Investigating the Effectiveness of DCA Attacks on a White-Box Cryptography Implementation

Khelif Jawed, Le Diréach Gurvan

October 12, 2023

Abstract

This project was realized within the context of the UE "Research Project" of the Master Cybersecurity of the University of Rennes. The code of the White-Box implementation is available on this git [1] and the code of the attack as well as the tools and traces are available on this git [2].

1 Introduction

There are many ways to protect the cryptographic keys of encryption algorithms. One approach tries to hide a cryptographic key in the software implementation of the encryption algorithm. This is the white-box implementation. The idea is to embed the secret key of the algorithm inside the cryptographic operations in order to make it difficult to extract the key even with access to the source code. Often this method is coupled with reverse engineering countermeasures against obfuscation (code and control structure). Our goal is to show that a white-box implementation alone is fallible and can be broken by a side-channel attack. We first study and implement a symmetric cryptographic algorithm, the AES-128, and then attack this implementation by side-channel by analyzing the memory calls produced by the implementation.

2 White Box model

For this project, we implemented a white-box AES 128 based on James Muir's paper [3] Before diving into the implementation details, let us introduce the basic concept of a white box.

White boxes are cryptographic algorithms designed to resist an attack model known as the white-box attack context. In this model, the attacker has all the details of the implementation, including the input, output, and intermediate state of computation. Their goal is to extract the key. A practical example from day-to-day life is software that is part of some digital rights management (DRM), where the content is sent encrypted and decrypted by the software while it is being viewed (Netflix, spotify, etc.).

2.1 Advanced Encryption Standard

Now that we have discussed the context, we can talk about the implementation. Table-based implementations are lookup tables that map plaintext to ciphertext under a fixed key. For practical reasons, we cannot afford to store all possible output for a fixed key. Thus, we will use smaller, obfuscated tables that will be described below.
To implement this, we need to make some brief modifications to the classical AES 128 encryp-

tion . Here is the classical scheme.

- Key Expansion: The 128-bit key is expanded into a key schedule consisting of 11 round keys, each 128 bits long.
- Initial Round: The input data is XORed with the first round key.
- Main Rounds: The input data is processed through 9 rounds of substitution, permutation, mixcolums, and XOR operations using the round keys.
- Final Round: The input data is processed through a final round of substitution, permutation, and XOR operations using the last round key.

To achieve this, we will implement the following functions where the `state` variable represents the intermediate state of the input data.

- `AddRoundKey` takes a 16-byte round key, k_r, and uses exclusive-or to update the 16-byte `state`.

- `SubBytes` uses the substitution table, `S-Box`, to substitute each byte of the `state` with the result of the S-Box transformation.

- `ShiftRows` rearranges the bytes of the state using the following permutation:

$$\sigma_{SR} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 5 & 6 & 7 & 4 \\ 10 & 11 & 8 & 9 \\ 15 & 12 & 13 & 14 \end{pmatrix}$$

- `MixColumns` updates the `state` four bytes at a time. Each set of four bytes is treated as a polynomial over the finite field $GF(2^8)$ and multiplied with a fixed polynomial from $GF(2^8)$.

We will make the following modifications. First, we will insert the `AddRoundKey(`key_0`)` in the loop and move the `AddRoundKey(`key_9`)` out of it. Then, we will swap `AddRoundKey()` and `ShiftRows()` . To make this, the key inputted in the `AddRoundKey` should be shifted (b_k in the figure below). This will work fine due to the linearity of `ShiftRows()`.

Input: plaintext
Output: ciphertext
state ← plaintext;
for $r = 1$ **to** 9 **do**
  ShiftRows(state);
  AddRoundKey(state, $b_{k(r-1)}$);
  SubBytes(state);
  MixColumns(state);
  ShiftRows(state);
end
AddRoundKey(state, b_{k9});
SubBytes(state);
AddRoundKey(state, k_{10});
ciphertext ← state;

Algorithm 1: AES modified encryption algorithm

After modifying the classical scheme of AES, we will combine certain steps and compute all possible outputs into a table. As a security measure against cryptanalysis, we must never look twice at the same place. This means that we need a separate table for each step.

2.2 Look-up tables

The first technique is called `T-box`. It combines `ShiftRow` and `AddRoundKey` into a series of 16 lookup tables, with one table for each byte of the state. In each round, a different T-box table is used.

Next, we will create the `Tyi` tables that map the `T-box` output to the `MixColumns` computation. Since `MixColumns` updates the state four bytes at a time, we will have a table for each four bytes.

To implement this using lookup tables, we need to decompose matrix and vector multiplication into an exclusive-or of values as follow :

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \left(x_0 \oplus \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} x_1 \oplus \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} x_2 \oplus \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} x_3 \oplus \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix} \right)$$

Thus we will have four lookup table, each one will be the computation of one of the vector lines and the matrix column it will map 8 bit to a 32 bit . Therefor each round 1 to 9 will have a 4 `Tyi` tables composed of `Ty0 Ty1 Ty2 Ty3` lookup tables.

Finally, we will have XOR tables. These tables are used to perform XOR computations for the four look-up tables in the `Tyi` tables.
Each table takes two 4-bit inputs and produces a 4-bit output. Since we want to XOR 32-bit values, we will need 8 (32 XOR 32 bits) * 3 (number of XOR operations in the `Tyi` tables) * 4 (number of tables) for each round.

