1. Blockchain

The completed classes as well as the function used to generate the comparative time table (***generate_time_table()***) can be found in the *blockchain.py* script located in the *Blockchain folder.*

This function generates a comparative table (both in *.txt* and *.csv* format, stored in the *Blockchain/Time comparisons*) of the cost of signing 100 messages of 256 bits using the ***sign*** and the ***sign_slow*** functions from the ***rsa_key*** class. These messages were signed using keys with 512,1024,2048 and 4096 bits respectively.

To obtain more reliable metrics, the mean execution time of 10 message signatures for each key length - method of signing has been used instead of obtaining the *ET* once. The library used for time tracking is ***time***.

Key length (bits)	Without CRT (s)	With CRT (s)
512	0.0641	0.0267
1024	0.3370	0.1333
2048	2.4114	0.7607
4096	17.2228	5.2659

As it can be seen looking at the table above, the *ET* of signing without using the *Chinese Remainder Theorem* is higher for all the key lengths than using it. This was an expected result, as CRT allows us to notably reduce the exponent size and thus, speed up the computations.

This can be clearly seen when comparing the different key length sizes. As this number increases, the execution time also does. This is due to the fact that by increasing the modulo length we are also increasing the exponent of the modular power (in both cases).

2. RSA
In this section, two encrypted files with AES, two AES keys encrypted with RSA, and their respective public keys were provided.

The objective consisted in deciphering the AES files exploiting vulnerabilities from the RSA key generation. Thus, in the following pages, I will indicate which vulnerabilities have been found in each subsection and how they have been used to obtain the original file.

2.1 Ron was wrong, Whit is right

To solve this problem, a previous read of this paper was needed.

The main idea I extracted from the paper consisted in that a large number of RSA keys weren't secure due to failures in the random factors generation. This was demonstrated practically after analyzing several RSA keys and checking that some of them shared a common factor in the module. This allows us to obtain the other factor trivially without needing to factorize the public modulus (what is RSA security based on).

Thus, to solve this problem, I computed the Greatest Common Divisor between my public key modulus and all the ones from my colleagues until I found one different from 1 (or my modulus itself). Then, once a factor is found *(p)*, we can obtain *q* as np, compute *(n)*, and then obtain the private key exponent using Bézout's Identity.

Once we have our RSA key, we can decipher the file that contains the key used for the AES message encryption and use it to decrypt our original file.

The code used to perform such procedures can be found in the *RW.py* file on the *RSA* folder.

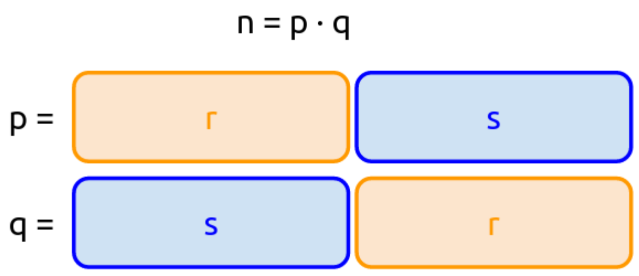
In my case, the original file consisted of a *.png* photograph of street art.

Decrypted image Ron and Whit section.

