Alpen-Adria-Universität Klagenfurt

Towards secure software implementations of masking schemes in Assembly

By

Nima Mahdion

Supervised by

Univ.-Prof. Elisabeth Oswald

Institute of Information Technology (ITEC)
Alpen-Adria-Universität Klagenfurt, Austria

July 10, 2024

Contents

1 Setup **5**

1.1 Codes/implementation 5

1.1.1 .S file 5

1.1.2 .c file 5

1.1.3 .h file 5

1.2 Measurement setup 9

1.2.1 The spot of the trigger: uELMO (simulator) 9

1.2.2 The spot of the trigger: in .c file on the device 11

1.2.3 The spot of the trigger: in .S file on the device 12

2 Field multiplication **14**

2.1 GFMUL as a function 15

2.2 GFMUL as a label 16

2.3 GFMUL as a Sequence of instructions 17

2.4 Comparison 18

3 Split-t-test **19**

4 ISW **21**

5 ISW-2- no function/label call **22**

5.1 ISW-2-no-function-call-0: basic code 22

5.1.1 Experimental result: 28

5.1.1.1 Setup: 28

5.1.1.2 Signal to Noise Ratio (SNR): 29

5.1.1.3 t-test: 33

6 ISW-3- no function/label call **52**

6.1 ISW-3-V0: basic code 53

6.1.1 Experimental result: 66

6.1.1.1 Signal to Noise Ratio (SNR): 67

6.1.1.2 t-test: 67

7 BBPP **117**

8 BBPP-2- no function/label call **118**

8.1 BBPP-2-no-function-call-0: basic code 118

8.1.1 Experimental result: 125

8.1.1.1 Setup: 125

8.1.1.2 t-test: 125

9 BBPP-OPT-3- no function/label call **126**

9.1 BBPP-OPT-3-no-function-call-0: basic code 127

9.1.1 Experimental result: 141

9.1.1.1 Signal to Noise Ratio (SNR): 142

9.1.1.2 t-test: 143

10 DOM-indep **160**

11 DOM-indep-2- no function/label call **161**

11.1 DOM-indep-2-no-function-call-0: basic code 161

11.1.1 Experimental result: 167

11.1.1.1 Setup: 167

11.1.1.2 Signal to Noise Ratio (SNR): 167

11.1.1.3 t-test: 169

12 DOM-indep-3- no function/label call **178**

12.1 DOM-indep-3-no-function-call-0: basic code 179

12.1.1 Experimental result: 193

12.1.1.1 Signal to Noise Ratio (SNR): 194

12.1.1.2 t-test: 194

13 HPC1 **201**

14 HPC1-OPT-2 **202**

14.1 HPC1-OPT-2-no-function-call-0: basic code 202

14.1.1 Experimental result: 209

14.1.1.1 Setup: 209

14.1.1.2 Signal to Noise Ratio (SNR): 209

14.1.1.3 t-test: 211

15 HPC1-OPT-3- no function/label call **225**

15.1 HPC1-OPT-3-no-function-call-0: basic code 226

15.1.1 Experimental result: 241

15.1.1.1 Signal to Noise Ratio (SNR): 242

15.1.1.2 t-test: 242

16 PINI1 **245**

17 PINI1-2 **246**

17.1 PINI1-2-no-function-call-0: basic code 246

17.1.1 Experimental result: 255

17.1.1.1 Setup: 255

17.1.1.2 t-test: 255

18 PINI1-3- no function/label call **258**

18.1 PINI1-3-no-function-call-0: basic code 259

18.1.1 Experimental result: 279

18.1.1.1 Signal to Noise Ratio (SNR): 280

18.1.1.2 t-test: 280

19 PINI2 **282**

20 PINI2-2 **283**

20.1 PINI2-2-no-function-call-0: basic code 283

20.1.1 Experimental result: 293

20.1.1.1 Setup: 293

20.1.1.2 t-test: 293

21 PINI2-3- no function/label call **295**

21.1 PINI2-3-no-function-call-0: basic code 296

21.1.1 Experimental result: 319

21.1.1.1 Signal to Noise Ratio (SNR): 320

21.1.1.2 t-test: 320

1. Setup

1.1 Codes/implementation

Each implementation contains 3 files, .S, .c and .h

1.1.1 .S file

.S file contains the assembly implementation of the gadget and can be used for different ARM Cortex-M0/3. For all gadget the main frame of .S file is the same as it is shown in Listing 1.1. But, for different gadget, the part Gadget_Instructions (in Listing 1.1) is different. We will illustrate the section Gadget_Instructions for individual gadgets. Notice:The assembly part is written by using Thumb-16 instructions. In Thumb-16 registerlist in push/pop can not contain High registers. As a result, instead of push/pop r4-r11, the following instructions are used.

1.1.2 .c file

Defining the inputs/outputs of the gadget, calling the gadget, and also, the way of communication with our board (SCALE board) is placed in .c file (Listing 1.2). For different devices (boards), the communication part has to be changed.

Furthermore, in the implementations, the inputs/outputs of all gadgets are considered as one-byte values. Thus, the variables (shares and randomnesses) need to be one-byte. As we are using a 32-bit (4-byte) processor, experiences have shown that there are some leakages based on:

1) Bytes in a word have interactions with each other.

2) ldrb/strb load the whole 32-bit word into the read/write bus and then select the desired byte.

Thereby, we need to remove the effects of the rest 3 bytes by setting them into zero. This strategy ensure that just the one-byte values contribute to the leakages. To reach this goal, we define variables in a way in which just the LSB (Least Significant Byte) set into non-zero. The definition is put in .c file (Listing 1.2) to ensure that the bytes of shares (of a, b, c) are not in a word.

1.1.3 .h file

Declaring the gadget which is implemented in .S file (Listing 1.3) is located in .h file. In addition, this file places a constant table, which is used for Field multiplication (Section 2).

Listing 1.1: .S file

```
.syntax unified
  .text
  .thumb
  .extern table  @ table is used for GFMULT (ai * bj multiplications)

.global Gadget_name
.func Gadget_name
Gadget_name:

  push  {lr}
  # push  {r4-r11} ########
  push  {r4-r7}
  mov   r4, r8
  mov   r5, r9
  mov   r6, r10
  mov   r7, r11
  push  {r4-r7}



################################
Gadget_Instructions
################################


 @ Trigger instructions



  # pop  {r4-r11} ########
  pop  {r4-r7}
  mov   r8, r4
  mov   r9, r5
  mov   r10, r6
  mov   r11, r7
  pop   {r4-r7}
  pop  {pc}
  .endfunc

.end
```

Listing 1.2: .c file

```c
#include <stdio.h>
#include <stdlib.h>
#include "Gadget_name_lib.h"

    int share_n = n;
    int rnd_n = m;

    uint8_t shares_a[4*share_n];
    uint8_t shares_b[4*share_n];
    uint8_t shares_ab[4*share_n];
    uint8_t rnd[4*rnd_n];

        // Receiving shares of a
        for (int i = 0; i < share_n; i++) {
            shares_a[4*i] = (uint8_t) scale_uart_rd(SCALE_UART_MODE_BLOCKING);
            shares_a[4*i+1]=0;
            shares_a[4*i+2]=0;
            shares_a[4*i+3]=0;
        }

        // Receiving shares of b
        for (int i = 0; i < share_n; i++) {
            shares_b[4*i] = (uint8_t) scale_uart_rd(SCALE_UART_MODE_BLOCKING);
            shares_b[4*i+1]=0;
            shares_b[4*i+2]=0;
            shares_b[4*i+3]=0;
        }

        // Receiving random numbers
        for (int i = 0; i < rnd_n; i++) {
            rnd[4*i] = (uint8_t) scale_uart_rd(SCALE_UART_MODE_BLOCKING);
            rnd[4*i+1]=0;
            rnd[4*i+2]=0;
            rnd[4*i+3]=0;
        }


        Gadget_name(shares_a, shares_b, rnd), shares_ab);

        // Sending shares of c (c = a * b), c shares_ab
        for (int i = 0; i < share_n; i++) {
            scale_uart_wr(SCALE_UART_MODE_BLOCKING, ((char) shares_ab[4*i]));
            shares_ab[4*i+1]=0;
            shares_ab[4*i+2]=0;
            shares_ab[4*i+3]=0;
        }


