Cryptography

Lab 2

Secret Key

Víctor Novelle Moriano

October,2021

FIB

1. Chacha20

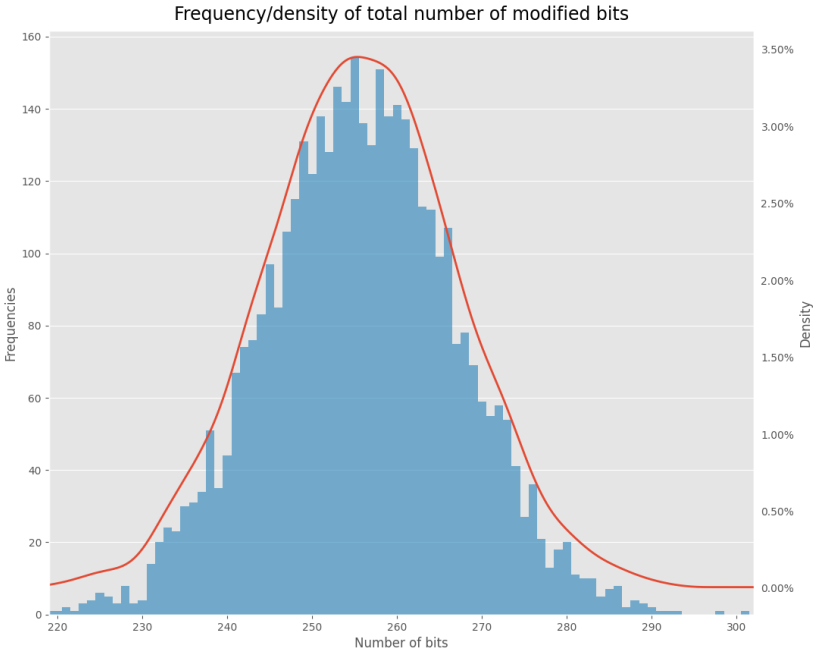
The Chacha20 implementation used for performing the different sections of this exercise can be found on the following Github repository. Notice how in the repository implementation, in the state matrix 64 bits are used for the counter and another 64 for the Nonce. This has been modified, assigning only 32 bits for the counter and 96 for the Nonce, replicating the state structure presented on the lecture notes.

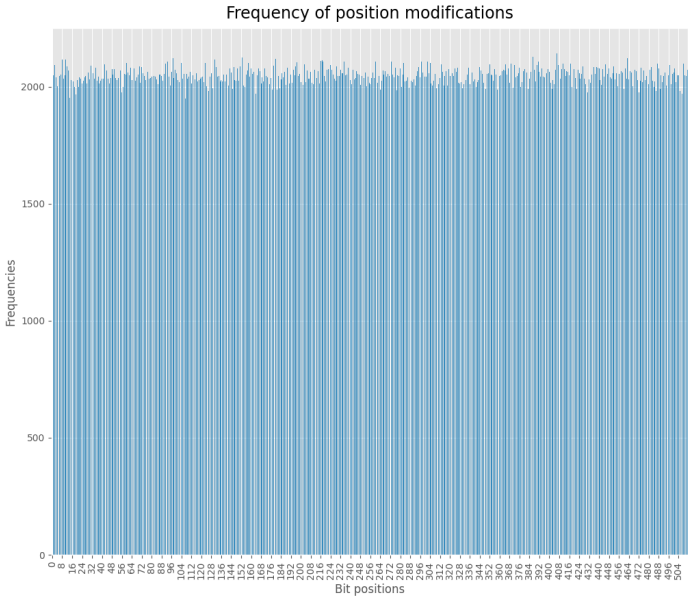
1.1 Small changes propagation

The function ***histograms*** from the script ***Chacha20.py*** generates the histogram of the total number of modified bits as well as the number of times a position has been modified.

This function encrypts a 64 zero byte text with a different Counter value each time, starting from 2 and finishing in 4096. For each execution, the obtained cyphertext is compared to the one obtained using Counter= 1 and the positions that have been modified are stored as well as the total number of different bits. Afterwards, both histograms are created using the ***generate_plot*** function, which stores them in the *Images* folder.

The obtained plots are the following:





As it can be seen, the average number of different bits between the first encrypted text and the ones obtained performing the Chacha20 algorithm only modifying the Counter value is 256. Moreover, all the bits are modified evenly. This is a good result, as it shows that there isn't a simple correlation between the expanded bits using different counters.

