CPA Attack to Embedded AES Algorithm

HSES

2022 - 2023 Q2

Lourdes Bruna

Òscar Pérez

Albert Vilardell

Table of Contents

Introduction.......... 3
Dataset 1.......... 3
Dataset 2.......... 5
Improving dataset 1.......... 7
Using Rust instead of Python.......... 8
Correlation comparison.......... 9
Performance comparison.......... 10
Conclusions.......... 12

Introduction

Correlation Power Analysis (CPA) is a type of side-channel attack that can be used to exploit the correlation between the power consumption of a cryptographic device and its secret key during the execution of cryptographic algorithms like AES encryption.

In this project, we are given some traces about the power consumption of a PIC18F4520 microcontroller. The traces belong to the point between SubBytes and ShiftRows. The goal is to obtain the key of the AES encryption. Two datasets are provided. The first one is well clocked while the second is not. It is interesting to see whether this technique adds a layer of difficulty in the process of obtaining the key.

This document explains how we obtained the keys of both datasets. After, some modifications to the code are discussed and implemented, so that the process is faster. Then, there is an analysis on whether the correlation results are different in both datasets, and the performance of the different versions of the programs. Finally, we draw some conclusions about the project.

Dataset 1

In this first dataset, traces are well clocked. Thus, we can directly calculate the correlation coefficient between the aligned power consumption traces (experimental current consumption) and the corresponding hypothetical power consumption traces (Hamming Weight model estimating current consumption). In order to do it, firstly we need to generate the HW model:

$$\boxed{POWER \propto HW\,(\,SBOX\,(P \oplus K)\,)}$$

The power consumption is proportional to the number of active lines in the bus and, for AES encryption, the SBOX corresponds to the following array:

```
sbox = [
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16]
```

Thus, the model is generated in the following way:

```
model = []

for x in range(150): # For all the cleartexts
    trace = []
    for y in range(256): # For all the keys
        xor = (cleartext[x][z])^(y)
        HW = bin(sbox[xor]).count('1') # SBOX + count the number of 1s (Hamming weight)
        trace.append(HW)
    model.append(trace)
```

After transposing the model, we can calculate the correlation coefficient and choose the biggest one. We log all the coefficients greater than `0.7`, but we only keep the highest one.

```
for k in range(50000):
    corr_i = []
    trace_i = []
    for j in range(150):
        trace_i.append(trace_byte[j][k])

    for i in range(256):

        corr0 = np.corrcoef(trace_i, model_transposed[i])

        value = abs(corr0[0][1])
        if (value >= 0.7):
            print("       Potential match at t = "+str(k)+" with key = "+str(i)+" and correlation of "+str(value))
            if (value >= max): # We store the value with the highest correlation (i.e. the key)
                max = value
                key_i = i
```

With this algorithm, we obtain the key:

```
[65, 117, 115, 116, 114, 97, 108, 111, 112, 105, 116, 104, 101, 99, 117, 115]
```

The addition of the bytes is indeed the checksum, `1712`.

However, this process is too slow. We need approximately **9 hours** to find all the key bytes (more than 30 min per byte).

This program is located in the file `dataset1_key.ipynb`. If it is opened with a Jupyter notebook, the original output can be seen.

Dataset 2

In this case, the dataset is not well clocked. This means that power consumption traces are not aligned, as one can see in figure 1.

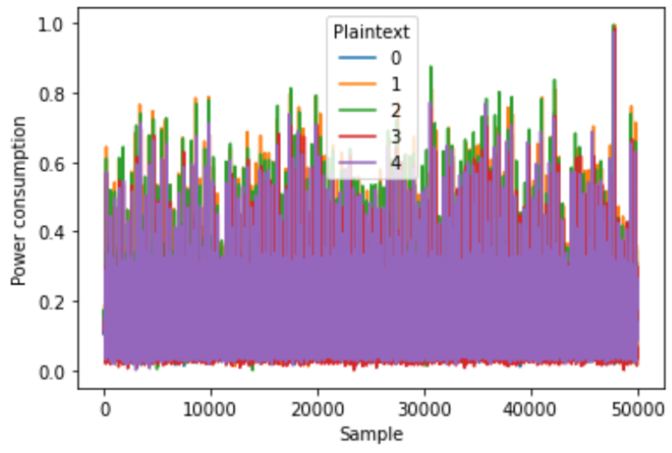

Figure 1. The 5 first plaintext traces from the file *trace0.txt*.