Currently, we have no protection against key extraction attacks (see section 4.1 [3]) and therefore, we will use internal encoding

2.3 Internal encoding

We use internal encoding to randomize the tables. We have two types of encoding: random bijection and mixing bijection. The first one is for confusion between the key and the tables, and the second one is for the diffusion between the input and output of the tables.

2.3.1 Random Bijection

The goal is to make each lookup table statistically independent from the key. This makes it harder for an attacker to perform a statistical attack. To achieve this, we will apply an encoding to each lookup table. From T we get a new table T' using two random bijection f,g.

$$T' = g \circ T \circ f^{-1}$$

For implementation reasons the encoding of a the lookup table will be split in nibbles and should cancel each other if there output is feed as an input. see section 4.1 [3]

2.3.2 Mixing bijections

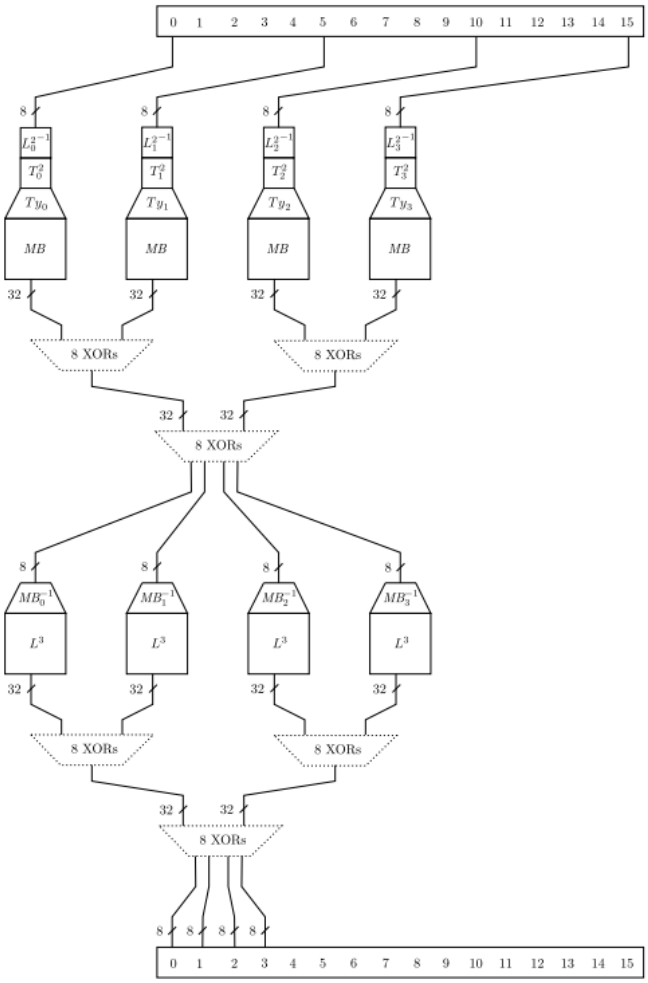
The encoding should be linear, so due to commutativity, we can perform XOR operations. We will have two types L and MB, applied in interior rounds (i.e., rounds 2 to 9).The encodings are invertible matrices over `GF(2)`.

For L at the beginning of each round, we will apply the decoding operation of the current round on the state. At the end of the round, we apply the encoding of the next round.

For each interior rounds will generate a random matrix and its inverse and store them in a tuple. To encode the lookup table, we multiply each input by the matrix. To decode, we multiply each encoded input by the matrix's inverse.

Thus, we have encoded our lookup tables for interior rounds. The lookup tables for the exterior rounds (i.e., round 1 and 10) will not be encoded to let to an implementation of external encoding see section 4.3 [3].

As for L. For MB we generate a tuple of matrices and their inverse for each `Tyi` table, and then apply it as previously described. At the output of the XOR operation, we apply the inverse. Because the output is 32-bit, we use the linearity property to correctly apply it. see section 4.2 [3]



This is representation of mixing bijection network for round 2

3 Side channel

This section describes what the side channel is in the context of a white-box and defines the DCA attack in this context.

3.1 Side Channel Attacks in White-Box Cryptography

In cryptography, a side-channel attack is a method that exploits vulnerabilities in the implementation of a cryptographic algorithm, rather than weaknesses in the algorithm itself. These vulnerabilities can originate from either software or hardware flaws. The attacker's objective is to exploit meta-information leaked by the implementation (such as calculation time, current consumption, electromagnetic emission, and so on) in order to make the cryptographic implementation vulnerable to attack.

3.2 Differential Computation Analysis

First of all it is necessary to know that the DCA (Differential computation Analysis) attack is a software version of the DPA (Differential Power Analysis) attack [4], where instead of measuring the current consumption, it measures the memory calls (which is the logical result of the current consumption). It was first describe by Bos et al [5].

So what is DCA? First of all, we suppose to have access to the implementation of a known cryptographic algorithm (here, WB-AES-128). From there, $I(P_i, k)$ designates an intermediate state of the algorithm where P_i designates an input plaintext and k a fraction of the secret key (in this case we will use a byte of this key). If we assume that this intermediate state depends on the memory accesses with in addition some random noise, we can imagine the situation by this formula:

$$L(I(P_i, k)) + y$$

where $L(I(P_i, k))$ is the trace of the memory calls during the intermediate state and y is the random noise.
We admit this noise null because by repeating this situation a great number of times, by average, noise will become nullified because it is random.

