VERISICC
Deliverable L3.3
Component Characterization

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

The deliverable L3.3 is the output of VERISICC task T3.3 focusing on the side-channel characterization of a chip. Our main goal is to acquire real side-channel leakages of a specific chip and study its nature as deeply as possible. As far as this is possible, the information gathered will help the design of efficient masking schemes adapted to this specific chip.

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).

The recent development of automated masking tools makes it possible to plug real leakage information inside the design of a masking scheme itself. The masking scheme will then be re-generated for each target chip given its *real* leakage information. For a given target chip this approach would result in schemes more robust or more efficient, or even both at the same time.

However, the side-channel characterization of a chip is no easy task. The scarce literature on the subject (see *e.g.* [MOW17]) shows it well. The study faces two big problems: (1) the complexity of modern microcontrollers (in terms of number of registers, memory size, instruction sets, etc.) makes a comprehensive side-channel caratherization barely possible and (2) the Hardware design details of the vast majority of microcontrollers are not publicly available which greatly complicates the side-channel analysis and its interpretations.

In the present study we will focus on the *OpenCard*, details of the target chip as well as the setup work for side-channel acquisitions are provided in Section 2. To precisely define the scope of the study, we identified a list of target functions ranging from single instructions to small gadgets useful in the implementation of masking schemes for symmetric cryptography. The targeted functions are listed in Section 3 and the source code can be found in the study material provided with this document [1].

The side-channel setup (especially the trigger setup) as well as the analysis of several millions of traces showed to be more time consuming than expected and only a small part of the target implementations were actually studied. We focused on the study of single instructions (Section 4) as it is the core of the characterization task, then we selected a single gadget to analyse it in depth (Section 5). The main results and conclusions drawn are sum up below:

- The study of single instructions (surrounded by a fine grain trigger system), detailed in Section 4, was conducted on three instructions (namely `adds`, `eors` and `ands`) on 32-bits registers.
  - All instructions and register choices show significant leakage of the least significant byte of the 32-bit inputs/outputs. The 3 most significant bytes seem to induce leakages too weak to be captured with 2M traces.
  - Observed leakages are well modelled by the Hamming Weight.
  - The register choice shows to have an impact on the side-channel leakage, we could identify a configuration when the instruction's second argument is stored in register `r1` that could be of interest for a masking scheme developer.

[1] `L3.3_caracterisation_des_composants_sources.tgz`

  - The study of transition leakages from two consecutive instructions over two sets of independent registers confirms the necessity to take transition leakages into account when designing masking schemes for the *OpenCard*.

- The study of a masked bit-wise AND gadget with 4 shares, detailed in Section 5.
  - In this context, a strong leakage related to all 4 bytes of the manipulated registers can be captured (by opposition to the single instruction case where only the least significant byte is leaking its value). We could not explain this difference and let for future work the analysis of this phenomenon.
  - The observed side-channel leakage does not bear (as far as we can tell with 2M traces) information on the xor difference of several bytes inside a single register. Hence, storing different shares of a sensitive variable in a single register does not seem to be an issue for the *OpenCard*.
  - Our high-order analysis of the masked bit-wise AND gadget with 4 shares does not show sensitive leakage. Considering an attacker willing to acquire several millions of side-channel power traces, the analysis shows that this implementation is robust against a non-profiled attacker.

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.

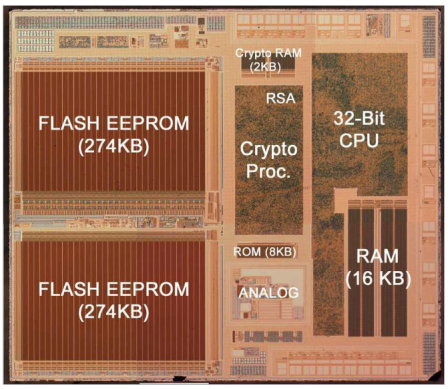

Figure 1: General view of the die after sample opening; mag 50X

The code for this characterisation 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 28MHz configuration is chosen for the study to reduce to acquisition time.

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 hardware 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 characterisation (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*.

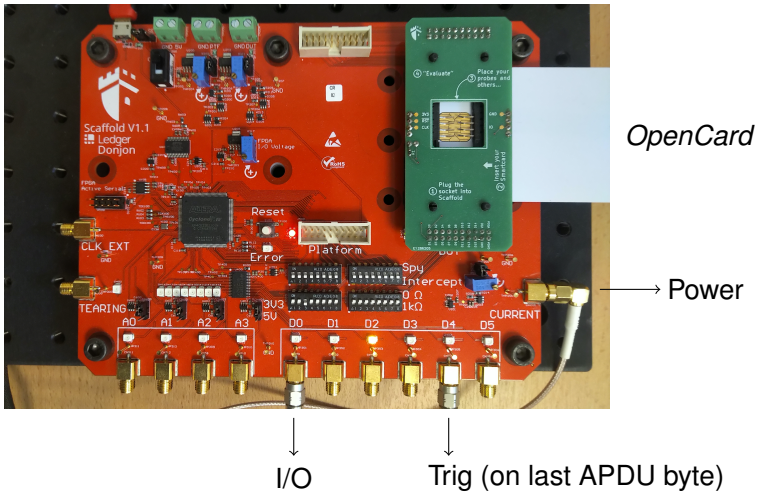

Figure 2: General View of the Scaffold Setup

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 single (or few successive) instructions runs which makes the Scaffold trigger functionality useless:

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

- 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 program return value and exit code.

Here is the detailed procedure:

- at card initialisation:
  - 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 target execution: set register GPIODAT1 to 0xFFFF00FF (drop I/O PIN to low level, see [Bei, 10.2.3])
  - after target execution: set register GPIODAT1 to 0xFFFFFFFF (raise I/O PIN back to high level, see [Bei, 10.2.3])

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 macros are used to get a square signal around a target execution: `TRIG00` is run right before the execution and `TRIGFF` is run right after.

```
;; GPIO P4~P5 data register
#define GPIODAT1 0xF8C08
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Macro TRIG00
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    MACRO
    TRIG00 $tmp
#ifndef NO_TRIG
    ldr $tmp, =GPIODAT1
    movs r12, #0xFFFF00FF
    str r12, [$tmp]
#endif
    MEND
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Macro TRIGFF
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    MACRO
    TRIGFF $tmp
#ifndef NO_TRIG
    ldr $tmp, =GPIODAT1
    movs r12, #0xFFFFFFFF
    str r12, [$tmp]
#endif
    MEND
```

An example of the trigger mechanism is given in the following asm code, the target execution is a single `adds` instruction.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EXAMPLE CODE ADD Instruction
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
adds_r0r1
    TRIG00 r3
    adds r0, r1
    TRIGFF r3
    bx lr
```

Figure 3 depicts the side-channel acquisition of the ADD instruction execution. The red signal is the I/O line while the blue signal is the power consumption of the DUT. We identified by a gray box, the approximate area where the target instruction is actually executed.

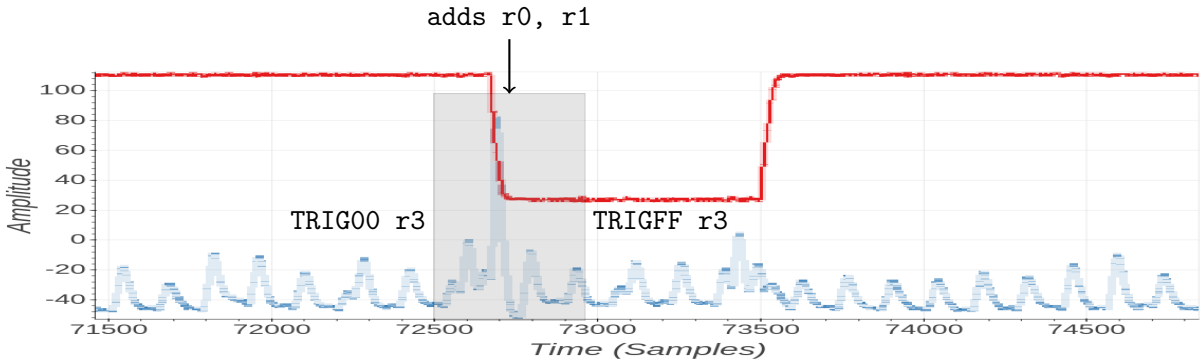

Figure 3: First Power Trace on ADD Instruction

3. Implementations

We list here all functions implemented on the OpenCard for the characterization study. All sources can be found in the study material provided with this document [4].

[4] `L3.3_caracterisation_des_composants_sources.tgz`

3.1. Internal Pseudo-Random Generator

Pseudo-random values are generated with Marsaglia's `xorshift96` pseudo-random generator [Mar03], initialized with a 48-Byte seed.

Practically, a 256-Byte buffer is filled with pseudo-random at startup, from which random are read when needed. Every 256 bytes of random, the buffer is filled with 256 bytes of fresh randoms.

3.2. Whitening

Whitening functions are developed in assembly. They can be called from C code, e.g. between calls to basic instructions or gadgets described below.

- `whiten_scratch_registers`: set scratch registers `r0`, `r1`, `r2`, `r3`, `r12` to 0, flush pipeline and wait for all memory accesses to be done.
- `whiten_full_stack`: flush pipeline, complete all memory accesses and set 0 in stack. The number of bytes to clean is defined manually with a preprocessor macro.
- `whiten_16B_stack`: same as `whiten_full_stack` where 16 B of stack are cleaned.

3.3. Basic Instructions

Basic instructions are assembly functions, `instr_rirj`, where `instr` is either `adds`, `eors` or `ands`, and $0 \leq i < j \leq 7$. The function `instr_rirj` performs the instruction `instr` using registers `ri` and `rj` as input. The result is initially in `ri`, but is moved to `r0` in order to be returned and accessible from C code.

3.4. Gadgets

The following gadgets are provided, developed in assembly.

- `mult_gf256_log_alog`: multiplication in Rijndael field using log/alog tables: given $a, b \in$ GF(256), it returns $a \cdot b$.
- `mult_2_nosec`: insecure multiplication in Rijndael field on 2 shares, without randomness. Given $(a_0, a_1), (b_0, b_1) \in$ GF(256) $\times$ GF(256), it returns (c_0, c_1) such that $c_0 \oplus c_1 = (a_0 \oplus a_1) \cdot (b_0 \oplus b_1)$.

  More precisely,

  - $c_0 = a_0 \cdot b_0$,
  - $c_1 = a_1 \cdot b_1 \oplus (a_0 \cdot b_1 \oplus a_1 \cdot b_0)$,

  where " $\cdot$ " is the multiplication `mult_gf256_log_alog`.
- `mult_2`: ISW multiplication on 2 shares. Given $(a_0, a_1), (b_0, b_1) \in$ GF(256) $\times$ GF(256), it returns (c_0, c_1) such that $c_0 \oplus c_1 = (a_0 \oplus a_1) \cdot (b_0 \oplus b_1)$.

  More precisely,

  - $c_0 = a_0 \cdot b_0 \oplus r$,
  - $c_1 = a_1 \cdot b_1 \oplus [(r \oplus a_0 \cdot b_1) \oplus a_1 \cdot b_0]$,

where r is a random in GF(256) and " $\cdot$ " is the multiplication `mult_gf256_log_alog`.

- `square_2`: squaring on 2 shares. Given (a_0, a_1), it returns $(c_0, c_1) = \left(a_0^2, a_1^2\right)$, where the square is computed with `mult_gf256_log_alog`.
- `refresh_2`: mask refreshing on 2 shares. Given (a_0, a_1), it returns $(c_0, c_1) = (a_0 \oplus r, a_1 \oplus r)$, where r is a random in GF(256).
- `and_8bits_4shares`: computes a sharing (4 shares of 1 B each) of `a&b`, each input being given by 4 shares of 1 B each, concatenated in one 4 B register. This implementation comes from [BDF+17].
- `and_8bits_4shares_1reg`: computes a sharing (4 shares of 1 B each) of `a&b`, each input being given by 4 shares of 1 B each, each share being in a separate register from the others.

Some levels of whitening can be defined, in order to clear registers and memory handling temporary variables before reusing them.

3.5. Complex Function: The Inverse in GF(256)

The inverse computation in the Rijndael field GF(256) relies on the `square_2`, `mult_2` and `refresh_2` functions described above. The inverse of $a \in$ GF(256) is obtained by computing successively

- a^2 with a squaring
- a^3 with a multiplication, after refreshing a
- a^{12} with 2 successive squarings, after refreshing a^3
- a^{15} with a multiplication
- a^{240} with 4 successive squarings, from a^{15}
- a^{252} with a multiplication
- a^{256} with a multiplication

4. Leakage Analysis of Basic Instructions

4.1. Introduction

As explained in Section 3.3, we will study the side-channel leakage of three instructions (namely `adds`, `eors` and `ands`) on registers `ri` and `rj` for $0 \leq i < j \leq 7$.

4.2. Single Instruction on 32 bits

In the first experiment, only `r0` and `r1` are used by the three instructions, we focus on the instructions themselves. The acquisition parameters are detailed in Table 1.

operation	`adds`/`eors`/`ands` `r0` `r1`
equipment	PicoScope 6424E, Scaffold
inputs	`r0` and `r1` filled with 32 bits fresh randoms
number of operations	100000
length	50us
sampling rate	1.25GSa/s
samples per trace	625KSamples
channel(s)	I/O, Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-320, 80]$mV
file size	125GB
acquisition time	about 2.5 days

Table 1: Acquisition Parameters Single Instructions `r0`-`r1`

Each of the 100000 operations of this acquisition campaign is constituted of 21 calls to the three instructions `adds`/`eors`/`ands`, as follows (the asm macro for each instruction can be found in the study material provided with this document [5]):