```

Listing 1.3: .h file

```c
#include <scale/scale.h>
#include <stdio.h>
#include <stdint.h>
extern void Gadget_name(uint8_t* a, uint8_t* b, uint8_t* r, uint8_t* c);
unsigned char table[798] = {
    0x00, 0x00, 0x19, 0x01, 0x32, 0x02, 0x1a, 0xc6, 0x4b, 0xc7, 0x1b, 0x68, 0x33, 0xee,
    0xdf, 0x03, 0x64, 0x04, 0xe0, 0x0e, 0x34, 0x8d, 0x81, 0xef, 0x4c, 0x71, 0x08, 0xc8,
    0xf8, 0x69, 0x1c, 0xc1, 0x7d, 0xc2, 0x1d, 0xb5, 0xf9, 0xb9, 0x27, 0x6a, 0x4d, 0xe4,
    0xa6, 0x72, 0x9a, 0xc9, 0x09, 0x78, 0x65, 0x2f, 0x8a, 0x05, 0x21, 0x0f, 0xe1, 0x24,
    0x12, 0xf0, 0x82, 0x45, 0x35, 0x93, 0xda, 0x8e, 0x96, 0x8f, 0xdb, 0xbd, 0x36, 0xd0,
    0xce, 0x94, 0x13, 0x5c, 0xd2, 0xf1, 0x40, 0x46, 0x83, 0x38, 0x66, 0xdd, 0xfd, 0x30,
    0xbf, 0x06, 0x8b, 0x62, 0xb3, 0x25, 0xe2, 0x98, 0x22, 0x88, 0x91, 0x10, 0x7e, 0x6e,
    0x48, 0xc3, 0xa3, 0xb6, 0x1e, 0x42, 0x3a, 0x6b, 0x28, 0x54, 0xfa, 0x85, 0x3d, 0xba,
    0x2b, 0x79, 0x0a, 0x15, 0x9b, 0x9f, 0x5e, 0xca, 0x4e, 0xd4, 0xac, 0xe5, 0xf3, 0x73,
    0xa7, 0x57, 0xaf, 0x58, 0xa8, 0x50, 0xf4, 0xea, 0xd6, 0x74, 0x4f, 0xae, 0xe9, 0xd5,
    0xe7, 0xe6, 0xad, 0xe8, 0x2c, 0xd7, 0x75, 0x7a, 0xeb, 0x16, 0x0b, 0xf5, 0x59, 0xcb,
    0x5f, 0xb0, 0x9c, 0xa9, 0x51, 0xa0, 0x7f, 0x0c, 0xf6, 0x6f, 0x17, 0xc4, 0x49, 0xec,
    0xd8, 0x43, 0x1f, 0x2d, 0xa4, 0x76, 0x7b, 0xb7, 0xcc, 0xbb, 0x3e, 0x5a, 0xfb, 0x60,
    0xb1, 0x86, 0x3b, 0x52, 0xa1, 0x6c, 0xaa, 0x55, 0x29, 0x9d, 0x97, 0xb2, 0x87, 0x90,
    0x61, 0xbe, 0xdc, 0xfc, 0xbc, 0x95, 0xcf, 0xcd, 0x37, 0x3f, 0x5b, 0xd1, 0x53, 0x39,
    0x84, 0x3c, 0x41, 0xa2, 0x6d, 0x47, 0x14, 0x2a, 0x9e, 0x5d, 0x56, 0xf2, 0xd3, 0xab,
    0x44, 0x11, 0x92, 0xd9, 0x23, 0x20, 0x2e, 0x89, 0xb4, 0x7c, 0xb8, 0x26, 0x77, 0x99,
    0xe3, 0xa5, 0x67, 0x4a, 0xed, 0xde, 0xc5, 0x31, 0xfe, 0x18, 0x0d, 0x63, 0x8c, 0x80,
    0xc0, 0xf7, 0x70, 0x07,

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

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

1.2 Measurement setup

Target: A 32-bit ARM Cortex-M3 microprocessor (SCALE board)

Oscilloscope: Pico oscilloscope 5000a.

Frequency = $2MHz$ (a clock takes $0.5\mu S$)

Sampling rate = $250MSa/s$

Each clock is sampled with 125 points

The power consumption (traces) are recorded in 16-bit integers (from the Picoscope).

The number of cycles (time executions):

1.2.1 The spot of the trigger: uELMO (simulator)

We put the main instructions in .S in uELMO, between the two lines: 22 (befor loading the inputs) and 32 (after the last STRB instruction).

Listing 1.4: Trigger in uELMO

```
.syntax unified
  .text
  .thumb
  .extern table  @ table is used for GFMULT (ai * bj multiplications)

.global Gadget_name
.func Gadget_name
Gadget_name:

    push  {r4-r7}
    mov   r4, r8
    mov   r5, r9
    mov   r6, r10
    mov   r7, r11
    push  {r4-r7}

    some instructions

# Start trigger ##############################
    ldr  r4, =0xE0000004
    movs r5, #1
    str r5, [r4, #0]
##############################################
nop
nop
nop
nop

@@@@@ Loading arguments @@@@@@@@@@@@
Gadget_Instructions
Last  strb

# End trigger ################################
    ldr  r4, =0xE0000004
    movs r5, #0
    str r5, [r4, #0]
##############################################
        pop  {r4-r7}
        mov   r8, r4
        mov   r9, r5
        mov   r10, r6
        mov   r11, r7
        pop   {r4-r7}
        pop  {pc}
        .endfunc
        .end
```

1.2.2 The spot of the trigger: in .c file on the device

We put start/end trigger before/after calling the gadget in .c file (Listing 1.5, lines 37 and 41). Notice: We have to remove the trigger instructions in in .S file (Listing 1.1 Line 26). In this method, about 80 cycles are related to the calling the gadget, push and pop instructions, that in the trace recording step they are ignored.

Listing 1.5: Trigger in .c file

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "Gadget_name_lib.h"
4     int share_n = n;
5     int rnd_n = m;
6     uint8_t shares_a[4*share_n];
7     uint8_t shares_b[4*share_n];
8     uint8_t shares_ab[4*share_n];
9     uint8_t rnd[4*rnd_n];
10        // Receiving shares of a
11        for (int i = 0; i < share_n; i++) {
12            shares_a[4*i] = (uint8_t) scale_uart_rd(SCALE_UART_MODE_BLOCKING);
13            shares_a[4*i+1]=0;
14            shares_a[4*i+2]=0;
15            shares_a[4*i+3]=0;
16        }
17        // Receiving shares of b
18        for (int i = 0; i < share_n; i++) {
19            shares_b[4*i] = (uint8_t) scale_uart_rd(SCALE_UART_MODE_BLOCKING);
20            shares_b[4*i+1]=0;
21            shares_b[4*i+2]=0;
22            shares_b[4*i+3]=0;
23        }
24        // Receiving random numbers
25        for (int i = 0; i < rnd_n; i++) {
26            rnd[4*i] = (uint8_t) scale_uart_rd(SCALE_UART_MODE_BLOCKING);
27            rnd[4*i+1]=0;
28            rnd[4*i+2]=0;
29            rnd[4*i+3]=0;
30        }
31 scale_gpio_wr( SCALE_GPIO_PIN_TRG, true  ); // Start trigger
32        Gadget_name(shares_a, shares_b, rnd), shares_ab);
33 scale_gpio_wr( SCALE_GPIO_PIN_TRG, false ); // End trigger
34        // Sending shares of c (c = a * b), c shares_ab
35        for (int i = 0; i < share_n; i++) {
36            scale_uart_wr(SCALE_UART_MODE_BLOCKING, ((char) shares_ab[4*i]));
37            shares_ab[4*i+1]=0;
38            shares_ab[4*i+2]=0;
39            shares_ab[4*i+3]=0;
40        }
41
```

1.2.3 The spot of the trigger: in .S file on the device

The trigger (trigger instructions) is spotted in the .S file, shown in Listing 1.1, line 26. However, in this method the last 10 cycles are related to the trigger and can be ignored.

Listing 1.6: Trigger in .S file

```
 @ Trigger
##########################################################################

  @ Start of trigger
  ldr  r4, =0x50013ffc
  movs r5,#1
  ldr  r6, [r4, #0]  @ r6 = 0 : SCALE_GPIO_PIN_TRG = 0
  # test: str  r6, [r3, #0]  @ r6 = 0xfc0f0000
  eors r5, r6  @ r5 = 1 @ Start trigger: SCALE_GPIO_PIN_TRG = 1
  str  r5, [r4, #0]
  nop
  nop
  nop
  nop
  @ End of trigger
  str  r6, [r4, #0] @ End trigger: r6 = 0 : SCALE_GPIO_PIN_TRG = 0
```

Table 1.1 shows the number of cycles recorded on the device and the simulator for various gadgets and their different version corresponding to the spot for the trigger.

Table 1.1: Comparison of number of cycles for different gadgets and versions

Gadget	Shares	Version	Number of cycles		
			uELMO	On device: Trig in .c file with 4 nops before trig	On device: Trig in .S file
ISW	ISW-2	V0	80	160	82
		V1	81	160	82
	ISW_3	V0	216	303	235
		V1	221	308	240
		V2	228	314	250
Dom-indep	Dom-in_2	V0			
		V1			
	Dom-in_3	V0			
		V1			
HPC1	HPC1_2	V0			
		V1			
	HPC1_3	V0			
		V1			
PINI1	PINI1_2	L-V0			
	PINI1_3	L-V0			
PINI_2	PINI2_2	L-V0			
	PINI2_3	L-V0			
BBPP	BBPP_2	L-V0			
	BBPP_3	L-V0			

2. Field multiplication

Galois field multiplication based on Double Log-Exp [7], using lookup table (in Listing 1.2) for two shares is illustrated in Algorithm 1. This multiplication is used for computing the GF multiplication of one shares (a one share (8-bit value) and b one share (8-bit value)).

