Playing with PK schemes

Homework3 - CNS Sapienza

Martina Evangelisti 1796480

20/11/2020

1 Introduction

RSA, is an asymmetric cipher algorithm, based on two keys: one public and one private. RSA operating mechanism is based on the problem of prime factorization: it's difficult to find prime factors of a very large integer number. RSA scheme could be conceptually divided into two different tasks: key generation, encryption/decryption.

2 Implementation of Key generation

As i said before, RSA scheme is based on the usage of two different keys. The public key is so called because it can be known to everyone, insted the private key has to remain secret. The two keys are generated together in such a way that a message encrypted with the public key can only be decrypted with the private key and vice versa. The key generation algorithm, realized in the code by the function *key_gen()*, is made of the following steps:

- Generation of p and q: prime numbers of 1024 bits
- Computation of $n = p \cdot q$
- Computation of $\phi(n) = (p-1) \cdot (q-1)$
- Generation of the public key e, such that:
  - $e \in \{1, 2, ...\phi(n) - 1\}$
  - $gcd(e, \phi(n)) = 1$
- Computation of the private key d, as the multiplicative inverse of e:
  - $d \cdot e \equiv 1 \bmod \phi(n)$

The prime numbers p and q are generated using the function *randprime* made available by *simpy*, a Python library. The function *randprime* takes as input the two numbers start and end, that limit the range from which the random prime numbers are taken.
I chose 2^{1023} as left limit and $2^{1024} - 1$ as right limit of the range, in order to have numbers of 1024 bits.
$\phi(n)$ can be computed as the multiplication of p-1 and q-1 thanks to Fermat's Little Theorem.
The public key e is selected as a pseudo-random number between 3 and $\phi(n)$ but assuring that it respects the property $gcd(e, \phi(n)) = 1$ which means that e has to be coprime with $\phi(n)$.
The pseudo-random number is obtained from the function *randrange* of *random* Python library.
The private key d is e's multiplicative inverse, computed implementing the Extended Euclidean algorithm.

Extended Euclidean Algorithm (EEA)
Input: positive integers r_0 and r_1 with $r_0 > r_1$
Output: $\gcd(r_0, r_1)$, as well as s and t such that $\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$.
Initialization:
$s_0 = 1 \quad t_0 = 0$
$s_1 = 0 \quad t_1 = 1$
$i = 1$
Algorithm:

1 DO
1.1 $i = i + 1$
1.2 $r_i = r_{i-2} \bmod r_{i-1}$
1.3 $q_{i-1} = (r_{i-2} - r_i)/r_{i-1}$
1.4 $s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$
1.5 $t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$
WHILE $r_i \neq 0$
2 RETURN
$\gcd(r_0, r_1) = r_{i-1}$
$s = s_{i-1}$
$t = t_{i-1}$

Figure 1: Pseudo-code of EEA
[1]

3 Encryption and Decryption

The encryption function, *rsaep* in the code, takes a message representative x, the public key e and n and returns the ciphertext representative y computing:

$$y \equiv e_{k_{pub}}(x) \equiv x^e \ mod \ n \tag{1}$$

where x,y $\in Z_n$.

The decryption function, *rsadp* in the code, takes a ciphertext representative y, the private key d and n and returns the plaintext representative x computing:

$$x \equiv d_{k_{pr}}(y) \equiv y^d \ mod \ n \tag{2}$$

where x,y $\in Z_n$.

Since the exponentiation of large numbers is a critical operation i implemented the fast exponentiation method based on square and multiply, *SandM*.

Square-and-Multiply for Modular Exponentiation
Input:
base element x
exponent $H = \sum_{i=0}^{t} h_i 2^i$ with $h_i \in 0, 1$ and $h_t = 1$
and modulus n
Output: $x^H \bmod n$
Initialization: $r = x$
Algorithm:

```
1    FOR i = t − 1 DOWNTO 0
1.1      r = r^2 mod n
         IF h_i = 1
1.2          r = r · x mod n
2    RETURN (r)
```

Figure 2: Pseudo-code of SandM
[1]

3.1 PKCS1-v1_5

I implemented an encryption/decryption scheme based on padding PKCS1-v1_5 Standard. This standard can handle messages of length up to k - 11 octets, where k is the length of the RSA modulus expressed in octets.
Since i used two prime numbers of 1024 bit, n has a length of 2048 that corresponds to 256 octets, so k is equal to 256 and the messages should be smaller than 245 bytes.
In the encryption, *pkcs_v1_5_en* function in the code, the message is padded as it follows:

$$EM = 0x00 \;||\; 0x02 \;||\; PS \;||\; 0x00 \;||\; M \tag{3}$$

where PS is a pseudo-randomly generated string with length equal to k - *messageLen - 3* with non-zero bytes.

