VERISICC
Deliverable L4.3
Evaluation Report

CALL: FUI25

NAME OF THIS PROJECT: VERISICC

Leader of this deliverable

- Partner: NinjaLab
- Contact name: Thomas Roche
- Contact information: thomas@ninjalab.io

Leader of the project

- Company: CryptoExperts
- Contact name: Sonia Belaïd
- Contact information: sonia.belaid@cryptoexperts.com, 06 68 75 30 66

Partners

- SMEs: CryptoExperts and NinjaLab
- Big Business: IDEMIA
- Public Institutions: INRIA, ANSSI, and Université du Luxembourg

Version : 0.1

Table of contents

1 Introduction **3**

2 Device Under Test **4**
2.1 Product Presentation 4
2.2 Side-Channel Setup 4
2.3 Trigger Mechanism 5

3 Implementations **7**
3.1 Internal Pseudo-Random Generator 8
3.2 Unprotected PRESENT Sbox 8
3.3 2-Share PRESENT Sbox 8
3.4 3-Share PRESENT Sbox 9

4 Unprotected PRESENT Sbox **11**
4.1 First Observations 12
4.2 Leakage Analysis 15

5 2-Share PRESENT Sbox **18**
5.1 Acquisition Campaign 18
5.2 Leakage Analysis 21
5.3 A Sensitive Leakage 27

6 3-Share PRESENT Sbox **32**
6.1 Acquisition Campaign 32
6.2 First-Order Leakage Analysis 35
6.3 Second-Order Leakage Analysis 41

1. Introduction

The deliverable L4.3 is the output of VERISICC task T4.3 focusing on the side-channel evaluation of formally proved masked implementations in practice. To this end, the study considers a specific chip and masked implementations provided by VERISICC project.

Masking schemes are proven secure in generic and idealistic models, this approach has the benefit to avoid designing masking schemes for each and every chip that might need to run secure cryptography. The downside is twofold: the idealistic models might not capture all real leakages (and then the masking scheme security proofs are worthless) or they might consider leakages that are not present in practice (and then add unnecessary constraints on the scheme, making it more costly than really needed).

Our goal in this study is precisely to estimate the distance between the classical idealistic models in which the masked implementation are proven and reality. In a previous task (see deliverable L3.3, the output of VERISICC task T3.3 focusing on the side-channel characterization of a chip) we showed that a chip can exhibit leakage functions much more complex than expected and then might not be properly captured by the model. In the present study, we will take another approach: we will consider masked implementations that enjoy a security proof and look for sensitive leakages on our selected chip.

The study will use the same target chip and side-channel setup than for the characterization task (T3.3, deliverable L3.3) except for the choice of a more stable external clock. All details are recalled in Section 2.

The PRESENT Sbox has been chosen as target cryptographic primitive for this study, three implementations are considered, they are detailed in Section 3 and the source code can be found in the study material provided with this document [1]. The first implementation is unprotected, it will set the reference for our study. The second and third implementations (respectively 2-share and 3-share masking schemes) result from the work of Barthe *et al.* [BGG+21].

The main results and conclusions are summarized below:

- The acquisition chain is setup for highly precise power consumption execution traces. It results in a moderate speed acquisition process and a large quantity of data to analyze.
- As expected from a similar setup, the side-channel traces show coherent leakage with the chip characterization results (T3.3, deliverable L3.3).
- The 2-share PRESENT Sbox implementation shows to leak sensitive information (*i.e.* first-order leakage of unmasked data) on the *OpenCard*.
- The root of the unintended leakages is tracked down with the help of the asm code together with a fine-grained timing analysis of the execution traces. However, without further experiments and/or a precise knowledge of the *OpenCard* MCU design, it is barely possible to properly extend the leakage model with this new observation (although patching the leakage for this specific case seems completely doable).
- Finally, an extensive leakage assessment of the 3-share implementation with respect to 1st- and 2nd-order leakages shows no sensitive leakages. Hence, for this implementation, the leakage model used by the developers seems well adapted, or – in other words – this implementation does not fall (by chance) into the unknown leakage characteristics of the *OpenCard*.

[1] `L4.3_evaluation_sources.tgz`

2. Device Under Test

2.1. Product Presentation

The device (so called *OpenCard* or simply DUT in the following) we rely on is a 0.13um 32-bit Contact Smartcard IC developed by Beijing ChipCity Technology Co., Ltd. [Bei]. The IC has been EAL4+ certified in Asia (out of the European SOG-IS scheme). It features an ARM core SC 100 with 18 KB of RAM, 8 KB of ROM and 548 KB of FLASH. Figure 1 gives an overview of the IC die and of the main blocks.

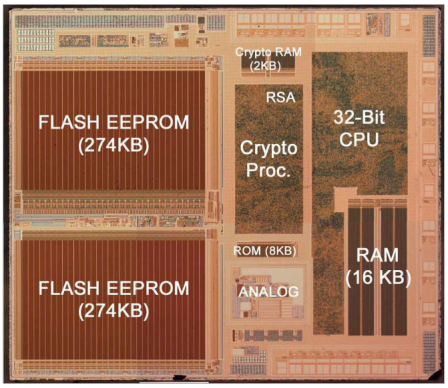

Figure 1: General view of the die after sample opening; mag 50X

The code for this characterization task is compiled from a Keil simulator using as a target an ARM7TDMI-S before being uploaded on the *OpenCard*. The third level of optimization is activated together with the pre-processing option `cpreproc` and the `interwork` qualifier to get ARM/thumb interworking support.

The *OpenCard* offers ISO7816 communication interface and can run on either internal (several clock frequency are available 3.5/7/9/14/28MHz) or external clock. The clock is configured through the *System Control Register* SFR/SCSYS [Bei, 4.2.2], the external clock configuration is chosen for the study to avoid the internal clock jitter. Indeed during the previous characterization task – T3.3, see deliverable L3.3 – the internal clock of the smartcard showed a small jitter that required side-channel trace resynchronization. The external clock is set to 1MHz.

2.2. Side-Channel Setup

The NinjaLab acquisition chain is detailed below

- DELL Precision Tower 3420 desktop computer equipped with a 3.6GHz Intel Core i7-7700 processor, 32GB of RAM and a 2TB hard disk drive;
- Pico Technology PicoScope 6424E oscilloscope, with a 500MHz frequency bandwidth, sampling rate up to 5GSa/s, 4 channels and a channel shared memory of 4G samples [Pic19];
- Scaffold board [Led]

The Scaffold board, developed by Ledger, is made for security research on embedded devices. It allows to easily communicate with a smartcard (thanks to its smartcard kit[2] and its ISO7816 module[3]) and offers a precise measure of the DUT power consumption. The Scaffold setup is depicted on Figure 2. To simplify our analysis we will focus on power consumption ignoring Electromagnetic Radiations (EM for short). Indeed EM measurements add a degree of freedom, the EM probe position, and substantially increase the effort for characterization (already enormous as we will see in the following). Hence, for all the study, the reader should not forget that our conclusions only hold for power consumption and not all side-channels that an adversary might eavesdrop on the *OpenCard*.

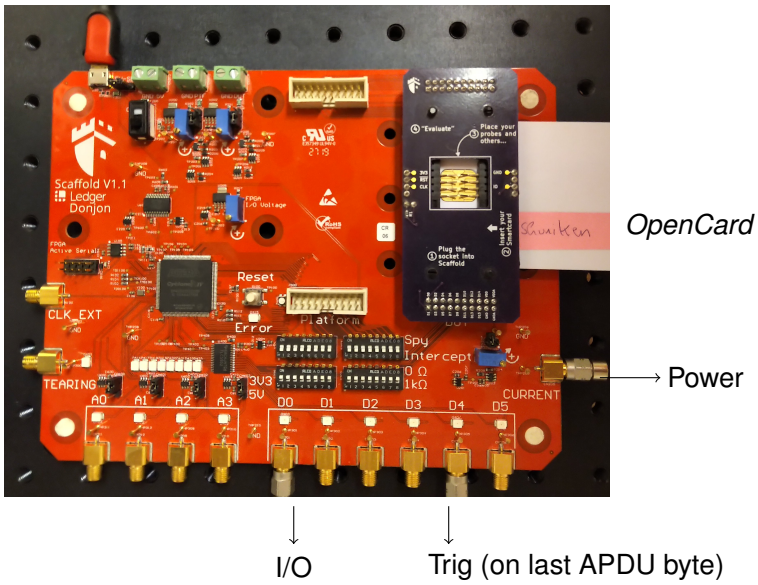

Figure 2: General View of the Scaffold Setup

The Scaffold board possesses a simple way to set a trigger signal at the beginning of a specific APDU transmission. However, our goal here is mainly to observe the side-channel leakage of short execution runs which makes the Scaffold trigger functionality useless:

- the trigger will be far from what we need to observe (with respect to the target observation length);
- each and every target execution on the card should be launched by an APDU command (to set the trigger signal). It will greatly impact the acquisition time as the ISO7816 communication protocol is pretty slow.

To avoid the above mentioned issues, we developed a custom trigger signal.

2.3. Trigger Mechanism

From the *OpenCard* datasheet [Bei], the smartcard I/O PIN can be dynamically re-configured as a GPIO. The idea is then to re-program the I/O PIN as a GPIO during the program execution (potentially executing many target instructions sequences) and send trigger signal before and after each target execution. Then, program the I/O PIN back to the ISO7816 communication engine to send the

[2] https://donjonscaffold.readthedocs.io/en/latest/kit_smartcard.html
[3] https://donjonscaffold.readthedocs.io/en/latest/iso7816_module.html

program return value and exit code.

Here is the detailed procedure:

- at card initialization:
  - set the 18th bit of register SCCM0 to 1, leave the other bits unchanged (GPIO ck enable, see [Bei, 4.2.1])
  - set register GPIODIR1 to 0xFFFF00FF (GPIO mode "out", see [Bei, 10.2.4])
- when receiving an APDU command:
  - after the APDU is fully received and before processing: set register SCGCON to 0x00000400 (GPIO configuration of the I/O PIN, see [Bei, 4.2.12])
  - after processing and before the result APDU is sent: set register SCGCON to 0x00000000 (ISO7816 I/O configuration of the I/O PIN, see [Bei, 4.2.12])
- to send a trig signal:
  - before and after the target execution: set register GPIODAT1 to 0xFFFF00FF (drop I/O PIN to low level, see [Bei, 10.2.3]), then set register GPIODAT1 back to 0xFFFFFFFF (raise I/O PIN back to high level)

Register	address	desc
SCCM0	0x0F0000	Clock Management Register 0
GPIODIR1	0x0F8C0C	GPIO P4 P5 direction control register
SCGCON	0x0F0040	GPIO Enable Register
GPIODAT1	0x0F8C08	GPIO P4 P5 data register

The following asm function is used to indicate the beginning or the end of a target execution, the C snippet of code is an example of the trigger mechanism usage, the target execution is a bitsliced unprotected PRESENT Sbox computation (full code can be found in the supplementary material provided with this document [4]).

```
;; GPIO P4~P5 data register
#define GPIODAT1 0xF8C08
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Function TRIG
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
TRIG
  TRIG00 r3
  TRIGFF r3
  bx lr
```

```
unsigned long src_bs[4], dst_bs[4];
unsigned long src[2], dst[2];

// src is filled with fresh randoms
[...]

// from 2 32-bit values
// to   4 16-bit bitsliced values
toBitslice (src_bs, src);

TRIG();
// execute 16 PRESENT Sbox calls
presentUnprotected (dst_bs, src_bs);
TRIG();

fromBitslice (dst, dst_bs);
```

[4] L4.3_evaluation_sources.tgz

Where `TRIG00` (resp. `TRIGFF`) macro forces the I/O line to drop from its default high level to low (resp. from low to high).

Figure 3 depicts the side-channel acquisition of the bitsliced unprotected PRESENT Sbox computation. The blue signal is the I/O line while the red signal is the power consumption of the DUT.

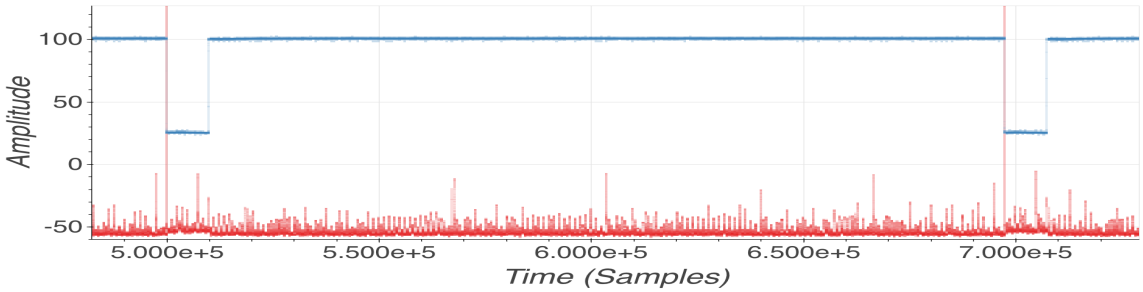

Figure 3: Unprotected PRESENT Sbox – I/O (blue) and Power (red) Trace

3. Implementations

The PRESENT Sbox has been chosen as target cryptographic primitive for this study, three implementations are considered. The asm code can be found in the study material provided with this document [5].

The first implementation is unprotected, it will set the reference for our study. The second and third implementations result from the work of Barthe *et al.* [BGG+21].

A call to these primitives on the smartcard is wrapped inside a series of APDU commands:

- Three APDU commands allow to set the three 4-Byte seeds of the xorshift96 PRNG (see below Section 3.1);
- An APDU command selects the PRESENT Sbox implementation;
- An APDU command sets the number of executions of the PRESENT Sbox:
- An APDU command launches the run of the selected PRESENT Sbox for the chosen number of executions. Between two executions, the inputs and an entropy buffer are refreshed with randoms (from successive PRNG calls).

All three PRESENT Sbox implementations are bitsliced and take 64-bit inputs stored as 4 16-bit bitsliced inputs. The 4 bitsliced inputs are stored in 32-bit `unsigned long` variables (denoted $\{A, B, C, D\}$) where only the 16 least significant bits are used. For each bit position $0 \leq i < 16$, the 4 bits at position i (denoted (A_i, B_i, C_i, D_i)) constitute a PRESENT Sbox input.

The masked implementations work on shared input variables (2 or 3 shares), their variables (*e.g.* any of their inputs $\{A, B, C, D\}$) then represent arrays (of length 2 or 3) of `unsigned long` variables. The jth share of variable X is then referred to as $X[j]$ and the unmasked value X is the exclusive-or

[5] `L4.3_evaluation_sources.tgz`

of its shares. Also, the masked implementations require fresh randoms that are used internally to ensure probing security. Before each call to these implementations, all necessary randoms are generated from successive PRNG calls and stored in the `entropy` buffer.

In all the following, $\oplus$ will denote the exclusive-or operation between two `unsigned long` variables while $\&$ will denote the bitwise-and operation.

3.1. Internal Pseudo-Random Generator

Pseudo-random values are generated with Marsaglia's `xorshift96` pseudo-random generator [Mar03], initialized with a 12-Byte seed.

When randoms are needed, a call to the macro `GET_RAND` iterates the pseudo-random generator and extracts 32 bits from its internal state.

3.2. Unprotected PRESENT Sbox

The bitsliced unprotected implementation of the PRESENT Sbox is a straightforward C implementation. Computing the 16 Sbox outputs (as 4 `unsigned long` variables denoted $\{E, F, G, H\}$)) is done as follows:

$$\begin{aligned}
E &\leftarrow B\&C \oplus A\&B\&D \oplus A\&C\&D \oplus B\&C\&D \oplus A \oplus C \oplus D \oplus 0xFFFFFFFF\\
F &\leftarrow A\&C \oplus A\&D \oplus C\&D \oplus A\&B\&D \oplus A\&C\&D \oplus A \oplus B \oplus 0xFFFFFFFF\\
G &\leftarrow A\&B \oplus A\&C \oplus A\&B\&D \oplus A\&C\&D \oplus B\&C\&D \oplus A \oplus C\\
E &\leftarrow B\&C \oplus A \oplus B \oplus D
\end{aligned}$$

The operation order does not matter here and then we let the C compiler do its work.

3.3. 2-Share PRESENT Sbox

The 2-share version of the PRESENT Sbox is an asm implementation borrowed from 2021 Barthe *et al.*'s paper [BGG+21]. This implementation uses a decomposition into three subfunctions of the PRESENT Sbox, as follows:

$$Sbox : \mathsf{A} \circ \mathsf{G} \circ \mathsf{G} \circ \mathsf{B},$$

where $\mathsf{A}, \mathsf{B}, \mathsf{G}$ are vectorial Boolean functions from $\{0,1\}^4$ into itself and A, B are affine while G is quadratic. The masking scheme requires 28 fresh random bytes (in addition to the 8-Byte random that is needed to share the 4 16-bit input variables $\{A, B, C, D\}$).

All in all, between two calls to the 2-share PRESENT Sbox implementation, 48 random bytes are extracted thanks to 12 calls to the PRNG:

- 8 bytes to generate new inputs $\{A, B, C, D\}$;
- 8 bytes to share the inputs $\{A, B, C, D\} \rightarrow \{(A[0], A[1]), (B[0], B[1]), (C[0], C[1]), (D[0], D[1])\}$;
- 32 bytes for the probing security, stored in a 8 length array R of `unsigned long` variables (the first 4 bytes, *i.e.* the first cell of R, are not used). Moreover,
  - a 32-bit random is consumed in B;
  - three 32-bit randoms are consumed for each call to G;

For reasons of completeness, and because the asm implementation might not be trivial to understand, we unfold here the implementations of the A, B, G subroutines. Note that the computation order of the successive operations is here crucial to assess 1st order security.