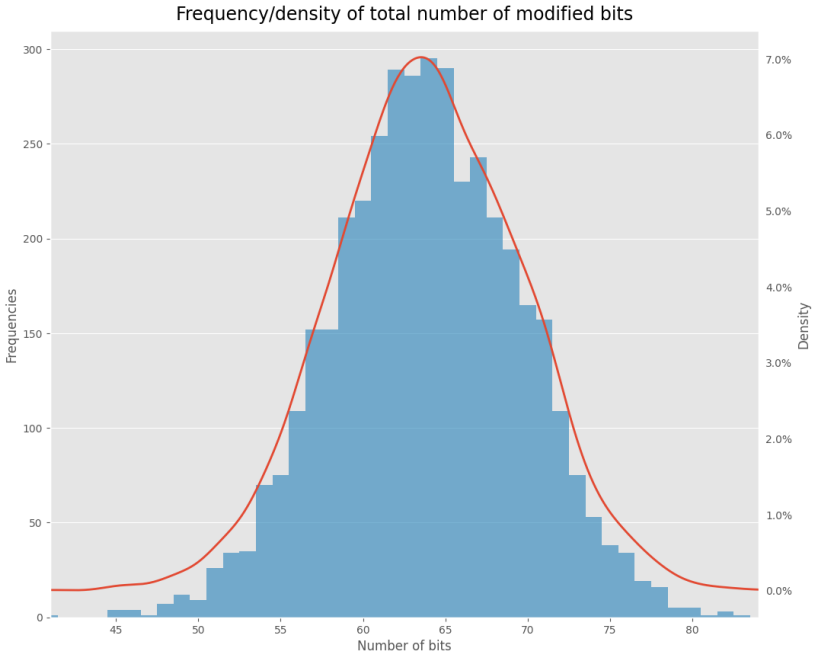
1.2 Effects of elemental functions

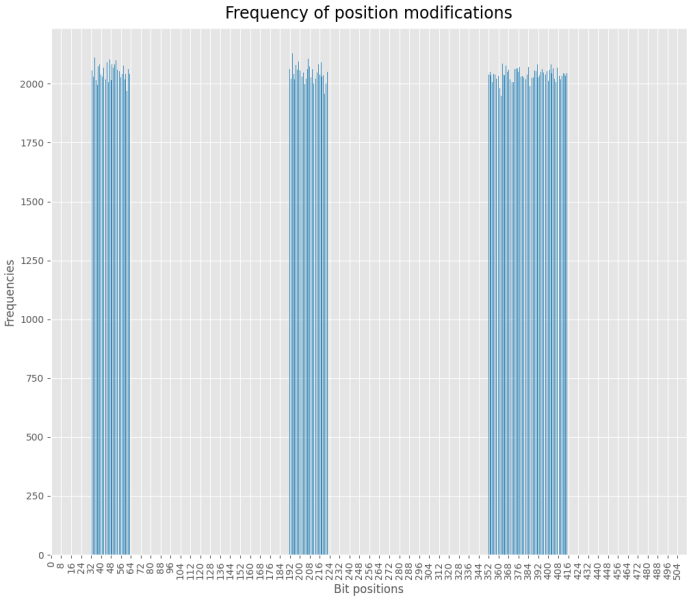
I would like to emphasize the fact that to perform this section no additional functions have been used. The ***histograms*** function has a parameter named *method* that indicates on which states the QUARTERROUND operations are applied. This parameter can have the following values:

- **0:** Traditional implementation of the Chacha20, where the QUARTERROUNDS of both Column and Diagonal rounds are applied (Used in the previous section).
- **1:** The QUARTERROUNDS from Column rounds aren't executed.
- **2:** The QUARTERROUNDS from Diagonal rounds aren't executed.
- **3:** The first QUARTERROUND operation from both Column and Diagonal rounds isn't executed.

As in the previous sections, the histograms are stored in the *Images* folder. To distinguish them, each one is stored indicating which operations weren't performed.