To solve it, we only extract the useful information from each trace. This useful information can be found on the rising edge of the clock (figure 2), when memory transactions are carried out, as they are responsible for a considerable power consumption:

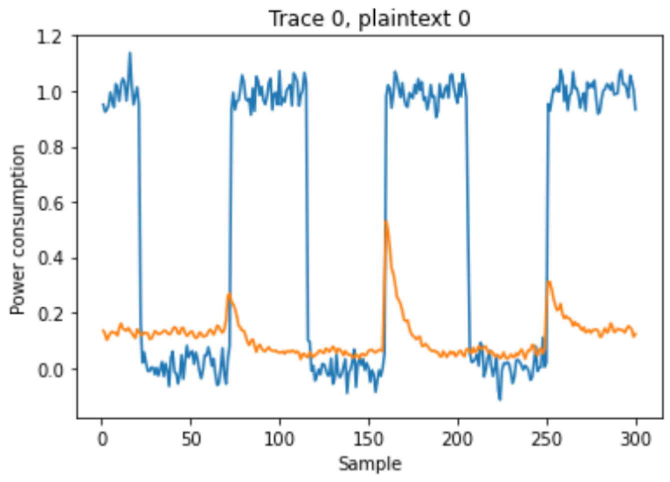

Figure 2. First plaintext trace and its clock signal from the file *trace0.txt*.

Thus, we search for the moment when the clock increases from a value lower than `0.8` to a higher value. In other words, the transition from `0` to `1`. Then, we save only 5 trace values before the transition and 5 more after the transition (this means 10 values for each rising edge of the clock):

```
limit=0.8
trace_flank = []

for i in range(150):
    aux = []
    for j in range(50000):
        if (j > 5 and j < 49995 and clock_byte[i][j-1] < limit and clock_byte[i][j] >= limit):
            for x in range(10):
                aux.append(trace_byte[i][j-5+x])
    trace_flank.append(aux)
```

We repeat the process for all the traces. In this way, the resulting traces (i.e. extracted information) are synchronized because we are using the power consumption values corresponding to the rising edges of the clocks (figure 3).