$TableLOG$ contains all of the possible values of $log_g(n)$ for $n = 0, ..., 255$.
$TableALOG$ holds all of the possible values of $exp_g(n)$ for $n = 0, ..., 510$.

$$a.b = \begin{cases} exp_g(log_g(a) + log_g(b) \bmod 255) & \text{if } a \neq 0 \; and \; b \neq 0 \\ 0 & Otherwise \end{cases}$$

Algorithm 1 GF multiplication based on Double Log-Exp

Require: $a, b \in GF(2^8)$

Ensure: $c = a.b \in GF(2^8)$

```
a' ← TableLOG[a]                          ▷ Lookup table of Logarithm
b' ← TableLOG[b]
s ← a' + b'
s ← a' + b'
if a = 0 or b = 0 then
    c ← 0
else
    c ← TableALOG[s]                      ▷ Lookup table of Exponentiation
end if
return c
```

For implementing GF multiplication based on Double Log-Exp (GFMUL), several ways are considered.

2.1 GFMUL as a function

Listing 2.1: GFMUL: Function (gfmul-f)

```
  @ Computing a*b:

# uint8_t gfmul(uint8_t a, uint8_t b); For calculating: a * b
# gfmul(in_r0:a, in_r1: b, out_r0: return)

.global gfmul
.func gfmul
gfmul:
  push   {lr}
  push         {r2-r5}
    # r0 = a, r1 = b

    ldr  r2, =table     @ r2 = &table, The address of the  "table"
    ldrb r3, [r2, r0]   @ r3 = table[a]
    ldrb r4, [r2, r1]   @ r4 = table[b]
    adds r3, r4         @ r3 = s = table[a] + table[b]
    # Get the antilog: s = table[s+256]
    movs r5, #250       @ r5 = 250
    adds r5, #6         @ r5 = 256
    adds r3, r5         @ r3 = s + 256
    ldrb r4, [r2, r3]   @ r4 = s = table[s+256]
    # Checking if a = 0 or b = 0, return 0; without conditional branch
    negs r5, r0         @ r5 = -a
    movs r2, #32        @ r5 = 32
    asrs r5, r2         @ r5 = -a >> 32
    ands r5, r1         @ r5 = tmp = b & (-a >> 32)
    negs r5, r5         @ r5 = -tmp
    asrs r5, r2         @ r5 = -tmp >> 32
    ands r4, r5         @ r4 = s & (-tmp >> 32)
    movs r0, r4         @ r0 = (a * b)
    # r0 = (a * b)

  pop {r2-r5}
  pop {pc}

.endfunc
```

2.2 GFMUL as a label

Listing 2.2: GFMUL: Lable (gfmul-l)

```
  @ Computing a*b:

gfmul:
  push         {r2-r5}
    # r0 = a, r1 = b

    ldr  r2, =table     @ r2 = &table, The address of the  "table"
    ldrb r3, [r2, r0]   @ r3 = table[a]
    ldrb r4, [r2, r1]   @ r4 = table[b]
    adds r3, r4         @ r3 = s = table[a] + table[b]
    # Get the antilog: s = table[s+256]
    movs r5, #250       @ r5 = 250
    adds r5, #6         @ r5 = 256
    adds r3, r5         @ r3 = s + 256
    ldrb r4, [r2, r3]   @ r4 = s = table[s+256]
    # Checking if a = 0 or b = 0, return 0; without conditional branch
    negs r5, r0         @ r5 = -a
    movs r2, #32        @ r5 = 32
    asrs r5, r2         @ r5 = -a >> 32
    ands r5, r1         @ r5 = tmp = b & (-a >> 32)
    negs r5, r5         @ r5 = -tmp
    asrs r5, r2         @ r5 = -tmp >> 32
    ands r4, r5         @ r4 = s & (-tmp >> 32)
    movs r0, r4         @ r0 = (a * b)
    # r0 = (a * b)

  pop {r2-r5}
  bx lr

```

2.3 GFMUL as a Sequence of instructions

Listing 2.3: GFMUL: The sequence of instructions (gfmul-s)

```
@ Computing a*b:

# r3 = &table, r4 = a, r5 = b
ldrb r0, [r3, r4]   @ r0 = table[a]
ldrb r1, [r3, r5]   @ r1 = table[b]
adds r0, r1         @ r0 = s = table[a] + table[b]
# Get the antilog: s = table[s+256]: g ^ s
movs r1, #250       @ r1 = 250
adds r1, #6         @ r1 = 256
adds r0, r1         @ r0 = s + 256
ldrb r1, [r3, r0]   @ r1 = s = table[s+256]
# Checking if a = 0 or b = 0, return 0; without conditional branch
negs r0, r4         @ r0 = -a hex(256-dec(a))
movs r4, #32        @ r4 = #32
asrs r0, r4         @ r0 = -a >> 32
ands r5, r0         @ r5 = tmp = b & (-a >> 32)
negs r5, r5         @ r5 = -tmp
asrs r5, r4         @ r5 = -tmp >> 32
ands r5, r1         @ r5 = s & (-tmp >> 32) = (a * b)
# r5 = (a * b)
```

2.4 Comparison

For using gfmul-f and gfmul-l, a and b must be in registers $r0$, $r1$, and the result $(a.b)$ is in the register $r0$ (Listing 2.4). there is no limitations on registers for using gfmul-s.

Listing 2.4: Calling gfmul-s and gfmul-l

```

# r0 = a (b)
# r1 = b (a)

bl    gfmul

# result:
#r0 = a * b


```

Regarding Listing 2.1, 2.2 and 2.3 gfmul-f, gfmul-l and gfmul-s need less cycles for doing the multiplication, respectively.

3. Split-t-test

Finding leakages via splitting t-test in to three t-test s:
Finding/Removing the leakages caused by 2 shares out of 3 shares. For this we apply 3 t-test s on 2 shares from 3 shares.As there are 3 shares, 3 possibilities exist for two shares out of the 3 shares $(sh_0, sh_1), (sh_0, sh_2)$ and (sh_1, sh_2). Then we apply t-test on the three possibilities.

t-test on 3 shares out of 3 shares:

$a = a_0 \oplus a_1 \oplus a_2$

$b = b_0 \oplus b_1 \oplus b_2$

random set:

a random: a_0, a_1 and a_2 random

b random: b_0, b_1 and b_2 random

fixed set:

a fix: a_0, a_1 random, $a_2 = a \oplus a_0 \oplus a_1$

b fix: b_0, b_1 random, $b_2 = b \oplus b_0 \oplus b_1$

Then, applying (just one time) t-test on random set and fixed test

t-test on 2 shares out of 3 shares:

$a = a_0 \oplus a_1 \oplus a_2$

$b = b_0 \oplus b_1 \oplus b_2$

$(sh_i, sh_j)(i \neq j)$

random set:

$(x_i \oplus x_j)$ random: x random, and $x_{3-(i+j)} = x \oplus (x_i \oplus x_j)$

fixed set:

$(x_i \oplus x_j)$ fix: x random, and $x_{3-(i+j)} = x \oplus (x_i \oplus x_j)$

1) (sh_0, sh_1)

random set:

$(a_0 \oplus a_1)$ random: a random, and $a_2 = a \oplus (a_0 \oplus a_1)$

$(b_0 \oplus b_1)$ random: b random, and $b_2 = b \oplus (b_0 \oplus b_1)$

fixed set:

$(a_0 \oplus a_1)$ fix: a random, and $a_2 = a \oplus (a_0 \oplus a_1)$

$(b_0 \oplus b_1)$ fix: b random, and $b_2 = b \oplus (b_0 \oplus b_1)$

Then, applying (just one time) t-test on random set and fixed test

2) (sh_0, sh_2)

random set:

$(a_0 \oplus a_2)$ random: a random, and $a_1 = a \oplus (a_0 \oplus a_2)$

$(b_0 \oplus b_2)$ random: b random, and $b_1 = b \oplus (b_0 \oplus b_2)$

fixed set:

$(a_0 \oplus a_2)$ fix: a random, and $a_1 = a \oplus (a_0 \oplus a_2)$

$(b_0 \oplus b_2)$ fix: b random, and $b_1 = b \oplus (b_0 \oplus b_2)$

Then, applying (just one time) t-test on random set and fixed test

3) (sh_1, sh_2)

random set:

$(a_1 \oplus a_2)$ random: a random, and $a_0 = a \oplus (a_1 \oplus a_2)$

$(b_1 \oplus b_2)$ random: b random, and $b_0 = b \oplus (b_1 \oplus b_2)$

fixed set:

$(a_1 \oplus a_2)$ fix: a random, and $a_0 = a \oplus (a_1 \oplus a_2)$

$(b_1 \oplus b_2)$ fix: b random, and $b_0 = b \oplus (b_1 \oplus b_2)$

Then, applying (just one time) t-test on random set and fixed test

4. ISW

ISW multiplication is introduced in 2003 [10] and generized in [12]. Algorithm 2 represents this scheme for $d + 1$ shares.