```
GET_RAND_32 (randa); // get the next 4 bytes of random_buffer
GET_RAND_32 (randb); // get the next 4 bytes of random_buffer
a = randa;
b = randb;

adds_r0r1(a, b);

a = randa;
b = randb;

eors_r0r1(a, b);

a = randa;
b = randb;

ands_r0r1(a, b);
```

We then ended up with about 2M executions for each instruction. Each instruction call being surrounded by a trigger signal on the I/O channel, the identification of the 2M sub-traces for each instruction is trivial. Moreover, the clock stability allows to bypass the otherwise necessary traces re-synchronization. Figure 4 displays 100 superposed sub-traces among the total of 6M sub-traces. The vertical dashed lines identify the trigger position whereas the gray box pinpoints the expected area of the instruction execution. The 100 sub-traces relate to all three target instructions, it is easy to see that `adds`, `eors` and `ands` take the same time to execute (1 cycle in ARM instruction set).

[5] see the `basic_instructions.s` asm file.

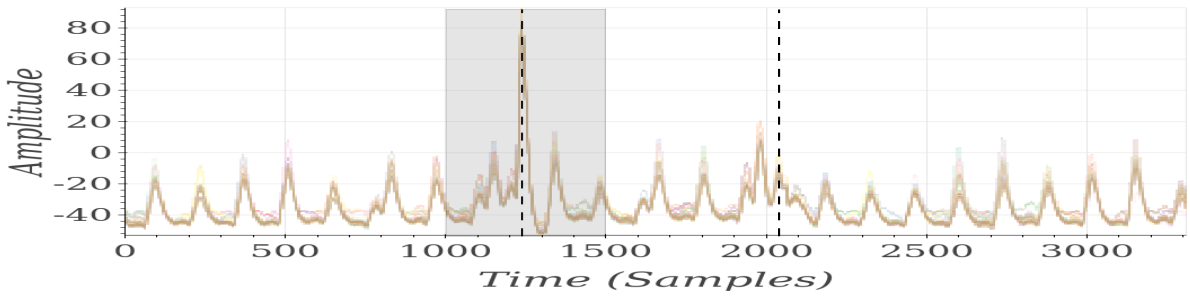

Figure 4: Single Instruction - 100 Superposed Power Traces

4.2.1. Leakage at Byte Level

The following Figures 5, 6 and 7 show the SNR computation results for the three instructions respectively (`eors`, `adds` and `ands`). In each figure, the 100 superposed traces are recalled in the first sub-figure, then three sub-figures relate to the SNR results on the first input value (a on 4 bytes stored in `r0`), the second input value (b on 4 bytes stored in `r1`) and the output value (c stored in `r0` after the instruction execution).

For each target variable on 4 bytes, 4 SNRs are estimated independently on each of the 4 bytes assuming no specific leakage model, *i.e.* the SNR is computed over the 256 possible values taken by a byte. The 4 SNR results for each target variable are displayed on the same sub-figure, appearing respectively red, blue, green and purple from the least significant Byte to the most significant Byte. Let us remark that the output of the `adds` instruction impacts the carry flag, hence adding a bit to the output value. The SNR on the carry bit value is then added to Figure 6 last sub-figure (appearing in orange).

Let us summarize what can be observed from the SNR results:

- What appears clearly on Figures 5, 6 and 7 is the fact that only the least significant byte shows significant leakage in the identified area, the three most significant bytes seem not to leak at all. This is rather unexpected and we could not explain this phenomenon[6] without a precise understanding of the underlying Hardware design of the DUT.
- The SNR of the 4-byte output c of instruction `and` (Figure 7 last sub-figure) shows a noise level higher that the other variables. This is expected and simply due to c bias: its bytes do not take all values in $[0, .., 255]$ with the same probability. For similar reasons, the LSB of c (the red curve) shows strong leakages before it is actually computed. In fact, the value $a\&b$ directly depends on value a (whatever the value of b) and b. Therefore, the strong SNR related to c before its computation corresponds actually to the manipulation of a and b.
- For all instructions, it is pretty clear that the value stored in register `r0` leaks more often than the one in register `r1` (this is particularly visible before the instruction execution).

[6] A intensive effort was devoted to find a *bug* creating this result in the *OpenCard* code and statistical treatment, without success.

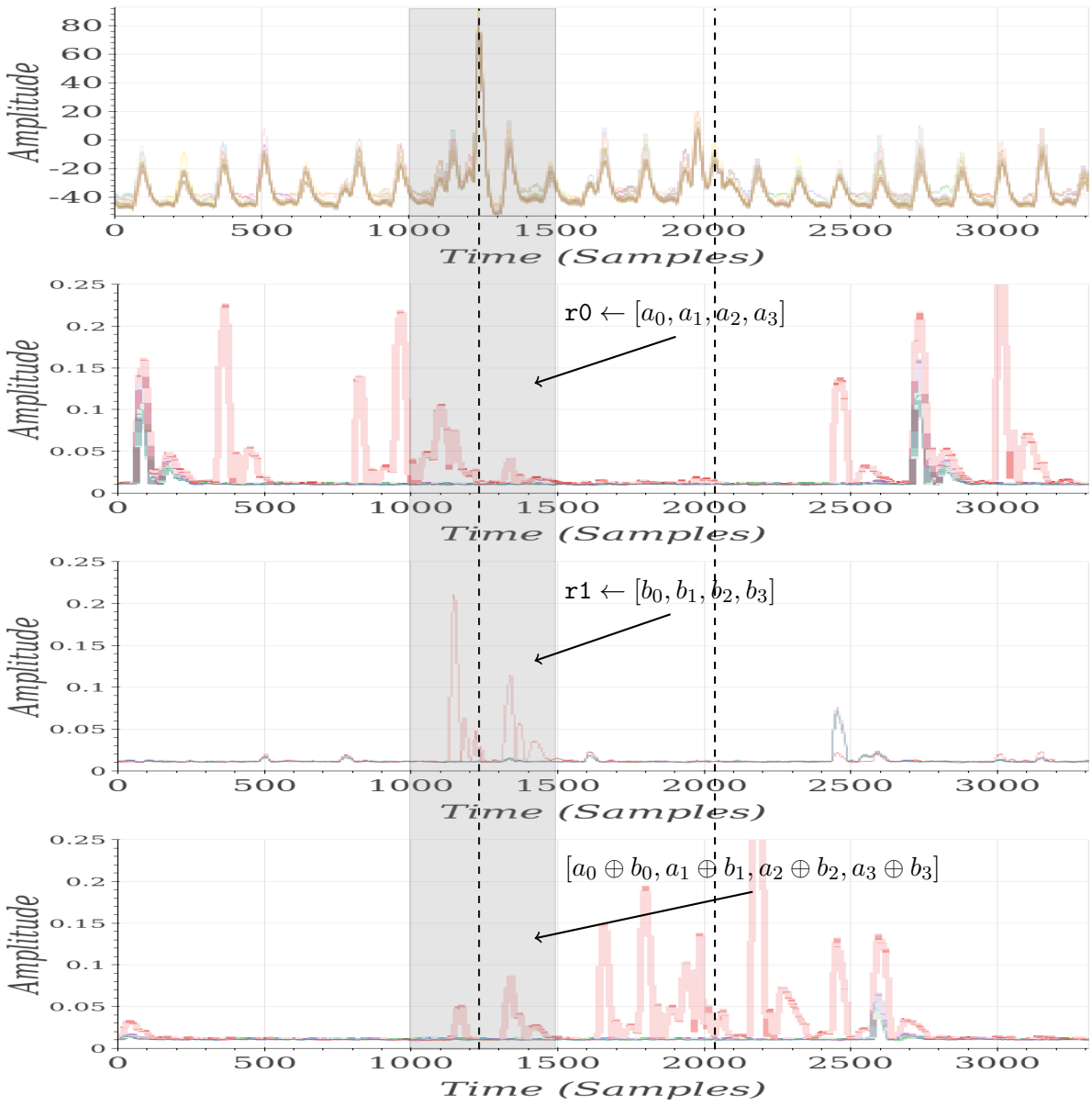

Figure 5: SNR Results – Instruction `eors r0, r1`

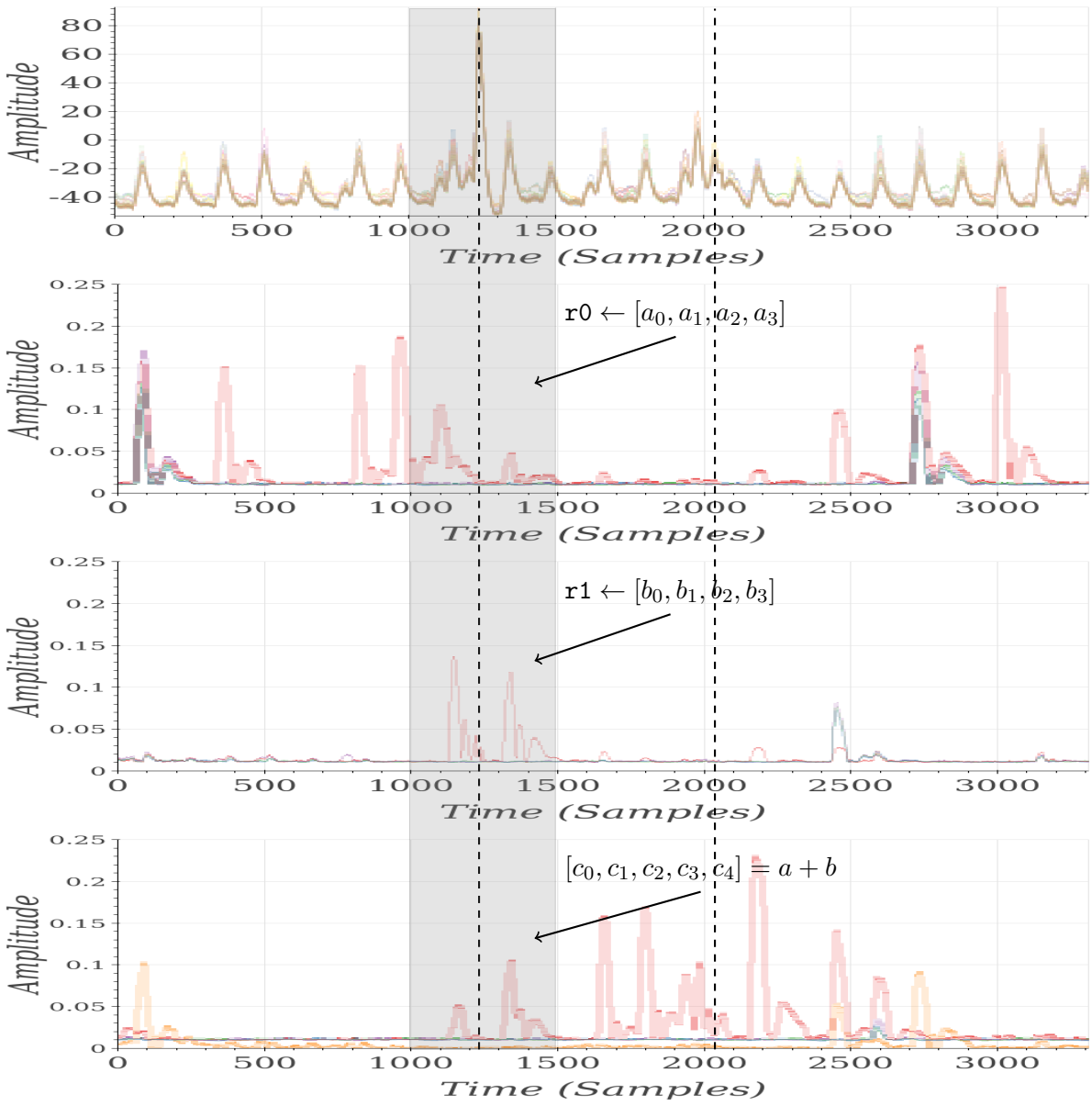

Figure 6: SNR Results – Instruction `adds r0, r1`

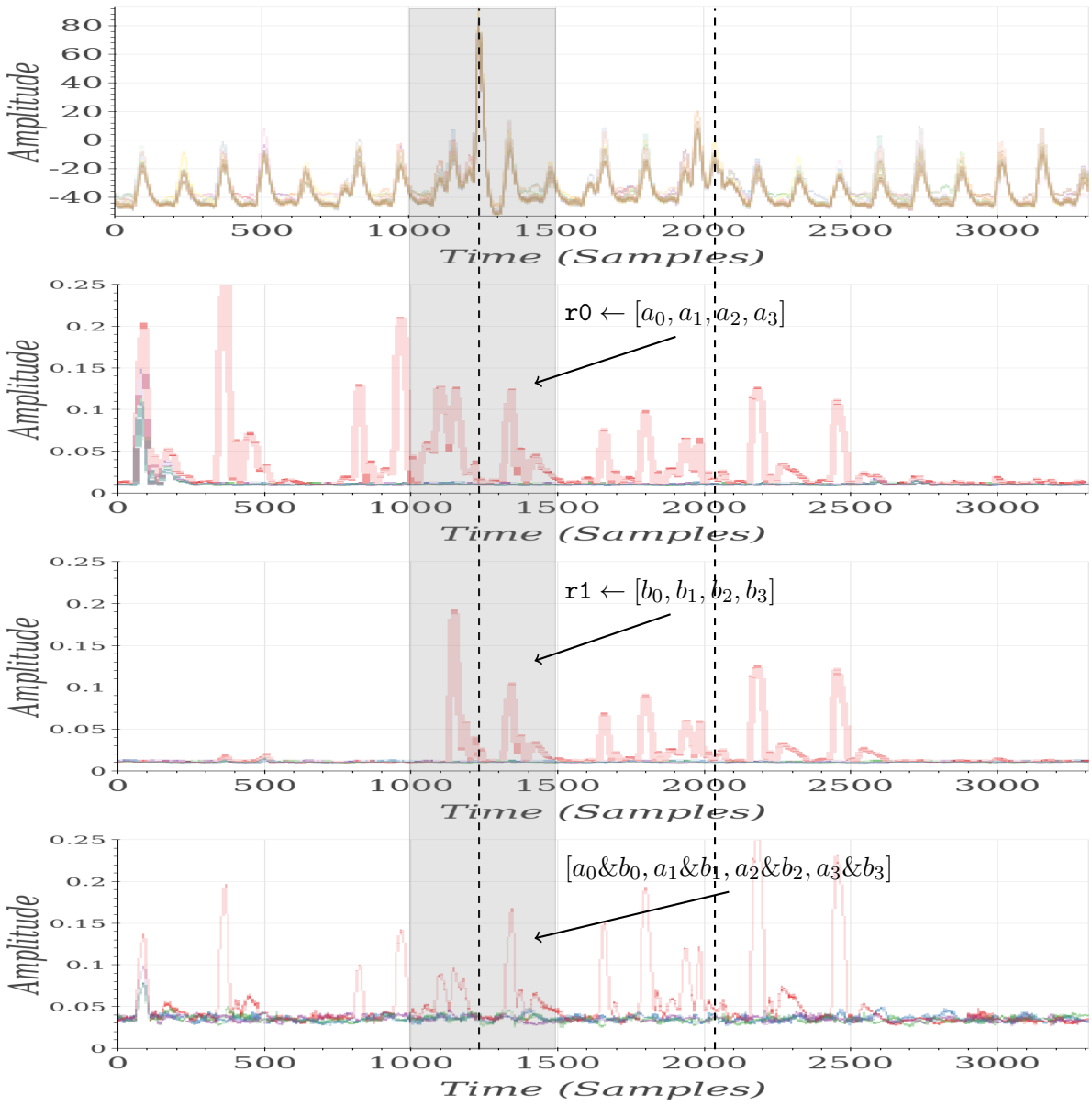

Figure 7: SNR Results – Instruction `ands r0, r1`