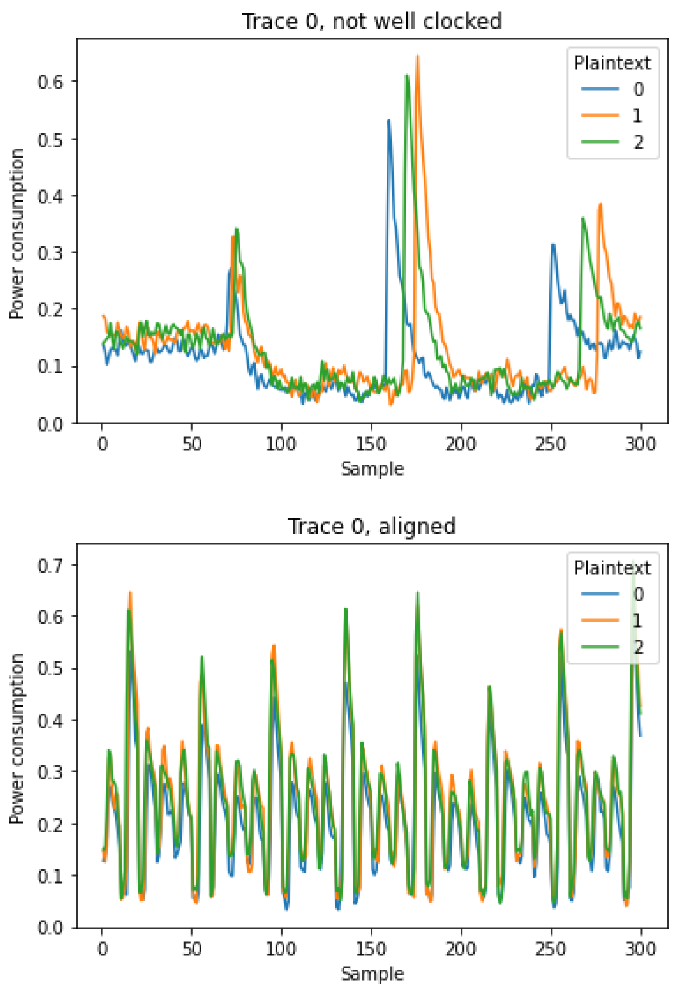

Figure 3. Plaintext traces before extracting the useful information (up) and resulting plaintext traces after extracting the useful information (down).

Finally, the CPA attack is carried out in the same way that with dataset 1, but only using the extracted information instead of the whole trace. It results in analyzing less data. Instead of `50.000` time instants for each key byte, we only extract about `5.000`, which results in **10 times less** executions of the algorithm.

As a result, we obtain the key:

```
[84, 104, 97, 116, 115, 32, 109, 121, 32, 75, 117, 110, 103, 32, 70, 117]
```

The addition of the bytes is indeed the checksum, `1434`.

The execution time is about an hour, which is much faster than in dataset 1 given the fact that we are correlating less data, as we have previously explained.

This program is located in the file `dataset2_key.ipynb`. If it is opened with a Jupyter notebook, the original output can be seen.

Improving dataset 1

If we want to improve computation time for dataset 1, we can also extract useful information. In this case, we do not have the clock signal but, since traces are synchronized, we can directly extract the values where there are power consumption peaks (figure 4).

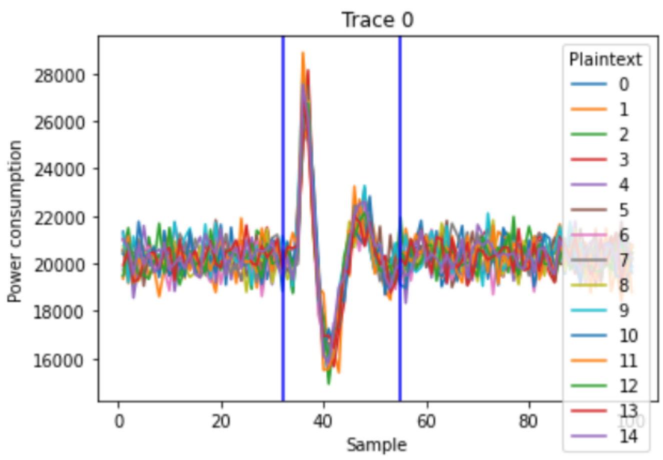

Figure 4. The 15 first plaintext traces from the file *trace0.txt*.

As one can see, peaks are produced around sample 40 (we can take a range between 20 and 60) and, since there are 100 samples in a period, peaks are repeated always at samples ended in 40:

```
trace_flank = []

for i in range(150):
    aux = []
    for j in range(40,50000,100):
        for x in range(40):
            aux.append(trace[i][j-20+x])
    trace_flank.append(aux)
```

The resulting trace is shown in figure 5.

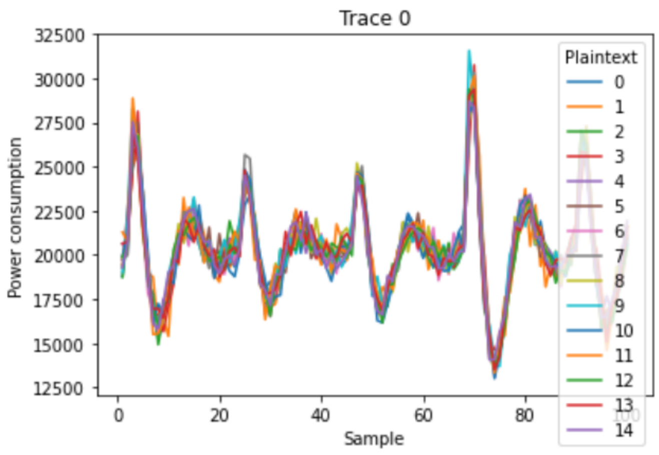

Figure 5. Resulting plaintext traces after extracting the useful information.