Algorithm 2 ISW multiplication gadget

Require: shares a_i, b_i $(0 \leq i \leq d)$ such as $a = \oplus_{i=0}^{d} a_i$, $b = \oplus_{i=0}^{d} b_i$
Ensure: $c_0, ..., c_d$ such that $c = \oplus_{i=0}^{d} c_i = a \cdot b$

```
for i = 0 to d do
    for j = i + 1 to d do
        r_ij ←$ {0,1}^n
        r_ji ← [r_ij ⊕ (a_i · b_j)] ⊕ (a_j · b_i)
    end for
end for
for i = 0 to d do
    c_i ← (a_i · b_i)
    for j = 0 to d do
        if i ≠ j then
            c_i ← c_i ⊕ r_ij
        end if
    end for
end for
return c_0, ..., c_d
```

5. ISW-2- no function/label call

The ISW multiplication for two shares (ISW-2) is illustrated in Algorithm 3 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 3 ISW-2

Require: shares a_0, a_1, b_0, b_1, rnd

Ensure: c_0, c_1

1: $c_0 \leftarrow (a_0.b_0) \oplus rnd$

2: $c_1 \leftarrow [((a_0.b_1) \oplus rnd) \oplus (a_0.b_1)] \oplus (a_1.b_1)$

In the implementation of ISW-2, for the GF multiplications $(a_i.b_j)$, the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

5.1 ISW-2-no-function-call-0: basic code

Listing 5.1 demonstrates the first implementation of ISW-2 based gfmul-s (ISW-2-no-function-call-0). The instructions in Listing 5.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 5.1: ISW-2-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @ r8 = &c


@@@@ Saving the value 256 in r11. This decreases one cycle.
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r4, #250       @ r4 = 250, 256 is not possible, so it is split: 250 + 6
  adds r4, #6         @ r4 = 256
  mov  r11, r4        @ r11 = 256


  ldr  r3, =table    @ r3 = &table, The address of the "table", for ai * bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r9, r3        @ r9 = &table


nop
nop
nop
nop



@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
@@@@ r8 = &c, r9 = &table, r11 = 256
@@@@@
```

```
@@@
```

```
@@@@ r4 = a1, r5 = b0, r6 = a0, r7 = b1,
@@@@ r8 = &c, r9 = &table, r11 = 256
@@@@
```

```
@@@
```

```
ldrb r5, [r1, r4]   @ r5 = table[a1]
ldrb r3, [r1, r7]   @ r3 = table[b1]

adds r5, r3         @ r5 = s = table[a1] + table[b1]

@@@@ Get the antilog: s = table[s+256]: g + s
mov  r2, r11        @ r2 = 256

adds r5, r2         @ r5 = s + 256

ldrb r3, [r1, r5]   @ r3 = s = table[s+256]

# @@@@ Checking if a1 = 0 or b1 = 0, return 0; without conditional branch
negs r5, r4         @ r5 = -a1 hex(256-dec(a1))

# @ asrs rd, <#imm> --> #imm: 0-31
# movs r6, #32        @ r6 = #32
asrs r5, r6         @ r5 = -a1 >> 32

ands r5, r7         @ r5 = tmp = b1 & (-a1 >> 32)
negs r5, r5         @ r5 = -tmp

# @ asrs rd, <#imm> --> #imm: 0-31
asrs r5, r6         @ r5 = -tmp >> 32

ands r3, r5         @ r3 = s & (-tmp >> 32) = (a1 * b1)


mov  r1, r8         @ r1 = &c

@@@@ c1
eors r3, r0         @ r3 =  [((a0 * b1) + rnd) + (a1 * b0)] + (a1 * b1)