As an attacker we have access to three values P_i which is a known random plaintext, Ci which

is the encrypted result of P_i after its passage through the implementation and Ti which is the set of memory calls during the passage of P_i through the implementation (so $[L(I(P_i, k)) + y]$ is only one of Ti). Our goal is to find the right value of k (which is our only unknown) by testing all these possible values and by comparing them with Ti.

The idea of DCA is to analyze all these Ti traces, using Hamming weights. For that we must already divide all our possible intermediate values in two groups (that we will name A_0 and A_1) according to a defined property, in our case we use the value of the first bit of the hypothesis of k in the intermediate state (cf. $I(P_i, k)$). This gives us a set A of 256 intermediate values (i.e. 256 hypothesis of k, because our k is a byte $(2 = 256)$), the whole separated in two distinct subsets A_0 and A_1. If our assumption of k is correct then the difference between the means of A_0 and and the mean of A_1 will converge to a value. On the other hand, if the assumption of k is incorrect, then the data located in the two sets will be as random and thus the average of the two subsets will converge to 0. This method allows us to find k as long as we have a sufficient number of traces (and therefore a sufficient number of known P_i).

Once our first k is found, we reiterate on the next part of the secret key that we have to find (the next byte) and so on until we have our complete key.

4 Gathering Traces

In this section, we will describe which tools and how we proceeded to obtain our traces in order to perform our statistical analysis.

To put into practice the DCA attack, we must already have an environment, allowing us to emulate an architecture, and, also allowing us to recover as easily as possible, the traces resulting from the memory calls during the execution of our implementation. Several candidates exist (Unicorn, Qiling, ...) but for our project we decided to go for Pin tool.

Pin is a tool developed by Intel that allows to emulate the behavior of an Intel processor. Thanks to its API Pin allows among other things to retrieve the content of registers and memory at any time. For the sake of time and simplicity we used an open source software suite under free license named Tracer.

Tracer is a github project [6] composed of 3 tools using the Pin API tool from Intel. TracerPIN is a tool to generate execution traces of a running process. When we run the AES with TracerPIN, it will return a log file in either text or sqlite format. The logged information is listed in different types:

- `[*]` Arguments
- `[-]` Information on base image and libraries
- `[!]` Information on filtered elements
- `[T]` Thread event
- `[B]` Basic block
- `[C]` Function call
- `[R]` Memory read operation
- `[I]` Instruction execution
- `[W]` Memory write operation

This tool also has more advanced options allowing us to log only certain address ranges.

TracerGrind is a valgrind plugin that behaves similarly to TracePIN, however we will not use it for our project.

TraceGraph is a GUI for visualizing execution traces produced by TracerPin and TracerGrind.

4.1 Analyse trace and finding pattern

Once our aes execution trace is generated with TracePin. We will focus on the intermediate state where the input value is XORed with the secret key, that is the "AddRoundKey" step. But how to recover this portion of the trace? For that we will have to define a repetitive pattern that allows us to identify the "AddRoundKey" step from the TracePin logs. We start in a visual way thanks to trace graph. Then we refine by searching among the memory calls for address ranges that are called several times (because the key is composed of several bytes, and the operations will be performed in the same way for each of them).
If we take the example of the aes classic (not white-box) our trace will look like this:

[See Anexe 1]

We can see the different steps related to the aes (AddRoundKey[1], ShiftRow[2], SubByte[3] and MixColumn[4]) in the different rounds (10 for the aes-128), the 10 round is not visible because the MixColumn is not applyed:

[See Anexe 2]
[See Anexe 3]

As for our intermediate state we focus only on the first round of the "AddRoundKey" step, we keep this:

[See Anexe 4]

We can see at first that a certain pattern is repeated 16 times, which corresponds to the number of bytes composing the secret key (128/8 = 16), so we can deduce that each entry of this pattern corresponds to the AddRoundKey with one of the bytes of the secret key. This is confirmed when we look at the end of each operation where we can see an XOR between two elements.

[See Anexe 5]

Looking at the AddRoundKey step but this time from the Stack side, we can then see the byte of the secret key that is used for the XOR.

This example is a simple case because it is a classic aes encryption, but in the context of a white box, things are slightly different. In our white-box implementation we use T-boxes which will combine the ShiftRow step with the AddRoundKey step which will create 4 Tyi tables (with 4 bytes) and then do the MixColumns step and this for rounds 1 to 9. The value is not directly readable anymore.
When we look at the execution trace of the aes in white box we get this :

[See Anexe 6]

We can see our stack on the right and our instructions on the left. In the instruction part we can see 36 "stairs" which correspond well to the operations on the 4 Tyi of each round (4*9 = 36).
If we focus on the first round and we are interested in these "stairs". We can see the steps where the 4 bytes come out of the T-box [1] followed by the XOR's before encapsulation with invert of MB an $L2$ [2], invert of MB and $L1+1$ [3], XOR after encapsulation with invert of MB an $L2$ [4], and ended with the assignment of these 4 bytes [5] ([See Anexe 7]).

5 Practical Attack

This section describes the different steps to attack the first key byte using the algorithm and its improvements from the paper [7].