In this way, if we apply the CPA, we obtain the same key:

```
[65, 117, 115, 116, 114, 97, 108, 111, 112, 105, 116, 104, 101, 99, 117, 115]
```

The computation time is reduced until approximately **1 hour**. This program is located in the file `dataset1_extract_info.ipynb`. If it is opened with a Jupyter notebook, the original output can be seen.

Using Rust instead of Python

As we have already seen, although improving dataset 1 execution time by selecting only the useful information, we still got a considerably slow result.

We concluded that the limiting factor was the programming language selected itself, Python. So, we tried to move the python program to a faster programming language, Rust.

After moving the dataset 1 program to Rust we obtain all key bytes in just **210 seconds (3.5 min)**, instead of 9 hours as in Python.

The rust program is located in `dataset1/rust/main.rs.`

The same was done for dataset 2, generating the traces with an auxiliary python script, following the same logic explained before for locating traces at clock rising intervals, located in `dataset2/rust/main.py.`

Using the same Rust program as dataset1, but with the generated traces from the python program, we are able to obtain all key bytes in just **19 seconds**, instead of 1 hour as in Python.

Correlation comparison

Dataset 2 was designed to harden the process of obtaining the key. Some changes to the code were required in order to extract it. However, we have considered it interesting to compare the correlation obtained in both datasets (figure 6). For reference, the ideal value is always 1.

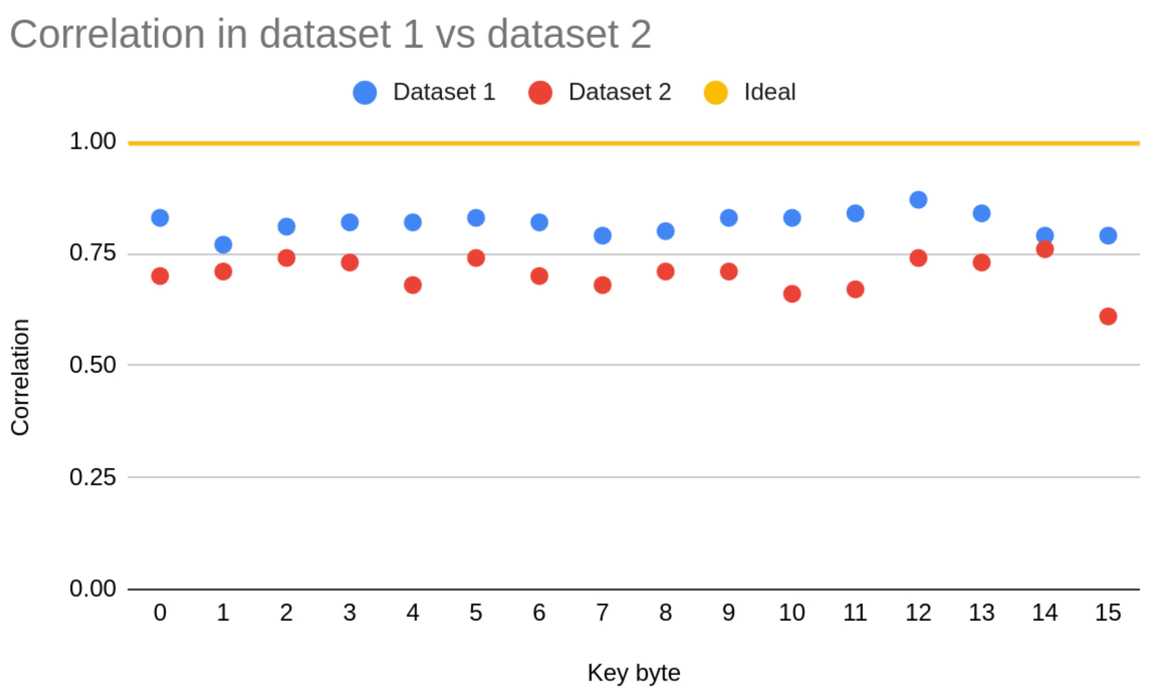

Figure 6. Correlation values in datasets 1 and 2 for each key byte.