To apply the *rsaep* and *rsadp* function, which are respectively the RSA encryption and decryption function, we need to derive an integer which is representative of the message and we also need a way of coming back to the input string.
To do this transformations i realized the *os2ip* function, which stands for *octet string to integer primitive*, and the *i2osp* function that means *integer to octet string primitive* as described in the RFC: 8017 [2].

I2OSP (x, l)

Input: x nonnegative integer to be converted

l intended length of the resulting octet string

Output: X corresponding octet string of length l

Errors: "integer too large"

Steps:

1. If $x \geq 256^l$, output "integer too large" and stop.
2. Write the integer x in its unique l-digit representation base 256:

$$x = x_{l-1}\, 256^{l-1} + x_{l-2}\, 256^{l-2} + \ldots + x_1\, 256 + x_0$$

where $0 \leq x_i < 256$ (note that one or more leading digits will be zero if $x < 256^{l-1}$).

3. Let the octet X_i have the value x_{l-i} for $1 \leq i \leq l$. Output the octet string

$$X = X_1\, X_2 \ldots X_l.$$

Figure 3: Pseudo-code of is2osp function
[3]

OS2IP (*X*)

Input: *X* octet string to be converted

Output: *x* corresponding nonnegative integer

Steps:

1. Let $X_1 X_2 \ldots X_l$ be the octets of X from first to last, and let x_{l-i} have value X_i for $1 \le i \le l$.
2. Let $x = x_{l-1}\, 256^{l-1} + x_{l-2}\, 256^{l-2} + \ldots + x_1\, 256 + x_0$.
3. Output x.

Figure 4: Pseudo-code of os2ip function
[3]

In the *pkcs_v1_5_en* function the *os2ip* is applied to the EM message and the result is used as the message representative that is the input of the RSA encrypt function.
The ciphertext representative, obtained from the encryption, is transformed into the real ciphertext with the *i2osp* function.
In a symmetric way, in *pkcs_v1_5_de*, the ciphertext representative is computed with the *os2ip*, the RSA decryption algorithm is applied to it and the EM is reconstructed with the *i2osp*.
The EM is tested to ensure that it has the form of the *equation 3* shown previously, and the original message, M, is extracted from it.

3.2 Basic attacks against RSA

Most of the attacks exploit weaknesses in the parameters used in RSA implementation and not in the algorithm itself.
One typical attack is based on the modulus factorization: knowing the value of the modulus n, the ciphertext y and the public key e, the value of the private key d is computed testing the following property:

$$d \cdot e \equiv 1 \; mod \; \phi(n) \tag{4}$$

To do that, the attacker has to factorize n decomposing it into the primes p and q in order to compute $\phi(n)$. If this value is found, the attacker can compute the private key d and decrypt the ciphertext y. The only way of preventing this mathematical attack is to have large modulus of 1024 or more bit.

In order to let RSA be a little more secure the value of the public key e should not be too small, at least equal or larger than 3. In fact if the message representative x and e are small, the cipher y is: $y = x^e < n$ and the original text x is easily obtained from: $\sqrt[e]{y}$.

In general RSA could be attacked even with brute force: trying all possible private keys. To protect against this type of attack, the key space has to be large, but there's always the need to find a trade off between the security and the computational cost of the algorithm. In fact using keys with large sizes will slow the encryption and the decryption process.

If the same modulus n is used for different users the system is insecure. One user can use his own keys to factor the modulus and compute the two prime factors p and q, then he can recover everyone's private key computing the multiplicative inverse of their public key e.

There is another category of attacks which doesn't focus on the inner structure of the RSA function but on its implementation. RSA could be attacked even if it is implemented with a Padding scheme. One example is the Timing attack which derives the private key d by measuring the time that the decryption takes. One simple way to defend against this type of attack is making the time spent in the decryption constant by adding a delay.

A lot of other attacks has been found and analyzed but so far it doesn't exist a devastating attack. The previously described attacks, and all the ones that has been found, are useful to describe what has to be avoided when implementing RSA. Indeed there are some proper implementation of RSA that are considered secure and should be trusted.