2-share implementation of A

$$
\begin{aligned}
G[0] &\leftarrow A[0] \\
E[0] &\leftarrow C[0] \oplus A[0] \\
H[0] &\leftarrow C[0] \oplus A[0] \oplus 0xFFFFFFFF \oplus D[0] \\
F[0] &\leftarrow B[0] \oplus 0xFFFFFFFF
\end{aligned}
$$

$$
\begin{aligned}
G[1] &\leftarrow A[1] \\
E[1] &\leftarrow C[1] \oplus A[1] \\
H[1] &\leftarrow C[1] \oplus A[1] \oplus D[1] \\
F[1] &\leftarrow B[1]
\end{aligned}
$$

2-share implementation of B

$$
\begin{aligned}
H[0] &\leftarrow D[0] \oplus B[0] \oplus 0xFFFFFFFF \\
E[0] &\leftarrow A[0] \oplus B[0] \\
F[0] &\leftarrow B[0] \oplus C[0] \oplus R[1] \\
G[0] &\leftarrow C[0]
\end{aligned}
$$

$$
\begin{aligned}
H[1] &\leftarrow D[1] \oplus B[1] \\
E[1] &\leftarrow A[1] \oplus B[1] \\
F[1] &\leftarrow B[1] \oplus C[1] \oplus R[1] \\
G[1] &\leftarrow C[1]
\end{aligned}
$$

$$
R \leftarrow R + 4
$$

2-share implementation of G

$$
\begin{aligned}
H[0] &\leftarrow (D[0]\&B[0]) \oplus R[2] \oplus C[0] \oplus (D[1]\&B[0]) \\
E[0] &\leftarrow (C[1]\&B[0]) \oplus (D[0]\&B[0]) \oplus R[1] \oplus (C[0]\&B[0]) \oplus A[0] \oplus (D[1]\&B[0]) \\
G[0] &\leftarrow B[0] \\
F[0] &\leftarrow (A[0]\&B[0]) \oplus D[0] \oplus R[3] \oplus (B[0]\&A[1])
\end{aligned}
$$

$$
\begin{aligned}
H[1] &\leftarrow (D[0]\&B[1]) \oplus R[2] \oplus C[1] \oplus (D[1]\&B[1]) \\
E[1] &\leftarrow (C[0]\&B[1]) \oplus (D[0]\&B[1]) \oplus R[1] \oplus (C[1]\&B[1]) \oplus A[1] \oplus (D[1]\&B[1]) \\
G[1] &\leftarrow B[1] \\
F[1] &\leftarrow (A[1]\&B[1]) \oplus D[1] \oplus [(B[1]\&A[0]) \oplus R[3]]
\end{aligned}
$$

$$
R \leftarrow R + 12
$$

3.4. 3-Share PRESENT Sbox

The 3-share version of the PRESENT Sbox is an asm implementation borrowed from 2021 Barthe *et al.*'s paper [BGG+21]. As for the 2-share version, this implementation uses the decomposition into

three subfunctions of the PRESENT Sbox:

$$Sbox : \mathsf{A} \circ \mathsf{G} \circ \mathsf{G} \circ \mathsf{B},$$

where $\mathsf{A}, \mathsf{B}, \mathsf{G}$ are vectorial Boolean functions from $\{0,1\}^4$ into itself and A, B are affine while G is quadratic. The masking scheme requires 104 fresh random bytes (in addition to the 16-Byte random that are needed by the sharing of the 4 16-bit input variables $\{A, B, C, D\}$).

All in all, between two calls to the 3-share PRESENT Sbox implementation, 132 random bytes are extracted thanks to 33 calls to the PRNG:

- 8 bytes to generate new inputs $\{A, B, C, D\}$;
- 16 bytes to share the inputs $\{A, B, C, D\} \rightarrow$ $\{(A[0], A[1], A[2]), (B[0], B[1], B[2]), (C[0], C[1], C[2]), (D[0], D[1], D[2])\}$;
- 108 bytes for the probing security, stored in a 27 length array R of `unsigned long` variables (the first 4 bytes, *i.e.* the first cell of R, are not used). Moreover,
  - eight 32-bit randoms are consumed in B;
  - nine 32-bit randoms are consumed for each call to G;

For reasons of completeness, and because the asm implementation might not be trivial to understand, we unfold here the implementations of the $\mathsf{A}, \mathsf{B}, \mathsf{G}$ subroutines. Note that the computation order of the successive operations is here crucial to assess 2nd order security.

3-share implementation of A

$$\begin{array}{lcl} G[0] & \leftarrow & A[0] \\ E[0] & \leftarrow & C[0] \oplus A[0] \\ H[0] & \leftarrow & C[0] \oplus A[0] \oplus 0xFFFFFFFF \oplus D[0] \\ F[0] & \leftarrow & B[0] \oplus 0xFFFFFFFF \end{array}$$

$$\begin{array}{lcl} G[1] & \leftarrow & A[1] \\ E[1] & \leftarrow & C[1] \oplus A[1] \\ H[1] & \leftarrow & C[1] \oplus A[1] \oplus D[1] \\ F[1] & \leftarrow & B[1] \end{array}$$

$$\begin{array}{lcl} G[2] & \leftarrow & A[2] \\ E[2] & \leftarrow & C[2] \oplus A[2] \\ H[2] & \leftarrow & C[2] \oplus A[2] \oplus D[2] \\ F[2] & \leftarrow & B[2] \end{array}$$

3-share implementation of B

$$
\begin{aligned}
H[0] &\leftarrow D[0] \oplus B[0] \oplus 0xFFFFFFFF \oplus R[1] \\
E[0] &\leftarrow A[0] \oplus B[0] \oplus R[3] \\
F[0] &\leftarrow B[0] \oplus C[0] \oplus R[5] \\
G[0] &\leftarrow C[0] \oplus R[7]
\end{aligned}
$$

$$
\begin{aligned}
H[1] &\leftarrow D[1] \oplus B[1] \oplus R[2] \\
E[1] &\leftarrow A[1] \oplus B[1] \oplus R[4] \\
F[1] &\leftarrow B[1] \oplus C[1] \oplus R[6] \\
G[1] &\leftarrow C[1] \oplus R[8]
\end{aligned}
$$

$$
\begin{aligned}
H[2] &\leftarrow D[2] \oplus B[2] \oplus [R[1] \oplus R[2]] \\
E[2] &\leftarrow A[2] \oplus B[2] \oplus [R[3] \oplus R[4]] \\
F[2] &\leftarrow B[2] \oplus C[2] \oplus [R[5] \oplus R[6]] \\
G[2] &\leftarrow C[2] \oplus [R[7] \oplus R[8]]
\end{aligned}
$$

$$
R \leftarrow R + 32
$$

3-share implementation of G

$$
\begin{aligned}
E[0] &\leftarrow (C[1]\&B[0]) \oplus (D[1]\&B[0]) \oplus R[4] \oplus A[0] \oplus (C[0]\&B[0]) \oplus (D[0]\&B[0]) \\
&\quad \oplus R[5] \oplus (C[2]\&B[0]) \oplus (D[2]\&B[0]) \\
F[0] &\leftarrow (A[1]\&B[0]) \oplus R[7] \oplus (A[0]\&B[0]) \oplus D[0] \oplus R[8] \oplus (A[2]\&B[0]) \\
G[0] &\leftarrow B[0] \\
H[0] &\leftarrow (D[1]\&B[0]) \oplus R[1] \oplus C[0] \oplus (D[0]\&B[0]) \oplus R[2] \oplus (D[2]\&B[0])
\end{aligned}
$$

$$
\begin{aligned}
E[1] &\leftarrow (C[2]\&B[1]) \oplus (D[2]\&B[1]) \oplus R[5] \oplus A[1] \oplus (C[1]\&B[1]) \oplus (D[1]\&B[1]) \\
&\quad \oplus R[6] \oplus (C[0]\&B[1]) \oplus (D[0]\&B[1]) \\
F[1] &\leftarrow (A[2]\&B[1]) \oplus R[8] \oplus (A[1]\&B[1]) \oplus D[1] \oplus R[9] \oplus (A[0]\&B[1]) \\
G[1] &\leftarrow B[1] \\
H[1] &\leftarrow (D[2]\&B[1]) \oplus R[2] \oplus C[1] \oplus (D[1]\&B[1]) \oplus R[3] \oplus (D[0]\&B[1])
\end{aligned}
$$

$$
\begin{aligned}
E[2] &\leftarrow (C[0]\&B[2]) \oplus (D[0]\&B[2]) \oplus R[6] \oplus A[2] \oplus (C[2]\&B[2]) \oplus (D[2]\&B[2]) \\
&\quad \oplus R[4] \oplus (C[1]\&B[2]) \oplus (D[1]\&B[2]) \\
F[2] &\leftarrow (A[0]\&B[2]) \oplus R[9] \oplus (A[2]\&B[2]) \oplus D[2] \oplus R[7] \oplus (A[1]\&B[2]) \\
G[2] &\leftarrow B[2] \\
H[2] &\leftarrow (D[0]\&B[2]) \oplus R[3] \oplus C[2] \oplus (D[2]\&B[2]) \oplus R[1] \oplus (D[1]\&B[2])
\end{aligned}
$$

$$
R \leftarrow R + 36
$$

4. Unprotected PRESENT Sbox

The unprotected PRESENT Sbox implementation is our first target, this study will help select the acquisition parameters, validate the chip leakage strength and observe the leakage areas and then ease the analysis of the masked implementations (see Sections 5 and 6).

4.1. First Observations

As detailed in Section 2.3, our trigger mechanism necessitates to acquire two traces simultaneously: the Power channel that relates to the power consumption of the chip and the I/O channel that bears the ISO7816 communication as well as the trig signals before and after each call to the cryptographic primitive.
This setup allows to request, in a single command to the smartcard, several (say n) computations of the PRESENT Sbox (see Section 3 for more details on the available APDU commands). Then, during the post-processing of the two acquired traces, the I/O signal is used to split the power trace into n subtraces. From the knowledge of the PRNG seed, we can reconstruct the successive inputs (including the masking material) of the n PRESENT Sbox calls.

In a first experiment, we acquire a maximum number of PRESENT Sbox computations in a single trace. The acquisition parameters are detailed in Table 1, the sampling rate is chosen following the characterization study (see deliverable L3.3).

operation	Unprotected PRESENT Sbox
equipment	PicoScope 6424E, Scaffold
inputs	`src[0]` and `src[1]` filled with 32 bits fresh randoms
number of operations	170
length	500 ms
sampling rate	1.25GSa/s
samples per trace	625MSamples
channel(s)	I/O, Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-220, -20]$mV
file size	1.25GB
acquisition time	about 30s

Table 1: Acquisition Parameters Unprotected PRESENT Sbox – First Tentative

Figure 4 displays a single trace (containing 170 PRESENT Sbox successive calls) of the I/O channel for different levels of zoom. One can clearly see the successive trigger signals that will provide a very handy way to split the trace. In the last subfigure, the gray boxes represent the actual PRESENT Sbox computations while the area in between relates to

1. a call to the `fromBitslice` function that builds the 16 PRESENT Sbox outputs from their bitsliced format.
2. a comparison of the output to the expected one
3. two PNRG calls (*i.e.* two iterations of the `xorshift96`) to refresh the inputs
4. a table-based computation of the PRESENT Sbox to compute the reference output
5. a call to the `toBitslice` function that sends the 16 PRESENT Sbox inputs into their bitsliced format.

It results that each PRESENT Sbox computation takes about 200K Samples (*i.e.* 160 us at 1.25GSamples/s) while the time between two computations takes about 3.2M Samples (*i.e.* 16 times longer than the PRESENT Sbox computation itself).

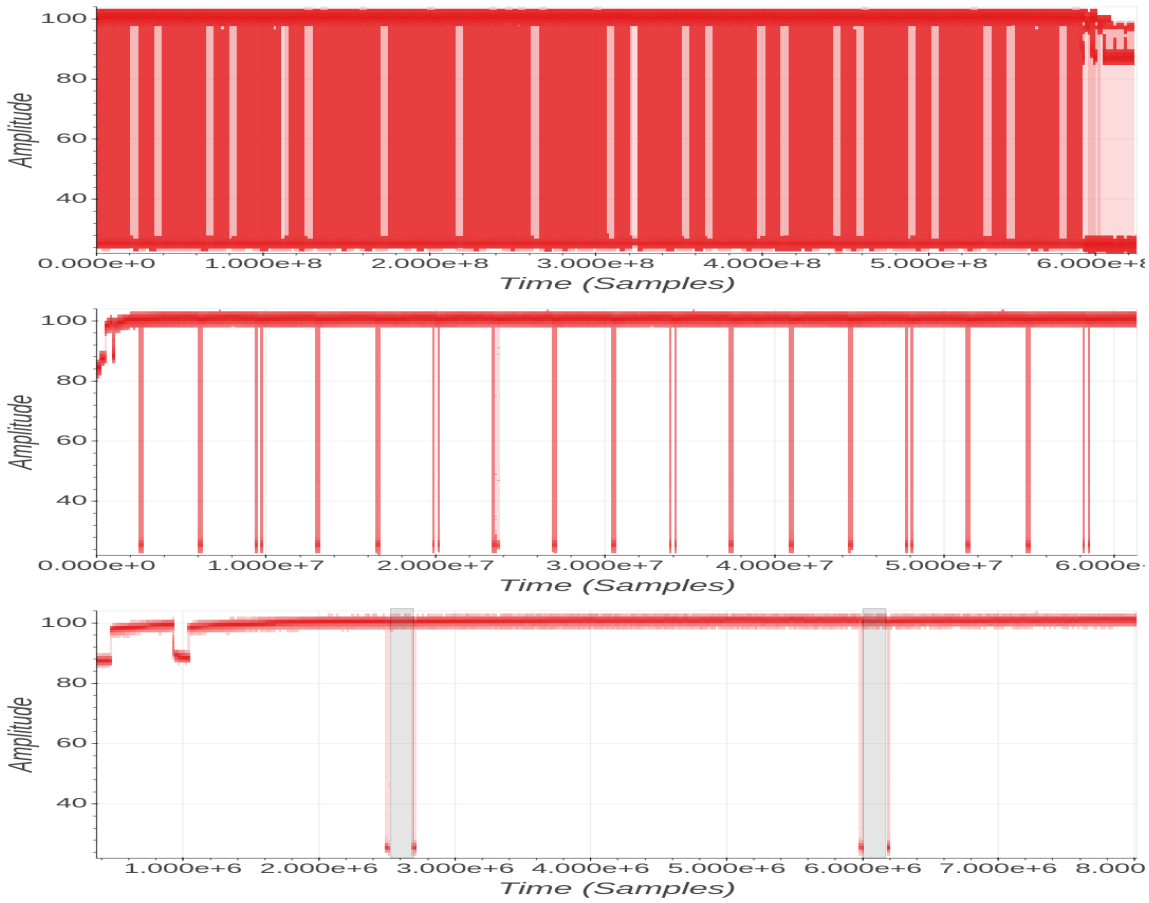

Figure 4: Unprotected PRESENT Sbox 170 Operations – I/O Trace – Full Trace (Top) - First Level of Zoom (Middle) - Second Level of Zoom (Bottom)

The above observation tends to show that for each stored power trace, only a small part will be useful. Hence, to reduce the storage cost, we modified our acquisition process to apply the splitting post-process directly on the acquired traces before writing them into a file. That way, we drastically reduced the size of the power traces but also could get rid of the I/O traces without storing them on disk.

This *online* splitting procedure allows to acquire 450 traces (each containing 170 PRESENT Sbox computations) in a single file of size 23G Bytes (instead of more than 500G Bytes). The parameters for this acquisition campaign are detailed in Table 2.

operation	Unprotected PRESENT Sbox
equipment	PicoScope 6424E, Scaffold
inputs	`src[0]` and `src[1]` filled with 32 bits fresh randoms
number of operations	450×170
length	500 ms
sampling rate	1.25GSa/s
samples per trace	300KSamples
channel(s)	Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-220, -20]$mV
file size	23GB
acquisition time	about 4h

Table 2: Acquisition Parameters Unprotected PRESENT Sbox

The resulting power traces are depicted in Figure 5 (and zooming into a small part of the traces in Figure 6), the second subfigure is the averaged trace over all acquisitions. It shows that selecting the external clock was a good choice (see Section 2.1) as it is stable enough to avoid traces resynchronization.