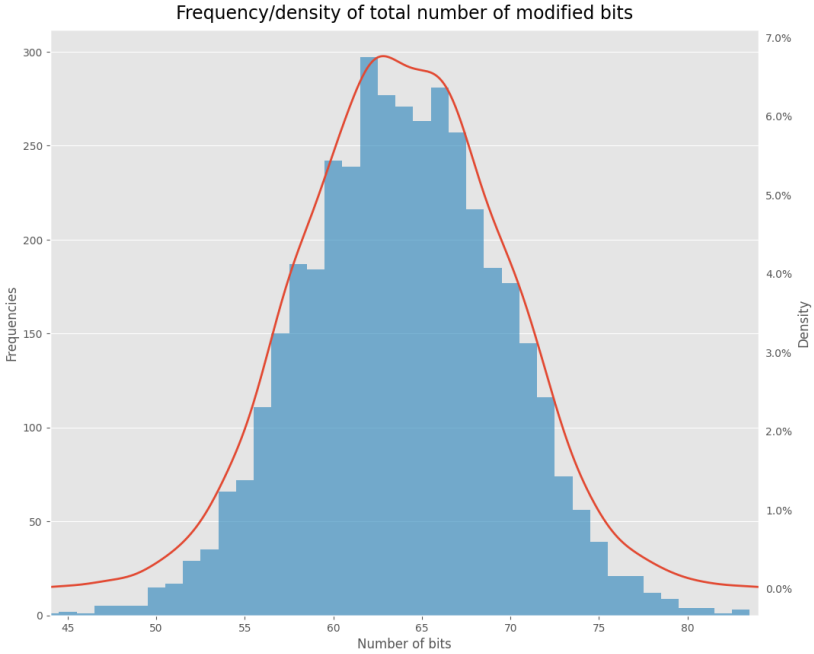
Without Column rounds

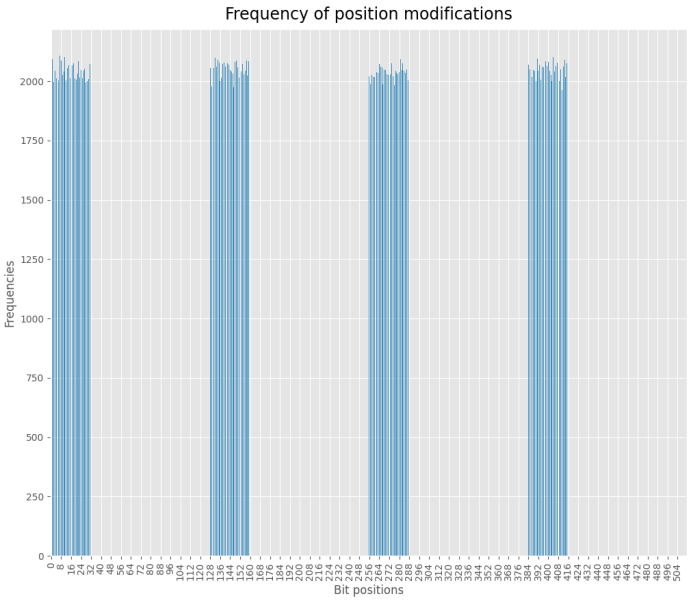




As it can be seen, the average number of modified bits has dropped drastically, being now around 64, as well as now, only a subset of bits are modified with the others remaining equal. The modified bytes are [4-8, 24-28,44-52]