The first step is to collect traces. Next, set the leak model as the selection function and apply it to the traces to identify which bit leaks the most in the model. The principle of DCA involves separating the traces into two sets: one where the encoded result of the selection func-

tion resides and one where it does not. If there is a correlation between the selection function and the traces, then the two sets can be distinguished. Finally, select the best hypothesized key based on this information.

5.1 Step 1

In the first step, we will describe what a trace is. As explained earlier, we used `TracerPin` to record memory access of the program and identify the pattern of the white box. Specifically, we focused on the output of the look-up tables. From those patterns, we recorded interesting information. We gather interesting information in a trace composed of a serialized address read and its value.

5.2 Step 2

Once we generate the set of traces
$\{\ t_e \mid 0 \leqslant e \leqslant 255\}$ from the set of plain text p_e composed $0 \leqslant e \leqslant 255$ and fixed key byte k.

We need to model the leak. First, we examine what is found in trace t_a from plaintext p_a. Based on the contents of the `Tyi` tables, the output is the result of `AddRoundKey`, `SubBytes`. From this we deduce that the value $\mathrm{SBox}(p_a \oplus \mathrm{k})$ should be present encoded in trace t_a.

Next, we will use a selection function to determine whether the value should appear in the trace or not. The decision will depend on the bit position and the key hypothesis.

$$\mathrm{Sel}(pe, kh, j) := \mathrm{SBox}(pe \oplus kh)[j] = b \in \{0, 1\}$$

5.3 Step 3

We will sort the trace to identify which bit leaks the most. Thus we need to repeat steps 3, 4, and 5 for each $0 \leqslant j \leqslant 7$.

Once we have this selection function, we will apply it to the set of traces T for a given key hypothesis kh and for a fixed bit j. Thus, we obtain two sets of traces, A_0 and A_1, as follows.

$$For\ b \in \{0, 1\},$$
$$A_b := \{te | 0 \leq e \leq 255, \mathrm{Sel}(pe, kh, j) = b\}.$$

5.4 Step 4

To get rid of the noise in traces . We compute the mean trace of each set A_0, A_1.

$$\text{For } b \in \{0, 1\},\ \bar{A}_b := \tfrac{1}{|A_b|} \textstyle\sum_{t \in A_b} t$$

5.5 Step 5

In this step we produce the difference of the two set mean trace . It will output if the two set are differentiable.

$$\Delta = |\overline{A}_0 - \overline{A}_1|.$$

5.6 Step 6

We now compare the difference of means traces obtained Δ_j for all target bits j for a given key hypothesis kh. We choose the bit j_m to be the one who lead to the highest peak in the difference of the mean's. And set key hypothesis kh score to this peak height as $H(\Delta_h) := H(\Delta_j)$. We will see how we use it in step 7.

$$\forall 1 \leq j \leq 8, H(\Delta_j) \leq H(\Delta_{jm})$$

5.7 Step 7

To have the best key byte hypothesis we compute a score for each one $0 \leq kh \leq 255$. The score as we saw in the previous step is the height of the highest peak in difference of mean's.

$$\forall 0 \leq h_0 \leq 255, H(\Delta_{h0}) \leq H(\Delta_h)$$

In unprotected implementation we would choose the best key byte hypothesis according to the highest score. In our case we applied several encoding thus we need to make some modification in the steps. The paper [1] describe the

effect of the encoding previously specify here []. Three theorem can outline those effects .

Theorem 5.1. *Given a T-box encoded via an invertible matrix A. The DCA attack returns a difference of means value equal to 1 for the correct key guess if and only if the matrix A has at least one row i with Hamming weight (HW) = 1. Otherwise, the DCA attack returns a difference of means value equal to 0 for the correct key guess.*

Theorem 5.2. *Given an OT-box which makes use of nibble encodings, the difference of means curve obtained for the correct key hypothesis k h consists only of values equal to 0, 0.25, 0.5, 0.75 or 1.*

Theorem 5.3. *Given an lOT-box which makes use of a matrix A and nibble encodings, the difference of means curve obtained for the correct key hypothesis k_h consists only of values equal to* $0, 0.25, 0.5, 0.75$ *or* 1*, if A has at least one submatrix Sub_d with one column $*a_j$, such that* $\text{rank}[Sub_d \ * a_j] = 3$.

We will briefly review of the three theorems: the first one describes the effect of mixing bijections on a DCA attack, the second one explains the effect of random bijections, and the third one analyzes the effect of both internal encoding.

5.8 Improvements

The theorems provide valuable insights that can be used to modify step 6 as follows.

- If $H(\Delta_j) > 0.3$, then we set the score of kh to $H(\Delta_j)$.
- If $0.2 \leq H(\Delta_j) \leq 0.3$, then we set the score of kh to $H(\Delta_{j0})$ where Δ_{j0} is the trace with it's peak closest to 0.25.
- Otherwise if $H(\Delta_j) < 0.2$, then we set the score of kh to $H(\Delta_{j0})$ where Δ_{j0} is the trace with it's peak closest to 0.

For step 7 we will do the same modification, if no key candidate stands out, we look for a candidate with the values closest to 0.25 or 0. Else we select the one with the highest score.

