Playing with PK schemes
Hw3 - CNS Sapienza

Gianfranco Romani 1814407

20 November 2020

1 Introduction

Before the introduction of asymmetric ciphers, a common key was needed by both parties to communicate through encrypted messages. This method needed safe ways to exchange the key and not always this was possible or easy. The problem was resolved by asymmetric cryptography with the use of two different keys, a public one, known by everyone, and the private one, known only by the addressee. There is no need for a safe channel to exchange the keys. The security of the system relays on the private key, so we need it to be computationally infeasible to determine. To reach this goal this kind of cryptography is based on mathematical problems that still don't have efficient solutions, but this makes asymmetric ciphers very slow compared to symmetric ciphers. Because of this limitation, these methods are used only to transfer small blocks of data, generally the key that will be used in a symmetric cipher, or to apply a digital signature.

In this report I am going to present my implementation of RSA and then I am going to compare its performances against real world implementation of RSA and AES.

2 RSA

RSA's effectiveness as an asymmetric cipher is based on the difficulty to compute the decryption without knowing the private key. This is guaranteed by the complexity to factorize a big integer into two prime numbers, computation that can take a very long time to be executed if the number that has to be factorize is at least 2048 bits long (1024 are not to be considered enough anymore).

To use RSA we need:

- the two prime numbers mentioned above, that we call p and b;
- their product n, called modulus;
- ϕ, which is given by $(p-1)*(q-1)$;
- e that has to be coprime with ϕ, i.e. gdc(e, ϕ)=1. It is used as the public key;
- d that is obtained from e such that $d*e=1\ mod\ n$. This is used as the private key.

Public key is composed by the couple *(n,e)*, whereas the private key is *(n,d)*. The encryption of messages is computed as the exponential of the plaintext, so the ciphertext y is given by $y = x^e\ mod\ n$. That implies that x has to be smaller than n or, after the decryption, we are going to have a message that is different from the original . Decryption is obtain by computing $x = y^d\ mod\ n$.

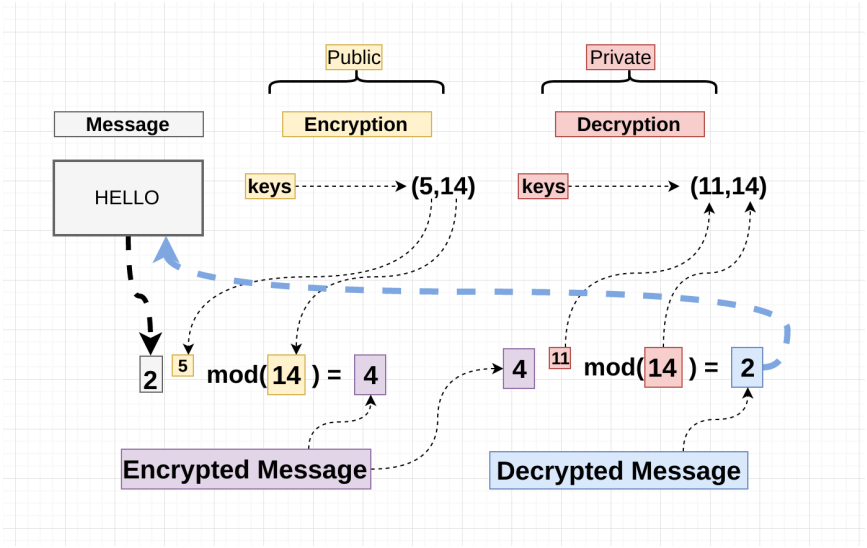

Figure 1: Scheme of how RSA works. Image from https://hackernoon.com/

3 Implementation

In my code are implemented three different ciphers, two taken from *PyCryptoDome* and my implementation of RSA. At the beginning of the exe-

cution of the program, it will be asked which of the three we want to use (to try my version type myRSA). The message to be encrypted and decrypted is fixed in this exercise (length of 120 characters).