4 Experimental analysis

To do an experimental analysis of the implementation of RSA that i realized i used a test file of: 245 Bytes because, using a modulus of 2048 bit size this is the maximum size that RSA can encrypt relying on:

$$max\_size = k - 11 \tag{5}$$

where k is the length of the RSA modulus expressed in octets.

First of all i profiled the performances of my implementation by timing the encryption and the decryption ten times.

TEST	encryption running time	decryption running time
1	0.0480420589447 sec	0.0486729145050 sec
2	0.0463571548462 sec	0.0452680587769 sec
3	0.0521159172058 sec	0.0528359413147 sec
4	0.0524129867554 sec	0.0532560348511 sec
5	0.0469748973846 sec	0.0467550754547 sec
6	0.0479140281677 sec	0.0477821826935 sec
7	0.0476710796356 sec	0.0485742092133 sec
8	0.0466740131378 sec	0.0506319999695 sec
9	0.0466210842133 sec	0.0462920665741 sec
10	0.0489311218262 sec	0.0481038093567 sec

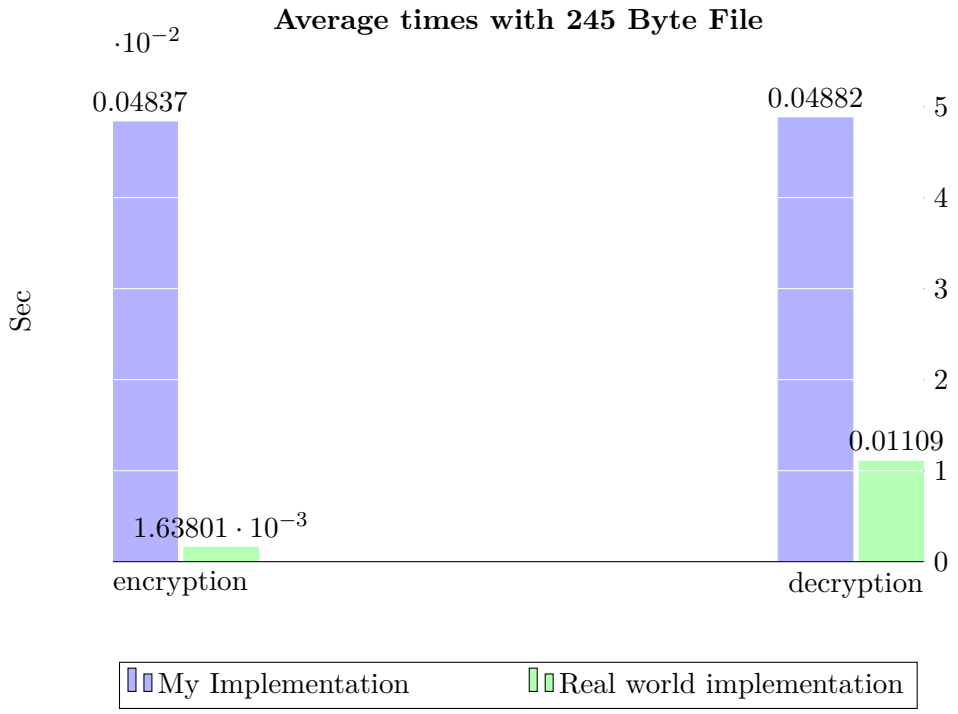
The slight difference between the running times of the different tests is due to the fact that the exponent e, which represents the public key, is a pseudo-random number and the computation is slower when e is greater.

4.1 Comparison with the performances of RSA real-world implementation

I tested the performances of the real-world RSA implementation made available by *PyCryptodome* Python package using the same file.

TEST	encryption running time	decryption running time
1	0.00150108337402 sec	0.00978803634644 sec
2	0.00172686576843 sec	0.0106139183044 sec
3	0.00150108337402 sec	0.0106041431427 sec
4	0.0015070438385 sec	0.011589050293 sec
5	0.00215101242065 sec	0.0113499164581 sec
6	0.0014169216156 sec	0.0133240222931 sec
7	0.00202393531799 sec	0.0139780044556 sec
8	0.00140285491943 sec	0.00962710380554 sec
9	0.00165009498596 sec	0.0100989341736 sec
10	0.00149917602539 sec	0.00994896888733 sec