6 Results

With the follows modification we are able to retrieve 94% of the key bytes . The reason for the 6% is the convergence error of $H(\Delta_{j0})$ to 0 or 0.25.
We can improve it by making a ranking key algorithm that will test the different possibility of convergence if $H(\Delta_j)$ is near 0.2.

7 Conclusion

Although the white-box model is a good idea, there is no long-term defense against attacks on white-box implementations. However, as we have just shown, white-box implementations implementations do not offer security even in the short term since the DCA attack we have presented can extract the secret keys in a few seconds. in a few minutes. An evolution track can be to try to obfuscate the patterns (masking usage of look-up table, etc..), and on the attacker side using more advance machine learning model can be interesting in the case where the implementation of the white box and its look-up tables network are unknown. Detecting those patterns would be time-saving.

References

[1] J. Khelif, "wb aes-128 rust," https://gitlab.istic.univ-rennes1.fr/jkhelif/white_box_chow, 2023, accessed: 2023-01-20.

[2] G. L. D. Jawed Khelif, "Sca on wb aes-128," https://gitlab.istic.univ-rennes1.fr/jkhelif/break_white_box, 2023, accessed: 2023-01-30.

[3] J. A. Muir, "A tutorial on white-box aes," 2013. [Online]. Available: https://eprint.iacr.org/2013/104.pdf

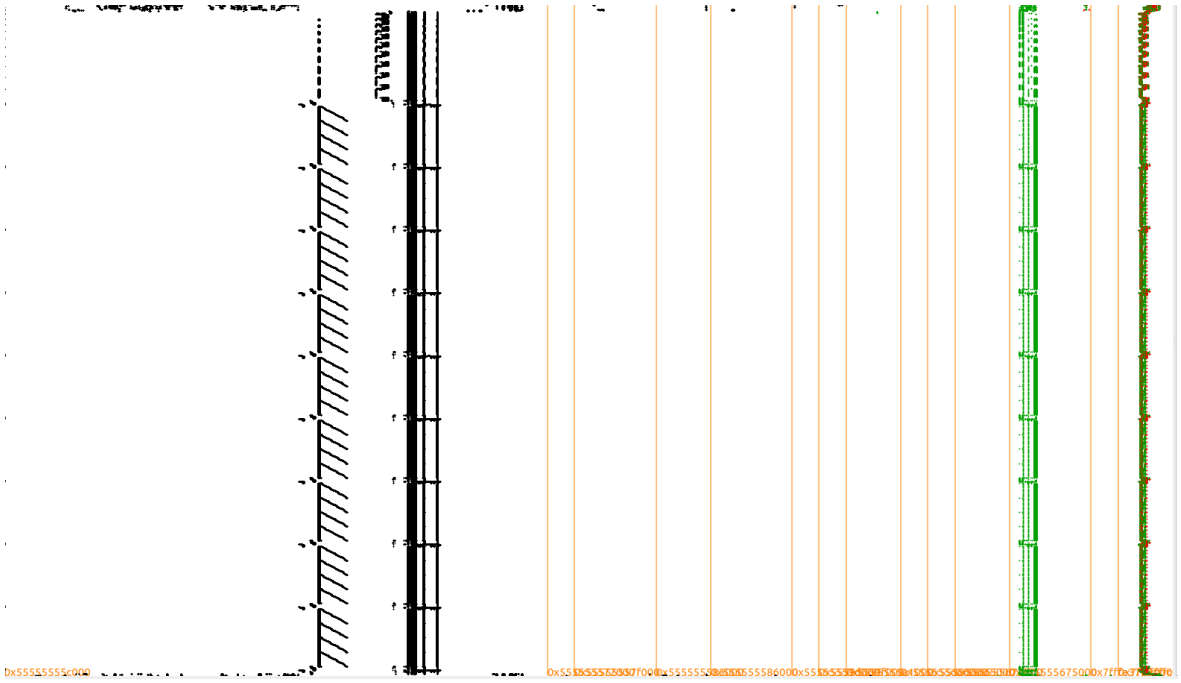
[4] yan1x0s. (2021) Side channel attacks — part 2 (dpa cpa applied on aes attack). [Online]. Available: https://yan1x0s.medium.com/side-channel-attacks-part-2-dpa-cpa-applied-on-aes-attack-66baa356f03f

[5] W. M. Joppe W. Bos, Charles Hubain and P. Teuwen, "A tutorial on white-box aes," 2015. [Online]. Available: https://eprint.iacr.org/2015/753.pdf