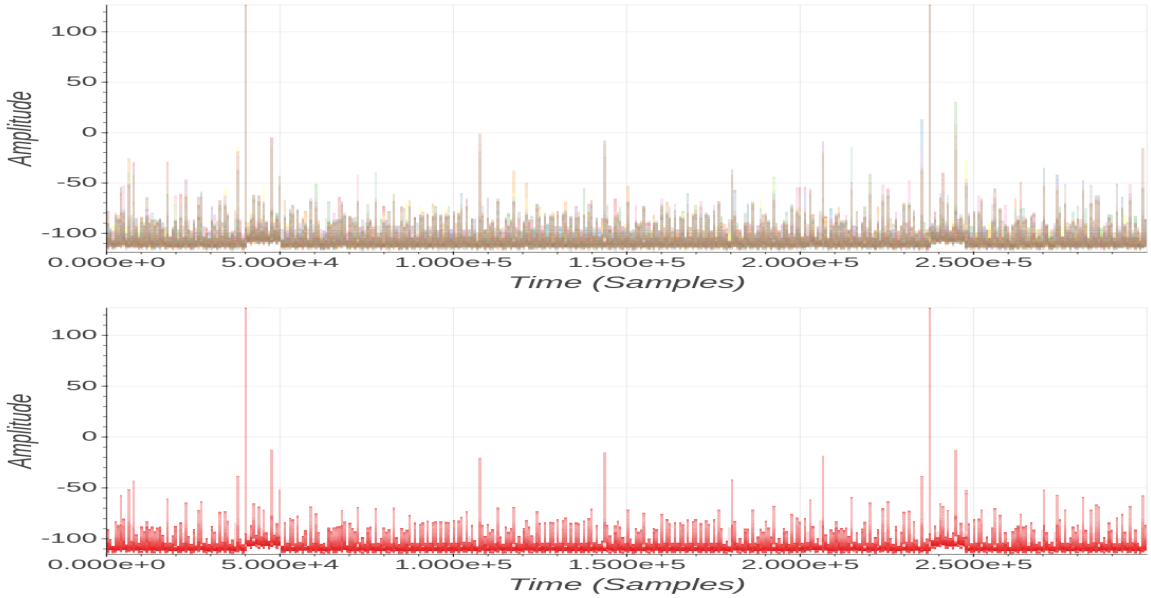

Figure 5: Unprotected PRESENT Sbox Single Operation – Power Traces – 10 Superposed Traces (Top) - Averaged over 76500 Traces (Bottom)

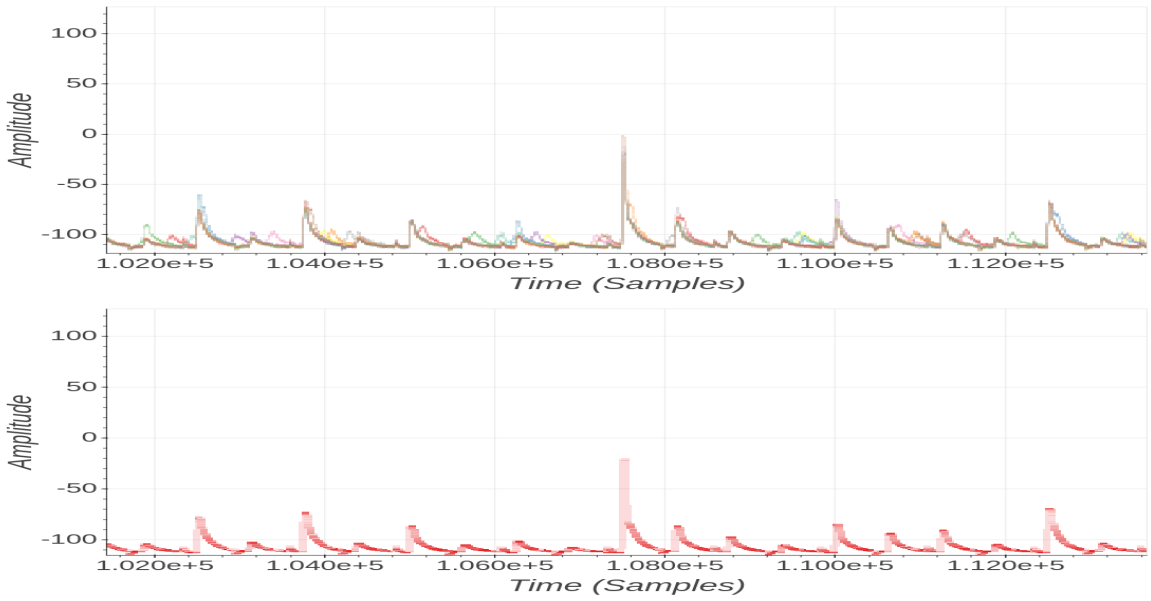

Figure 6: Unprotected PRESENT Sbox Single Operation – Power Traces Zoom – 10 Superposed Traces (Top) - Averaged over 76500 Traces (Bottom)

4.2. Leakage Analysis

The implementation under study is detailed in Section 3.2, the four 16-bit bitsliced inputs $\{A, B, C, D\}$ are sent to the four 16-bit bitsliced outputs $\{E, F, G, H\}$. Our leakage analysis first proceeds with a SNR evaluation of the bitsliced inputs/outputs at the byte level. In other words, each 16-bit variable $X \in \{A, B, C, D, E, F, G, H\}$ is divided into two 8-bit variables (X^L, X^H), and the 8-bit value SNR is computed for all 16 8-bit variables X^α for $\alpha \in \{L, H\}$. The results are depicted in Figure 7, where the first subfigure recalls the averaged trace, the second subfigure shows the 8 univariate SNR results for the 8 input variables and the last subfigure shows the univariate 8 SNR results for the 8 output variables. For clarity, both 8-bit variables of a single 16-bit variable X are represented with the same color.

From the input related SNRs (second subfigure) one can follow the successive usage of the input variables.
The output related SNRs (last subfigure) depict clearly the 4 output variable stores, in the correct order.

By zooming into the results, one can remark that the 2 8-bit variables of a 16-bit variable leak at the same time, which makes sense since these two variables are stored in a single 32-bit register and then accessed and modified simultaneously. Moreover, one can see that the leakage-related time samples are located in specific time slots in the traces. This will help reduce the traces size in the next sections. These observations are illustrated with two different levels of zoom in Figures 8 and 9.

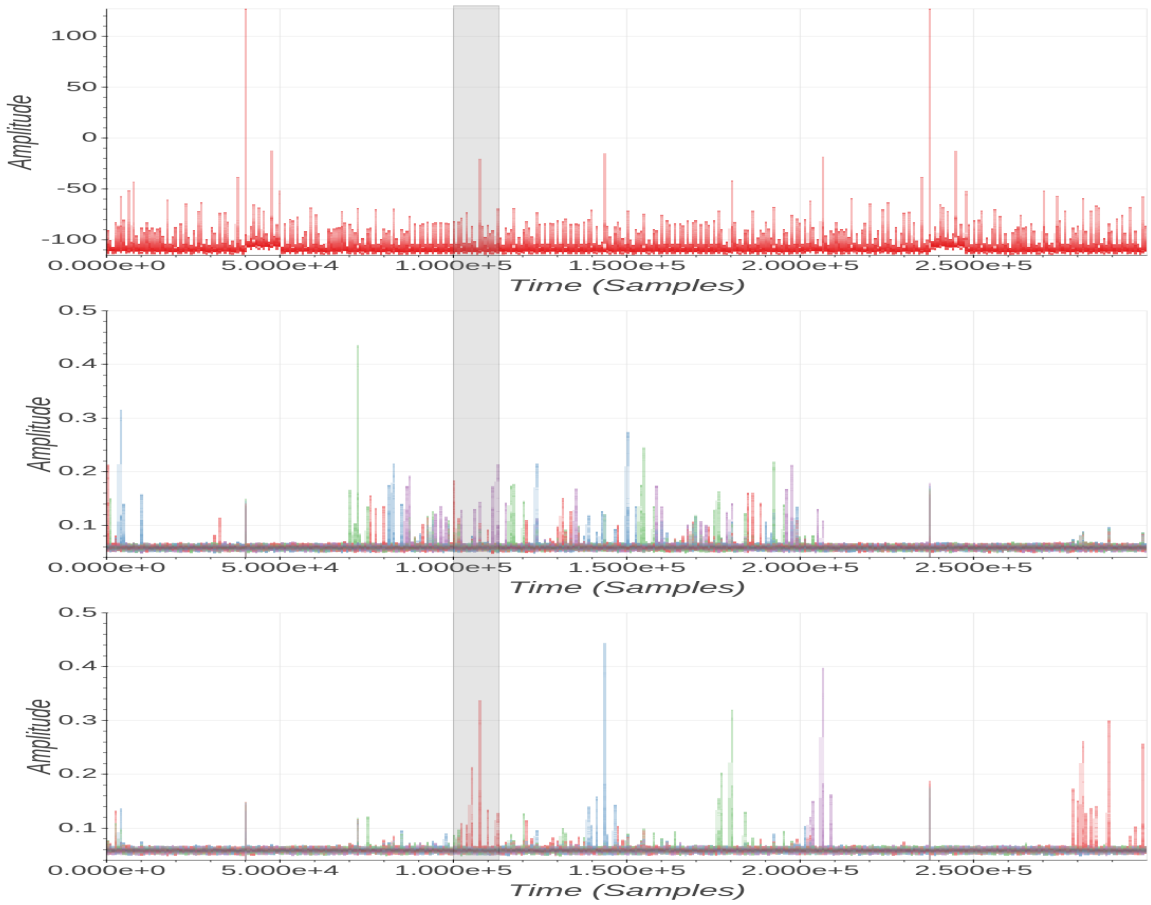

Figure 7: Unprotected PRESENT Sbox– SNR Results – Average Trace (Top) - SNR $\{A, B, C, D\}$ (Middle) - SNR $\{E, F, G, H\}$ (Bottom)

Figure 8 is a simple zoom inside the gray area identified in Figure 7, this corresponds to the time area where the first output variable (E) is stored in memory. This figure illustrates two important remarks on the traces and their relevant side-channel leakages:

1. the leakages relative to intermediate variables seem to solely appear in a specific area of the clock cycle (identified in Figure 8 by gray boxes). This remark allows to reduce the size of the traces and will be crucial for the analysis of the more expensive masked implementations.

2. the leakage related to a single intermediate value (say *e.g.* E in the last subfigure) spans for many clock cycles. This can be explained by a combination of factors:

   - some instructions (*e.g.* `load` and `store`) take several clock cycles to execute;
   - the pipeline length of the *OpenCard* IC is 3, so that an instruction start being treated before its real execution;
   - the power consumption measurement is *global*: any register or clock gate activity is captured. As a side note, one can see that the power information persistence is spread over several samples but does not overlap over multiple clock cycles.

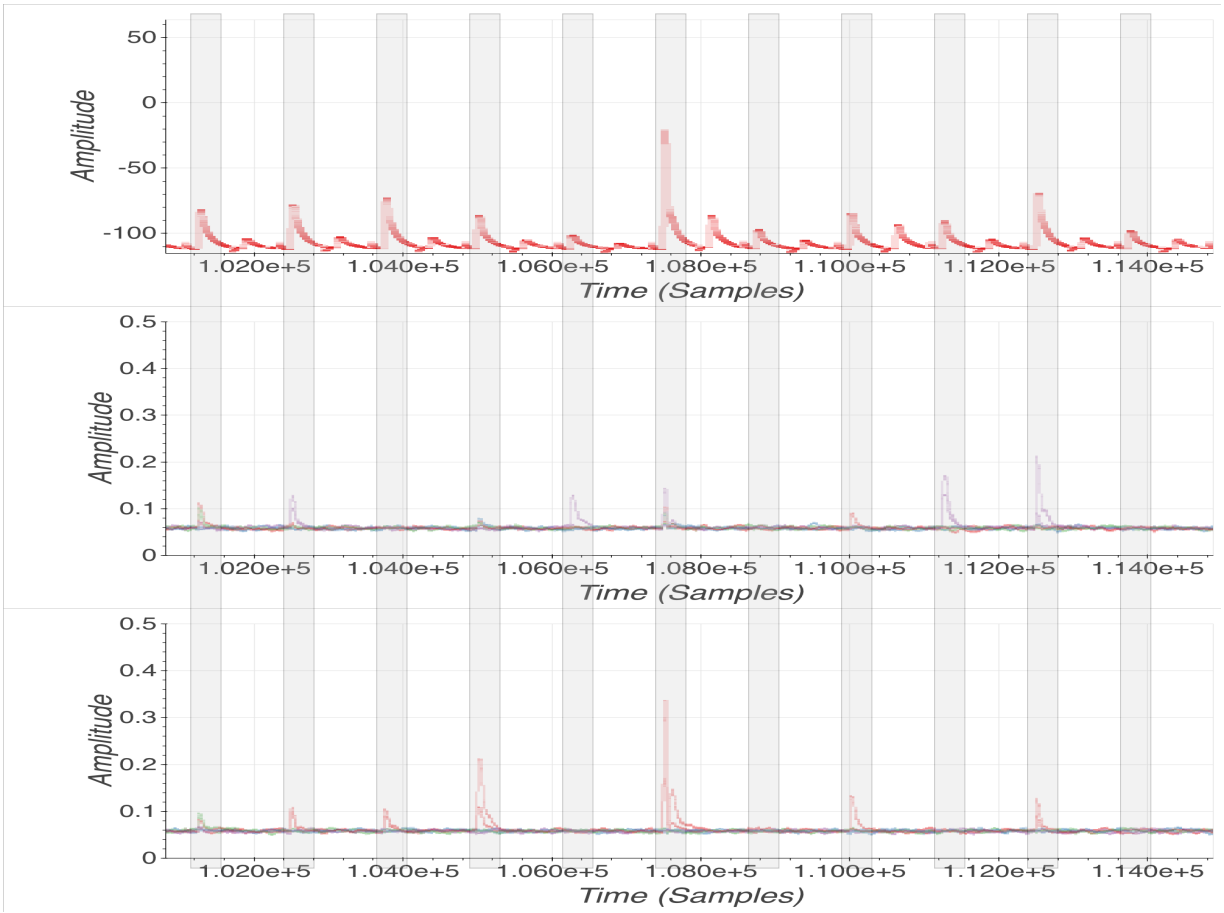

Figure 8: Unprotected PRESENT Sbox– SNR Results Zoom – Average Trace (Top) - SNR $\{A, B, C, D\}$ (Middle) - SNR $\{E, F, G, H\}$ (Bottom)

Figure 9 zooms in again inside Figure 8, around the highest SNR peak related to E. The leakage study is now at the bit level and we focus on the output variable here. For each of the 4 output variables ($\{E, F, G, H\}$), 16 Welch's T-Tests [Wel47] are estimated.

- In the second subfigure, all 16 T-Test results are displayed with same color. It tells us that, at this point in time, only the output variable E leaks some information (nothing really surprising here since the other output variables should not have been computed yet).
- In the last subfigure, the same 16 T-Test results are displayed but this time, the color red is given to the 8 least significant bits of the output variables, while the color blue indicates that the target bit comes from the 8 most significant bits of an output variable. This result tells us that the 8 least significant bits of a 32-bit register seem to have a stronger leakage than the next 8 bits. This observation is pretty stable for all the results observed in this study. This is also aligned with the results of the characterization task (see deliverable L3.3 for more details).

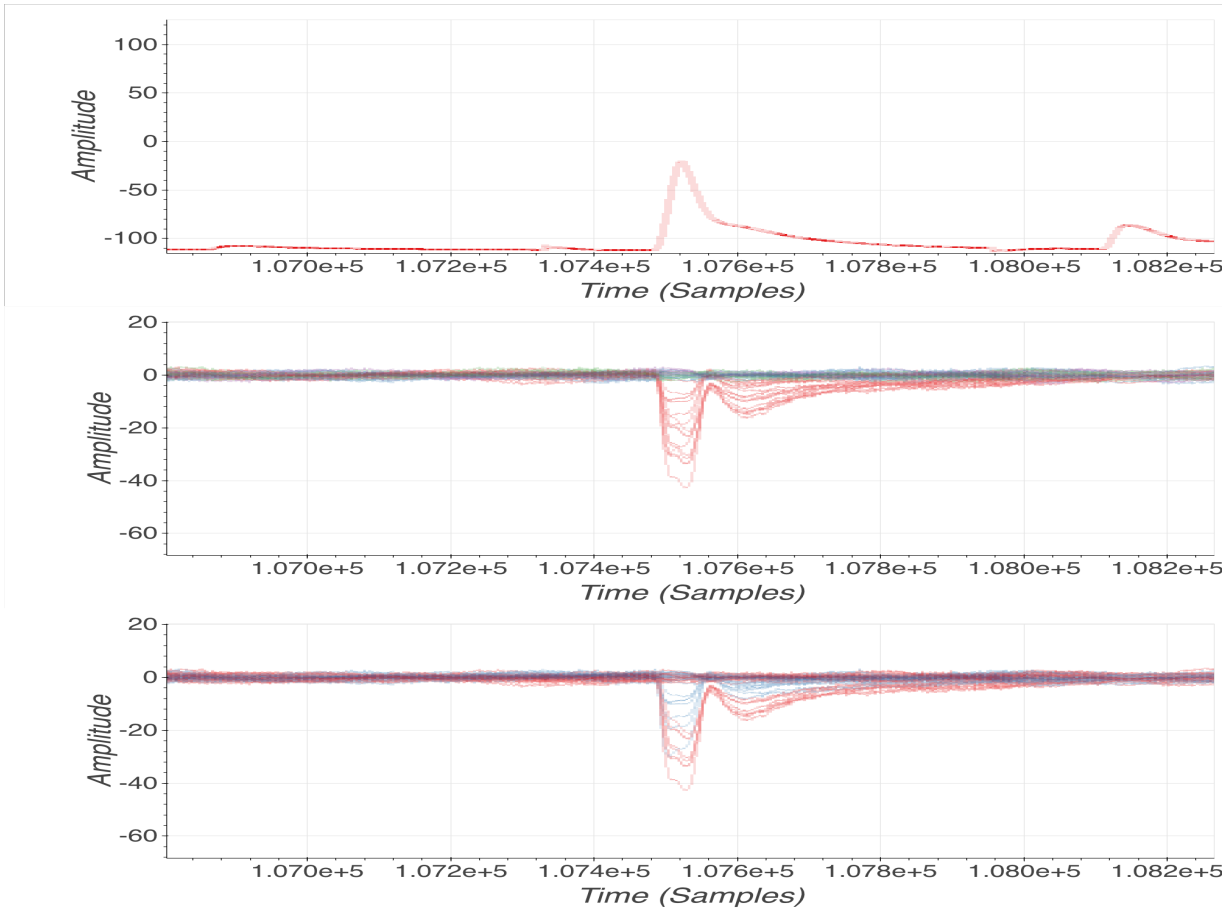

Figure 9: Unprotected PRESENT Sbox– bit-wise T-Test Results – Average Trace (Top) - 4×16 T-Tests $\{E, F, G, H\}$ (Middle) - 4×16 T-Tests $\{(E^L, E^H), (F^L, F^H), (G^L, G^H), (H^L, H^H)\}$ (Byte 0 vs. Byte 1) (Bottom)

5. 2-Share PRESENT Sbox

Thanks to the preliminary leakage analysis of the unprotected PRESENT Sbox implementation detailed in the previous section, we can now move to the first masked implementation of PRESENT Sbox. We here first present the acquisition campaign as well as the resulting traces before going into the details of the leakage analysis.

5.1. Acquisition Campaign

As in Section 4.1, our first experiment acquires a single power trace that contains a maximum number of computations (*i.e.* filling the oscilloscope memory). Due to the length of the computation and the requirement in fresh random for each execution, the length for an Sbox computation is much higher than for the unprotected case, and then the number of executions that can contain a single trace is limited to 80, the acquisition parameters are detailed in Table 3.

operation	2-Share PRESENT Sbox
equipment	PicoScope 6424E, Scaffold
inputs	`src[0]` and `src[1]` filled with 32-bit fresh randoms `entropy[i]` for $i \in \{0, \cdots, 7\}$ filled with 32-bit fresh randoms
number of operations	80
length	500 ms
sampling rate	1.25GSa/s
samples per trace	625MSamples
channel(s)	I/O, Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-220, -20]$mV
file size	1.25GB
acquisition time	about 30s

Table 3: Acquisition Parameters 2-Share PRESENT Sbox – First Tentative

Figure 10 displays the I/O channel of 80 successive executions of the 2-share PRESENT Sbox. It tells us that the computation length of a single PRESENT Sbox is now about 850K Samples (*i.e.* 680 us at 1.25GSamples/s, more than a factor 4 compared to the unprotected implementation). The length between two PRESENT Sbox calls takes now about 6M Samples (mostly fresh random generation from 12 PRNG calls).