To precisely understand where the instruction is actually executed, we display in the following figure 8, the SNR results (on the LSB only since the other bytes do not seem leak) inside the area identified around the trigger signal. Figure 8 provides, for each instruction, the SNRs on variables a, b and c in the same sub-figure (a is red, b is blue and c is green), allowing to better visualize the leakage timings. Apart from the `and` instruction where the leakage of c cannot be clearly differentiated from the one related to a or b, the SNRs peaks between time samples 1100 and 1200 (*i.e.* within a single cycle of the DUT) show the manipulation of a and b closely followed by the manipulation of c. This is exactly what we are expecting from the instruction *Execute* stage[7].

[7] the DUT possesses a 3-stage pipeline (Fetch, Decode, Execute)

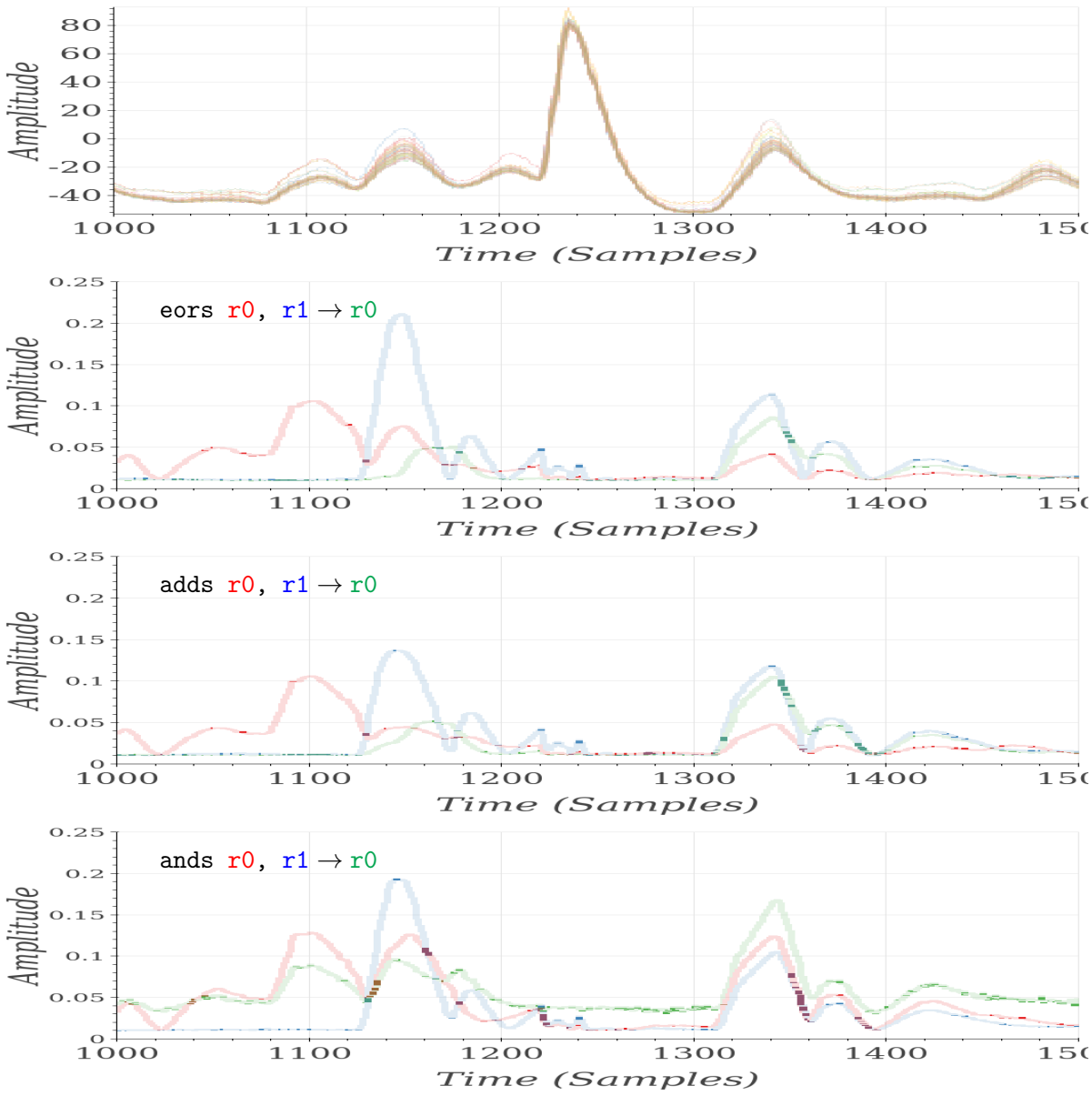

Figure 8: SNR Comparison on LSB

4.2.2. Leakage at Bit Level

In the previous section, a strong leakage related to the LSB of the input and output of the three studied instructions has been exhibited. We study here the leakage related to each of the 8 bits of the leaking LSB. The targeted variables now take binary values, we then use the T-Test to compare them. The purpose of this experiment is to better understand the real leakage function of the Byte value. This function could be estimated using linear regression or mutual information. However it is not clear how this knowledge could be used to improve the masking schemes and therefore we did not go that far. The idea here is simply to verify that all 8 bits have an impact on the leakage. The results show that all 8 bits impact the Byte leakage. Moreover, the leakage function is not far from the *Hamming Weight model* often observed in practice.

For illustration purpose, we display in Figure 9 the T-Test results for instruction `eors`. The 8 T-Test scores over the 2M Power traces are given for the LSB of the two inputs a and b and the output $c = a \oplus b$ (again none of the 3 MSB bits show significant leakage in the area of interest).

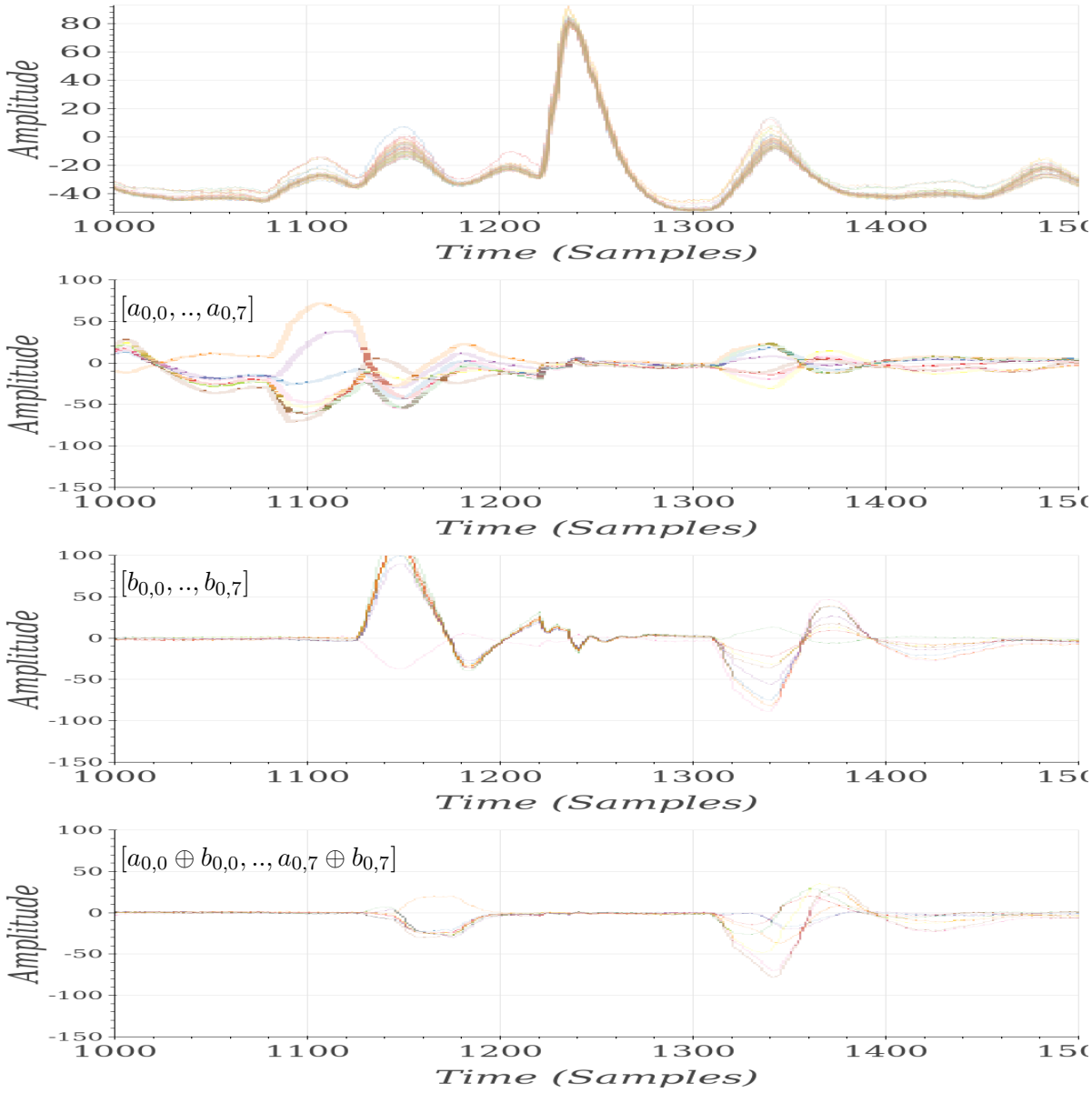

Figure 9: T-Test Results on each bit of LSB – Instruction `eors r0, r1`

4.3. Impact of Registers Choice

In the second experiment, we focus on the leakage of the LSB of a, b and c for the three instructions while varying the register choice. The acquisition parameters are detailed in Table 2.

operation	`adds`/`eors`/`ands` `ri` `rj` for $0 \leq i < j \leq 7$
equipment	PicoScope 6424E, Scaffold
inputs	`rj` and `rj` filled with 8 bits fresh randoms
number of operations	5000
length	20ms
sampling rate	1.25GSa/s
samples per trace	25MSamples
channel(s)	I/O, Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-320, 80]$mV
file size	233GB

Table 2: Acquisition Parameters Single Instructions `ri-rj`

Each of the 5000 operations of this acquisition campaign is constituted of 84 calls to the 28 combinations of register choices for each of the three instructions `adds`/`eors`/`ands` (the asm macro for each instruction can be found in the study material provided with this document [8]). In total, each side-channel trace contains $3 \times 28 \times 84$ successive calls to a single instruction.

We then ended up with 420000 ($= 5000 \times 84$) executions for each instruction call of the type `inst ri, rj`. Each instruction call being surrounded by a trigger signal on the I/O channel, the identification of the 420K sub-traces for each instruction is trivial. Again, the clock stability allows to bypass the otherwise necessary traces re-synchronization.

This experiment aims at comparing the leakage strength related to the register choice with respect to the instruction call. Let us emphasize that the results are intrinsically limited by the facts that (1) only the first 8 registers where tested, (2) `r0` was never tested as the second argument of an instruction, (3) `r7` was never tested as the first argument of an instruction.