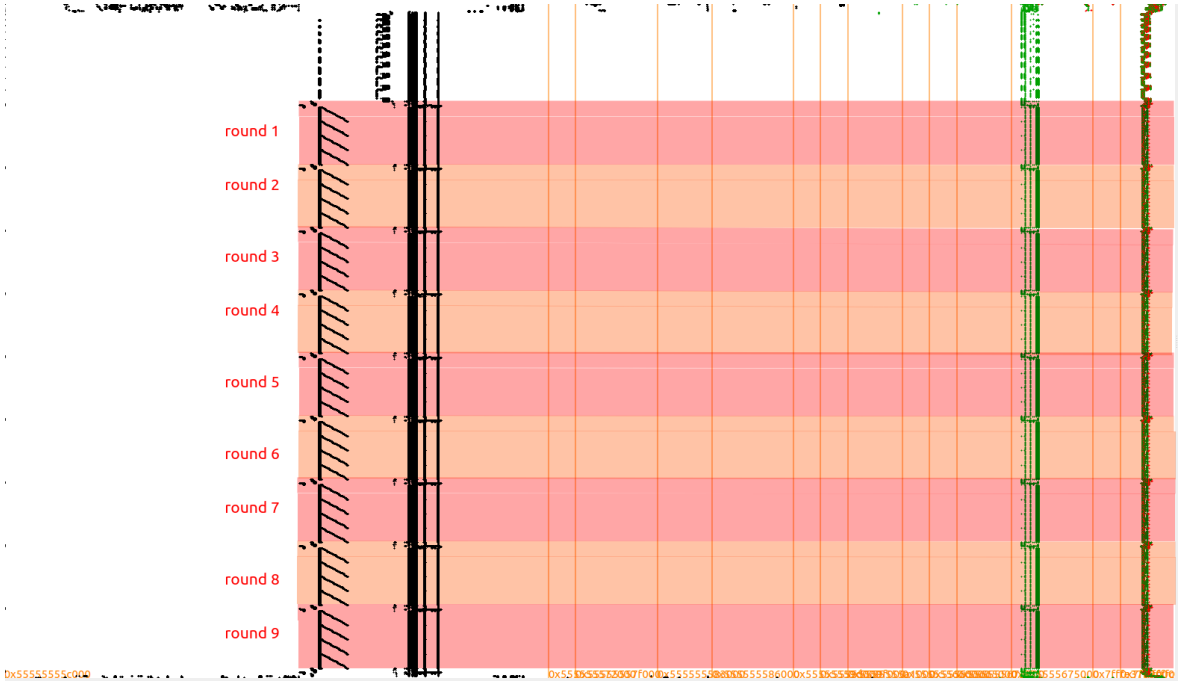
[6] P. Teuwen, "Tracer," https://github.com/SideChannelMarvels/Tracer, 2022, accessed: 2023-02-15.

[7] W. M. Estuardo Alpirez Bock, Chris Brzuska and A. Treff, "On the ineffectiveness of internal encodings - revisiting the dca attack on white-box cryptography," 2018. [Online]. Available: https://eprint.iacr.org/2018/301.pdf