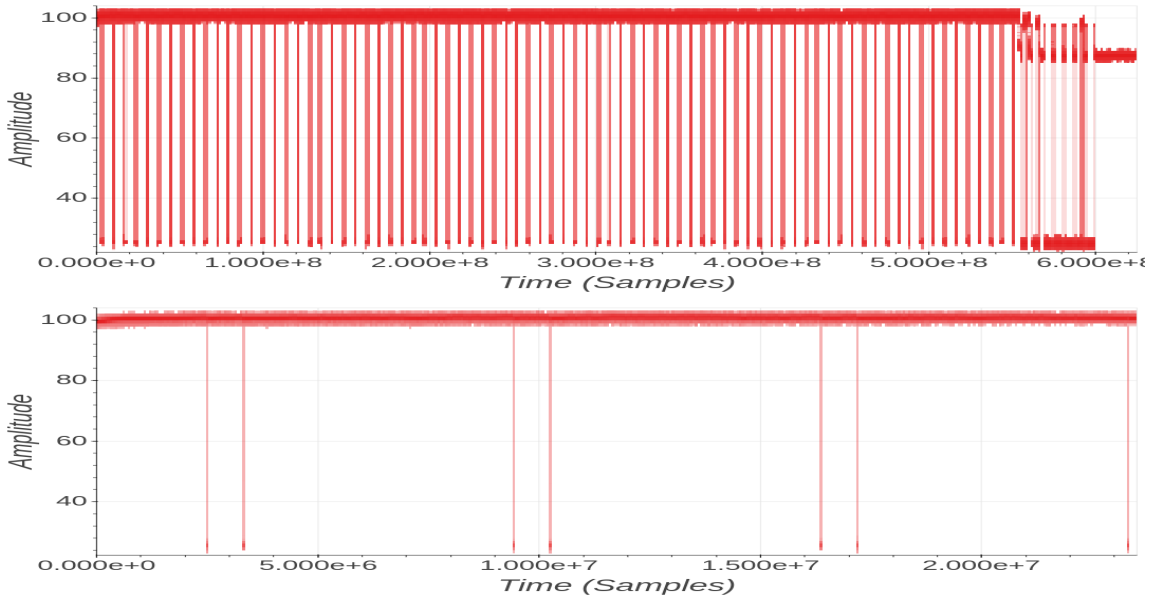

Figure 10: 2-Share PRESENT Sbox 80 Operations – I/O Trace – Full Trace (Top) - Zoom in Firsts Executions (Bottom)

From the above observations, similarly to the unprotected case, we will apply *online* the post-processing of dividing the power traces into 80 PRESENT Sbox computations and then reduce drastically the side-channel traces storage. The full acquisition campaign parameters are given in Table 4, in total 5000×80 PRESENT Sbox computations are observed for a total storage size of 360G Bytes and about 45h of acquisition.

operation	2-Share PRESENT Sbox
equipment	PicoScope 6424E, Scaffold
inputs	`src[0]` and `src[1]` filled with 32-bit fresh randoms `entropy[i]` for $i \in \{0, \cdots, 7\}$ filled with 32-bit fresh randoms
number of operations	5000×80
length	500 ms
sampling rate	1.25GSa/s
samples per trace	900KSamples
channel(s)	Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-220, -20]$mV
file size	360GB
acquisition time	about 45h

Table 4: Acquisition Parameters 2-Share PRESENT Sbox

Figure 11 displays the resulting power traces of a single PRESENT Sbox execution, the averaged trace over the whole 400K traces shows clearly that resynchronization is not necessary. The resulting traces are 900K Samples long, we will further reduce these traces to reduce the analysis time.

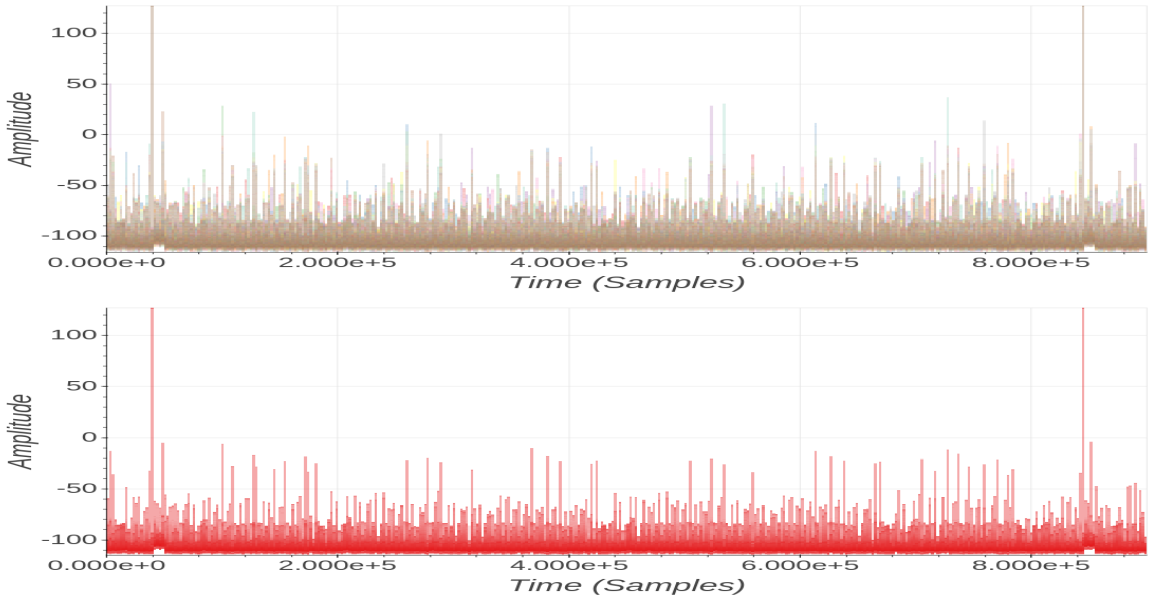

Figure 11: 2-Share PRESENT Sbox Single Operation – Power Traces – 10 Superposed Traces (Top) - Averaged over 400K Traces (Bottom)

From the analysis of the leakages observed in the unprotected case (see Figure 8, Section 4.2), we know that within a clock cycle, only a small part of the time samples are likely to bear any intermediate variable related leakages. The following figure 12 illustrates our extra post-processing on the power traces: first detect the center of the leakage area for all clock cycles (first and second subfigure) and then only keep 300 samples around each center. The resulting reduced traces are displayed in the last subfigure (for a zoom) and Figure 13 (for the whole averaged trace). The resulting traces are about 193K Samples long.

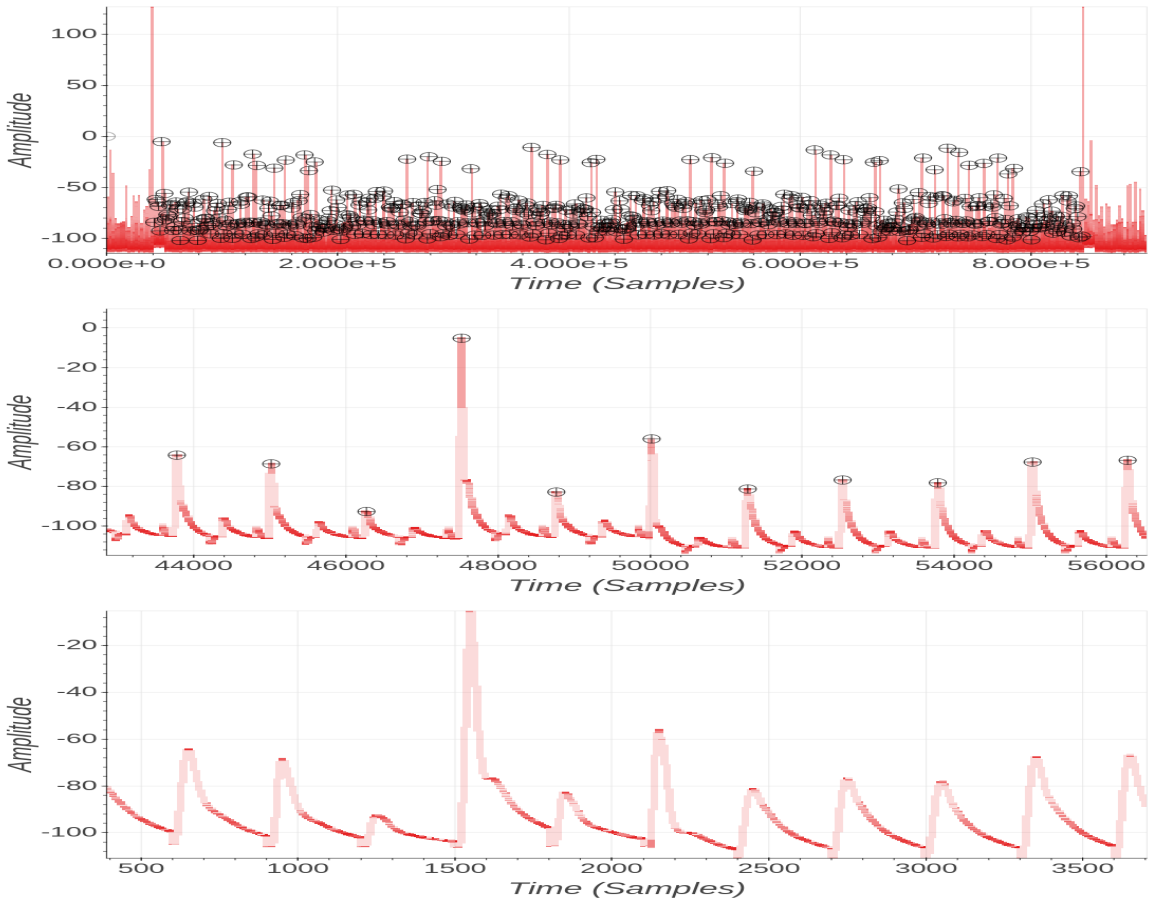

Figure 12: 2-Share PRESENT Sbox Single Operation – Power Traces with Syncro Points – Average Trace (Top) - Average Trace Zoom (Middle) - Average Reduced Trace Zoom (Bottom)

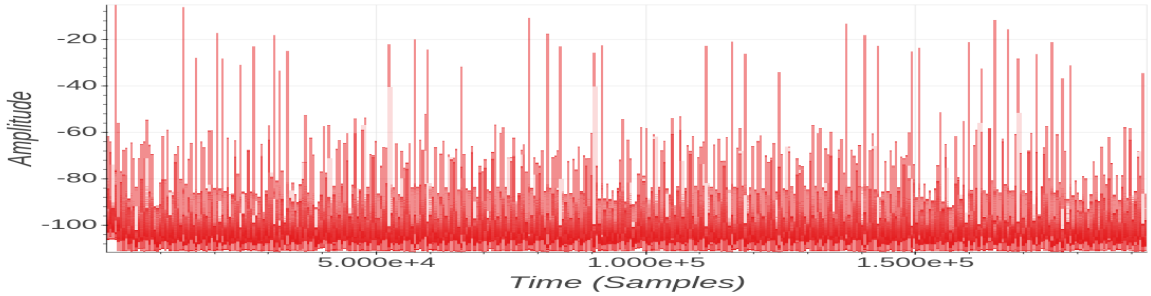

Figure 13: 2-Share PRESENT Sbox Single Operation – Power Traces – Average Reduced Trace

5.2. Leakage Analysis

Figures 14 to 18 display the SNR results on the whole acquisition campaign with respect to the 5 intermediate variables that naturally appear from the decomposition A ∘ G ∘ G ∘ B (see Section 3.3): the inputs, the output of B, the output of G ∘ B, the output of G ∘ G ∘ B and the output of A ∘ G ∘ G ∘ B. Each one of these intermediate variables can be represented as 4 16-bit vari-

ables (say $\{A, B, C, D\}$), each one is stored and manipulated as two shares $\{(A[0], A[1]), (B[0], B[1]), (C[0], C[1]), (D[0], D[1])\}$. Each 16-bit share leads to two 8-bit SNR computations. Hence each figure depicts the 8 SNRs for the first share (middle subfigure) and for the second share (last subfigure) for a given intermediate variable.

This analysis of the shares leakage throughout the PRESENT Sbox computation allows to roughly identify the succession of subfunction executions as illustrated on the figures.

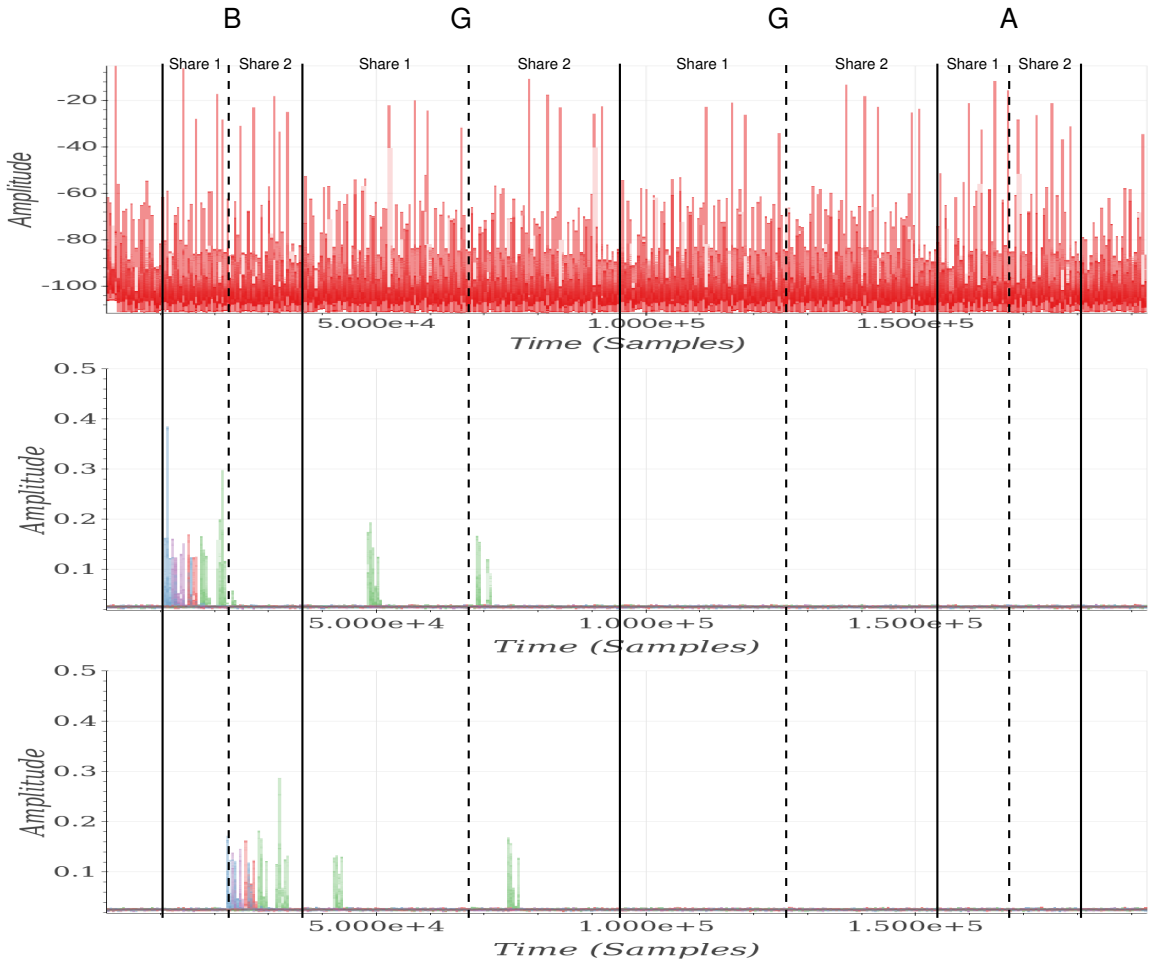

Figure 14: 2-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$ – Average Reduced Trace (Top) - SNR Inputs Share 1 (Middle) - SNR Inputs Share 2 (Bottom)

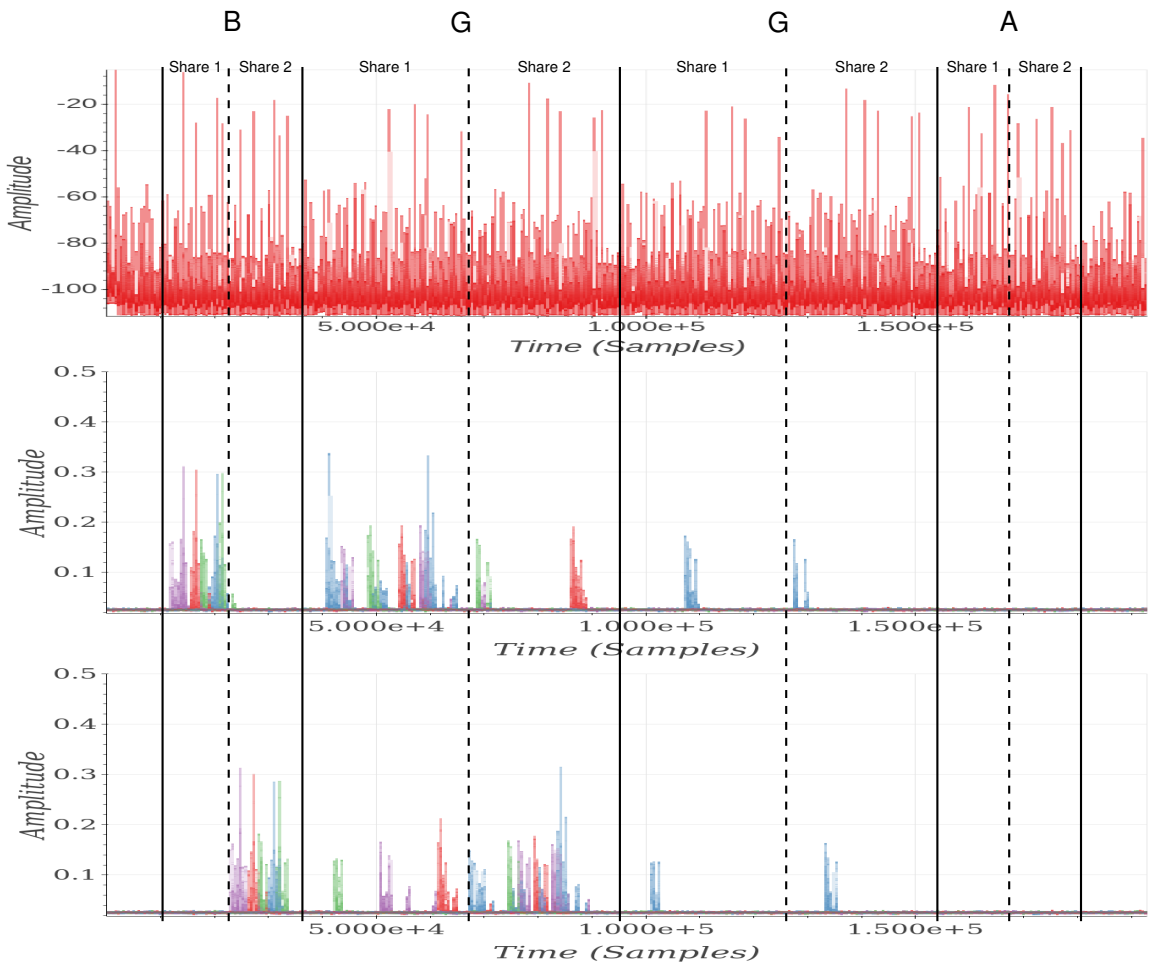

Figure 15: 2-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$ – Average Reduced Trace (Top) - SNR Outputs B Share 1 (Middle) - SNR Outputs B Share 2 (Bottom)