As one can see, dataset 2 has smaller correlations than dataset 1. So, dataset 2 also makes it harder to obtain the results as the correlation is not as *obvious* as in dataset 1.

Other relevant data, such as the minimum, maximum, average and standard deviation of correlation of the time instants from which the key is obtained is also attached in table 1.

Dataset	Minimum	Maximum	Average	Standard deviation
Dataset 1	0.77	0.87	0.82	0.025
Dataset 2	0.61	0.76	0.7	0.038

Table 1. Minimum, maximum, average and standard deviation of the correlation values for both datasets.

In summary, we can see that dataset 2 obtains correlations that are smaller and that vary more than in dataset 1.

Performance comparison

First, we have compared the performance of the three implementations of dataset 1 in Python and Rust (figure 7).

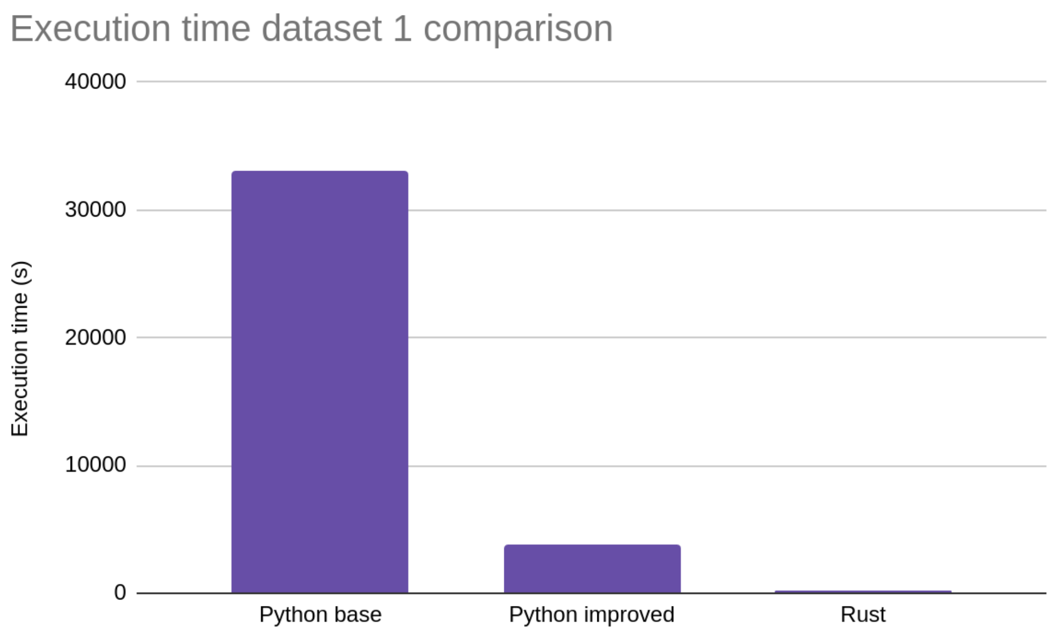

Figure 7. Dataset 1 execution time comparison.

The speedup obtained is **8.61x** for Python improved (respect to Python base) and **157.27x** for Rust (respect to Python base).

We have also analyzed the performance of the two implementations of dataset 2, the one in Python and the one in Rust.