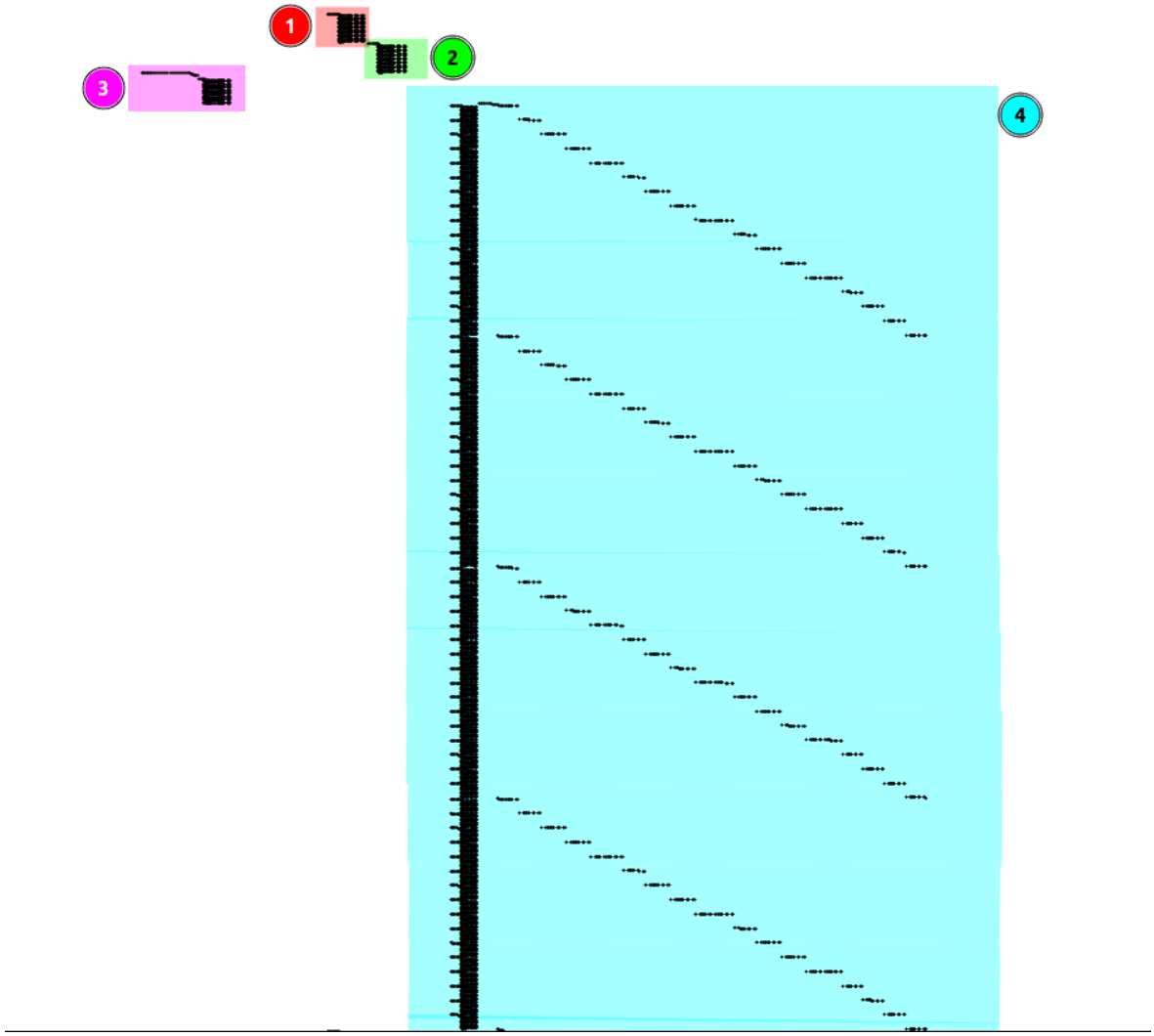
8 Annexes



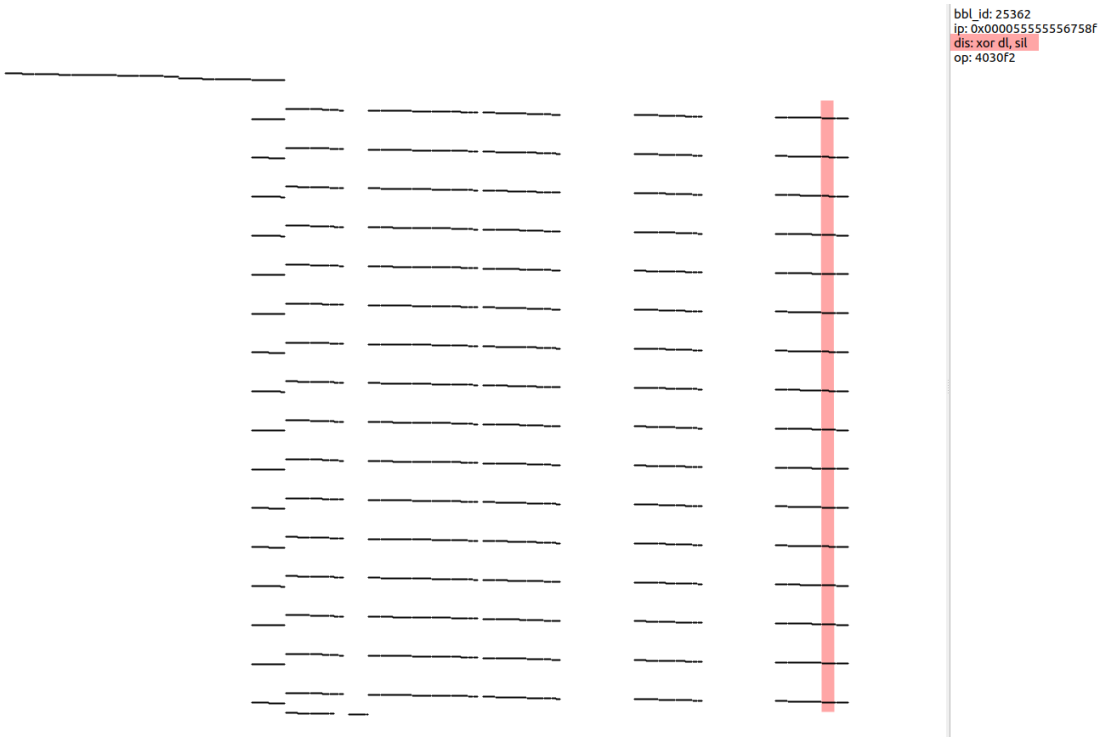
Annexe 1


round 1
round 2
round 3
round 4
round 5
round 6
round 7
round 8
round 9