We computed the SNR on the two input Bytes and the output Byte of each instruction call. The results are plotted on Figures 10, 11 and 12 respectively for instruction `eors`, `adds` and `ands`. For each SNR sub-figure, 24 score traces are displayed, one for each pair of registers (`ri`, `rj`) (for $0 \leq i < j \leq 7$). The SNR traces colors have been chosen to illustrate the unique clear and consistent behaviour we observed on these results. For all SNR sub-figure of Figures 10, 11 and 12, the 24 SNR traces take exactly 7 different colors, the color of SNR score corresponding to `inst ri, rj` is selected by the `ri` choice among its 7 possible values $\{\texttt{r0}, \texttt{r1}, \ldots, \texttt{r6}\}$. Hence, when `ri` is `r0`, the SNR scores are displayed in red; when `ri` is `r1`, the color is blue, and so on.

From Figures 10, 11 and 12, our main observation is that the choice of register does not have a strong impact on the leakage of the inputs/output of the instruction call but for a very specific case: when `ri` is `r1` (the blue scores in all SNR sub-figures). In this case, and whatever is the choice of `inst` or `rj`, the leakage related to b (second argument of `inst`, *i.e.* the one stored in `rj`) has a significative loss of SNR whereas the leakage related to a (the first argument) is increased. A careful analysis (comparing Figures 10 and 11) shows that, when `ri` is `r1`, the leakage related to b is in fact replaced by the leakage of $a \oplus b$ (somehow a register overwrite must happen that blinds the leakage of b by $a \oplus b$). This implies that the output $c = a \oplus b$ in the case of instruction `eors`, leaks before its very computation by the DUT. In the case of `adds`, the same observation holds but with a reduced leakage strength, indeed the values $a + b$ and $a \oplus b$ are not equal but have strong data dependency.

[8] see the `basic_instructions.s` asm file.

This observation could be exploited by a developer of the *OpenCard* that needs to manipulate a sensitive value x and mask it with a random r. When doing so, the developer needs to execute the instruction `eors ri, rj`. To reduce the leakage related to the value x, he should store r in register `r1` and x in register `rj` ($j \neq 1$) and execute `eors r1, rj`.

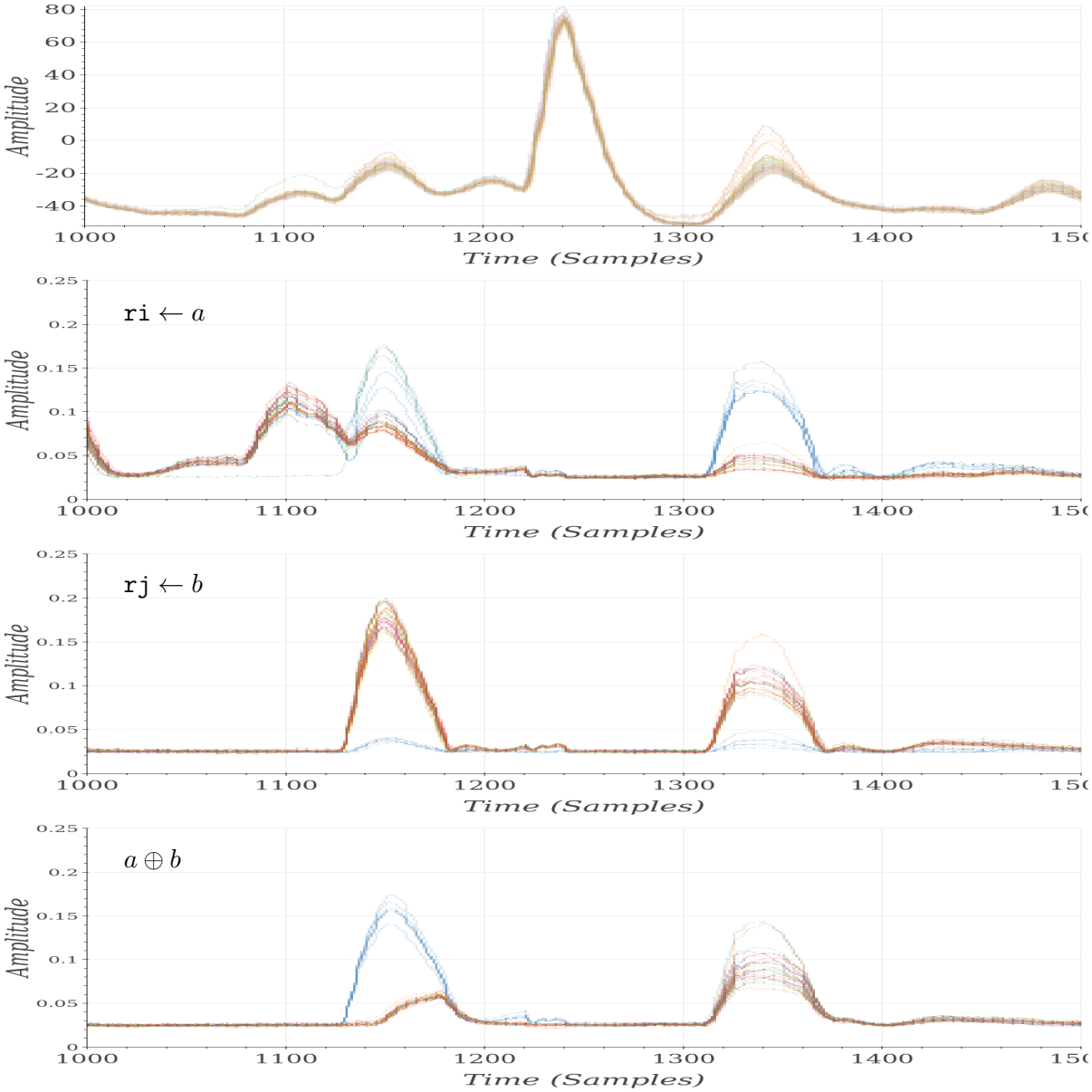

Figure 10: SNR Results – `eors ri, rj`

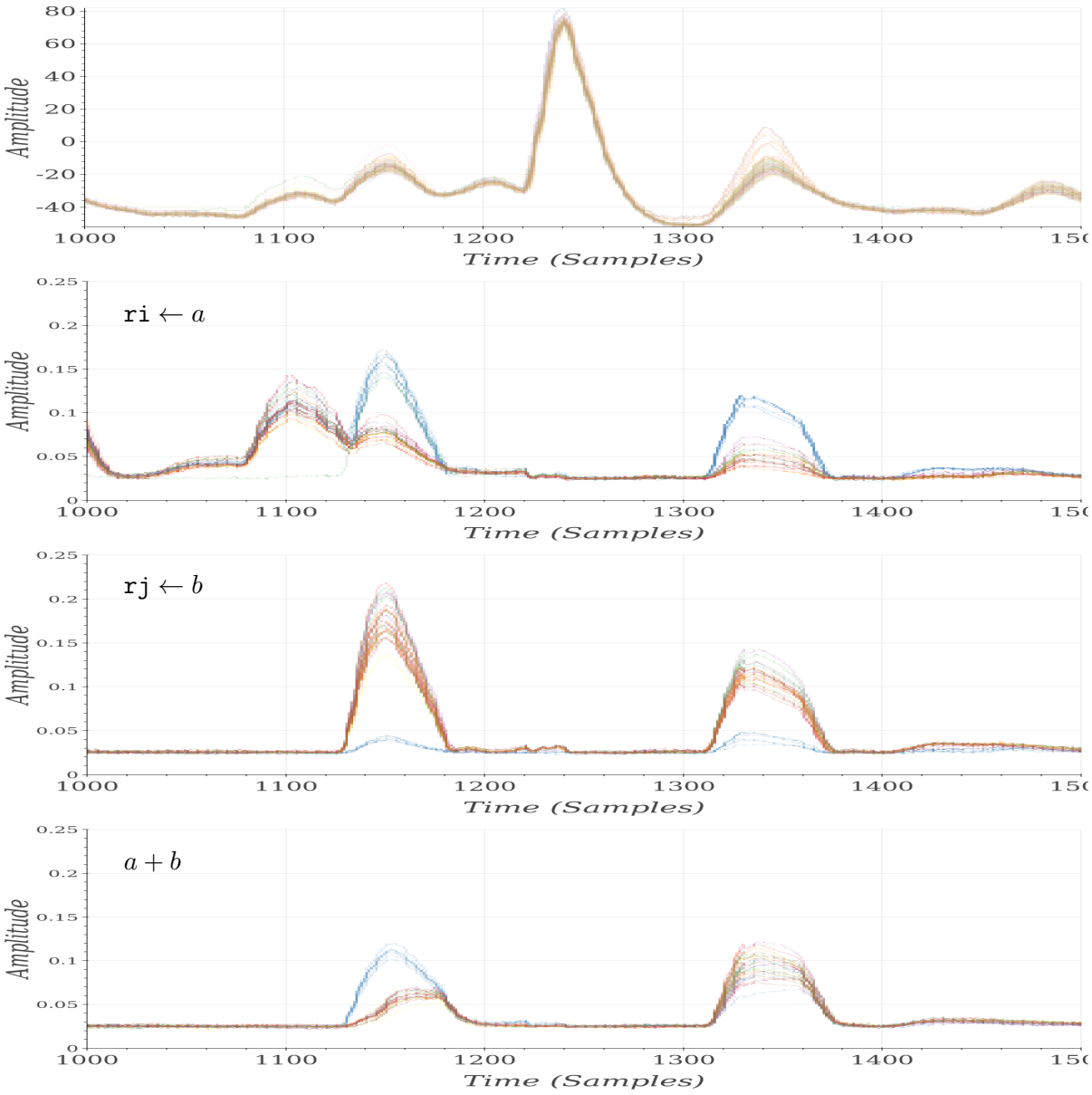

Figure 11: SNR Results – `adds ri, rj`

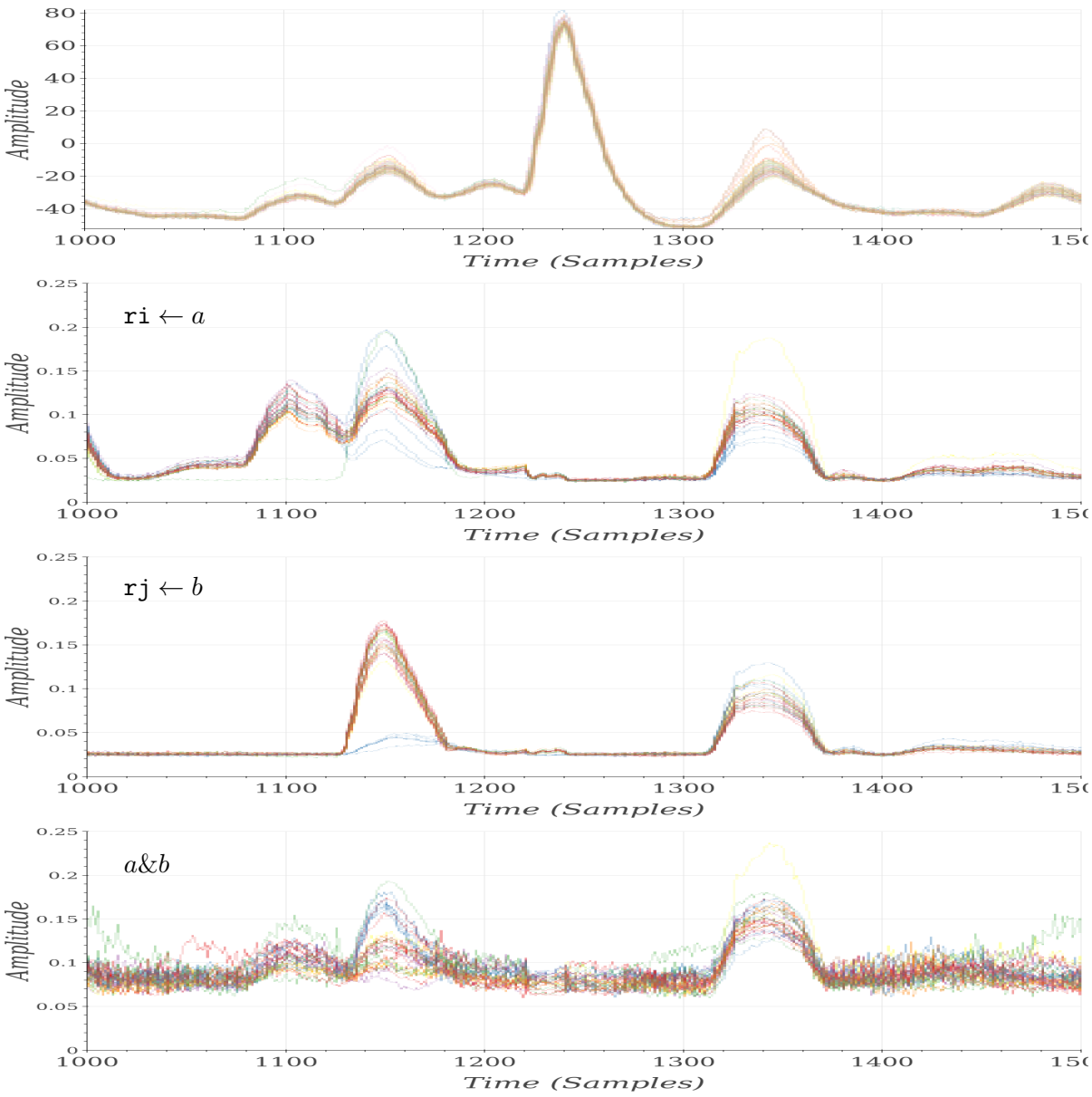

Figure 12: SNR Results – `ands ri, rj`

4.4. Two Successive Instructions

In the next experiment, we consider the succession of two instruction calls. All combinations of (`inst1`, `inst2`) are investigated. The acquisition parameters are detailed in Table 3.

operation	`adds/eors/ands r0, r1` `adds/eors/ands r2, r3`
equipment	PicoScope 6424E, Scaffold
inputs	`r0`, `r1`, `r2` and `r3` filled with 32 bits fresh randoms
number of operations	10000
length	50us
sampling rate	1.25GSa/s
samples per trace	625KSamples
channel(s)	I/O, Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-320, 80]$mV
file size	12GB

Table 3: Acquisition Parameters Two Successive Instructions

Each of the 10000 operations of this acquisition campaign is constituted of 10 calls to the 9 pairs of instructions (`inst1`,`inst2`) where `insti` $\in \{$`adds`, `eors`, `ands`$\}$. An example of an asm macro for these calls is given below (the asm macro for each double instructions can be found in the study material provided with this document [9]. The two consecutive instructions work on two different pairs of registers filled with independent values. The idea of this experiment is to analyse the interaction of these values and potential unexpected leakages.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EXAMPLE CODE ADD-ADD Instruction
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
adds_adds_r0r1r2r3
    push {r4}

    TRIG00 r4
    adds r0, r1
    adds r2, r3
    TRIGFF r4

    pop {r4}
    bx lr
```