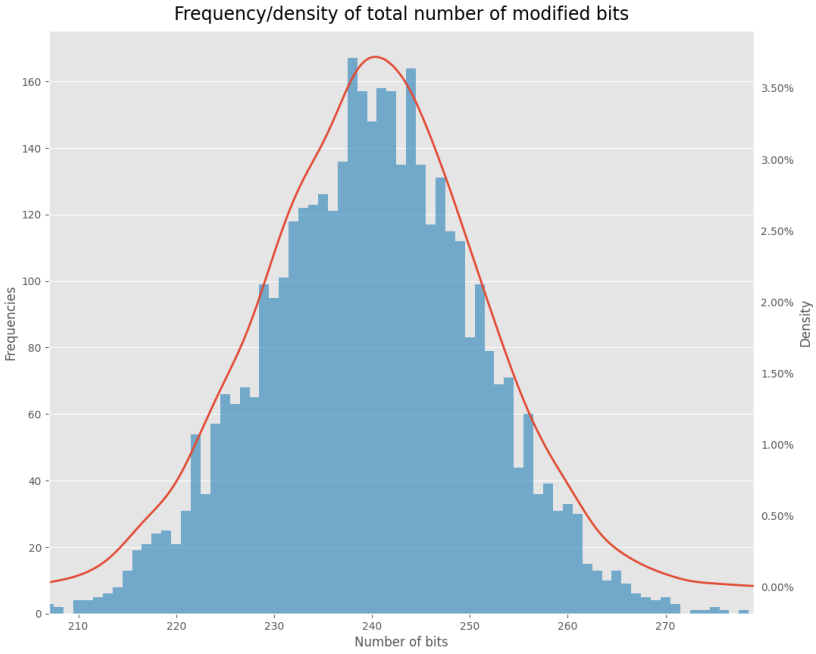
Without Diagonal rounds

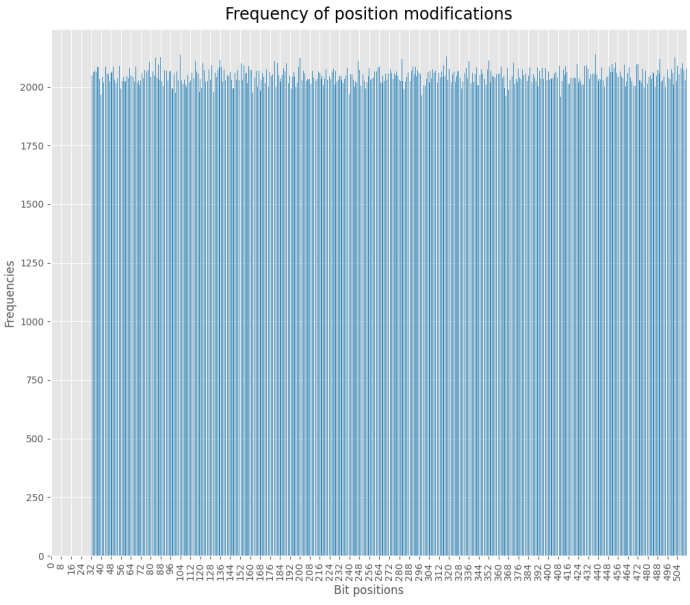




Regarding the first histogram, the obtained result is similar to the one seen when removing the Column rounds. The main difference is present on which bits are modified, being, in this case, the bytes [0-4,16-20,32-36,48-52].

Without Specific rounds





In this case, it can be seen the distribution of modified bits is displaced to the left comparing it with the original but notably less than in the previous cases. This is better seen observing the right table, as it shows us that the first four bytes aren't changed. This was an expected result, as deleting the selected QUARTERROUNDS provoke that the top-left cell from the state matrix remains always equal.

2. The finite field GF (2^8)

The requested functions for the sections **i)** to **v)** can be found in the ***GF.py*** script.
In the script there is also programmed an extra function, ***execute_test,*** which ensures that the following properties are respected:

- *GF_product_p (a,b) = GF_product_t(a,b)* $\forall a,b \mid a,b \in [0,256)$
- *GF_product_p (a,b) = GF_product_p(b,a)* $\forall a,b \mid a,b \in [0,256)$
- *GF_product_p (a,GF_invrs(a))* = 1 $\forall a \mid a \in [0,256)$

To execute the time comparison requested for this exercise the ***time_evaluation*** function has been used. This function generates a comparative table (both in *.txt* and *.csv* format, stored in the *Outputs/ET_factors* folder) of the cost of multiplying all the 256 values from the finite field against the provided fixed values (*0x02, 0x03, 0x09, …*) using the ***GF_product_p*** *(polynomial multiplication)* and the ***GF_product_t*** *(table look-up multiplication).*

To obtain more reliable metrics, the mean execution time of 1000 multiplications for each operation execution has been used instead of obtaining the *ET* once. The library used for time tracking is ***time***.

Factor	GF_product_t ET	GF_product_p ET
0x02	8.45317840576171E-05	0.000742123126984
0x03	9.07504558563232E-05	0.000792327165604
0x09	8.68737697601318E-05	0.000761516332626
0x0B	8.20512771606446E-05	0.000730665683746
0x0D	8.21163654327393E-05	0.000734364509583
0x0E	8.2033634185791E-05	0.000734432458878

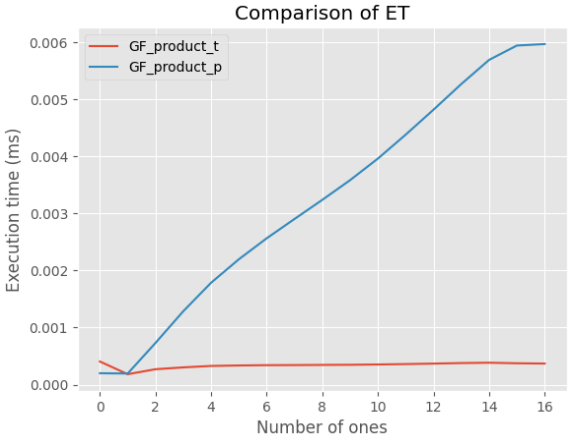
As it can be seen, for all the tested factors the ***GF_product_t*** function is faster than the polynomial multiplication one.

In addition to the requested time comparison, two additional tests were also performed:

In the first one, I wanted to check if the amount of 1 value present in both binary numbers affected the execution time of the multiplication. The reasoning behind this idea lays in that, in the **GF_product_p** multiplication, the *multiply_by_x* function is called each time there is a 1 on the *a* factor and, when more ones are present in the *b* factor, more times the if is executed.