strb r3, [r1, #4]   @ Storing c1, r3 = c1

nop
nop
nop
nop
```

5.1.1 Experimental result:

5.1.1.1 Setup:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $78\mu s$
Each clock is sampled with 125 points
Number of samples = 120×125 = 15000 samples
The bytes of shares (a, b, c are not in a word)
Figure 5.1 shows the recorded 15KS before raising-edge of the trigger.

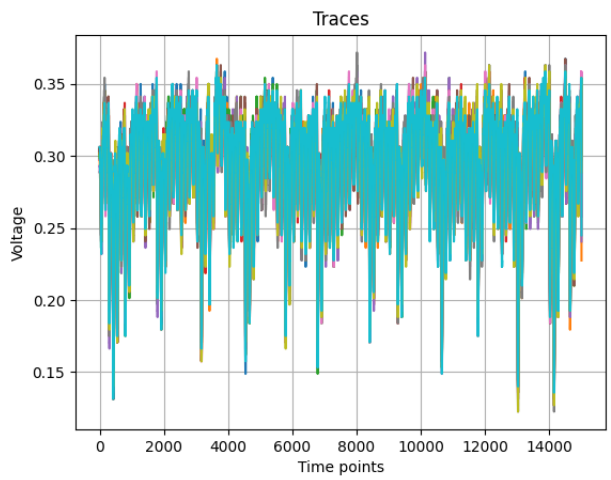


Figure 5.1: Traces isw-2-without-b-0 (Setup: 250MSa/s, 15kS, 4ns, 2MHz)

5.1.1.2 Signal to Noise Ratio (SNR):

Figure 5.2 shows the SNR for all the intermediate values. It can be concluded that the first 3750 (30×125) and also the last 1000 (8×125) samples do not have effect on the SNR, so, they can be removed.

Number of samples = $(120 - 30 - 8) \times 125 = 82 \times 125 = 10250$ samples

Figure 5.3 - 5.7 show the SNR of traces with 10250 samples for different intermediate values.

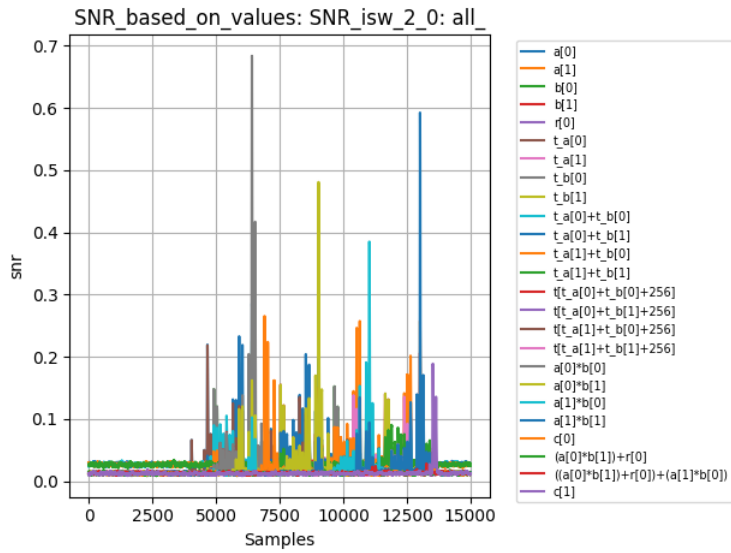


Figure 5.2: SNR isw-2-without-b-0 (Setup: 250MSa/s, 15kS, 4ns, 2MHz)

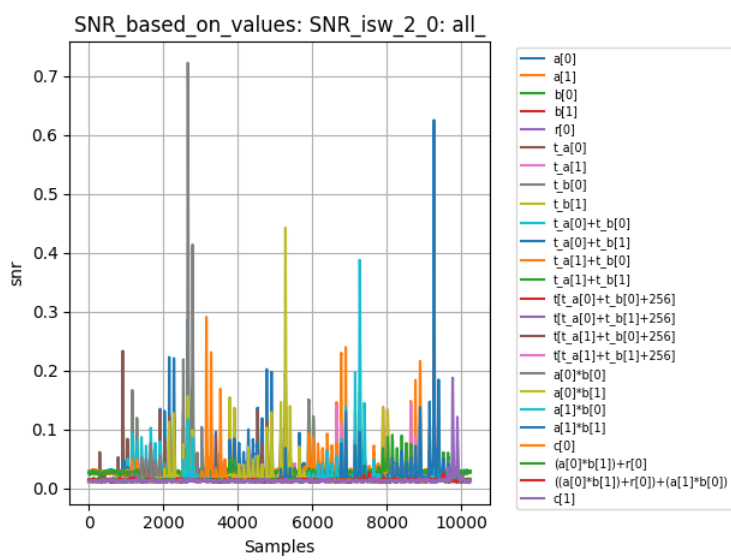
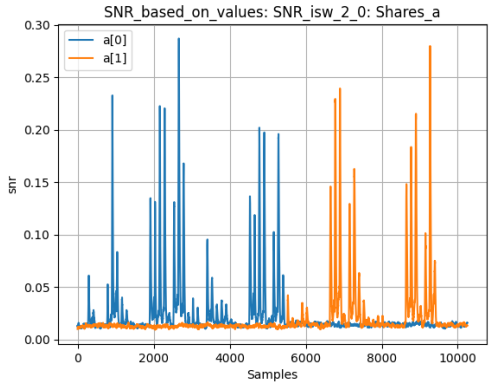
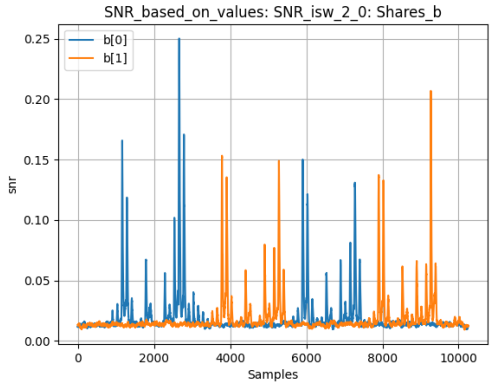


Figure 5.3: SNR isw-2-without-b-0 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

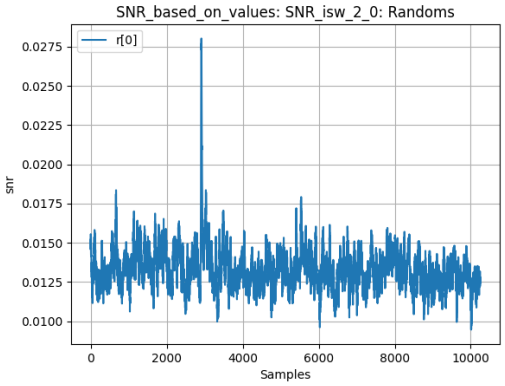
(a) Shares-a



(b) Shares-b



(c) Randoms



(d) Inputs of the gadget

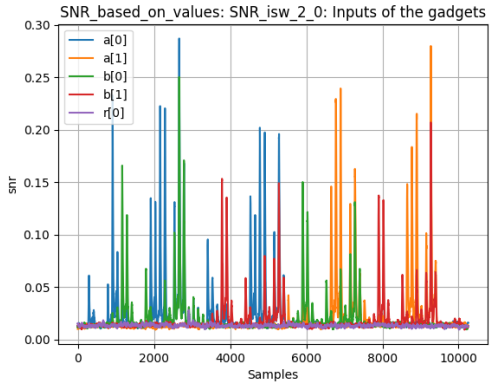
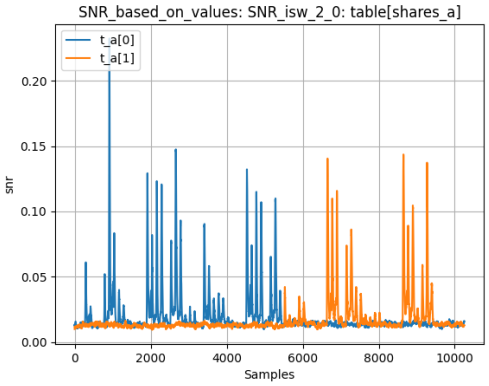
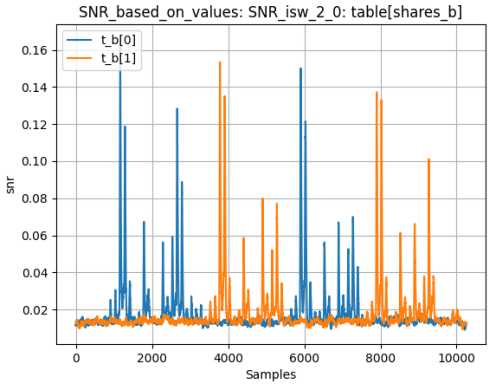


Figure 5.4: SNR isw-2-without-b-0-inputs (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

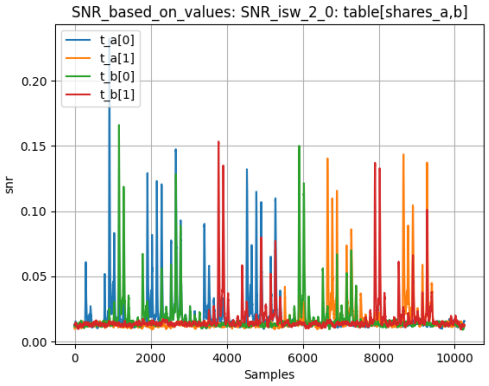
(a) Table of Shares-a



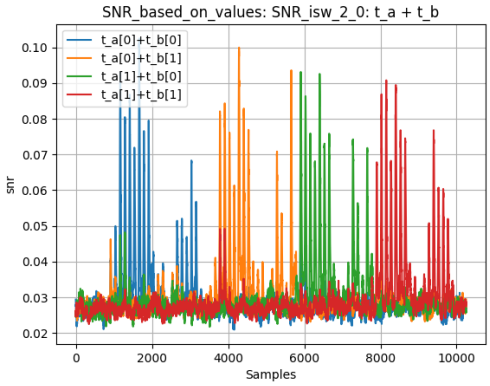
(b) Table of Shares-b



(c)



(d) Table of Shares-a Plus Table of Shares-b



(e) T of T-a plus T-b plus 256

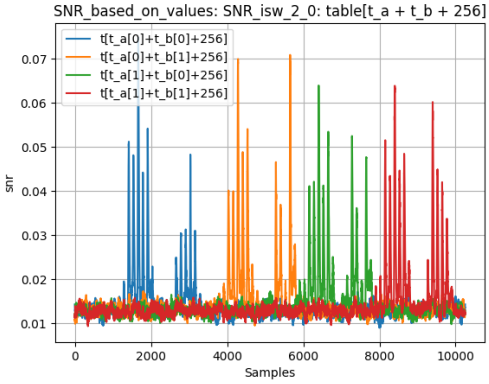


Figure 5.5: SNR isw-2-without-b-0-tables (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

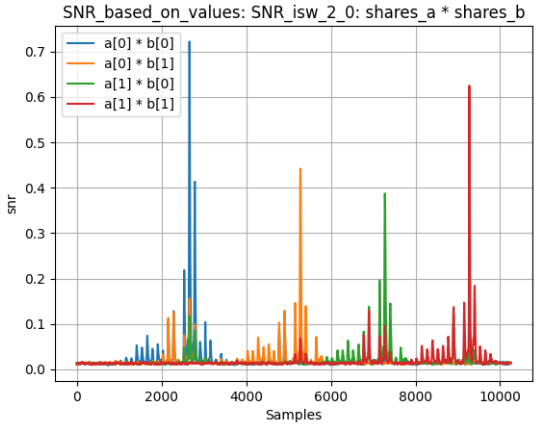
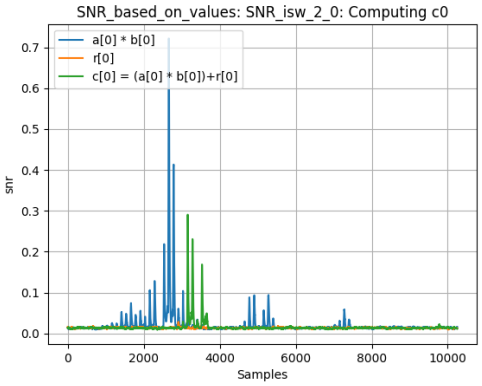
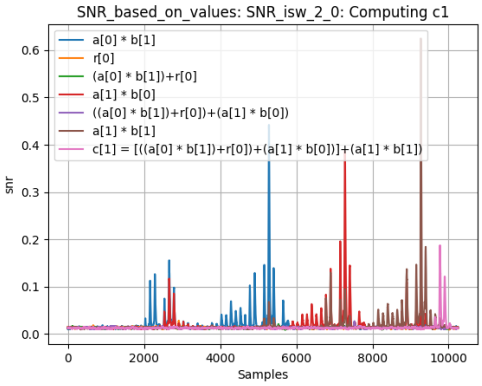


Figure 5.6: SNR isw-2-without-b-0 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

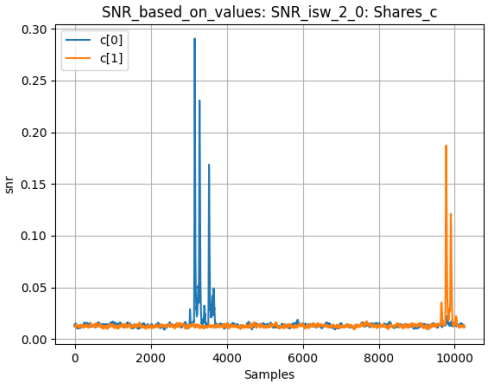
(a) Computing c0



(b) Computing c1



(c) Shares-c



(d) Outputs of the gadget

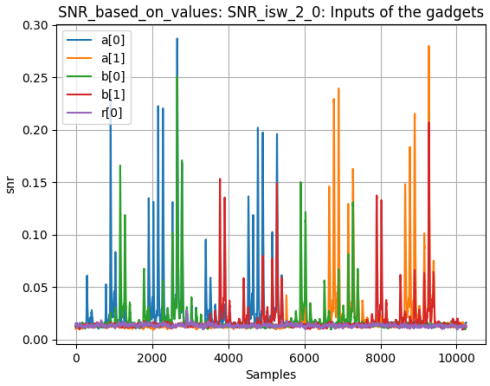
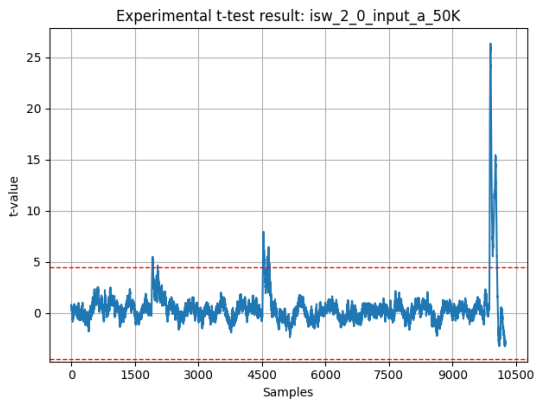


Figure 5.7: SNR isw-2-without-b-0-outputs (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

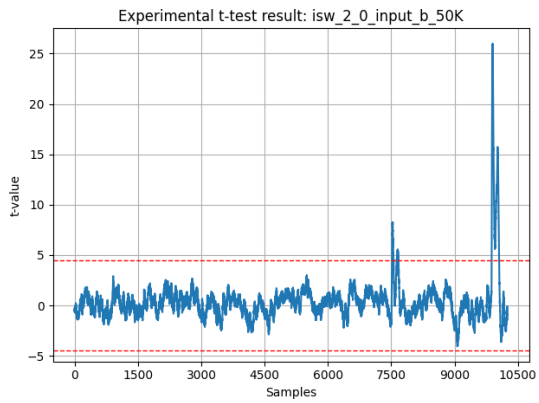
5.1.1.3 t-test:

Figure 5.8 shows the fixed-versus-random t-test result on both inputs a and b for 50k traces. The leakages occur at cycles: 16, 17, 37, 38, 61, 62, 80 and 81, Which are correspond to lines:
63, 110, 173, 174, 79, 206,
in ISW-2-no-function-call-0.

(a) t-test on input a



(b) t-test on input b



(c) t-test on both inputs a and b

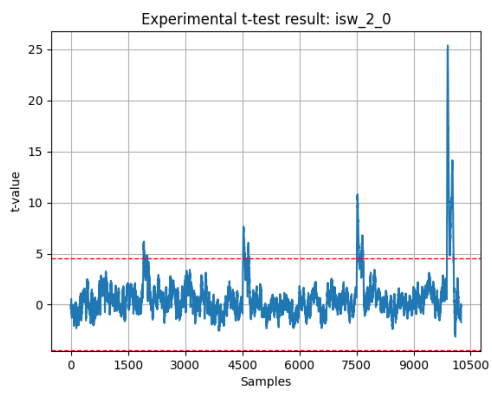


Figure 5.8: t-test isw-2-without-b-0-(Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Also, we used the leakage simulator uELMO, which reportes the same leakages (Listing 5.2).

Listing 5.2: Leakages reported by **uELMO** for isw-2-0