We then ended up with 100000 executions for each instruction pair. Each instruction pair being surrounded by a trigger signal on the I/O channel, the identification of the 100000 sub-traces for each of them is trivial. Again, the clock stability allows to bypass the otherwise necessary traces re-synchronization. Figure 13 displays 100 superposed sub-traces among the total of 900K sub-traces. The vertical dashed lines identify the trigger position whereas the gray box pinpoints the expected area of the two instruction executions.

[9] see the `basic_instructions.s` asm file.

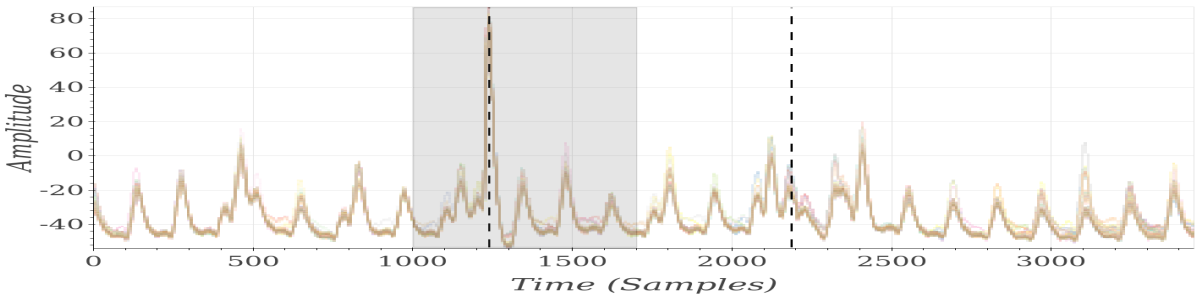

Figure 13: Two Consecutive Instructions - 100 Superposed Power Traces

The following figure 14 depicts the SNR results for the three instruction pairs that start with a `eors`. The other cases fully comply with these results and therefore are not displayed for conciseness. For the three SNR sub-figures of Figure 14, 6 SNR scores are displayed: the DUT is executing $e = a \oplus b$ and then $f = c \star d$, with $\star$ being either $\oplus, +$ or $\&$. The SNR computations with respect to the values taken by the LSB of $\{a, b, c, d, e, f\}$ are displayed for each case of operation $\star$. The SNR respective colors for $\{a, b, c, d, e, f\}$ are $\{\text{red}, \text{blue}, \text{green}, \text{purple}, \text{orange}, \text{yellow}\}$. The SNR of the 3 MSB of the target variables are not displayed since, similarly to previous observations, they do not show significant SNRs.

As expected, the leakage related to the first instruction (`eors`) corresponds exactly to what was observed in previous sections. The second instruction however, seems to show a different behaviour: the three variables (c, d, f) do not show leakages during the next DUT clock cycle (around sample 1340) but the one after that (around sample 1480) and they all three leak simultaneously (f does appear a bit later than c and d). We will need more SNR computations to explain what happens.

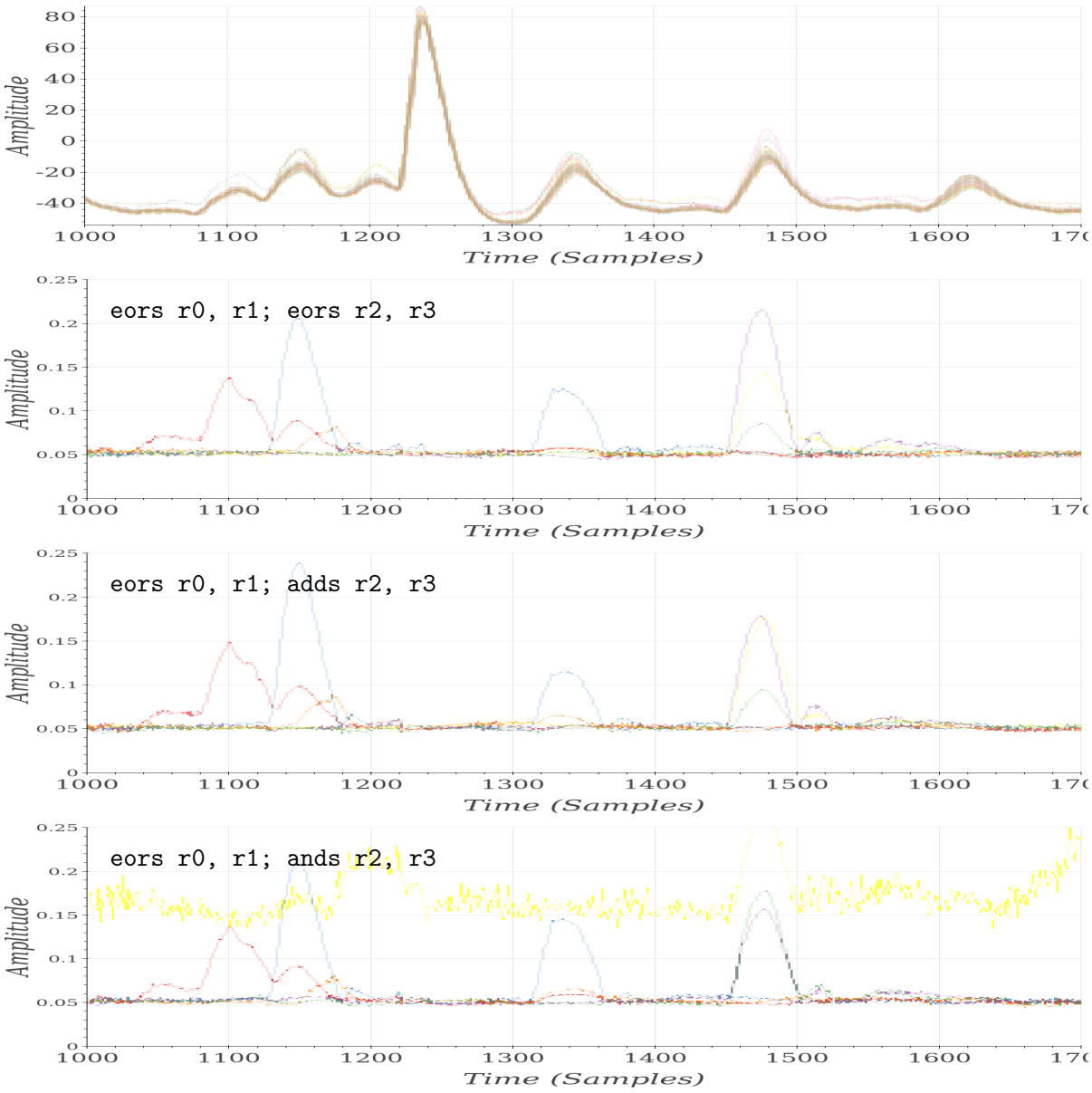

Figure 14: SNR Results – `eors r0, r1; inst2 r2, r3` – $\{a, b, c, d, e, f\}$

To study the interactions between the two consecutive instruction calls, we computed the SNR with respect to the values $v \oplus \{a, b, c, d, e, f\}$ with $v \in \{a, b, c, d, e, f\}$[10]. The next Figure 15 plots some of the results:

- The second sub-figure shows the SNR results of $a \oplus \{a, b, c, d, e, f\}$. The SNR peaks show a leakage relating to $a \oplus b$ and $a \oplus e$, which is expected.
- The third sub-figure shows the SNR results of $d \oplus \{a, b, c, d, e, f\}$. There, one can see that $d \oplus b$ (in blue) shows a SNR peak, meaning that the consecutive processing of b and d by two different instructions induces a leakage on $d \oplus b$. This fact has to be taken into account by a

[10]Of course, $v \oplus v$ will always show a null SNR

developer who must avoid the consecutive manipulation of two variables whose difference is sensitive.

- Finally, the last sub-figure shows the SNR results of $e \oplus \{a, b, c, d, e, f\}$. Here, the interesting observation is the yellow SNR (corresponding to $e \oplus f$) around sample 1360, its time position and shape is similar to what we identified as the *Execute* stage of a single instruction (see Section 4.2.1, Figure 8). We can deduce from this observation that the *Execute* stage of the second instruction really occurs at the next cycle after the *Execute* stage of the first instruction (as it should be), but since f takes the place of the previously computed e, the leakage relates to $e \oplus f$.

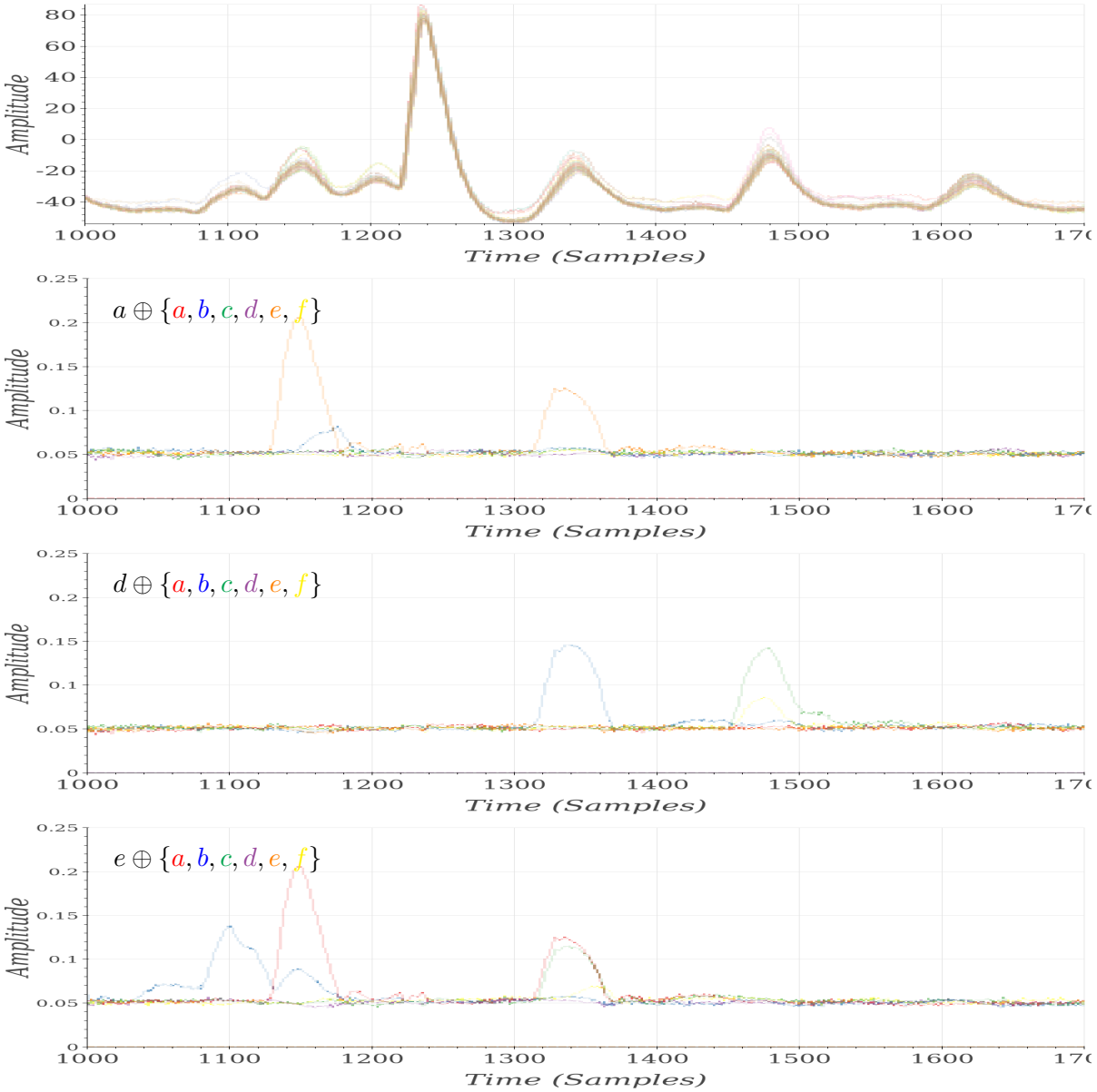

Figure 15: SNR Results – `eors r0, r1; eors r2, r3` – $v \oplus \{a, b, c, d, e, f\}$

This experiment concludes the study of basic instructions, in the next section we will have a look at a gadget (as defined in Section 3.4)

5. Leakage Analysis of Masked AND Gadget

5.1. Introduction

In this experiment we will study the `and_8bits_4shares` gadget as defined in Section 3.4, its asm implementation is given here (the asm macro for each gadget can be found in the study material provided with this document [11]):

```
GET_1B_RAND_asm r3
GET_1B_RAND_asm r2
add r3, r2, LSL #8
GET_1B_RAND_asm r2
add r3, r2, LSL #16
GET_1B_RAND_asm r2
add r2, r3, r2, LSL #24

ands r3, r0, r1
eors r4, r3, r2
ands r3, r0, r1, ROR #8
eors r4, r3
ands r3, r1, r0, ROR #8
eors r4, r3
eors r4, r2, ROR #8
ands r3, r0, r1, ROR #16
eors r0, r4, r3
```

where the `GET_1B_RAND_asm` asm macro picks a random byte based on the internal `xorshift96` pseudo-random generator (see Section 3.1) initialized with a fresh random seed every 16 calls to `and_8bits_4shares` gadget. The registers `r0` and `r1` contain the shared inputs a and b (the 4 shares are stored in a single register). At the end of `and_8bits_4shares`, the register `r0` contains the output $c = a\&b$ as 4 shares.
The acquisition parameters are detailed in Table 4.

[11] see the `snippets.s` asm file.

operation	AND 4-shares
equipment	PicoScope 6424E, Scaffold
inputs	`r0`, `r1` filled with 32 bits fresh randoms
number of operations	150000
length	1ms
sampling rate	1.25GSa/s
samples per trace	660KSamples
channel(s)	I/O, Power
channel(s) parameters	I/O: DC, voltage range $[-5, 5]$V Power: DC, voltage range $[-320, 80]$mV
file size	185GB
acquisition time	about 4 days

Table 4: Acquisition Parameters AND 4-shares

Each of the 150000 operations of this acquisition campaign is constituted of 16 calls to the `and_8bits_4shares` function. We then end up with about 2.4M executions of `and_8bits_4shares`. Each call being surrounded by a trigger signal on the I/O channel, the identification of the 2.4M sub-traces is trivial.