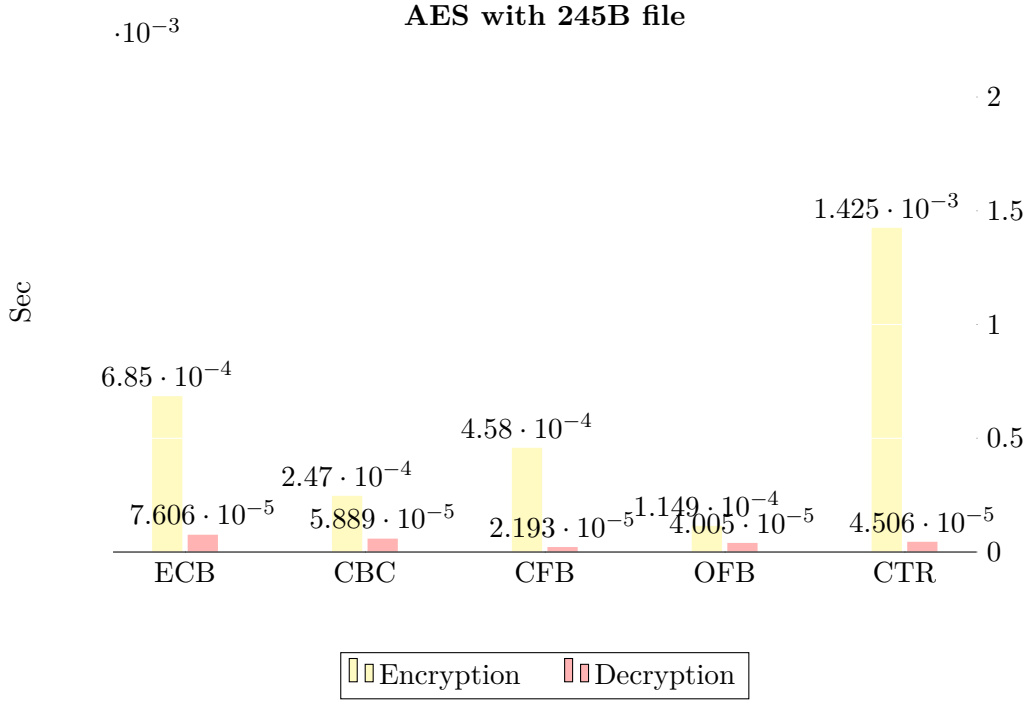
Relying on the values of the two tables previously shown, i computed the average time spent in encryption and in decryption by the two different implementations, and the comparison between them can be seen in the following plot.



The algorithm that i implemented is slower than the one made available by *PyCryptodome*.
This depends on the fact that *PyCryptodome* optimizes all mathematical operation such as the divisions made in *i2osp* function using a shift operator.
Moreover, RSA *PyCryptodome* implementation, as specified in the documentation [4], uses a default public key e equal to 65537, instead in my implementation this value is pseudo randomly chosen in range [3:$\phi(n)$] where $\phi(n)$ is the product of (p-1) and (q-1) which are values of 1024 bits.
This results in a value of the public key that is likely to be bigger than the one used in *PyCryptodome* implementation and as a consequence of that the computation in my implementation takes more time.

4.2 Comparison with the performances of AES real-world implementation

I tested the performances of the real-world AES implementation made available by *PyCryptodome* Python package.
The following plot shows AES performances on the same file of 245 Bytes used before, timing five different modes of operation.



As it's previously shown, my implementation of RSA scheme takes an average time of 0.0483714342117 seconds to encrypt and 0.0488172292710 secs to decrypt a file of 245 Bytes. Instead the implementation from *pycryptodome* library takes on average 0.00163800716400 secs to encrypt and 0.0110922098160 secs to decrypt the same file. Both these times are significantly greater than the ones performed by AES that we can see in the plot. All the modes of operation analyzed are remarkably faster than RSA.

Asymmetric schemes are slower than the symmetric ones. As we could notice in RSA the size of the keys is much larger than in AES where the length was 128/192/256 bit and the processes involved in an asymmetric schemes generate and extra overhead with respect to a symmetric scheme.
In fact, symmetric schemes are used to encrypt large chunks of data that needs to be transferred, instead asymmetric schemes are used in small transaction to guarantee authentication and for example to built a secure communication channel where the symmetric key could be exchanged.
Moreover in RSA we have a limit on the size of the data we want to encrypt [*equation 5*], and to encrypt big sizes we need a big modulus that would significantly slow the process.

References

[1] Understanding Cryptography by Christof Paar and Jan Pelzi

[2] Request for Comments: 8017. Internet Engineering Task Force (IETF)

[3] PKCS#1 v2.1: RSA Cryptography Standard. RSA Laboratories. DRAFT 1 - September 17, 1999

[4] PyCryptodome documentation https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html