```
Cycle 24: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x200001FD]=0x97
, Term=Glitchy decoding port 1 XOR current port 1, Tvalue=inf
Cycle 24: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x200001FD]=0x97
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 25: Instr=Execute: negs r0,r6
Decode: movs r3,#0x20
Memory: idle
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 43: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x200001FD]=0x97
, Term=Glitchy decoding port 1 XOR current port 1, Tvalue=inf
Cycle 43: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x200001FD]=0x97
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 44: Instr=Execute: negs r0,r6
Decode: movs r6,#0x20
Memory: idle
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 68: Instr=Execute: ldrb r3,[r1,r7]
Decode: adds r5,r5,r3
Memory: load [0x200000DC]=0x41
, Term=Decoding port 2 HD, Tvalue=inf
Cycle 84: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: store [0x20001FBC]=0xEE
, Term=Memory write buffer HD, Tvalue=inf
Cycle 85: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: idle
, Term=Memory write buffer delayed HD, Tvalue=inf
```

Reasons and solutions:
1)

Glitchy register access [5]: (Glitch because of decoding the current instruction in the decoding format of the previous instruction)

Cycles 16-17, 37-38
lines 63 and 110 in ISW-2-no-function-call-0.
Regarding Listing 5.3, during the execution of negs r0, r6 instruction, the movs r3, #32 instruction (which was encoded to 0x2320: 0010001100100000) is decoded in the Decoder stage of the pipeline. The 0x2320 (movs) is decoded at first in the negs style (5.9 a), then it is decoded according to the movs format (5.9 b), so that a glitch will occur here.
Assuming for INSRT Rd, Rm, Rd and Rm are loaded into $port1$ and $port2$, respectively. Regarding it, in the decoding stage of negs r0, r6, $port2$ is loaded with $r6$ ($r6 = a0$), now, when movs r3, #32 is decoded in the negs format , the content of the register with the address bits 5-3 (0x2320: 0010 0011 0010 0000, which is 100 = 4 and means register $r4$) is loaded into $port2$. As $r4 = a1$ and $r6 = a0$, so there is a leakage of the input a.
Also the same reason causes leakage in the movs r6, #32 (Line 110 in ISW-2-no-function-call-0), Listing 5.4.

Listing 5.3: First glitchy register access leakage in isw-2-0

```

# r0 = table[a0] + table[b0] + 256,
# r1 = table[table[a0] + table[b0] + 256],
# r2 = rnd,
# r3 = &table,
# r4 = a1,
# r5 = b0,
# r6 = a0,
# r7 = b1

  negs r0, r6          @ r0 = -a0 hex(256-dec(a0))
  movs r3, #32         @ r3 = #32 (0x2320)
#  Encoding movs r3, #32: 0x2320 : 0010 0011 0010 0000

```

From ARM Architecture: Negate is a pre-UAL synonym for RSB (immediate) with an immediate value of 0. For details see RSB (immediate).

For removing this kind of leakage, we can use two approaches as the follow:
1) Instead of using movs rx, #32, using the two instructions: movs rx, #16 and adds rx, #16
With this approach, when the movs is decoded in the negs format, the bits 5-3 would be 010 = 2 and means the content of register $r2$ is loaded into $port2$. As $r2 = rnd$ and $r6 = a0$, so there would be

Listing 5.4: Second glitchy register access leakage in isw-2-0

```
# r0 = table[a0] + table[b1] + 256,
# r1 = &table,
# r2 = rnd,
# r3 = table[table[a0] + table[b1] + 256],
# r4 = a1,
# r5 = b0,
# r6 = a0,
# r7 = b1