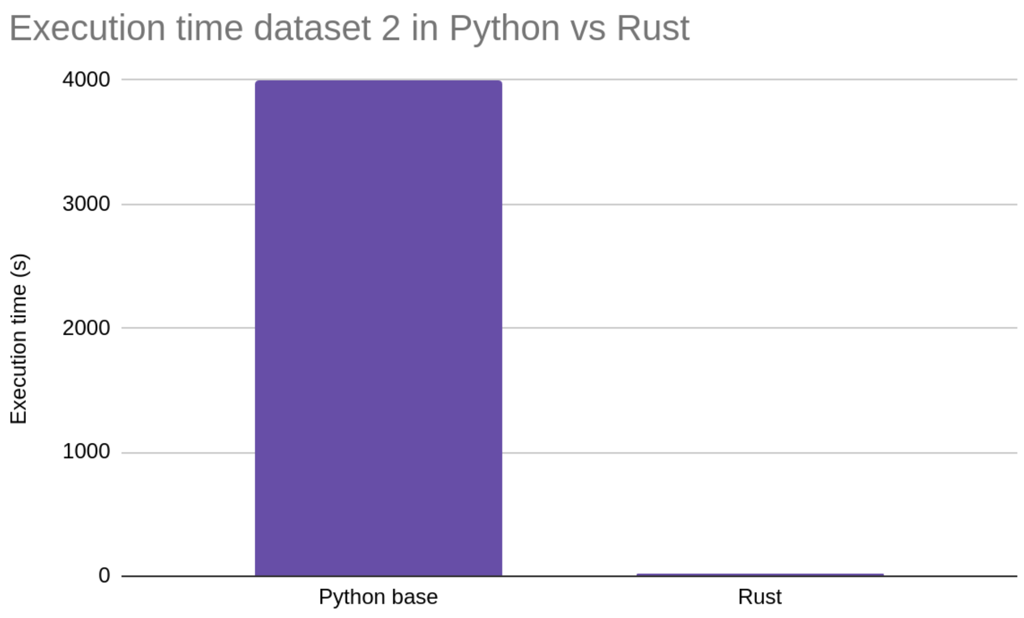

Figure 8. Dataset 2 execution time comparison in Python and in Rust.

The speedup obtained is **210x**.

After, we have compared the performance of the basic version of dataset 1 in Python and dataset 2 in Python.

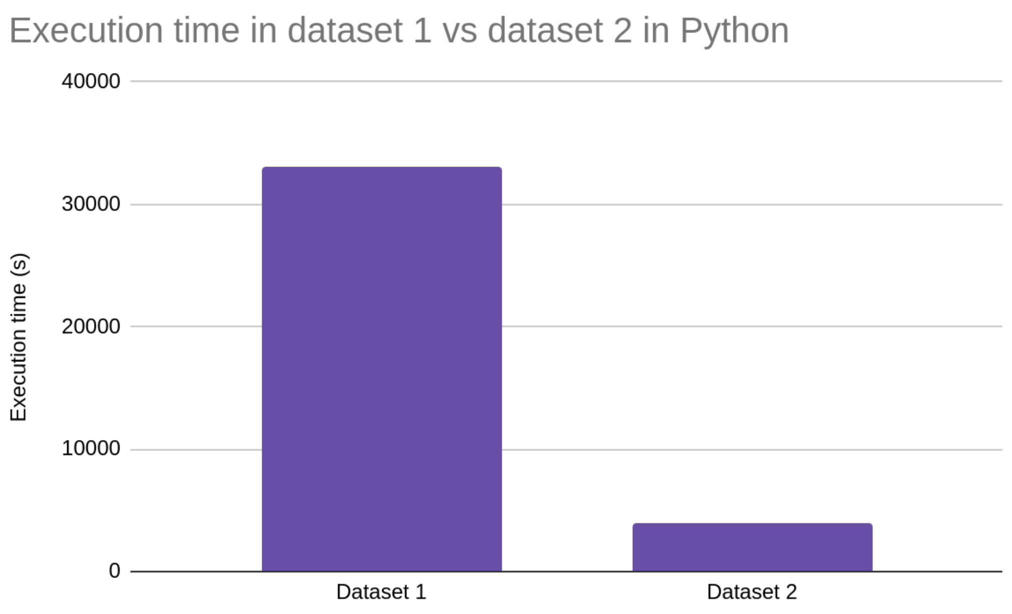

Figure 9. Dataset 1 vs Dataset2 execution time comparison for python version.

The speedup obtained is **8.26x**.

Lastly, we have compared the performance of dataset 1 and dataset 2 in Rust.