Following the features defined in the previous paragraph, the first thing done by the program is to compute the values needed to use RSA, using the function $compute\_values(n\_bits)$, which takes as parameter the number of bits for the keys. As first thing $compute\_values(n\_bits)$ generates random numbers using $random.randrange(2^{(n\_bits-1)}+1,\ 2^{n\_bits}-1)$ since the necessity to avoid small numbers. These numbers have to be prime to be chosen as values for p and q, so the function $miller\_rabin\_test(number, k)$ is used. As the name of the function can suggest, I decided to use the Miller-Rabin test to check if the numbers generated are (most likely) primes. I said most likely because M-R test is a probabilistic method, like the Fermat primality test, that has a 75% probability of success. Iterating this method for enough times makes us pretty sure that the number tested is prime or not. I have chosen to iterate the method for 40 times (number that is generally suggested) to obtain a probability of error of 4^{-40}. When the primes are computed, we can finally calculate n, ϕ and e. The last value of those three, e, is obtained by starting from 3 and checking if it is coprime with ϕ, otherwise we increase it by one. The last value needed is d. This is computed with the extended euclidean algorithm implemented in the function $extended\_euclidean\_algorithm(e, phi)$. Before that, I tried to use the functions $moltiplicative\_inverse(e, phi)$, that uses an iterative approach,and $pow(e, phi-2, phi)$ but the performances were in both cases way worse when the bits were more than 10.

Once we have all the values, the message is encrypted using the function $encryption(message, e, n)$ and then decrypted by $decryption(message, d, n, n_bits)$. In both, to calculate the exponentials, I don't use the *pow* function but a square and multiply approach that is much faster.

4 Comments

The limits of RSA scheme are several, as shown in the following paragraphs.

4.1 Performances

RSA can be very computationally onerous compared to other methods. The cause of that is mainly the generation of the keys. This problem can be restrained thanks to probabilistic methods to check whether a number is prime, gaining a lot in performances but losing in certainty. Fortunately we have

seen in the section above that the uncertainty is too small to consider it a real threat. Rabin-Miller algorithm has a complexity of $O(klog^3n)$ where n is the number tested for primality and k is the number of iterations performed. Another problem, that is particularly evident in my implementation, is given by the computation of the exponential with such big exponents. In the previous versions of my implementation, using *pow* for this task, several hours were needed only for decryption. Despite the optimizations applied, RSA (in my implementation particularly) is much slower than AES. The followings are the mean time of several executions for each algorithm for both encryption and decryption with a message of 120 characters:

- AES: ≈1.2 s for AES-256;
- RSA (library): ≈2.9 s for 2048 bits;
- RSA (mine): ≈ 6 s for 1024 bits, more than 2 minutes for 2048.

We could decide to improve performances for encryption (but making decryption slower) by using a small e and a big d (carrying out several attempts I noticed that in my implementation it happens often and e is 3 or 5). The inverse (d small, e big) would compromise the security of the cipher.

4.2 Message length

If the message x is bigger than $n = p*q$ than the decryption can't obtain the original message. To solve this problem we could truncate the message (not recommended), show an error message to the user or divide the message into blocks and encrypt them one at the time. In my implementation I decided to just show the error message (as happens when using the RSA in PYCryptoDome). With AES such problem does not exist.

4.3 Security

To consider RSA secure it should be hard to obtain d, ϕ, p or q. To do so p and q should be bigger than 1024 bits (done in my implementation), e has to be equal or bigger than 3 (done in my implementation) and both (p-1) and (q-1) should have large prime factors (it is not guaranteed in my implementation), otherwise some algorithms can perform factorization efficiently. Even with all these precautions RSA cannot be considered fully secure and the only way to be prevent attacks is to use well known implementations.

5 Conclusions

Given the problem about performances that RSA has and the limitations about the message's length, it can be difficulty used for long messages. Instead, an efficient use of this method could be the encryption of the key that is going to be used for a symmetric cipher. Another application in which RSA results very good at is digital signature, because only the person with the private key can generate a specific signature that can be verified by everyone who has the public key.

References

[1] https://en.wikipedia.org/wiki/RSA_(cryptosystem)

[2] https://pycryptodome.readthedocs.io/en/latest/src/examples.html

[3] https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test

[4] https://stackoverflow.com/questions/6325576/how-many-iterations-of-rabin-miller-should-i-use-for-cryptographic-safe-primes