5.2. Signal Overview

Figure 16 displays 100 superposed traces of the `and_8bits_4shares` execution. It appears that a small clock jitter induces a de-synchronization at the end of the traces, indeed the length of the `and_8bits_4shares` execution is about 20 times longer than the previous target executions.

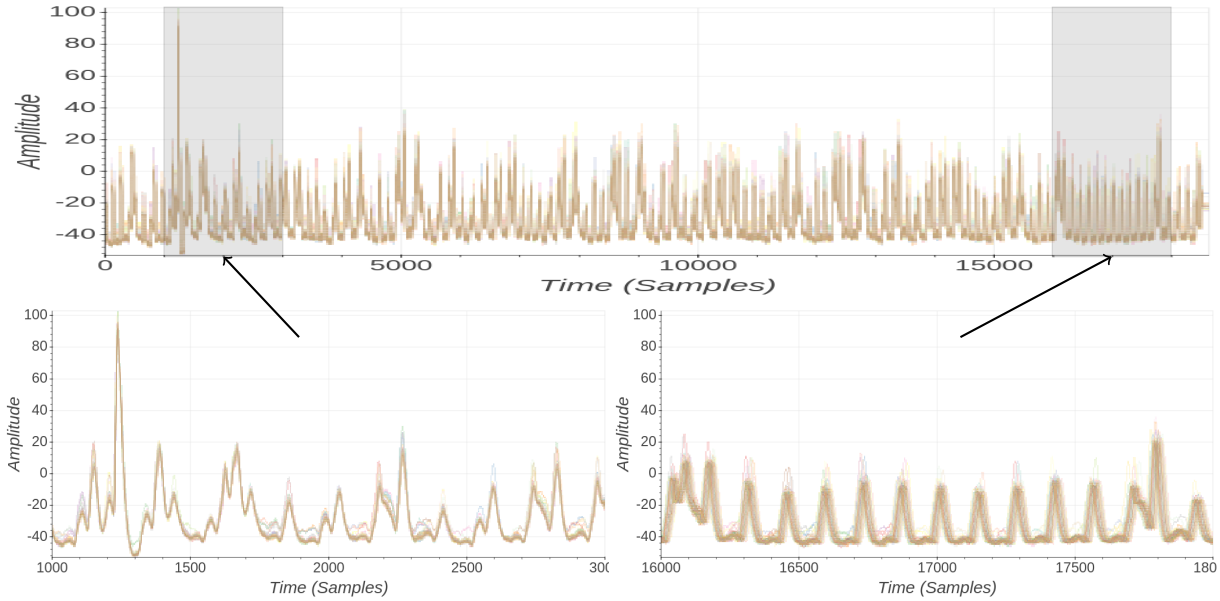

Figure 16: AND 4-shares – 100 Power Traces

5.2.1. Traces Resynchronization

To solve this synchronization issue and given that the de-synchronization is not visible over 1000 sample traces, we use a very simple and efficient re-synchronization algorithm: a synchronization point is positioned on the first trace every 1000 samples (as depicted on Figure 17) and all subsequent traces are re-synchronized based on those points. For each synchronization point, the exact position on the processed trace is found taking the best 500 Samples cross-correlation score over a sliding window of 50 Samples around the expected position.

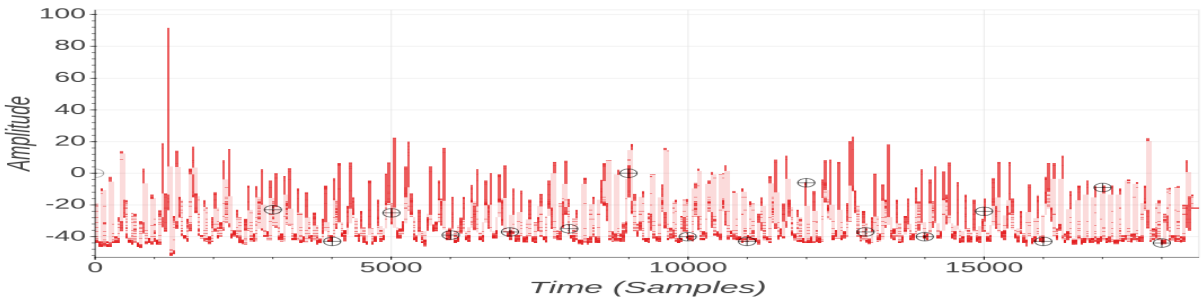

Figure 17: AND 4-shares – Synchronization Points

The resulting re-synchronized traces are plotted in Figure 18, one can verify that the first 100 traces are well synchronized over the full length of the `and_8bits_4shares` execution. Moreover, the next Figure 19 displays the average execution trace over the 2.4M re-synchronized traces, its steady amplitude over the full execution confirms that the re-synchronization procedure is sound.

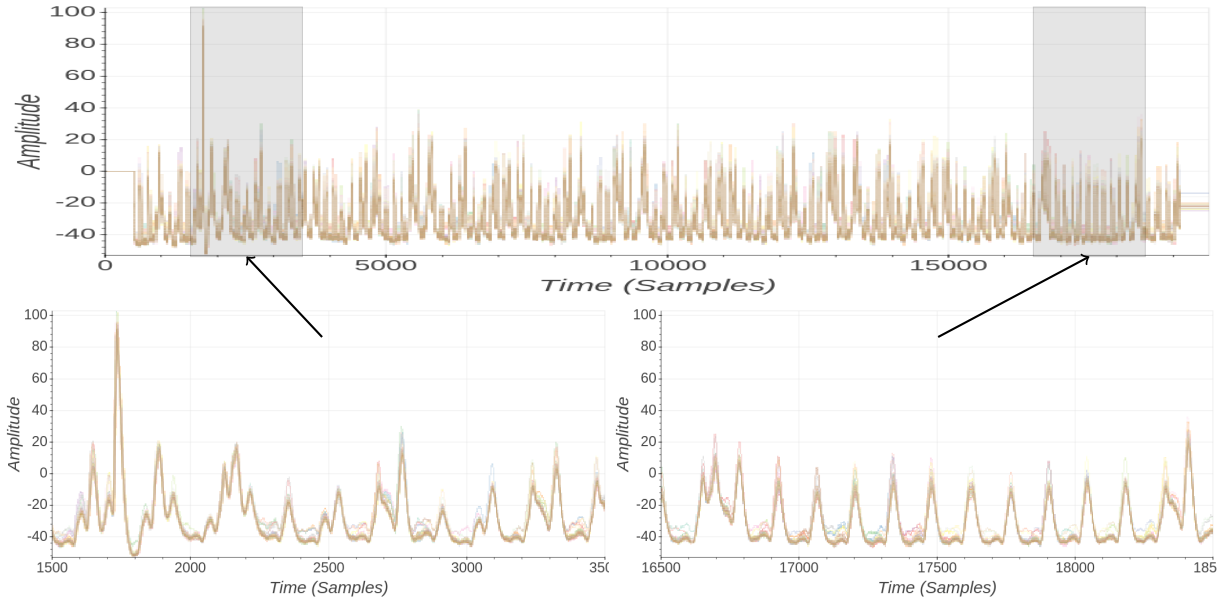

Figure 18: AND 4-shares – 100 Re-synchronized Traces

5.2.2. First Observations

A quick study of the `and_8bits_4shares` code and corresponding Power traces allows to easily identify the two parts of the execution (see Figure 19), *i.e.* (1) getting the randoms necessary for the computation and (2) computing the secure `AND` on 4 shares. The first part, not optimized is much longer than the computation itself. Our goal here is really to study the second part and not the random generation but we will see that it provides some insight on the DUT leakage.

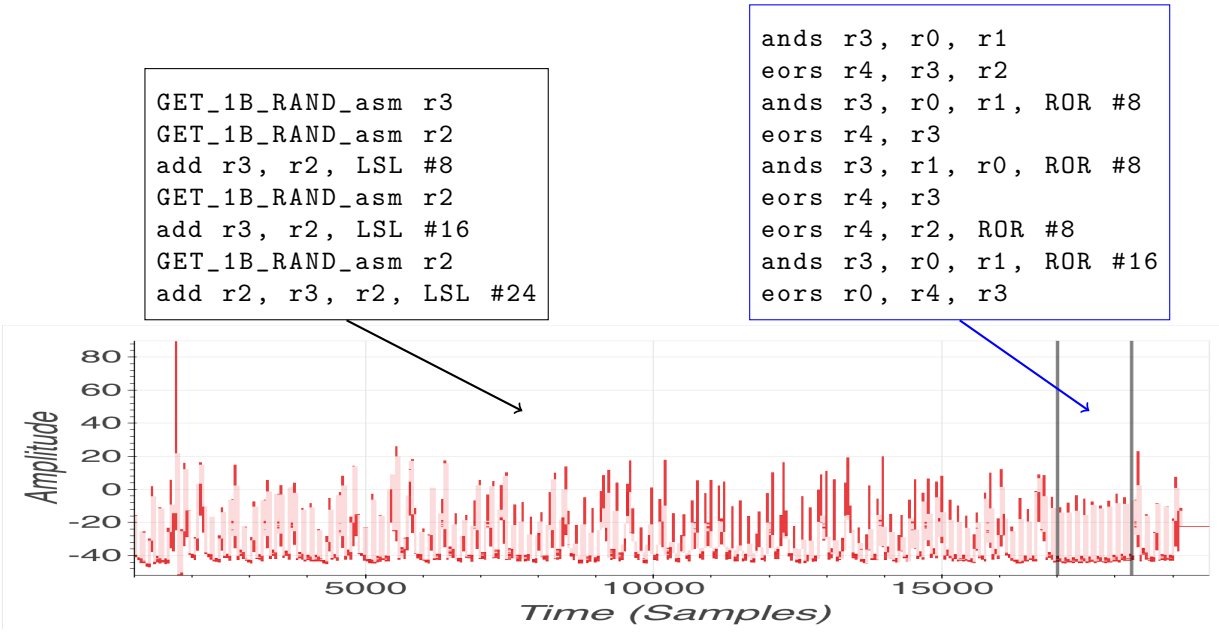

Figure 19: AND 4-shares – Average Re-synchronized Power Traces

Focusing on the second part of the `and_8bits_4shares` computation, Figure 20 depicts a tentative identification of the successive instruction calls. This identification has been done based on the Power trace study but also taking into account the side-channel analysis of the next section. We must emphasize that this is mainly a way to illustrate what we are looking at and could be wrong, there are no strong proof here but a body of weak evidences.

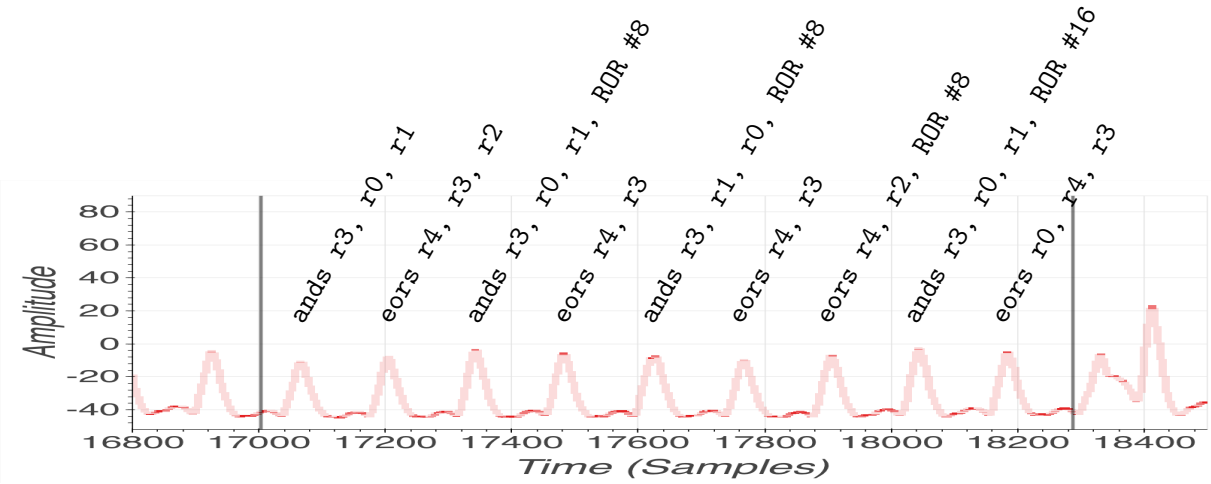

Figure 20: AND 4-shares – Average Re-synchronized Power Traces Zoom

5.3. Side-Channel Analysis

The side-channel analysis consists in computing the SNR of all intermediate variables (and their shares) of the `and_8bits_4shares` operation (again, we focus on the secure `AND` computation here, not the PRNG calls). In the following a, b are the shared inputs (stored respectively in `r0` and `r1` when calling `and_8bits_4shares`) and c the shared output (stored in `r0` when returning from `and_8bits_4shares`). We call s (to avoid confusion with registers names) the 4 random bytes generated from the 4 PRNG calls and stored in register `r2` before the secure `AND` computation. Figure 21 displays the SNR result over the whole `and_8bits_4shares` execution for the 3 inputs and the output of the secure `AND` computation whereas Figure 22 is a zoom in the secure `AND` computation. For all variables, the 4 Bytes as well as their xor is considered, there are then 5 SNR scores for each of them.

We sum up here the observations that can be made from this analysis:

- The first striking observation is that all 4 Bytes of `r0`, `r1` and `r2` are now leaking whereas all results presented previously consistently showed that solely the LSB was leaking. This contradiction could not find an explanation and we let for future work the tedious analysis that will give some sense to this phenomenon.

- Another clear observation is that the shared value of a is strongly leaking during the PRNG calls even though no manipulation of `r0` is done there (this is also true for the shared value of b, though in a milder fashion). This confirms a previous observation (see Section 4.2) that the values stored in `r0` will strongly leak even when no manipulation is intended.

- Finally, for all targeted variables, the xor of the 4 shares (*i.e.* the sensitive variables) shows no significant leakage through the whole execution. This is a positive result as it shows that storing all shares of a sensitive variable in the *OpenCard* does not compromise the soundness of the masking scheme. This observation might have a great impact on the efficiency of the masking scheme.