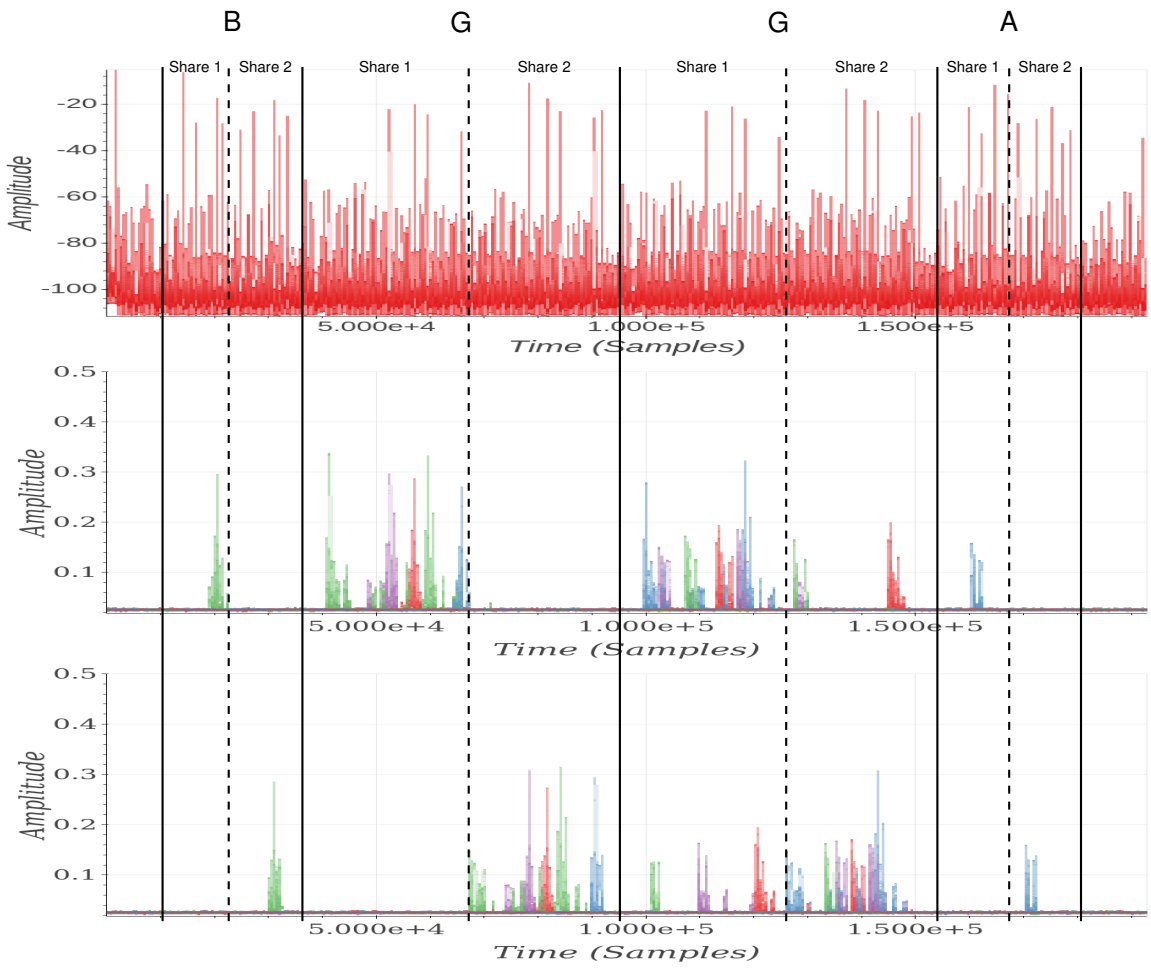

Figure 16: 2-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$ – Average Reduced Trace (Top) - SNR Outputs G $\circ$ B Share 1 (Middle) - SNR Outputs G $\circ$ B Share 2 (Bottom)

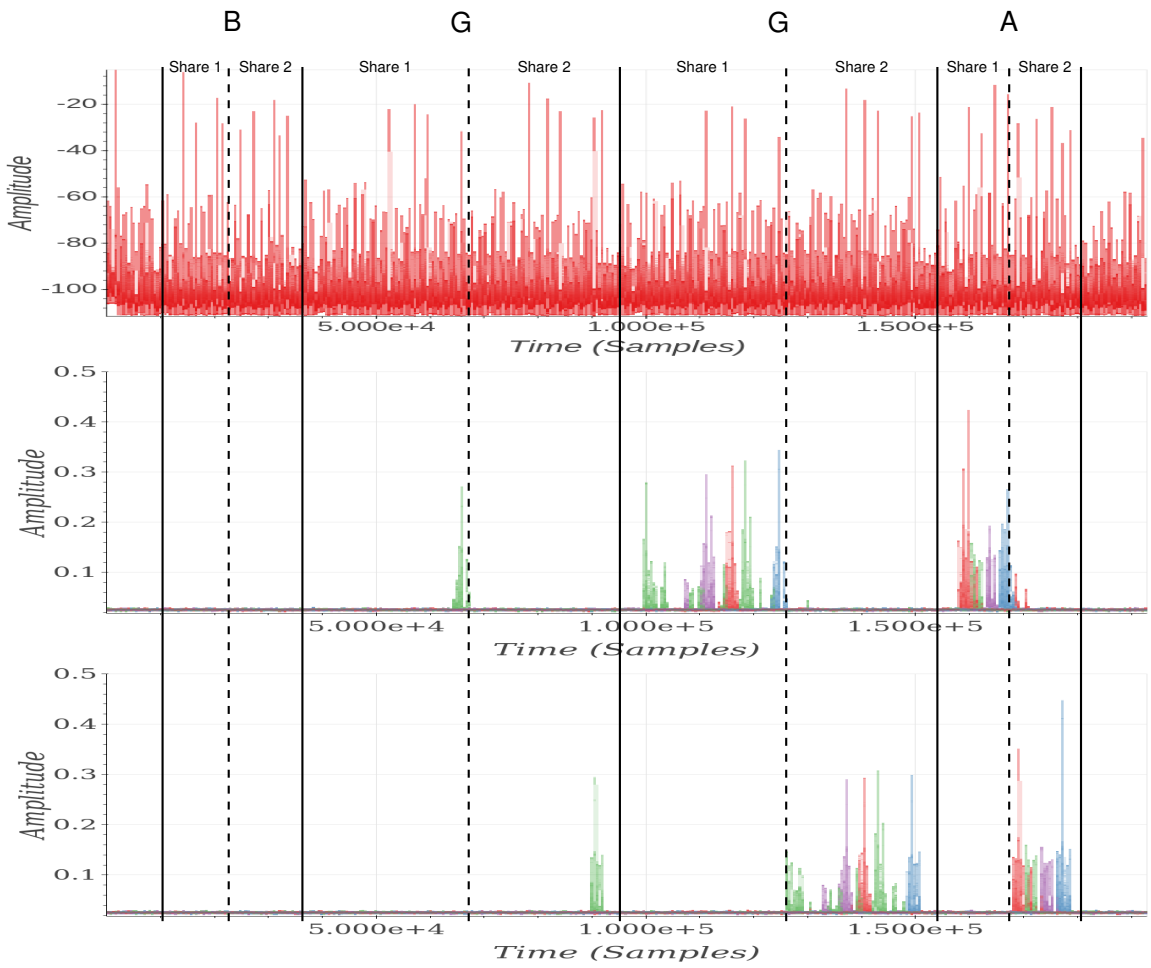

Figure 17: 2-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$ – Average Reduced Trace (Top) - SNR Outputs G $\circ$ G $\circ$ B Share 1 (Middle) - SNR Outputs G $\circ$ G $\circ$ B Share 2 (Bottom)

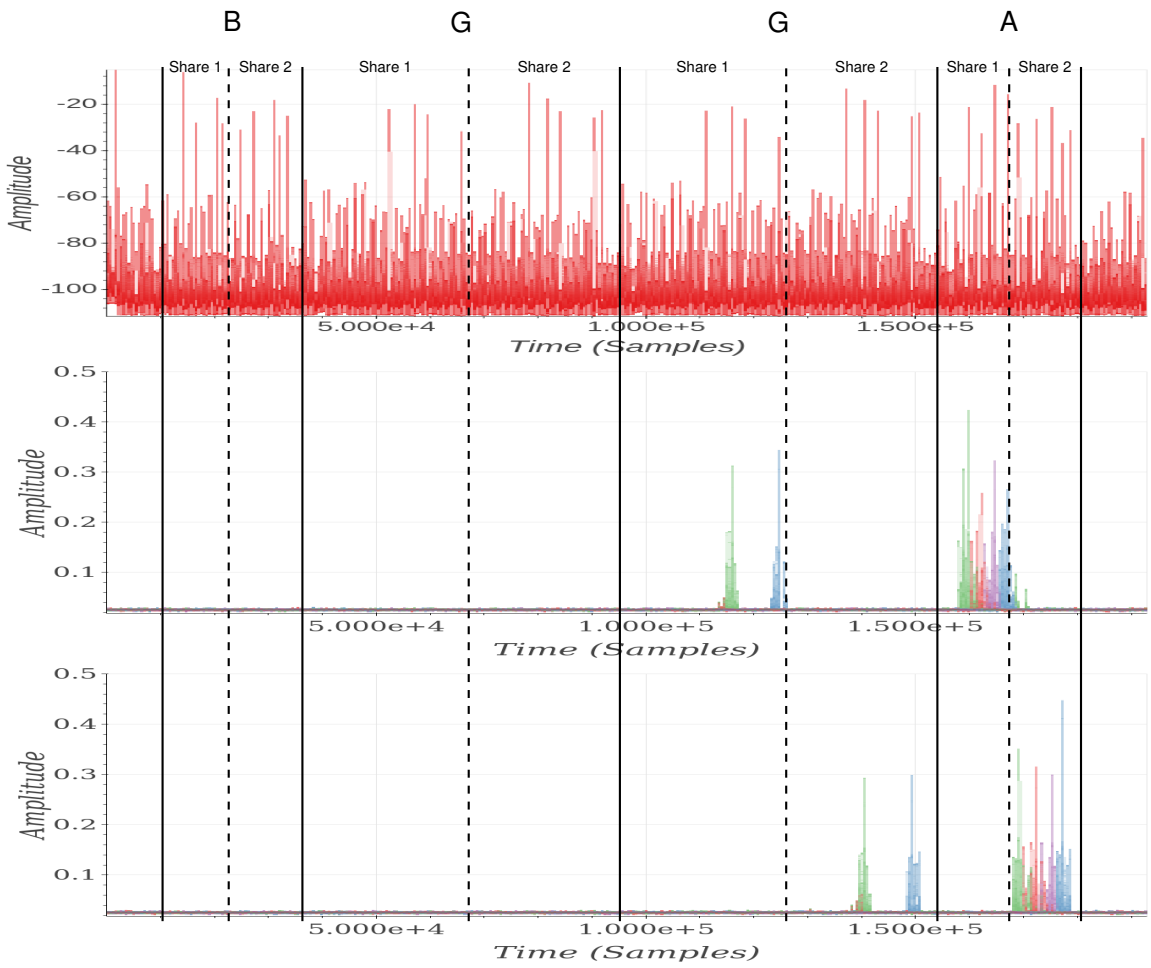

Figure 18: 2-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$ – Average Reduced Trace (Top) - SNR Outputs A ∘ G ∘ G ∘ B Share 1 (Middle) - SNR Outputs A ∘ G ∘ G ∘ B Share 2 (Bottom)

In Figure 19, the SNR results relate to the entropy bytes (stored in array R). In the second subfigure, the same color is used for the 2 bytes of each 16-bit randoms in R: there are 7 such randoms that are used during the computation, one is consumed during the computation of B and 3 are consumed during each call to G.
The last subfigure is simply a color change, here the red is used to identify the least significant byte of a random in R while blue indicates the most significant byte. This last result shows (again) well that the least significant byte has a stronger leakage than the most significant one. Figure 20 is a zoom into this representation around one of the SNR peaks. One can again observe that, when accessing a value stored in memory, its manipulation will affect the side-channel trace for many consecutive clock cycles (see Section 4.2 for similar observations on the unprotected case).

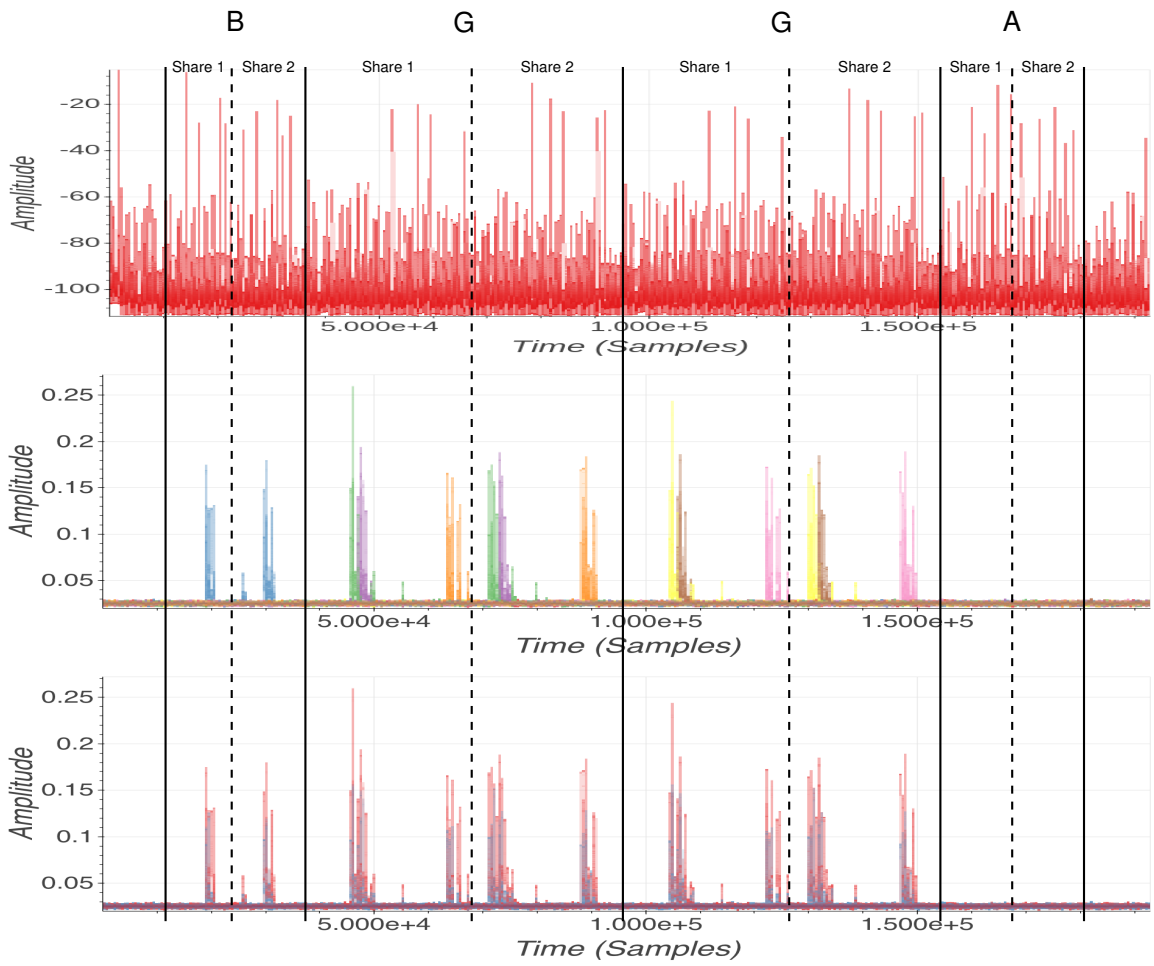

Figure 19: 2-Share PRESENT Sbox Single Operation – SNR Results – Average Reduced Trace (Top) - SNR Entropy (Middle) - SNR Entropy Byte 0 vs. Byte 1 (Bottom)

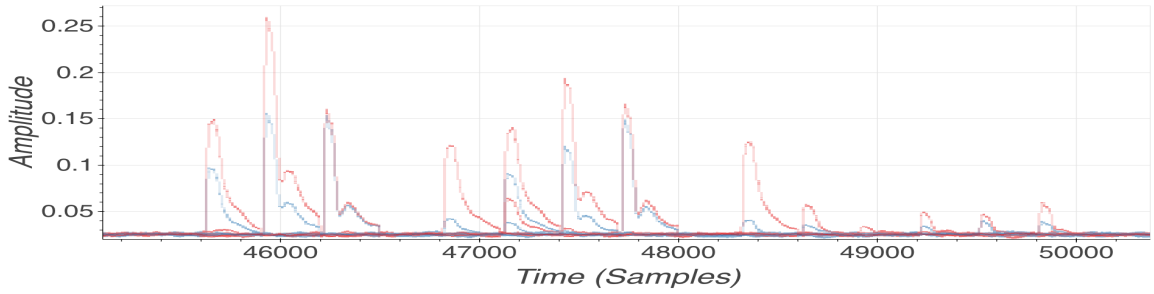

Figure 20: 2-Share PRESENT Sbox Single Operation – SNR Results – SNR Entropy Byte 0 vs. Byte 1 Zoom

5.3. A Sensitive Leakage

The above analysis studies the leakage related to the shares of the intermediate variables, it gives us a good understanding of the leakage and allows to identify the various subfunction executions. We

now look for sensitive variables (*i.e.* un-masked intermediate variables) that might leak despite the masking scheme. This analysis, throughout all simple (*i.e.* without considering Hamming Distance model) intermediate variables, showed a single leakage associate to a sensitive variable. It can be observed on the SNR results related to the bitsliced output of G ∘ G ∘ B (*i.e.* the input to A). Figure 21 displays this SNR result:

- for the whole trace (second subfigure) where the inputs to A computation, say the 16-bit variables $\{A, B, C, D\}$ are colored in red, blue, green and purple respectively. It tells us that only the input variable A is leaking.
- around the SNR peak (last subfigure) where the least significant byte of A is in red while the most significant byte is in blue. It says that this sensitive variable leaks the very same way than any other variables (the least significant byte leakage is stronger).