  negs r0, r6          @ r0 = -a0 hex(256-dec(a0))
  movs r6, #32         @ r3 = #32 (0x2620)
#  Encoding movs r6, #32: 0x2620 : 0010 0110 0010 0000
```

(a) Encoding NEG instruction

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

`RSBS <Rd>, <Rn>, #0`	Outside IT block.
`RSB<c> <Rd>, <Rn>, #0`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	1	Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

(b) Encoding MOV instruction

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

`MOVS <Rd>, #<imm8>`	Outside IT block.
`MOV<c> <Rd>, #<imm8>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Figure 5.9: Encoding format of NEG and MOV instruction (ARM Architecture)

no leakage of the input a (Figure 5.10 and 5.11).

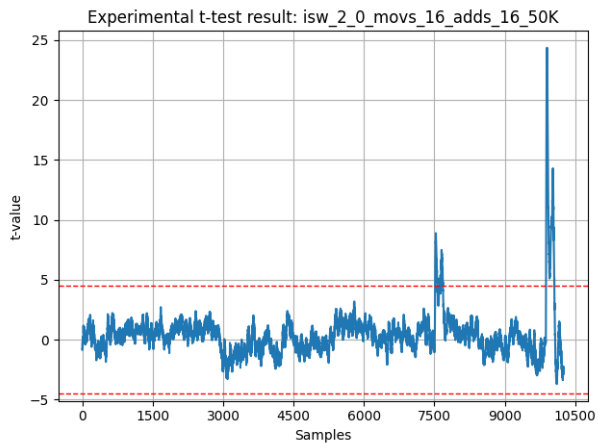


Figure 5.10: Experimental T-test result of isw-2-without-b-0 - patch: movs16 and adds16 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

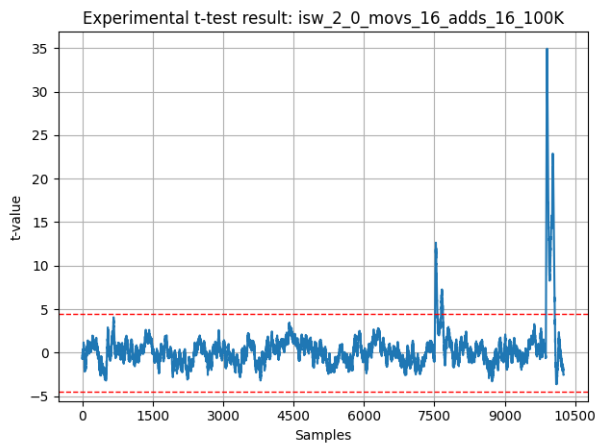


Figure 5.11: Experimental T-test result of isw-2-without-b-0 - patch: movs16 and adds16 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

2) Through the implementation of ISW-2-no-function-call-0, changing the content of $r3$ and $r4$ with each other. This approach does not need extra instruction. Figure 5.12 and 5.13 show the results. As a result, in Listing 5.3, $r3 = a1$, and $r4 = \&table$ as well as in Listing 5.4, $r3 = a1$, and $r4 = table[table[a0] + table[b1] + 256]$, so that there is no leakage on $r4$ and $r6$.
This way also removes the next leakage. WHY: DELAY on switching the values of wires in multiplexer tree.

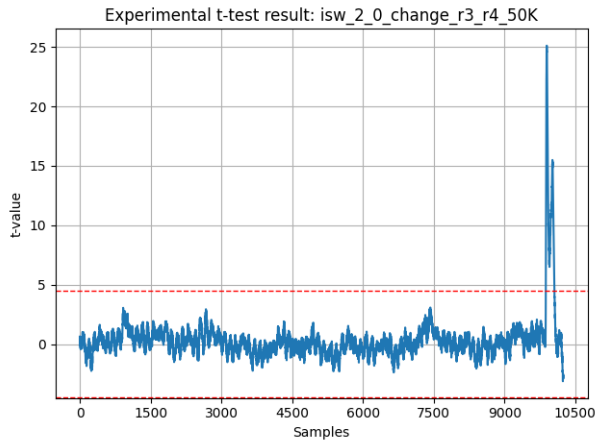


Figure 5.12: Experimental T-test result of isw-2-without-b-0 - patch: changing r3 and r4 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

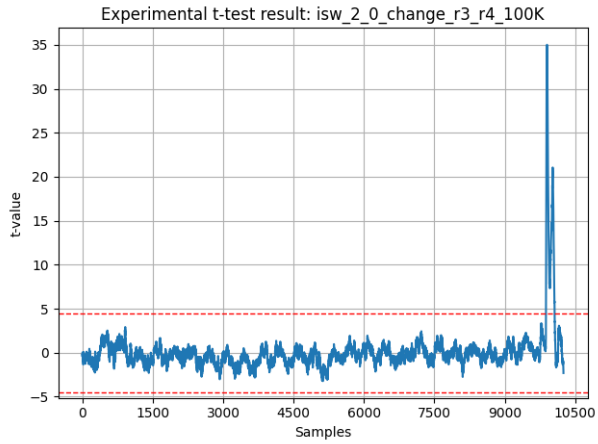


Figure 5.13: Experimental T-test result of isw-2-without-b-0 - patch: changing r3 and r4 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

2) Delay on switching values in SEL0 signal in multiplexer tree (Figure 14.9):

Cycles: 61-62
Lines 173, 174
The sources of this type of leakage are pointed in Listing 5.5.

Listing 5.5: Source of the leakages in isw-2-without-b-0, delay on switching

```
# r0 = ((a0*b1)+rnd)+(a1*b0),
# r1 = &table,
# r2 = ((a0*b1)+rnd),
# r3 = table[table[a1] + table[b0] + 256],
# r4 = a1,
# r5 = b0,
# r6 = 32,
# r7 = b1

  ldrb r5, [r1, r4]   @ r5 = table[a1]
  ldrb r3, [r1, r7]   @ r3 = table[b1]
  adds r5, r3         @ r5 = s = table[a1] + table[b1]
```

Description about the register tree:
As all codes are based on THUMB-16, just Low registers are considered in the figure.

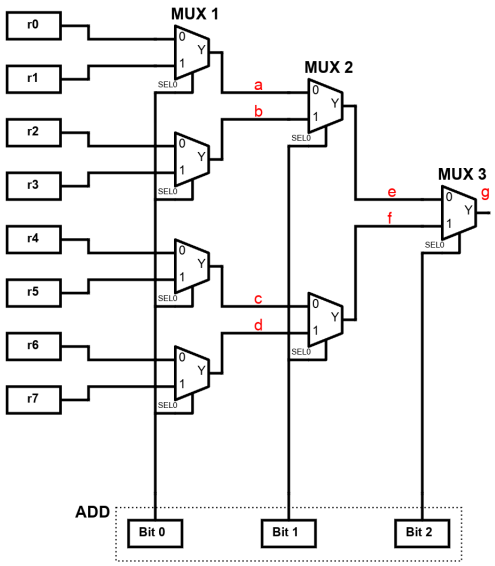


Figure 5.14: Multiplexer tree

From paper [5], Table 1:
In Decoding stage of ldrb Rd, [Rn, Rm] instruction:

Port1 $\leftarrow$ Rd
Port2 $\leftarrow$ Rn
Port3 $\leftarrow$ Rm
and From paper [6], register tree (Figure 14.9).
Regarding Listing 5.5,
According [5]: in the decode of the first ldrb, register $r4$ (address in binary: 100) is imported into $Port3$. Then in the next cycle, register $r7$ (address in binary: 111) (from the second ldrb) must be imported into $Port3$.
First cycle: According [6]: For $r4$, SEL0 signals (ADD: Bit0: 0, Bit1: 0, Bit2: 1) the multiplexers will be 001, so that $r4$ (=$a1$) goes on the wires c, f and g.
Second cycle: According [6]: For $r7$, SEL0 signals (ADD: Bit0: 1, Bit1: 1, Bit2: 1) the multiplexers will switch to 111, so that:

There is a delay in switching values in multiplexer values (Bit_i), and the leakage occurs on the wires of the multiplexer tree (wires: a-g in Figure 14.9, not in decoding ports or pipeline registers:
Bit0: $0 \rightarrow 1$:
Because of the delay, first $r4$ ($a1$) then $r5$ ($b0$) goes on wire c.
Because of the delay, first $r6$ (32) then $r7$ ($b1$) goes on wire d.
Bit1: $0 \rightarrow 1$:
Because of the delay, first the value of the wire c ($r5 = b0$) then the value of the wire d ($r7 = b1$) goes on the wire f.
As a result, on the wire f, input b ($b0 \oplus b1$) will leak.

There are 4 ways to remove this leakage as listed in Listing 5.6, 5.7, 5.8 and 5.9.
1) Changing the order of the 2 LDRB instructions (delay on switching wires) (Listing 5.6, Figure 5.15)
2) Clearing $b0$ from $r5$ (Listing 5.7, Figures 5.16, 5.17)
3) Clearing the read bus (Listing 5.8, Figures 5.18, 5.19))
4) Replacing $r3$ and $r4$, means, $r3 = a1$ (Listing 5.9, Figures 5.12, 5.13)
Why replacing the content of $r3$ and $r4$ removes this leakage???
In this way, the shares of input b ($b0$ and $b1$) will not be the inputs of the same multiplexer.

Listing 5.6: changing the order of the two ldrb instructions

```
# r0 = ((a0*b1)+rnd)+(a1*b0),
# r1 = &table,
# r2 = ((a0*b1)+rnd),
# r3 = table[table[a1] + table[b0] + 256],
# r4 = a1,
# r5 = b0,
# r6 = 32,
# r7 = b1

  ldrb r3, [r1, r7]   @ r3 = table[b1]
  ldrb r5, [r1, r4]   @ r5 = table[a1]

```

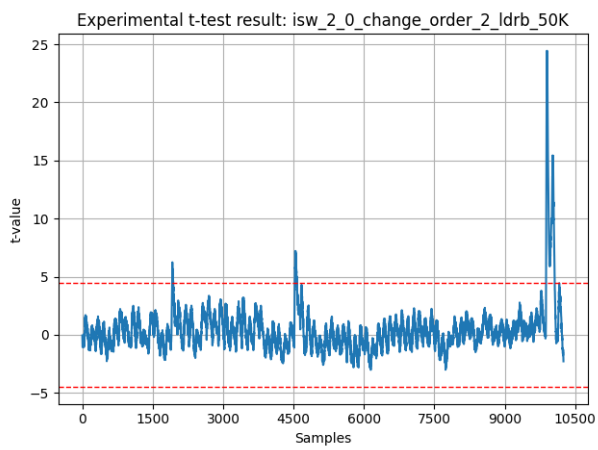


Figure 5.15: Experimental T-test result of isw-2-without-b-0 - patch: Changing the order of two ldrb(Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Listing 5.7: Clearing r5 (=b0), removing $b0$ on the wire c