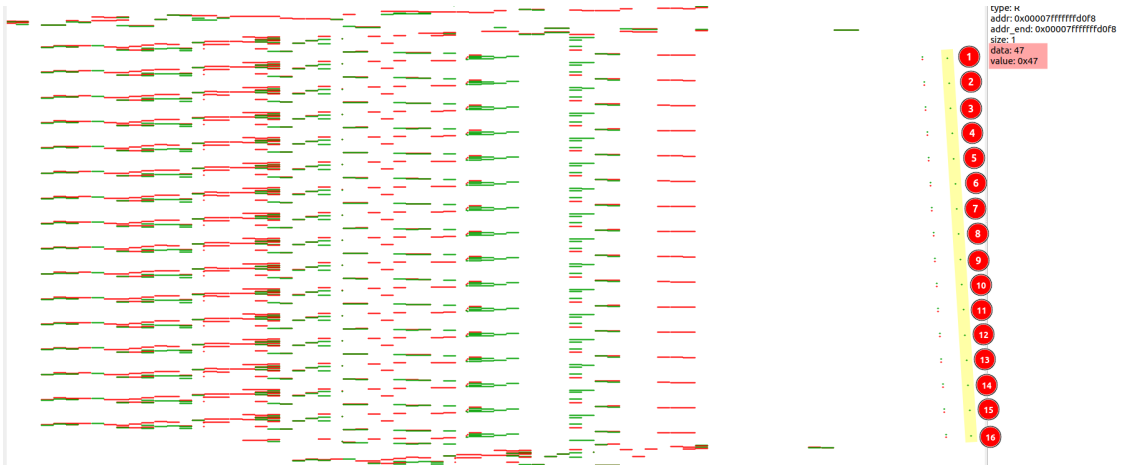
Annexe 2



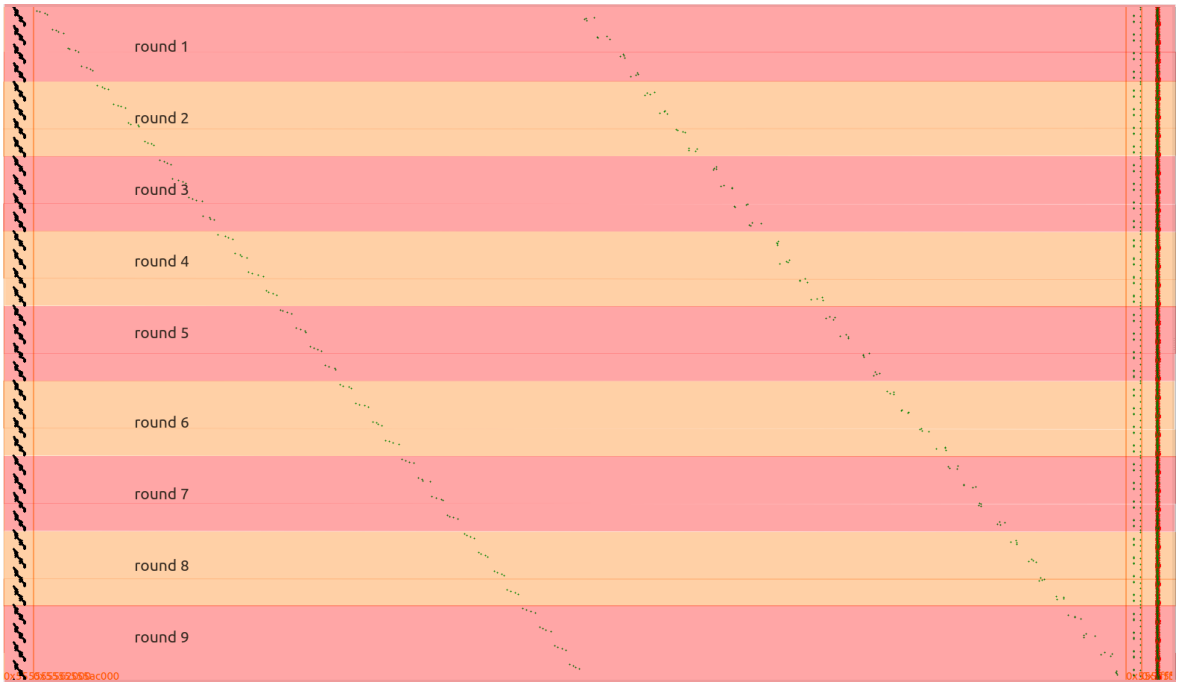
Annexe 3



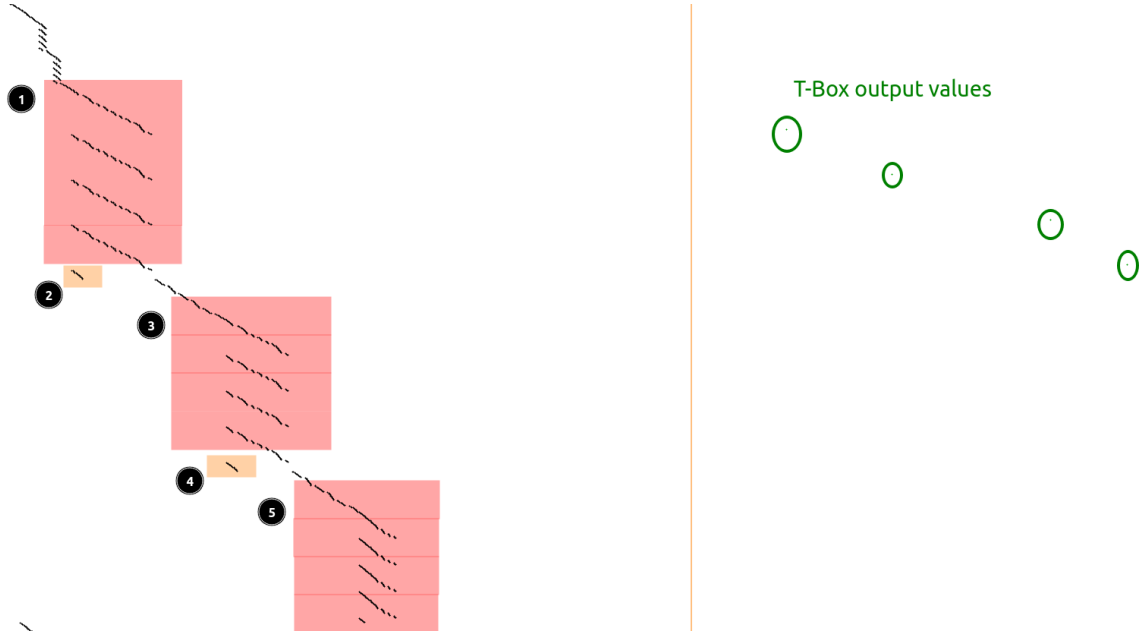
Annexe 4



Annexe 5



Annexe 6



Annexe 7