Thus, to compare and visualize the execution times depending on the number of ones in the factors the ***time_evaluation_extra*** function has been implemented. To do so, all the possible pairs of multiplications have been executed, averaging their ET performing 100 executions.

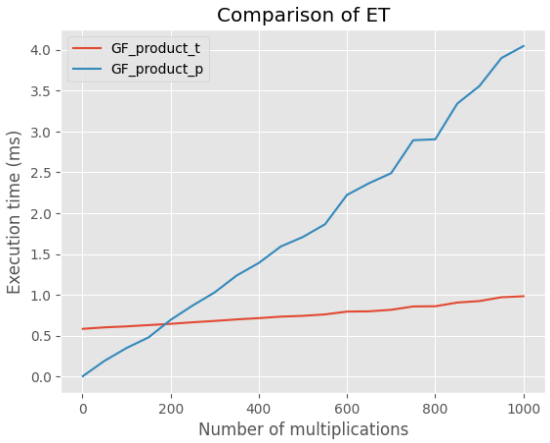
Then, the times depending on the number of ones are averaged taking into account how many operations with such number of 1s were executed. Lastly, the ET tables, as well as a comparison graphic are stored in the *Outputs/ET_ones* folder.



As it can be seen, as the number of ones is increased, the execution time of the polynomial multiplication also increases. Regarding the ***GF_product_t*** ET, it almost remains constant, as in every execution it only has to perform the look-ups on the tables, independently of the number of ones of each factor.

Lastly, I wanted to check when one function is more convenient than the other in a real set-up. If we want to perform some multiplications, in general, we won't know the amount of 1 bits in the factors and we will need to take into account the creation time of the tables, as even though the operation time is lower, considering the creation time it may be higher.

Thus, to evaluate this setup the ***time_evaluation_extra_2*** has been programmed. In this function, a random set of multiplications of size k is chosen. Then, the execution time of both multiplication functions is compared (taking into account the creation table time for the ***GF_product_t***) performing an average of 1000 executions. This procedure is repeated for different values of k starting from 1 and finishing in 1000, with a size 50 step. Finally, the ET tables, as well as a comparison graphic are stored in the *Outputs/ET_real* folder.



As it can be seen in the plot above, the table look-up approach is notably faster if a big amount of multiplications need to be performed. However, if 150 or fewer multiplications need to be executed, it is recommended to use the polynomial approach, as despite being slower per operation, the creation of the tables consumes more time.

3. Secret key cryptography

The functions used for performing both sections of this exercise can be found in the ***Decrypt.py*** file.

3.1

To perform this section, the code present in the ***Decrypt_1*** function has been used. On it, different AES operation methods have been used to try to decrypt the provided file (using also the provided key). To speed up the evaluation of the decrypted files and avoid checking manually several garbage documents, after the decryption with a method has been performed we check with *filetype* if the result has some known type. If so, we evaluate if the PKCS7 padding is correct and if it is the case, the result is stored in the *Decrypted* folder.

In my case, the original file was an insect *.jpg* image encrypted using AES in **CBC** operation mode.

3.2

To perform this section, the code present in the ***Decrypt_2*** function has been used.
For this part, we have to find the 16-byte key used for the encryption, as well as the 16 byte IV vector used.
If we don't have any more information, in the worst case we will need to test 2^{256} possible values. However, with the information provided in the articulation of the exercise, this value can be notably reduced:

- First, the key and the IV vector are constructed by applying SHA256 to a *PreMasterKey* of 16 bytes. Thus, we would need to test 2^{128} values.
- By construction, the *PreMasterKey consists* of two different bytes which are repeated 8 times and afterwards concatenated. Thus, we would need to test 2^{16} values.
- All the possible characters that can be used in the *Ki* keys can be represented with 7 bits. Thus, the first bit of the two bytes from the *PreMasterKey* is always 0. Thus, we only need to test $\mathbf{2^{14}}$ values.

Thus, in the ***Decrypt_2*** function, first, we generate one of the possible PreMasterKey from the 2^{14} ones, then the SHA256 is applied to the key and the decryption using the AES in CBC operation mode is tested.

As in the previous section, once a description try is successful, the result is saved in the *Decrypted* folder with the name *backdoor*.

In my case, the original file was the song *Pattern Of My Life* by *Annie Lenox* , in format *.mp4*. For encryption, the key used was **0xc9da747337426d77c253b1c07ab9ae96** and the IV was **0x854bfe872ba61c6a5ad6e3d0e1a71e9d.**