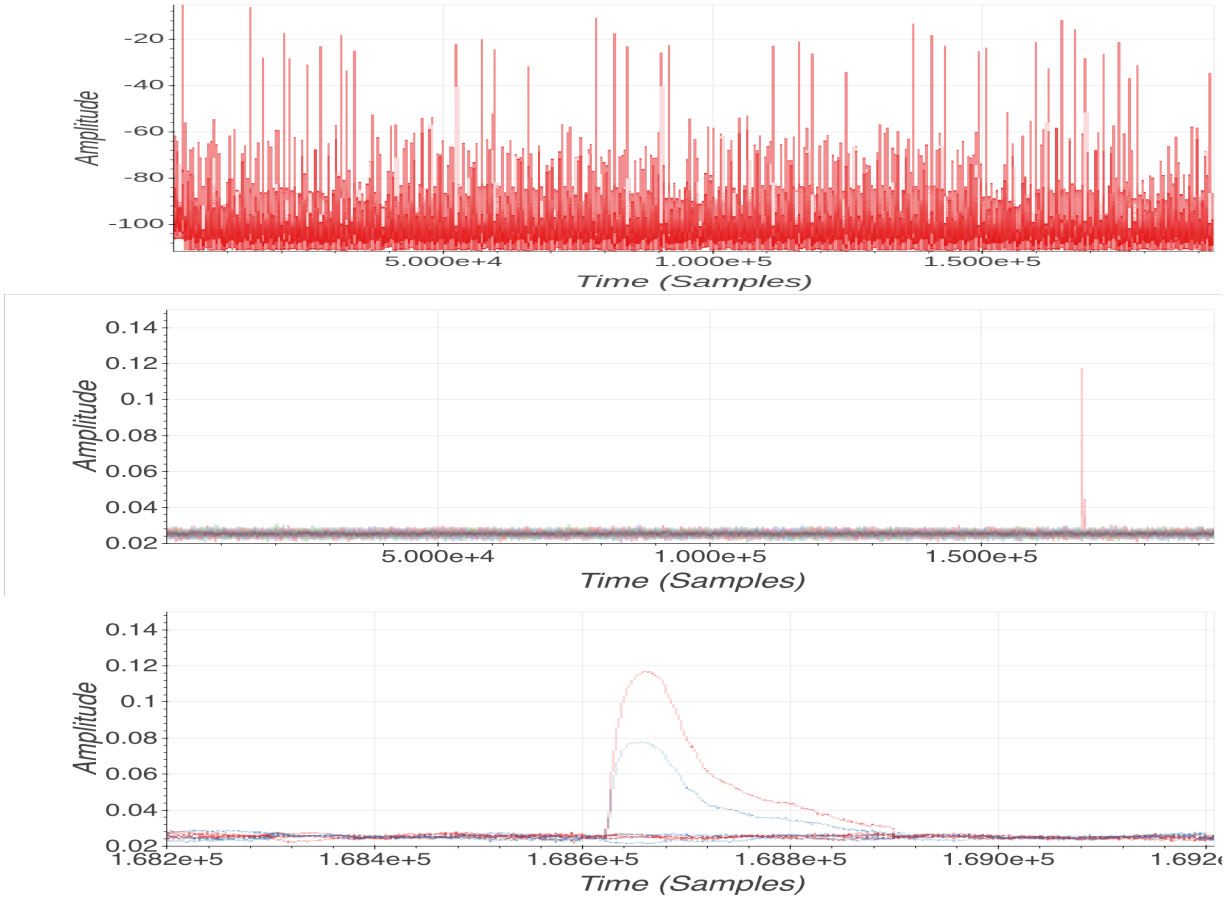

Figure 21: 2-Share PRESENT Sbox Single Operation – SNR Results – Average Reduced Trace (Top) – SNR Output G ∘ G ∘ B (Middle) – SNR Output G ∘ G ∘ B Byte 0 vs. Byte 1 Zoom (Bottom)

Thanks to the analysis of the shares in the previous section, it is easy to identify the leakage position inside the PRESENT Sbox execution. Figure 22 illustrates this identification: the leakage occurs inside the A execution, right before the second share variables are computed.

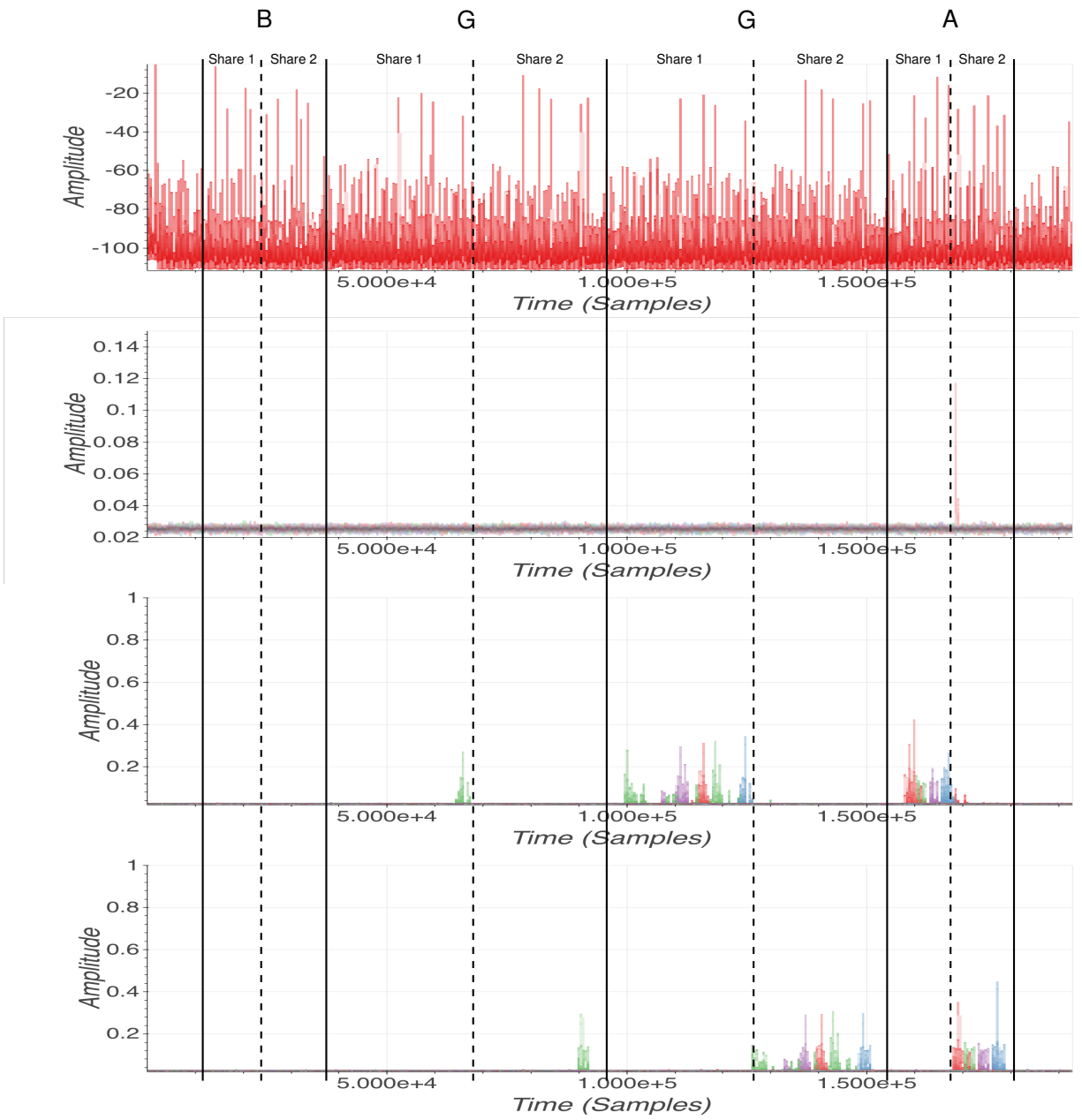

Figure 22: 2-Share PRESENT Sbox Single Operation – SNR Results – Average Reduced Trace (Top) - SNR Output G ∘ G ∘ B (Second) - SNR Output G ∘ G ∘ B Share 1 (Third) - SNR Output G ∘ G ∘ B Share 2 (Bottom)

Figure 23 zooms in around the sensitive variable SNR peak, it gives a good reason why this sensitive leakage appears: at the same time both shares of input A ($A[0]$ and $A[1]$) are leaking. The combination of these two leakages produces the leakage associated to the unmasked A. Remark that, at the leakage time position, all $A[0], A[1]$ and A have comparable leakage strength.

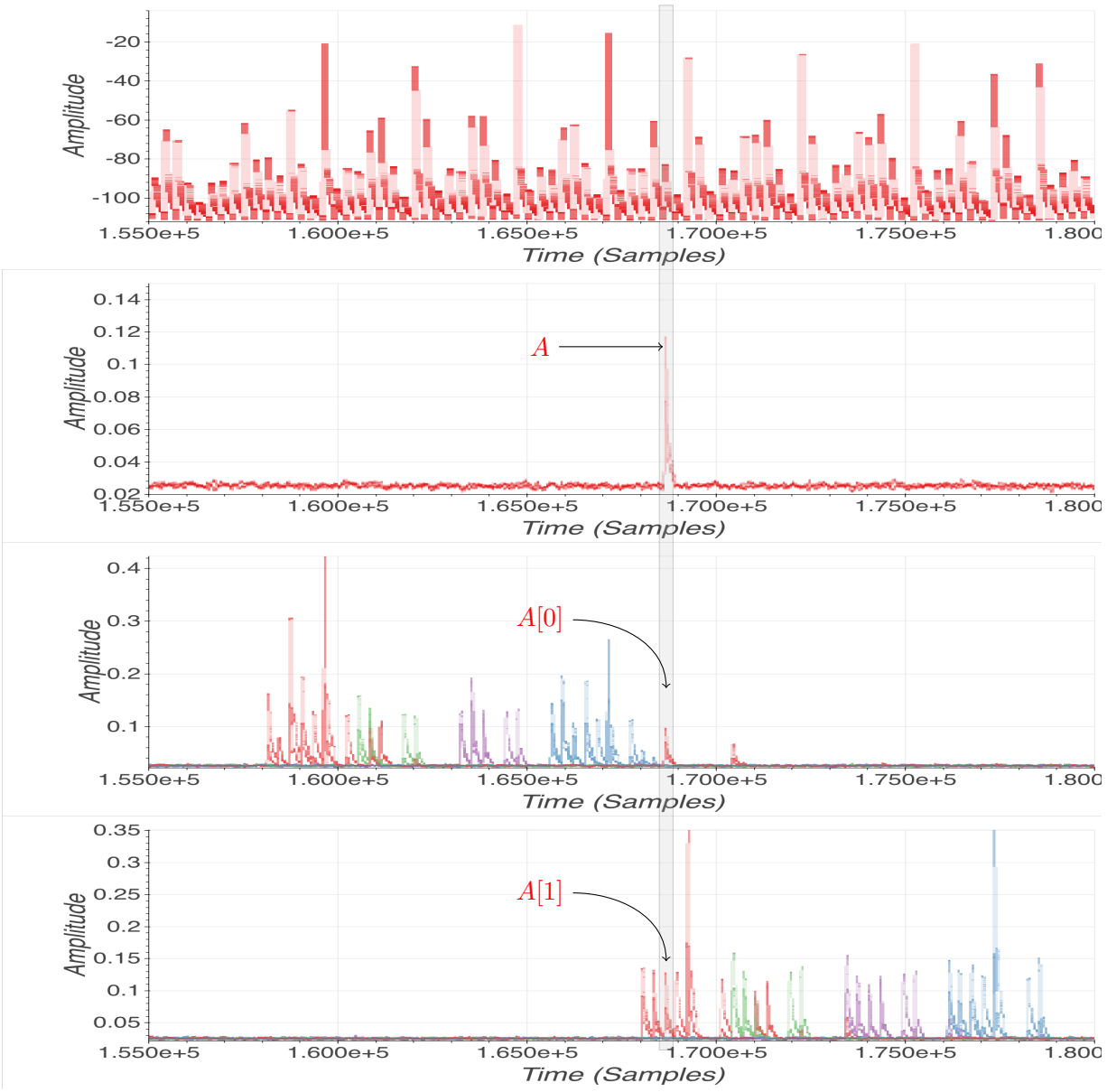

Figure 23: 2-Share PRESENT Sbox Single Operation – SNR Results Zoom Level 1 – Average Reduced Trace (Top) - SNR Output G ∘ G ∘ B (Second) - SNR Output G ∘ G ∘ B Share 1 (Third) - SNR Output G ∘ G ∘ B Share 2 (Bottom)

Now that the leakage time position is well estimated, we can look at the asm code of A to better understand why this unexpected leakage appears. The following snippet of code was extracted from the study material provided with this document [6]. With annotations, it indicates the likely position of the leakage (in red). The sensitive leakage appears during the manipulation of $A[1]$, its second share. The first share $A[0]$ is however not manipulated at this point but was loaded into register `r4` (and never erased from it) at the very beginning of A (first `LDR` instruction). The register `r4` will be loaded with a new value few instructions after the sensitive leakage occurs. We can make the hypothesis that this sequence of events creates the sensitive leakage. Without more experiments and

[6] `L4.3_evaluation_sources.tgz`

more information about the *OpenCard* IC design, it is not possible to have a more precise analysis of leakage source. However, one can safely conclude that such very specific leakages are far from being taken into account when designing a masking scheme. Our example here shows how disastrous the result can be.

It can be observed however that the unintended leakage appears due to a specific combination of events (the manipulation of `r4` containing $A[0]$ close enough to the loading of $A[1]$). A deeper analysis of this phenomenon might well conclude that this should not appear very often (and indeed we did not found another similar leakage in the rest of the implementation).

Interestingly enough, a leakage assessment of this exact implementation was done in [BGG+21] on two CM0+ MCUs (FRDM-KL82Z, STM32L073RZ) and concluded in the absence of sensitive leakages. But the *OpenCard* is actually based on a SC 100 MCU, the two architectures must have many differences (CM0+ is based on ARMv6-M architecture, while SC100 is based on ARMv4T architecture). For instance, the fact that the CM0+ possesses a 2-staged pipeline while the SC100 has a 3-staged pipeline might give some sense to this unintended leakage appearing solely on the *OpenCard*.

```
;; calcAOrder1(
;;   uint32_t *entropy,
;;   uint32_t outputs[4][2],
;;   uint32_t inputs[4][2]
;; )
;; r0 entropy ptr, r1 *outputs, r2 *inputs
calcAOrder1
        PUSH    {r4, r5, r6, lr}
        LDR     r4, [r2, #0]
        STR     r4, [r1, #16]   G[0] ← A[0]
        LDR     r5, [r2, #16]
        EORS    r5, r4
        STR     r5, [r1, #0]    E[0] ← C[0] ⊕ A[0]
        MVNS    r5, r5
        LDR     r6, [r2, #24]
        EORS    r5, r6
        STR     r5, [r1, #24]   H[0] ← C[0] ⊕ A[0] ⊕ 0xFFFFFFFF ⊕ D[0]
        LDR     r5, [r2, #8]
        MVNS    r5, r5
        STR     r5, [r1, #8]    F[0] ← B[0] ⊕ 0xFFFFFFFF
        ;; second share
        LDR     r5, [r2, #4]    ------------------------------> A[0] ⊕ A[1]
        STR     r5, [r1, #20]   G[1] ← A[1]
        ADDS    r3, r0, r1              ;; clear(opB)
        LDR     r4, [r2, #20]
        ANDS    r0, r0                  ;; clear(opB)
        EORS    r4, r5
        STR     r4, [r1, #4]    E[1] ← C[1] ⊕ A[1]
        ANDS    r0, r0                  ;; clear(opB)
        LDR     r5, [r2, #28]
        ANDS    r0, r0                  ;; clear(opB)
        EORS    r4, r5
        STR     r4, [r1, #28]   H[1] ← C[1] ⊕ A[1] ⊕ D[1]
        LDR     r5, [r2, #12]
        STR     r5, [r1, #12]   F[1] ← B[1]
        ANDS    r0, r0                  ;; clear(opA), clear(opB)
        STR     r0, [r0, #0]            ;; clear(opW)
        POP     {r4, r5, r6, pc}        ;; scrub(r4), scrub(r5), scrub(r6)
```

6. 3-Share PRESENT Sbox

This section is dedicated to the analysis of 3-share implementation of the PRESENT Sbox (see Section 3.4). We will first present the acquisition campaign as well as the resulting traces before going into the details of the leakage analysis.