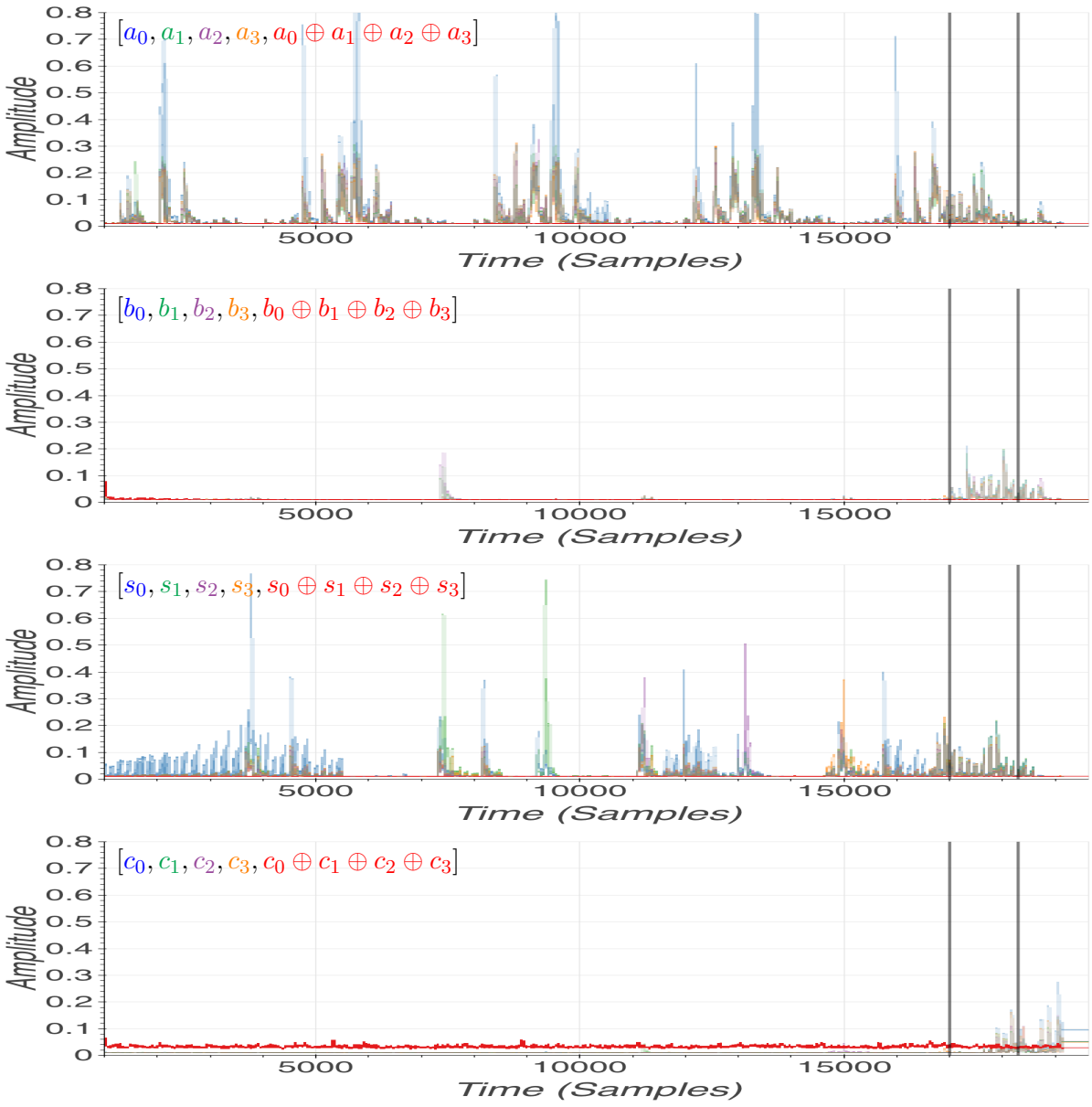

Figure 21: AND 4-shares – SNR Scores – Inputs $\{a, b, s\}$ and Output $\{c\}$

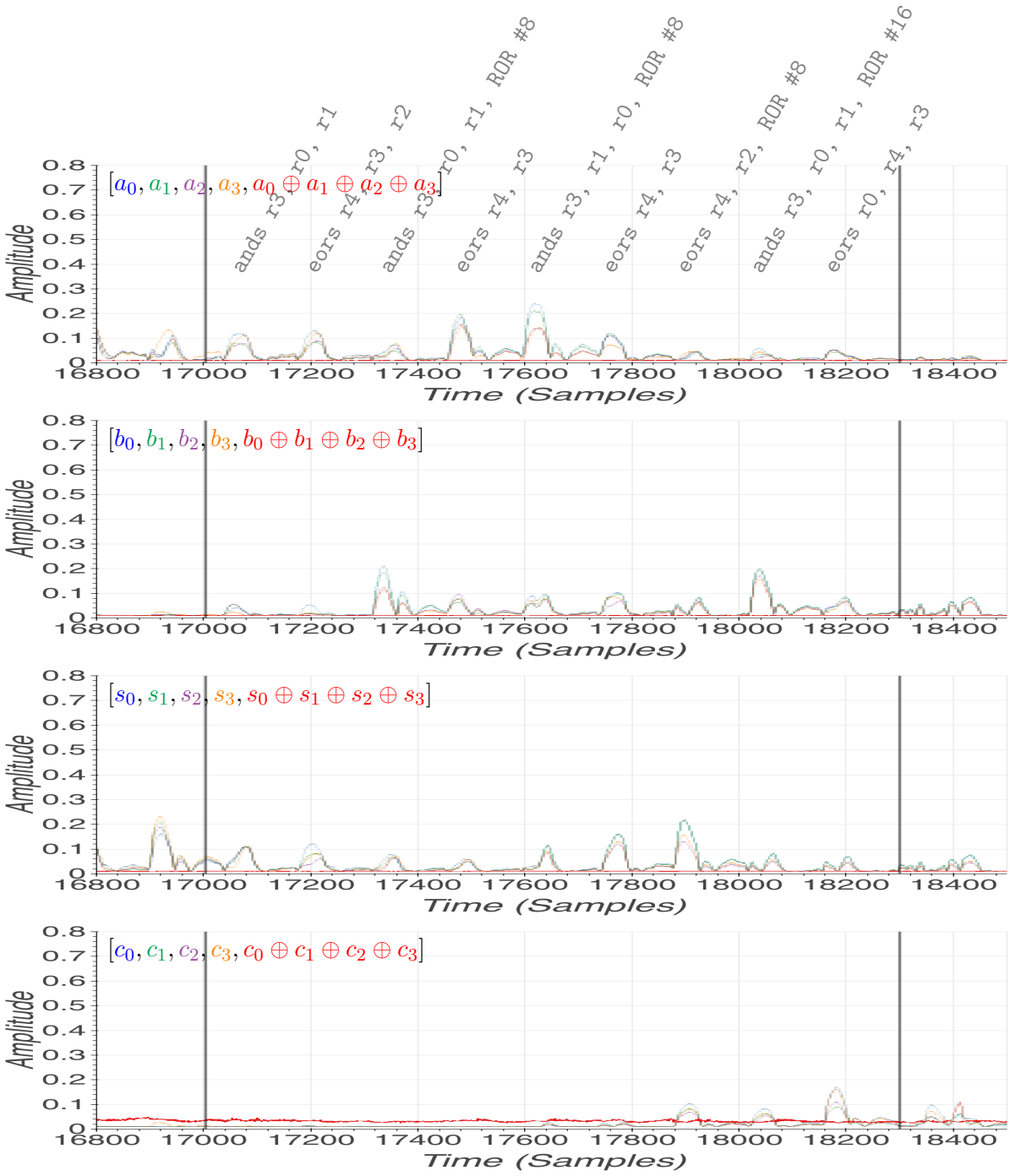

Figure 22: AND 4-shares – SNR Scores Zoom – Inputs $\{a, b, s\}$ and Output $\{c\}$

For completeness, we provide in the following figures 23 to 30, the 5 SNR scores of each of the instruction output (4 Bytes plus the xor of them) during the secure `AND` computation. One can see that, for some of them, the xor of the 4 Bytes (the red SNR in the figures) shows a leakage. Interpreting this leakage is not easy and should not be regarded as a flaw in the masking scheme implementation without a deeper study.