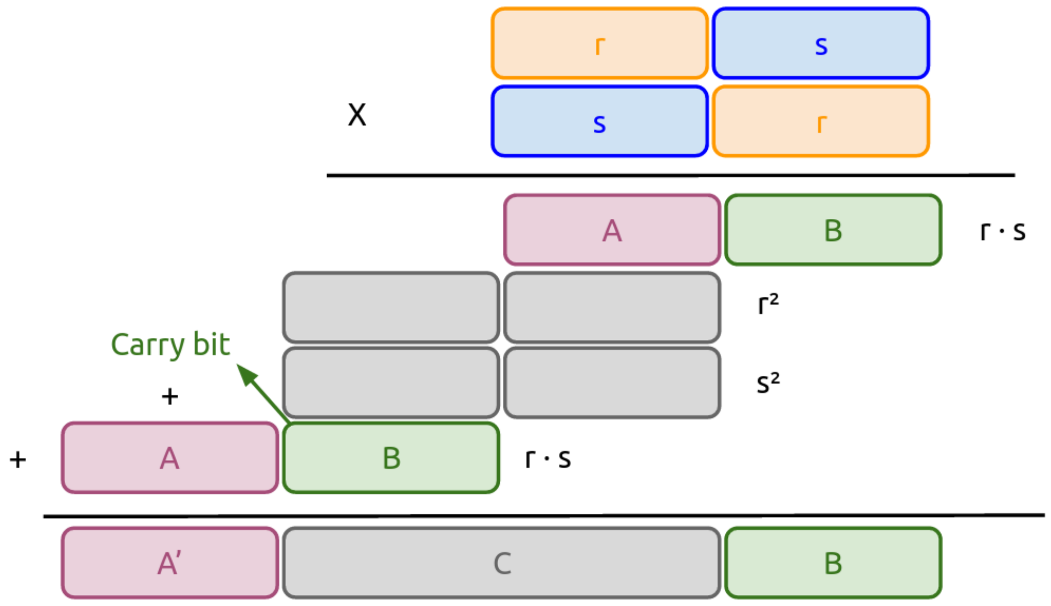
2.2 Pseudo RSA

In this case, in order to obtain the private exponent of the RSA key, we must exploit the fact that there exists a relation between the two factors, as well as we know how to construct them.

The key creation structure is the following:



Using this, we can extract information about certain bit blocks of the public modulus, as it can be seen in the following picture:



We are interested in obtaining the values of *(r + s)* and *(r · s)* in order to solve the following second-degree equation:

$$x^2 - (r+s)x - (r \cdot s) = 0$$

Equation 1

to obtain *r* and *s*. With them, we can construct both *p* and *q* and then proceed as we did in the Ron and Whit case.

However, we do not have *direct* access to such values. As stated above, *r · s* = AB, however we can only directly consult B. The first block of the modulus, which is A', is the A value with a carry bit (which can be 0,1 or 2 depending on the sum of the first block of C).

Now, focusing on C, we can observe that:

$$C = r^2 + s^2 + BA$$

Equation 2

from we can extract:

$$(r+s)^2 = r^2 + s^2 + 2rs \rightarrow (r+s)^2 = C - BA + 2AB$$

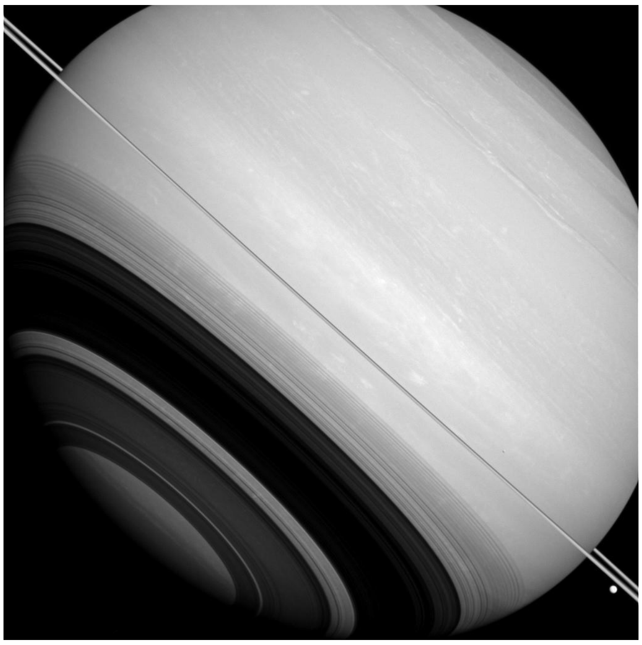
Equation 3

Thus, the procedure to find the *r · s* and *(r+s)* values is the following:

First, we select a carry bit value (0,1,2) and subtract it from the A' block (and add it to the C). Then, using both A' and B, the square root of *Equation 3* is computed. If the result is an integer, we can solve *Equation 1* and obtain the original message. If not, another carry bit is chosen and the procedure is repeated.

The code that performs all these actions can be found in the *pseudo.py* file on the *RSA* folder.

In my case, the original file consisted of a *.jpeg* photograph of Saturn taken by the Cassini-Huygens space probe on Aug. 14, 2014.



Circling Saturn - CICLOPS

More information about this picture can be found [here](#).