6.1. Acquisition Campaign

As for the other implementations, our first experiment allows to choose the maximum number of successive PRESENT Sbox computations that can fit in a single power trace. The acquisition parameters for this experiment are detailed in Table 5, this time only 50 computations fit into the trace (*i.e.* in the oscilloscope memory). Figure 24 display the I/O trace, it shows that:

- a single PRESENT Sbox computation takes about 1.4M samples, *i.e.* 1.1 ms at 1.25GSamples/s
- the random generation between two computations takes about 9.5M samples

operation	3-Share PRESENT Sbox
equipment	PicoScope 6424E, Scaffold
inputs `entropy[i]` for $i \in \{0, \cdots, 67\}$ filled with 32 bits fresh randoms	`src[0]` and `src[1]` filled with 32 bits fresh randoms
number of operations	50
length	500 ms
sampling rate	1.25GSa/s
samples per trace	625MSamples
channel(s)	I/O, Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-220, -20]$mV
file size	1.25GB
acquisition time	about 30s

Table 5: Acquisition Parameters 3-Share PRESENT Sbox – First Tentative

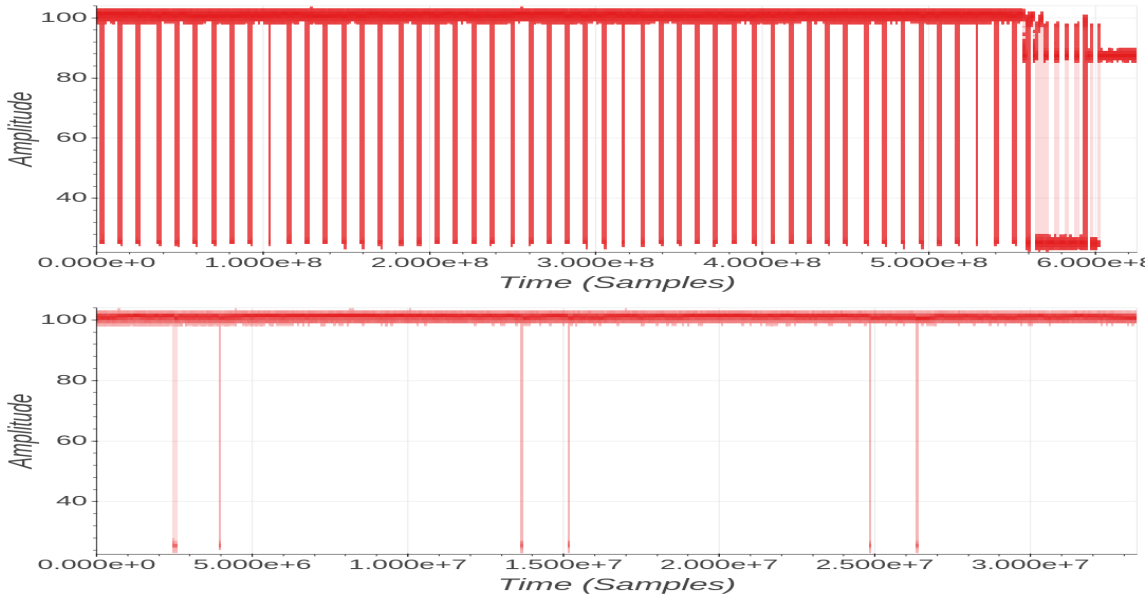

Figure 24: 3-Share PRESENT Sbox 80 Operations – I/O Trace – Full Trace (Top) - Zoom in Firsts Executions (Bottom)

As detailed in Section 5.1 for the 2-share version, we process the trace splitting online, then only storing the interesting part of the power traces. Table 6 sums up the acquisition campaign parameters, thanks to our online processing we can acquire a total of 10000×50 PRESENT Sbox

computations in about 5 days, the resulting file is about 800G Bytes. Figure 25 depicts the resulting traces and shows that no resynchronization is necessary.

operation	3-Share PRESENT Sbox
equipment	PicoScope 6424E, Scaffold
inputs `entropy[i]` for $i \in \{0, \cdots, 67\}$ filled with 32 bits fresh randoms	`src[0]` and `src[1]` filled with 32 bits fresh randoms
number of operations	10000×50
length	500 ms
sampling rate	1.25GSa/s
samples per trace	1.6MSamples
channel(s)	Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-220, -20]$mV
file size	800GB
acquisition time	about 115h

Table 6: Acquisition Parameters 3-Share PRESENT Sbox

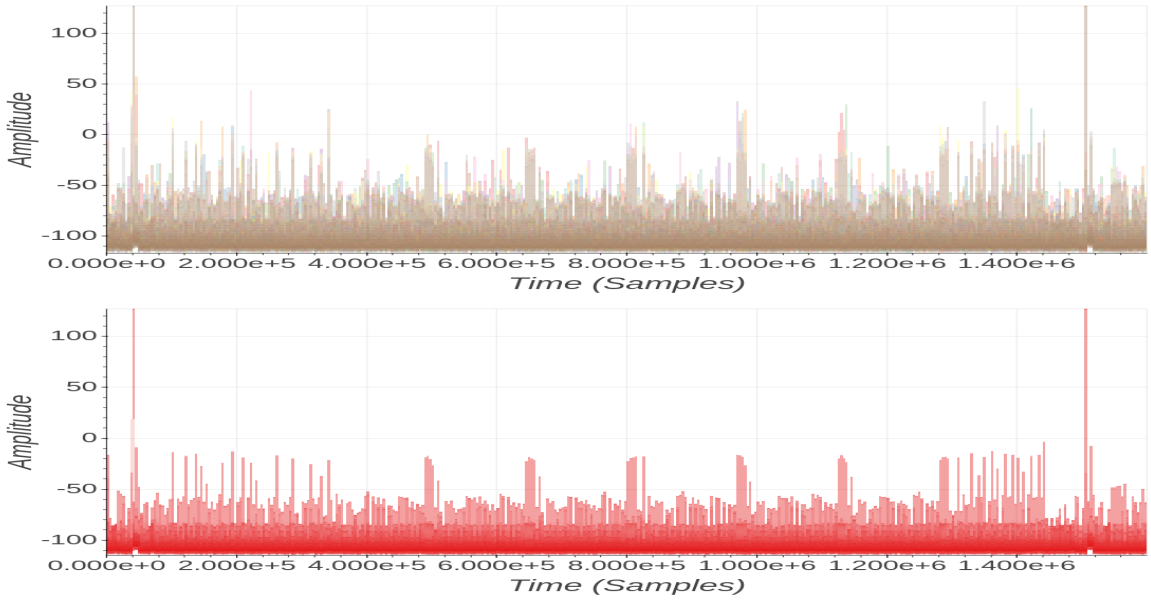

Figure 25: 3-Share PRESENT Sbox Single Operation – Power Traces – 10 Superposed Traces (Top) - Averaged over 400K Traces (Bottom)

Similarly to the 2-share implementation, the traces are further reduced to contain only the part of the clock cycles that bear side-channel leakages. The procedure is exactly the same and is summarized in Figure 26. The length of the traces is reduced from 1.6M samples to 352K samples. Figure 27 shows the resulting averaged trace over the 500K traces.

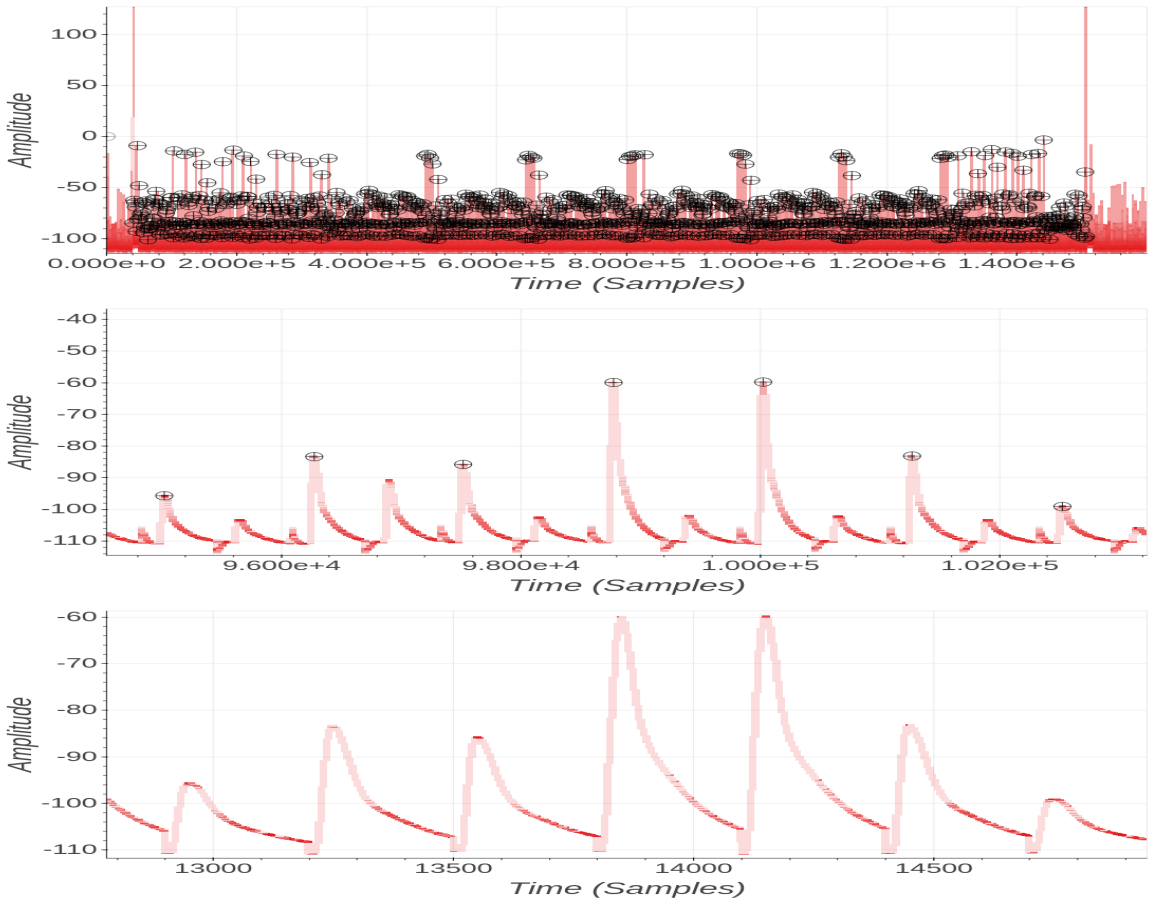

Figure 26: 3-Share PRESENT Sbox Single Operation – Power Traces with Syncro Points – Average Trace (Top) - Average Trace Zoom (Middle) - Average Reduced Trace Zoom (Bottom)

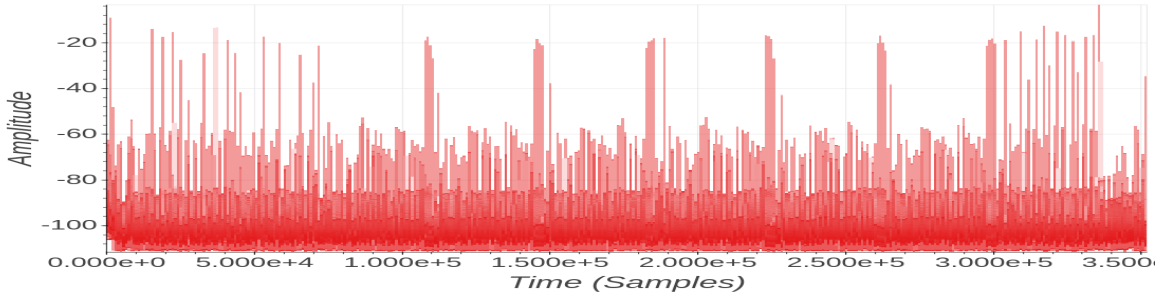

Figure 27: 3-Share PRESENT Sbox Single Operation – Power Traces – Average Reduced Trace

6.2. First-Order Leakage Analysis

Thanks to our knowledge of the masking material, we first study the leakage of the intermediate variable shares. For each input/output of the subfunctions A, B, G calls, we evaluate the two 8-bit SNR of each of the three shares. Results are depicted on Figure 28 to 32. This analysis allows a timing identification of the sequence of subfunction executions, as illustrated on the figures.

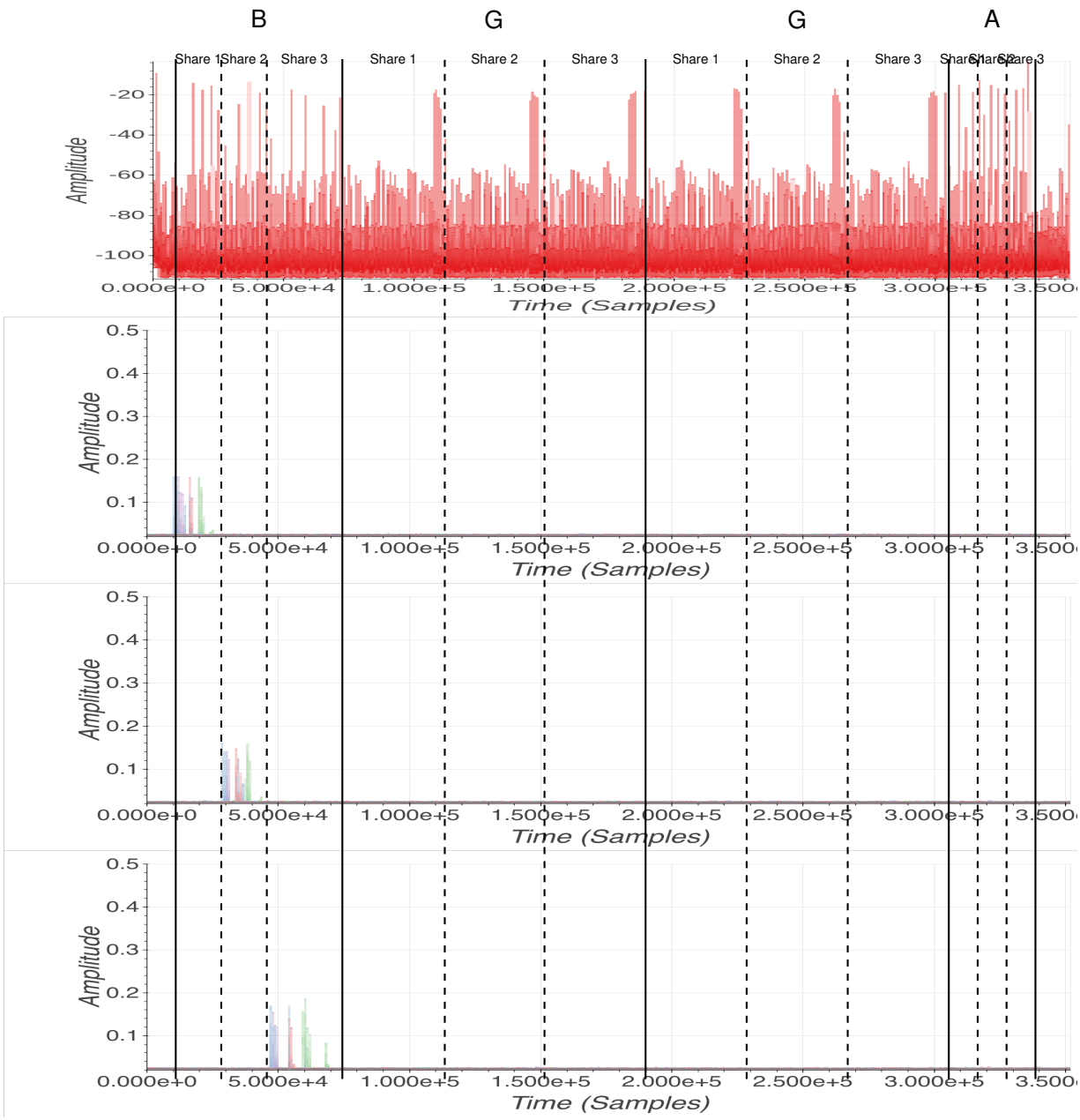

Figure 28: 3-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$– Average Reduced Trace (Top) - SNR Inputs Share 1 (Second) - SNR Inputs Share 2 (Third) - SNR Inputs Share 3 (Bottom)