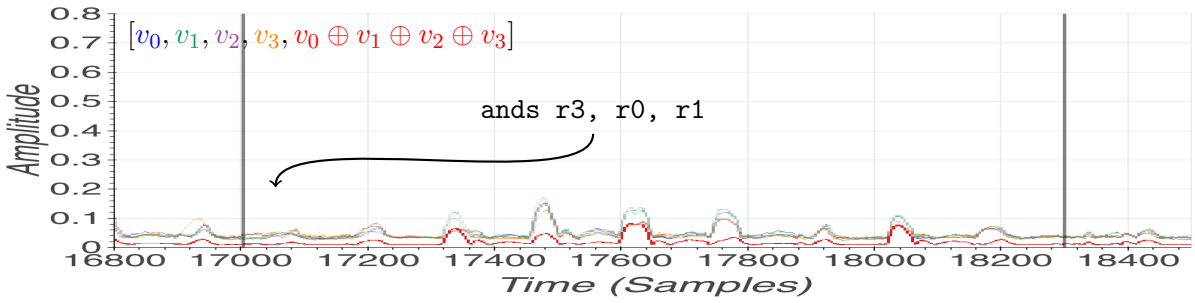

Figure 23: AND 4-shares Carac – SNR Scores – `ands r3, r0, r1`

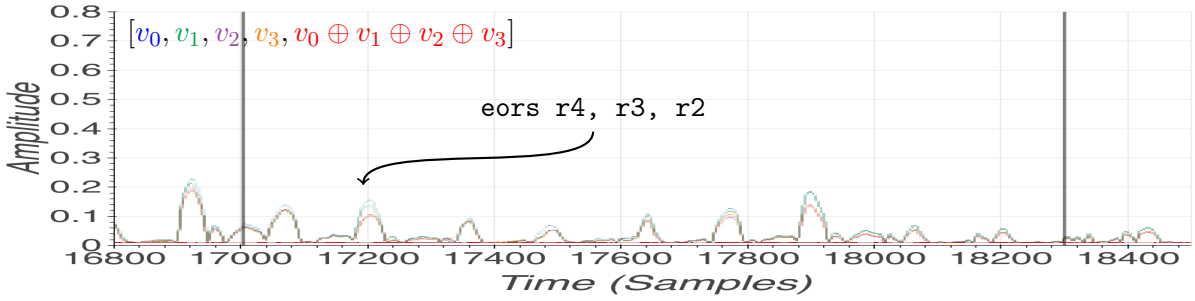

Figure 24: AND 4-shares Carac – SNR Scores – `eors r4, r3, r2`

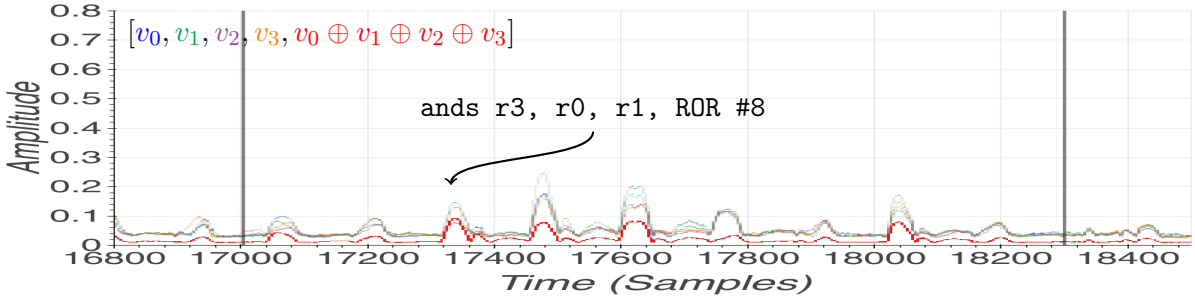

Figure 25: AND 4-shares Carac – SNR Scores – `ands r3, r0, r1, ROR #8`

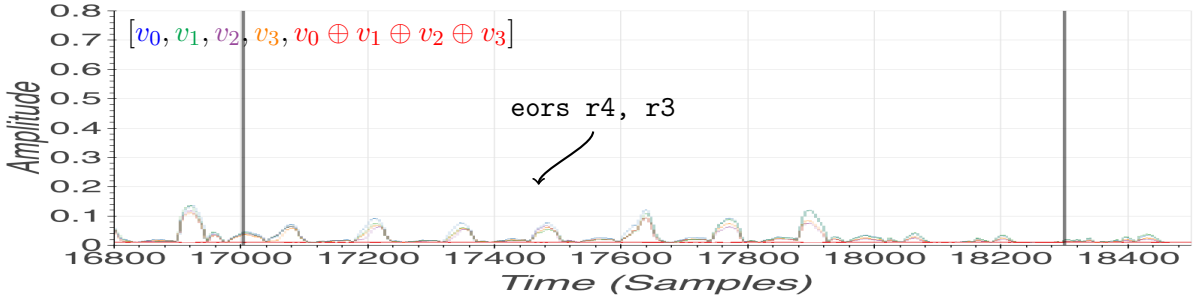

Figure 26: AND 4-shares Carac – SNR Scores – `eors r4, r3`

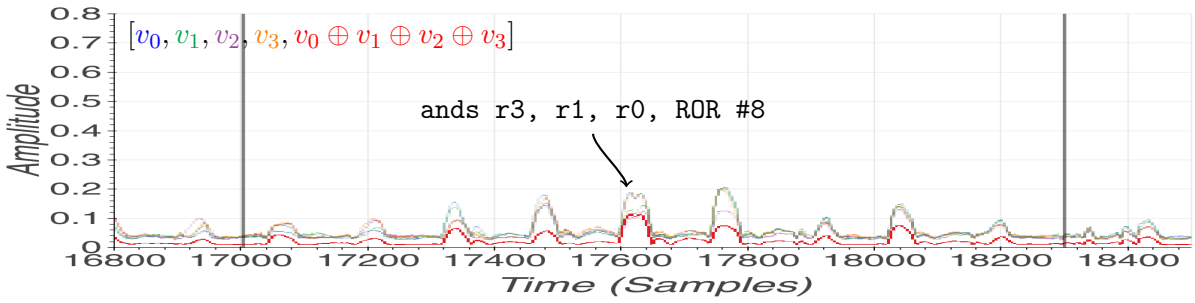

Figure 27: AND 4-shares Carac – SNR Scores – `ands r3, r1, r0, ROR #8`

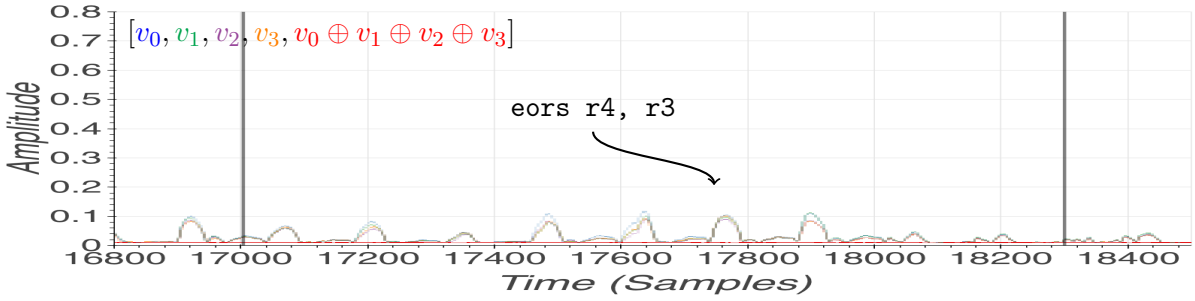

Figure 28: AND 4-shares Carac – SNR Scores – `eors r4, r3`

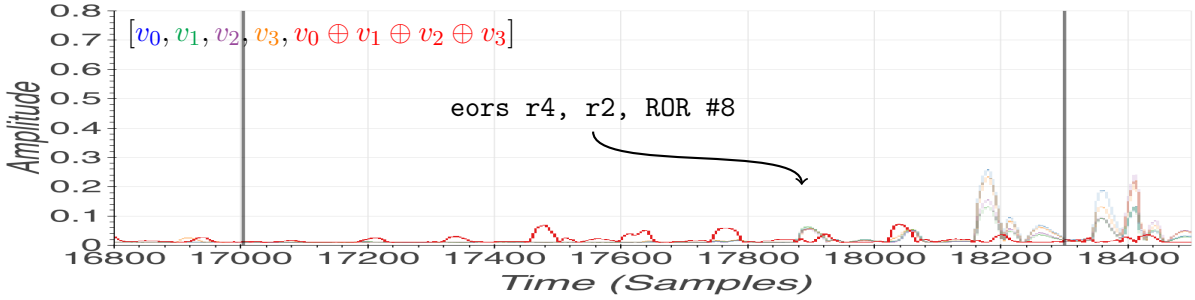

Figure 29: AND 4-shares Carac – SNR Scores – `eors r4, r2, ROR #8`

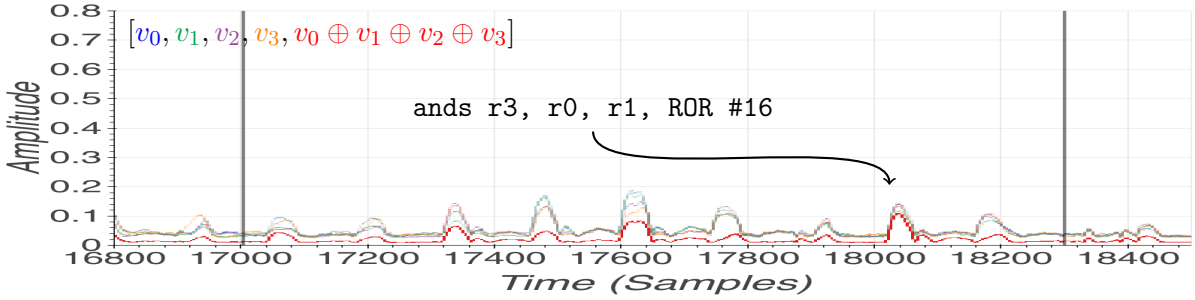

Figure 30: AND 4-shares Carac – SNR Scores – `ands r3, r0, r1, ROR #16`

5.4. Higher-Order Side-Channel Analysis

We conclude this side-channel analysis by the study of the Higher-Order leakage. In a first attempt, we only consider univariate higher-order moments of the Power traces, *i.e.* we compute the SNRs over centered powers of the traces considering each sample individually. Since we observed in previous section (*e.g.* Figure 22) that the 4 shares of sensitive variables leak simultaneously this approach makes sense while being much more affordable than estimating the multivariate higher-order moments (whose computation cost increases exponentially with the order).

Figure 31 (with a zoom in the secure `AND` computation in Figure 32) shows the SNR results. For each univariate statistical moment, the SNR of the unmasked values a, b and c is plotted. No significant leakage is observed (c being the output of $a\&b$ its value is not equiprobably distributed in $[0, .., 255]$, this explain the SNR shape).

Since we know that all shares of (say a) leak at the same exact time samples, we would expect (at least) the 4th order univariate analysis to exhibit a leakage. Our failure to do so certainly comes from the number of available traces: 2.4M traces is certainly not enough to see such a high-order leakage.

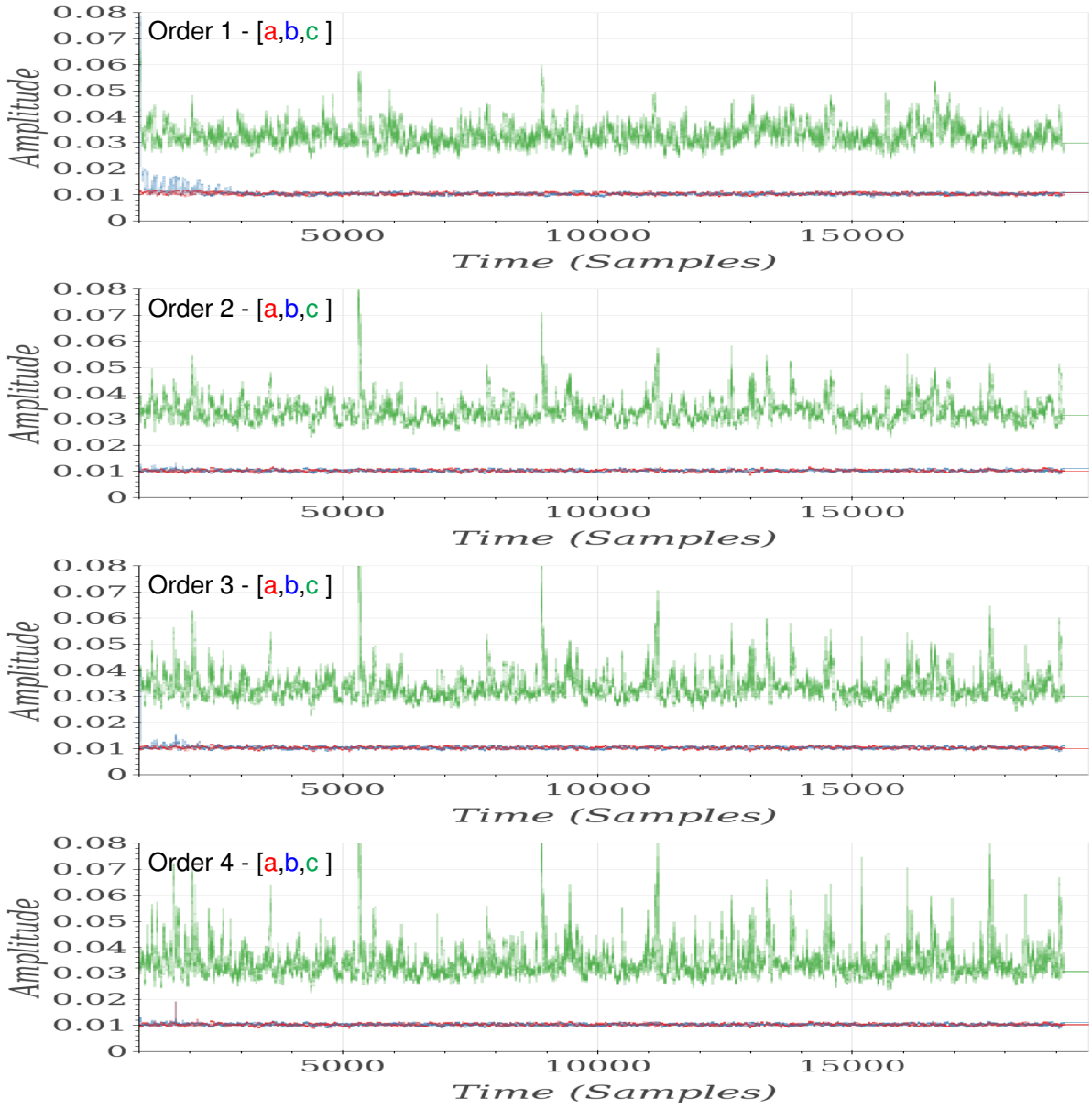

Figure 31: AND 4-shares – Univariate HO SNR Scores – Inputs $\{a, b\}$ and Output $\{c\}$

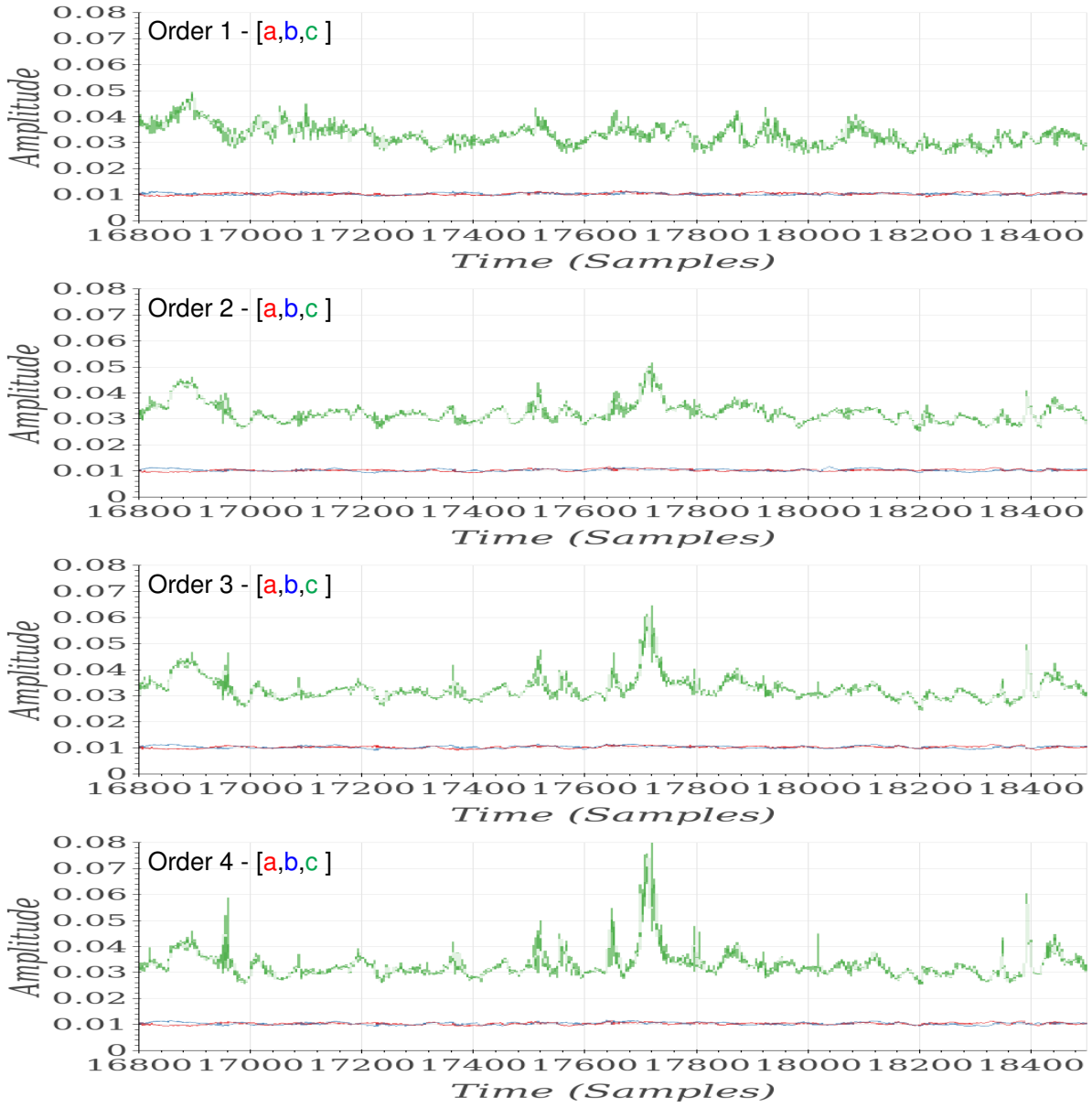

Figure 32: AND 4-shares – Univariate HO SNR Scores Zoom – Inputs $\{a, b\}$ and Output $\{c\}$

As previously stated, the higher-order multivariate analysis would require a tremendous computational effort if we want to consider all combinations of samples within the whole traces. We limited our analysis to the secure `AND` computation area for the 2nd-order bivariate analysis and did not found any leakage. For the third and fourth orders, we limited our analysis to tiny windows around the SNR peaks visible in Figure 22. Again, none of these SNRs showed significant scores for the unmasked values a, b or c.

References

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, 2017.

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

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

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

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 199–216. USENIX Association, 2017.

[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].