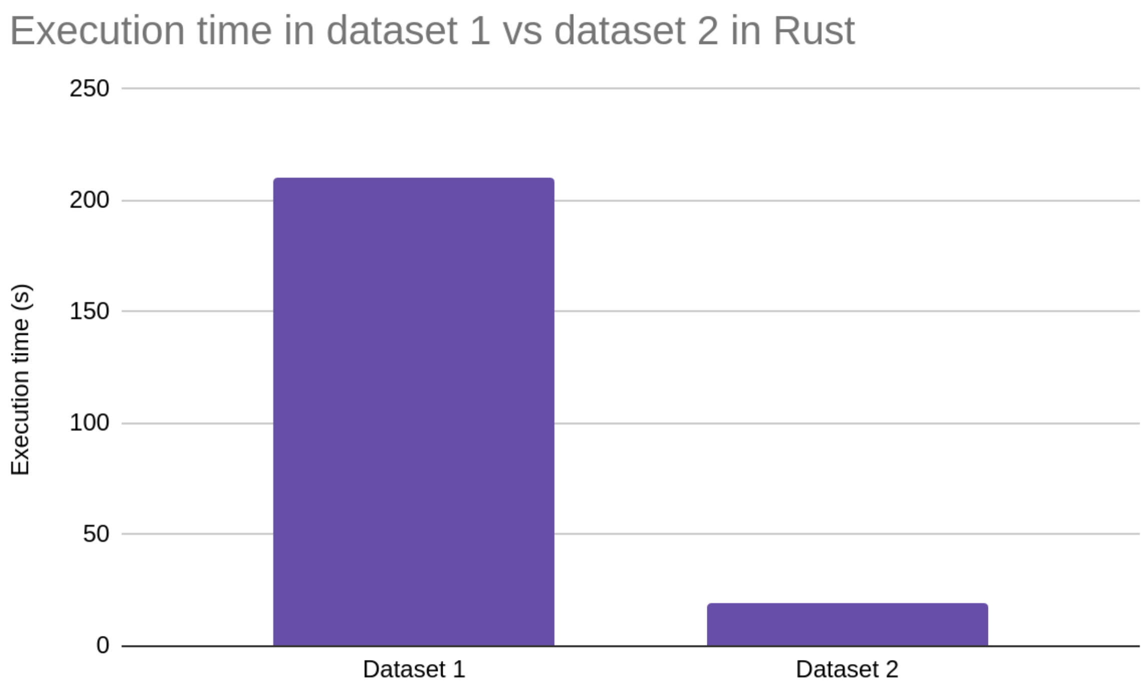

Figure 10. Dataset 1 vs Dataset2 execution time comparison for rust version.

The speedup obtained is **11.05x**.

As a summary, the execution time of all the versions and the file where they can be found is included in table 2.

Version	Execution time (s)	Execution time (HH:MM:SS)	Filename (they located in folder Dataset X)
Dataset 1 - Python base version	33026	09:10:26	`dataset1_key.ipynb`
Dataset 1 - Python improved version	3836	01:03:56	`dataset1_extract_info.ipynb`
Dataset 1 - Rust	210	00:03:30	`rust/main.rs`
Dataset 2 - Python	3997	01:06:37	`dataset2_key.ipynb`
Dataset 2 - Rust	19	00:00:19	`rust/main.rs`

Table 2. Summary of execution times obtained in each version.

Conclusions

We have been able to obtain the keys of the two datasets, which are included next.

The key obtained for dataset 1 is:

```
[65, 117, 115, 116, 114, 97, 108, 111, 112, 105, 116, 104, 101, 99, 117, 115]
```

The key obtained for dataset 2:

```
[84, 104, 97, 116, 115, 32, 109, 121, 32, 75, 117, 110, 103, 32, 70, 117]
```

We have realized that **misaligning the clock is a good technique against CPA attacks**. It makes it harder to perform the attack, and the correlation obtained is lower. However, it can be overtaken, as we have proven in this project.

We have also seen that the way the attack is implemented is very important. We do not recommend using Python for this kind of attack as it is slow. Thus, using Rust we can obtain a better execution time.