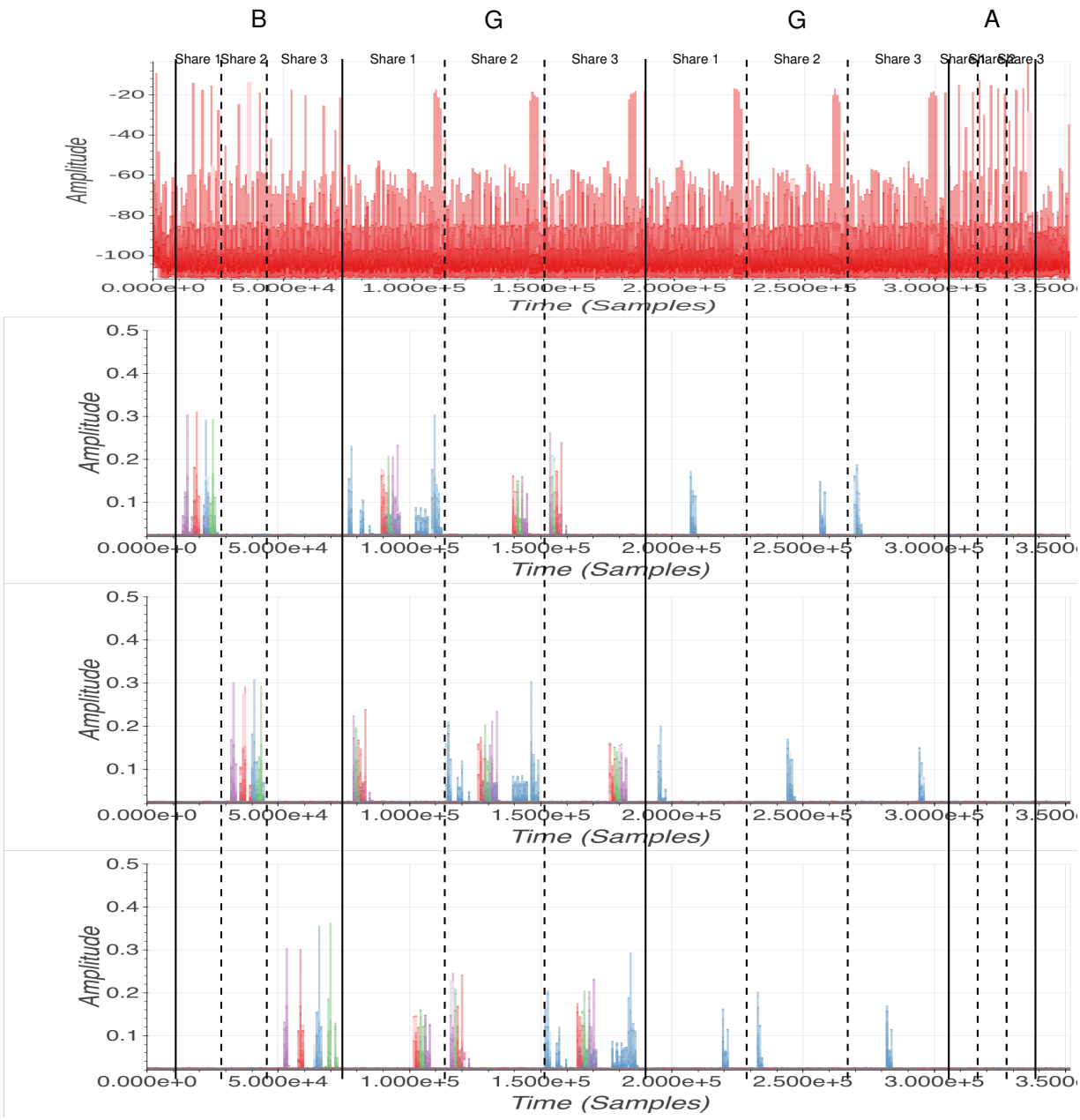

Figure 29: 3-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$– Average Reduced Trace (Top) - SNR Outputs B Share 1 (Second) - SNR Outputs B Share 2 (Third) - SNR Outputs B Share 3 (Bottom)

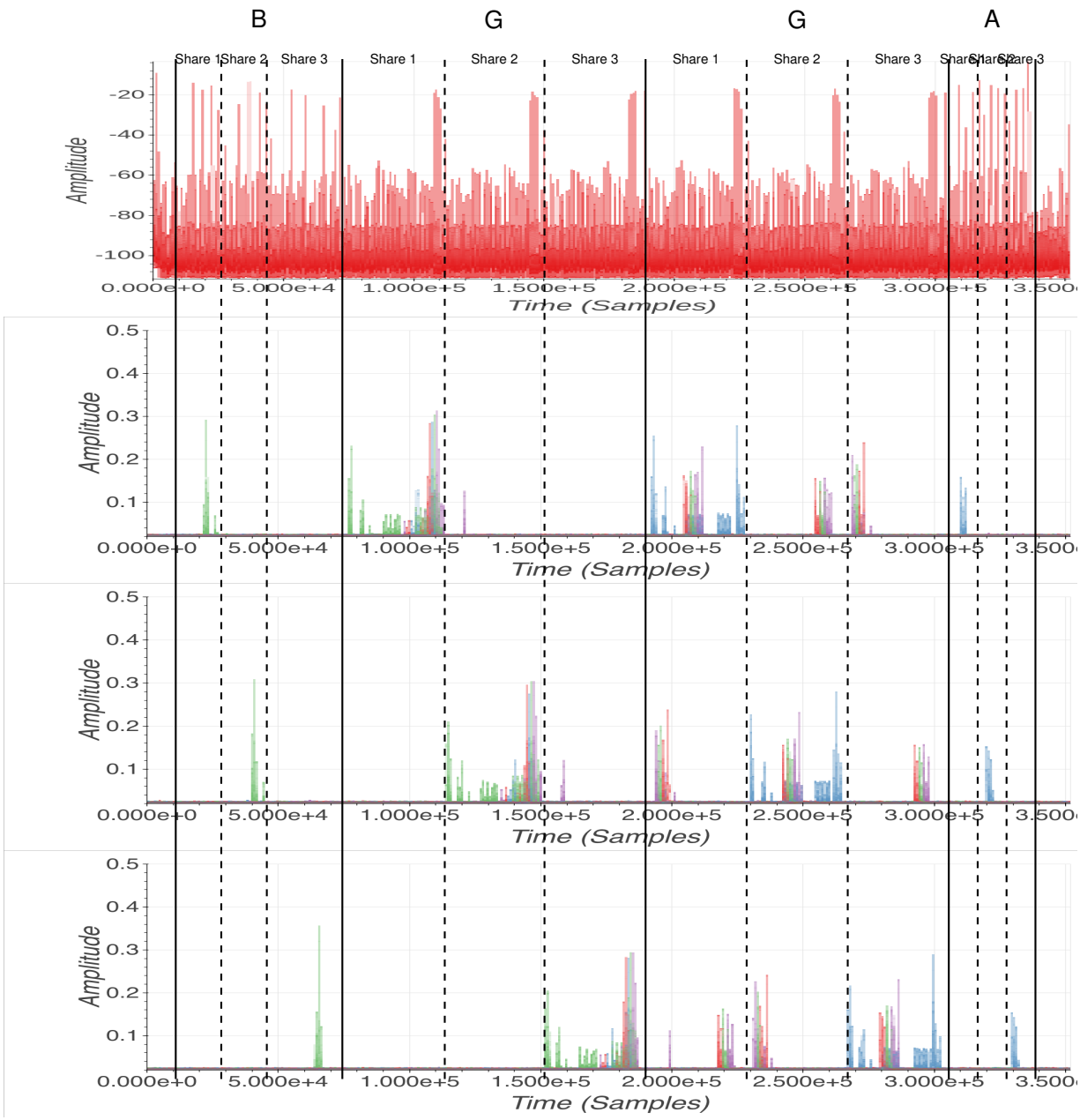

Figure 30: 3-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$– Average Reduced Trace (Top) - SNR Outputs $G \circ B$ Share 1 (Second) - SNR Outputs $G \circ B$ Share 2 (Third) - SNR Outputs $G \circ B$ Share 3 (Bottom)

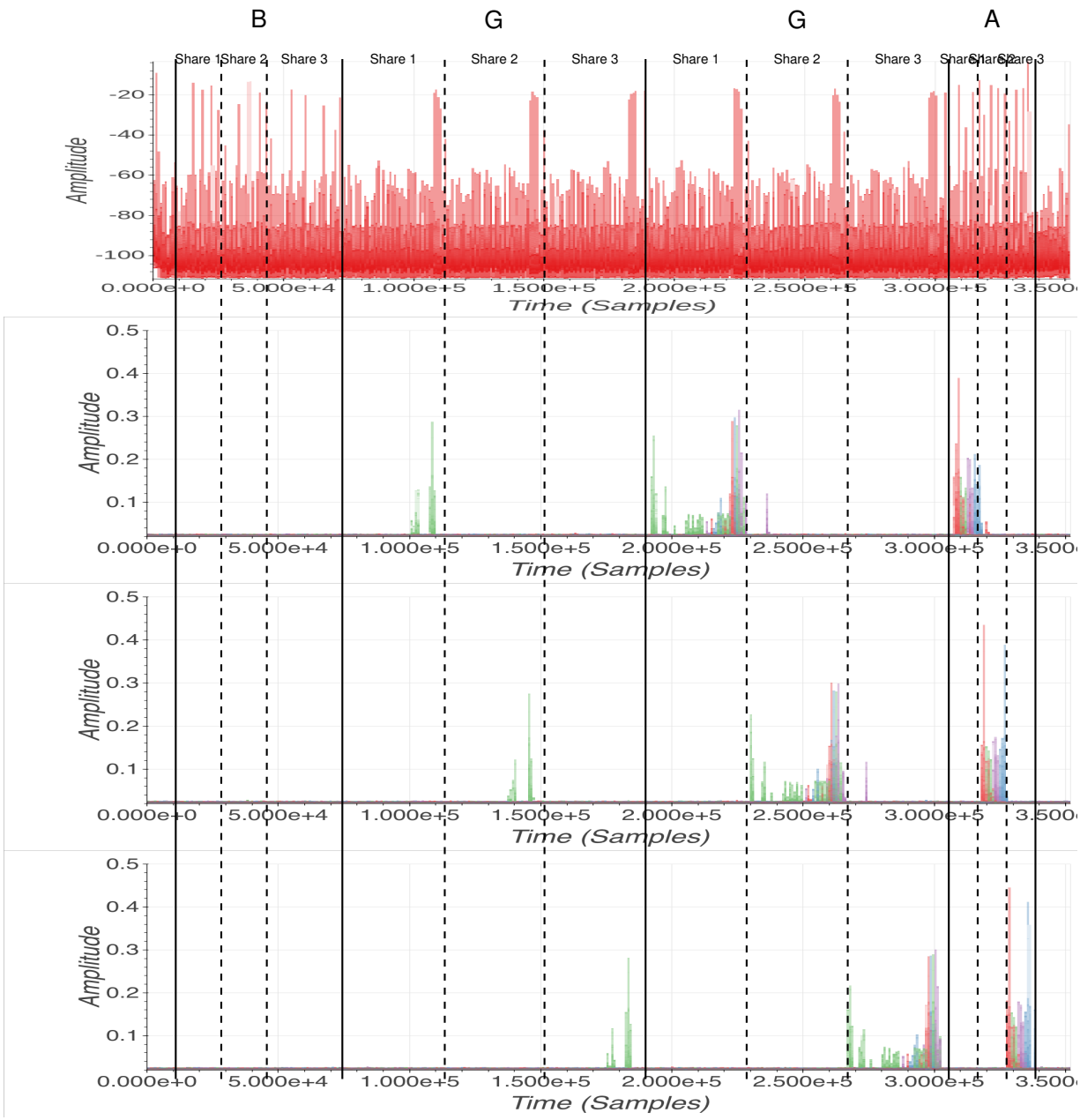

Figure 31: 3-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$– Average Reduced Trace (Top) - SNR Outputs $G \circ G \circ B$ Share 1 (Second) - SNR Outputs $G \circ G \circ B$ Share 2 (Third) - SNR Outputs $G \circ G \circ B$ Share 3 (Bottom)

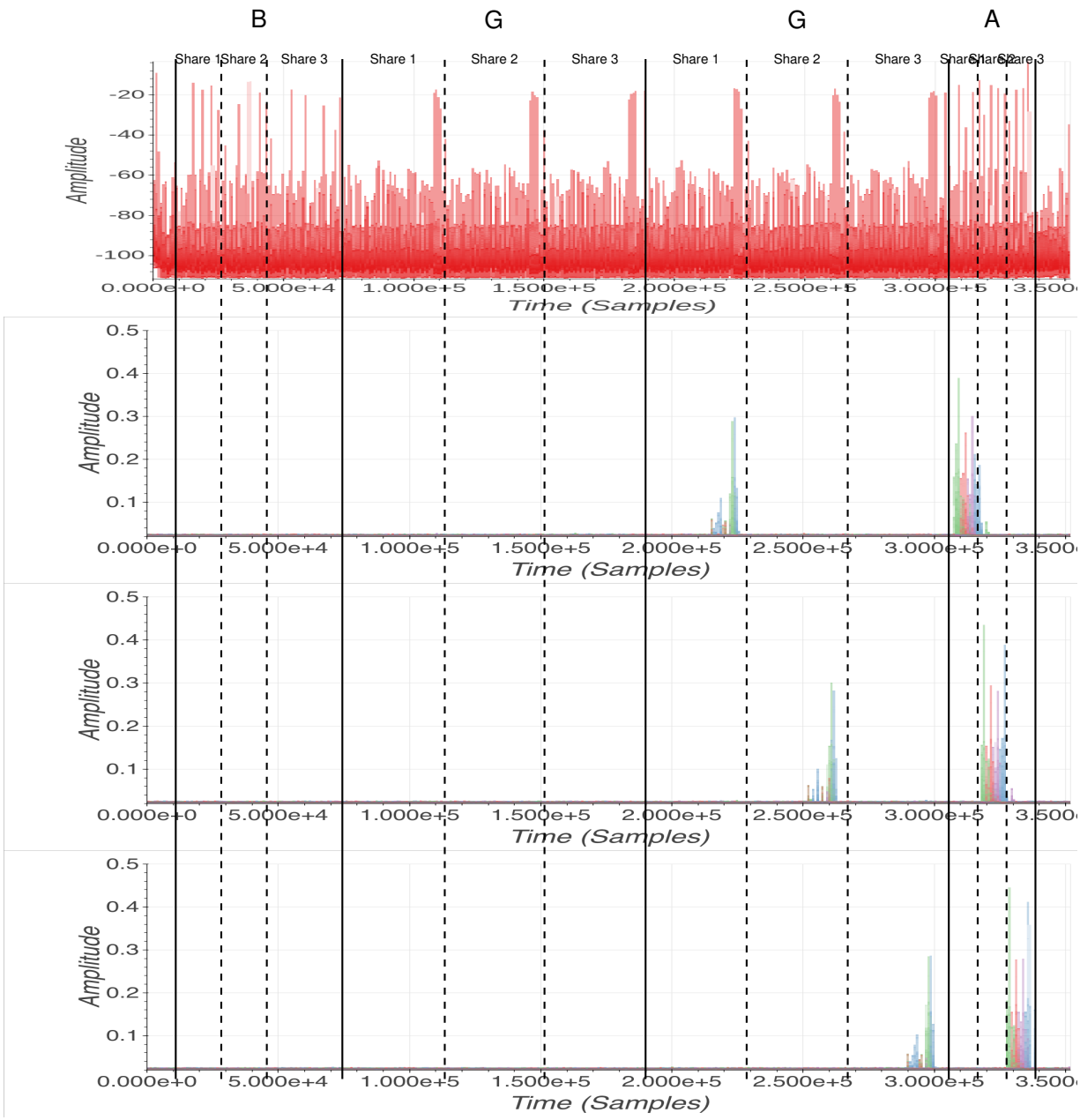

Figure 32: 3-Share PRESENT Sbox Single Operation – SNR Results $\{A, B, C, D\}$– Average Reduced Trace (Top) - SNR Outputs $A \circ G \circ G \circ B$ Share 1 (Second) - SNR Outputs $A \circ G \circ G \circ B$ Share 2 (Third) - SNR Outputs $A \circ G \circ G \circ B$ Share 3 (Bottom)

Now that the side-channel identification of the different parts of the execution is done, we study the unintended leakages, and first the unmasked intermediate variables. This study amounts to target each of the input/output variables of the subfunctions and study their leakage:

- the SNR analysis of the 4 bitsliced inputs/outputs $\{A, B, C, D\}$: these values are on 16 bits and then, as before, two 8-bit SNRs are estimated (the eight 8-bit SNRs are computed in about 1.5 hours, the whole SNR computation for all intermediate variables takes then about 7.5 hours)
- the SNR analysis of the 16 inputs/outputs of the PRESENT Sbox: these values are on 4

bits (the 16 4-bit SNRs are computed in about 3.5 hours. There is a bijection between each intermediate variables, then focusing on the inputs of B is enough to study the whole execution)

- the T-Test analysis of each bit individually of the bitsliced inputs/outputs $\{A, B, C, D\}$. There are, at each step of the computation, 64 such T-Test to compute (the 64 1-bit T-Tests are computed in about 7 hours, the whole SNR computation for all intermediate variables takes then about 35 hours)

None of the above leakage assessment could identify any significant sensitive leakage (contrarily to the 2-share implementation, see Section 5.3).

6.3. Second-Order Leakage Analysis

Looking for 2nd-order leakages, and then fully validate/invalidate the practical robustness of the 3-share implementation on the *OpenCard* is a bit more complicated than looking for 1st-order leakages. And, as we will see, a complete analysis is not possible within the scope of this study (for time and/or computational power limitations).

The first thing to check is *univariate* 2nd-order leakages. To this end, we simply have to re-compute the SNRs listed above on the side-channel traces where the 2nd statistical moment has been exposed: the averaged trace is subtracted from each trace and the result of squared (sample-wise).
As for the 1st-order leakage assessment, none of the SNR results show significant sensitive leakage.

Now, considering *multivariate* 2nd-order leakages, we should conduct the above analysis over the traces where all bi-variate 2nd statistical moments are exposed, *i.e.* where all pairs of samples are combined (usually using the centered product combination function). This re-combination could be done by only considering small windows where the 3 shares of an intermediate variable are manipulated. But even that way, this re-combination would inflate the trace length so much that the SNRs computation would be done in a reasonable amount of time (see the timings for the SNR computation in previous Section, they should be multiplied by a factor 10K to 100K for reasonable window sizes).

In our setup however, we can simplify the analysis: let us consider an intermediate variable X, shared as $(X[0], X[1], X[2])$, we can study the leakage related to the combination of any pairs of shares. For $0 \leq i < j \leq 2$, if a 1st-order leakage related to $X[i] \oplus X[j]$ is found, then a bivariate 2nd-order leakage can be easily built from the time samples where the last share leaks. This trick reduces the spectrum of 2nd-order leakages that can be detected but it would cover unintended leakages such as the one found in Section 5.3.
The full SNRs computation for each intermediate variable and each pair of shares was conducted (for all 8-bit SNRs, 4-bit SNRs and 1-bit T-Tests): no significant leakage was detected. This experiment concludes the leakage assessment of the 3-share implementation.

As mentioned in Section 5.3, the observed sensitive leakage in the 2-share implementation appears because the leakage model of the chip was not known by the developers, but the appearance of this unexpected collision of register values might not happen very often. And as a matter of fact, it does not seem to occur in the 3-share implementation.

References

[Bei] Beijing ChipCity Technology Co., Ltd. . CC32RS512 Contact Smart Card Chip User Manual.

[BGG+21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):189–228, 2021.

[Led] Ledger SAS. Scaffold. https://github.com/Ledger-Donjon/scaffold.

[Mar03] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[Pic19] Pico Technology. PicoScope 6000E Series datasheet. https://www.picotech.com/download/manuals/picoscope-6000e-series-data-sheet.pdf, 2019. [online; accessed 04-Nov-2020].

[Wel47] B. L. Welch. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1/2):28–35, 1947.