104847
93750117599362540084071219744865949578749268847251322032445022373781330157947987
98846758887382062256423368752848685643544126787698111800630799466171062646003972
67178204750220446538725417700901499335141646104497189624457608645674233103223213
1358979145159938084905370874094427337571891717415241483637462482339└─[sulley@par
rot]-[~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_implementation]
└─$

└─[sulley@parrot]-[~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_im
plementation]
└─$ cat private_key
12890314184780010047726025718407667913745357840863601710514537686741420437914940
47082922561669511986419339036658438263622383581110880743475729112912065241108703
27032626402895216153868613768299350391373397164895758547763294251627070603403670
83603600055525637467575691989179793055764634177870590010608884162569596390457709
20635465986080995357309657842045670516368540427642719329204342953265597823077040
86630016245216362671769869712446901977606071400282795754070694832167311844448529
83185098240998264466431744329780193327286194549163839418842314273520213194182742
65766562838728361294941881554030992844768568325556978801|19362208538147819635071
86513046419591719487329689161067364926901459908026952468322148781736016474686666
91623949678138211409669169314916653683983573418027793406262124114427048286334855
66089553378392884504893300115349051736186126569013167853992760402852077277685877
20029096000773522602805608638026919562791048479375011759936254008407121974486594
95787492688472513220324450223737813301579479879884675888738206225642336875284868
56435441267876981118006307994661710626460039726717820475022044653872541770090149
93351416461044971896244576086456742331032232131358979145159938084905370874094427
337571891717415241483637462482339└─[sulley@parrot]-[~/Sapienza/Computer_and_Netw
ork_Security/homeworks/hw3/rsa_implementation]
└─$

```

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.

```
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_implementation
$cat encrypted_rsaimplementation
63179737946353270686195589724522308518129766698971512912642983610963654998684421136458061273156250482247747042363479
18503570638943391503916642188950567855183323521723862618833813674811947907430252850258125473519548424244379436075647
60918930302710673730024358239773566340611145183625094508537233810189051194758154454444849504691531629968365845396926
84900337813766407308961622243843896280494791801917025701463716074193143746152585495517602905693194869904245829564529
59574683452619084717781772878634047262972761285061639159563289534271210310092119745091800608651888754217738550746308
976683599103216816822652090569495093 [sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_implementation
```

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.

```
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/openssl_aes
$time openssl enc -aes-256-cbc -pass pass:1954544 -in ../input -out encrypted_opensslaes
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

real    0m0.002s
user    0m0.002s
sys     0m0.000s
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/openssl_aes
$time openssl enc -aes-256-cbc -pass pass:1954544 -d -in encrypted_opensslaes -out decrypted_opensslaes
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

real    0m0.002s
user    0m0.002s
sys     0m0.000s
```

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

```
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/openssl_rsa
$time openssl rsautl -encrypt -inkey publickey.pem -pubin -in ../input -out encrypted_opensslrsa
real    0m0.002s
user    0m0.002s
sys     0m0.000s
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/openssl_rsa
$time openssl rsautl -decrypt -inkey privatekey.pem -in encrypted_opensslrsa -out decrypted_opensslrsa
real    0m0.003s
user    0m0.003s
sys     0m0.000s
```

Figure 4: OpenSSL RSA encryption and decryption speed results.

```

[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_implementation
$time ./RSA -e ../input encrypted_rsaimplementation public_key
-----
* * * Data was successfully encrypted. * * *
-----

real    0m0.002s
user    0m0.002s
sys     0m0.000s
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/hw3/rsa_implementation
$time ./RSA -d encrypted_rsaimplementation decrypted_rsaimplementation private_key
-----
* * * Data was successfully decrypted. * * *
-----

real    0m0.006s
user    0m0.006s
sys     0m0.000s

```

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.

```

Parrot Terminal
File Edit View Search Terminal Help
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/openssl_rsa
$time openssl genrsa -out privatekey.pem 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
e is 65537 (0x010001)

real    0m0.091s
user    0m0.074s
sys     0m0.008s
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/openssl_rsa
$time openssl rsa -in privatekey.pem -pubout -out publickey.pem
writing RSA key

real    0m0.004s
user    0m0.000s
sys     0m0.003s
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/openssl_rsa
$time openssl genrsa -out privatekey.pem 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
e is 65537 (0x010001)

real    0m0.079s
user    0m0.064s
sys     0m0.009s
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/openssl_rsa
$time openssl rsa -in privatekey.pem -pubout -out publickey.pem
writing RSA key

real    0m0.003s
user    0m0.002s
sys     0m0.000s
[sulley@parrot]~/Sapienza/Computer_and_Network_Security/homeworks/hw3/openssl_rsa
$

```

Figure 6: OpenSSL RSA key generation speed results.

```
Parrot Terminal
File Edit View Search Terminal Help
[sulley@parrot] ~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_implementation
$ time ./RSA -g public_key private_key 2048
-----
* * * Public and private keys generated. * * *
-----
real    0m0.033s
user    0m0.031s
sys      0m0.000s
[sulley@parrot] ~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_implementation
$ time ./RSA -g public_key private_key 2048
-----
* * * Public and private keys generated. * * *
-----
real    0m0.041s
user    0m0.034s
sys      0m0.004s
[sulley@parrot] ~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_implementation
$ time ./RSA -g public_key private_key 2048
-----
* * * Public and private keys generated. * * *
-----
real    0m0.032s
user    0m0.029s
sys      0m0.000s
[sulley@parrot] ~/Sapienza/Computer_and_Network_Security/homeworks/hw3/rsa_implementation
$
```

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](https://en.wikipedia.org/wiki/GNU_Multiple_Precision_Arithmetic_Library)
- [2] RSA (cryptosystem) operation, URL: [https://en.wikipedia.org/wiki/RSA\\_\(cryptosystem\)#Operation](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](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](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](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Padding)
- [8] Asymmetric Ciphers II, URL: [https://piazza.com/class\\_profile/get\\_resource/kf2apkmqhrq4qw/kgrqkx1l5wt3ux](https://piazza.com/class_profile/get_resource/kf2apkmqhrq4qw/kgrqkx1l5wt3ux)

## Appendix A Functions from source code

```
22
23  /*
24   * Recursive Extended Euclidean Algorithm for generating decryption key: d
25   */
26 void eea(const mpz_class &a, const mpz_class &mod, mpz_class &x, mpz_class &y)
27 {
28     if (a == 0)
29     {
30         x = 0, y = 1;
31         return;
32     }
33     mpz_class x1, y1;
34     eea(mod%a, a, x1, y1);
35     x = y1 - x1*(mod/a);
36     y = x1;
37 }
38
39 /*
40 * Find the Modular Inverse of e with respect to phi
41 * using Extended Euclidean Algorithm(eea)
42 */
43 mpz_class generate_d(const mpz_class &e, const mpz_class &phi)
44 {
45     mpz_class s, t;
46     eea(e, phi, t, s);
47     return (t%phi + phi) % phi;
48 }
49 |
```

Figure 8: Multiplicative inverse calculation.



```

92  /*
93  * Modular exponentiation operation(right-to-left binary exponentiation method)
94  */
95  mpz_class modular_exponentiation(mpz_class base, mpz_class exponent, const mpz_class &modulo)
96  {
97      mpz_class result = 1;
98      base = base % modulo;
99
100     while (exponent > 0)
101     {
102         if (exponent % 2 == 1)
103         {
104             result = (result*base) % modulo;
105         }
106         base = (base*base) % modulo;
107         exponent = exponent / 2;
108     }
109
110     return result;
111 }

```

Figure 9: Modular exponentiation calculation.