```
# r0 = ((a0*b1)+rnd)+(a1*b0),
# r1 = &table,
# r2 = ((a0*b1)+rnd),
# r3 = table[table[a1] + table[b0] + 256],
# r4 = a1,
# r5 = b0,
# r6 = 32,
# r7 = b1

  eors r5, r5         @ Clearing b0 from r5, r5 = 0
  ldrb r5, [r1, r4]   @ r5 = table[a1]
  ldrb r3, [r1, r7]   @ r3 = table[b1]
  adds r5, r3         @ r5 = s = table[a1] + table[b1]
```

Figure 5.16: Experimental T-test result of isw-2-without-b-0 - patch: clearing b0 from r5 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

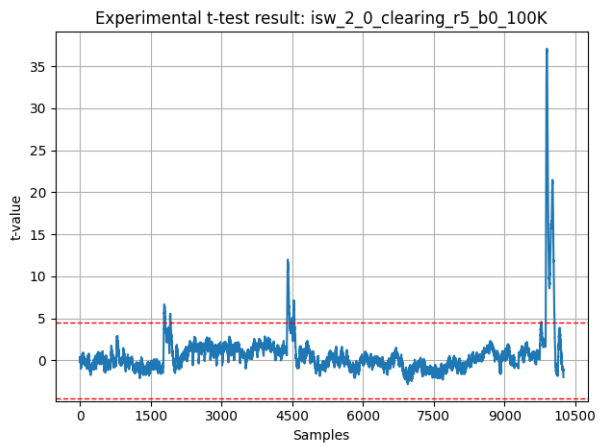


Figure 5.17: Experimental T-test result of isw-2-without-b-0 - patch: clearing b0 from r5 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Listing 5.8: Clearing the read bus, changing the values on the wires

```
# r0 = ((a0*b1)+rnd)+(a1*b0),
# r1 = &table,
# r2 = ((a0*b1)+rnd),
# r3 = table[table[a1] + table[b0] + 256],
# r4 = a1,
# r5 = b0,
# r6 = 32,
# r7 = b1

  ldrb r5, [r1, r4]   @ r5 = table[a1]
  ldrb r3, [r1, #0]   @ Clearing the read bus, r3 = table[0]
  ldrb r3, [r1, r7]   @ r3 = table[b1]
  adds r5, r3         @ r5 = s = table[a1] + table[b1]
```

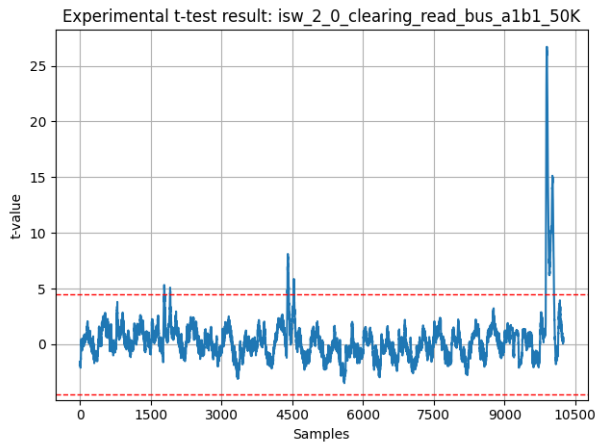


Figure 5.18: Experimental T-test result of isw-2-without-b-0 - patch: clearing b0 from r5 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

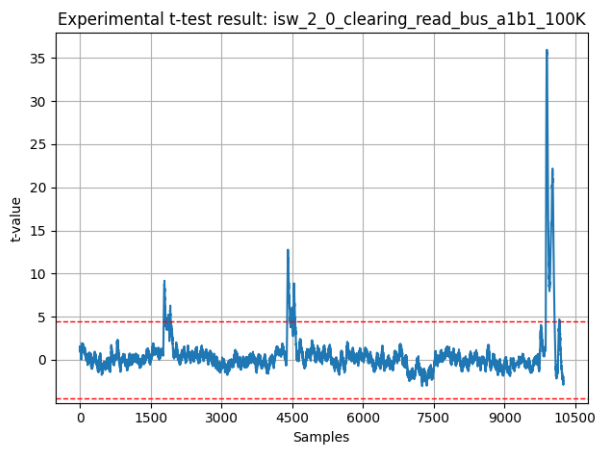


Figure 5.19: Experimental T-test result of isw-2-without-b-0 - patch: clearing b0 from r5 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Listing 5.9: Changing r3 and r4, removing the effect of switching delay

```
# r0 = ((a0*b1)+rnd)+(a1*b0),
# r1 = &table,
# r2 = ((a0*b1)+rnd),
# r3 = a1,
# r4 = table[table[a1] + table[b0] + 256],
# r5 = b0,
# r6 = 32,
# r7 = b1

  ldrb r5, [r1, r3]   @ r5 = table[a1]
  ldrb r4, [r1, r7]   @ r4 = table[b1]
  adds r5, r4         @ r5 = s = table[a1] + table[b1]
```

3) The write bus (str instruction):
When two secret shares are loaded into the write bus in succession, it causes leakages (Section **??**). In ISW-2 implementation (Listing 5.1), storing shares of c (Listing 5.10) which are the instructions in lines 79 and 206 in Listing 5.1 made this type of leakage.

Listing 5.10: Source of the leakages in isw-2-without-b-0, storing shares of c

```
1
2   strb r1, [r3, #0]   @ r3 = &c, Storing c0, r1 = c0
3
4     A sequence of instructions
5
6   strb r3, [r1, #4]   @ r1 = &c, Storing c1, r3 = c1
7
```

For leakages related to the write bus, clearing the write bus removes them (Listing 5.11). Figure 5.20 shows the result of applying this patch.

Listing 5.11: Patching the write bus leakage in isw-2-without-b-0 (storing shares of c)

```
1   strb r1, [r3, #0]   @ r3 = &c, Storing c0, r1 = c0
2
3     A sequence of instructions
4
5     strb r1, [r1, #4]   @ Clearing the write bus, r1 = &c
6
7   strb r3, [r1, #4]   @ r1 = &c, Storing c1, r3 = c1
8
```

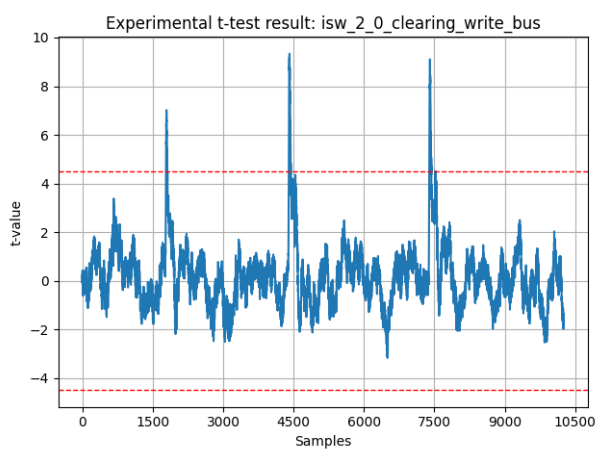


Figure 5.20: Experimental T-test result of isw-2-without-b-0 - patching the write bus (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Less leaky implementation of ISW-2 by applying the below patches (two patches): There are other ways, but according to our experiments, this way needs to insert less instructions.
1) Clearing the write bus
2) Changing $r3$ and $r4$ and clearing the write bus on isw-2-0. Figure 5.21 shows the t-test results on of isw-2-1 for 100K and 500K traces.

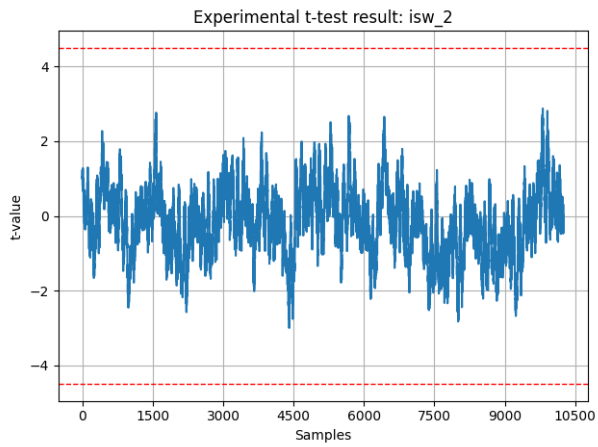


Figure 5.21: Experimental T-test result of isw-2-without-b-0 - patching the write bus and changing r3 and r4 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

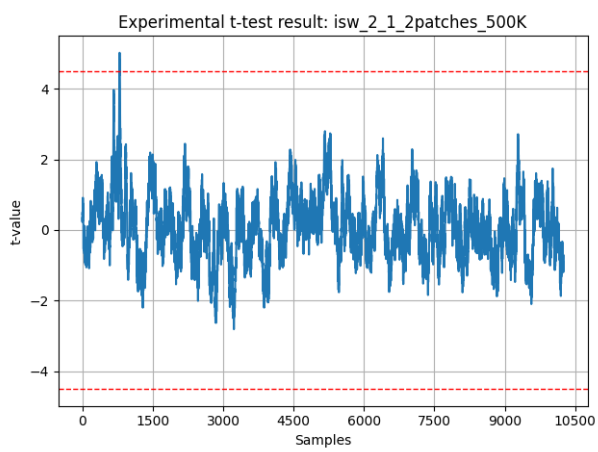


Figure 5.22: Experimental T-test result of isw-2-without-b-0 - patching the write bus and changing r3 and r4 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Tiny leakage: As it can be seen in the Figure 5.22 (also in Figure 5.23), by increasing the number of traces up to 500K, there is a tiny leakage between samples 0-1500. This is related to the Lines 47 and 48, listed in 5.13.
This leakage is caused by the switching delay on the values of wires of multiplexer tree (Figure 14.9). In first cycle, $r6$ (bin: 110, ADD: 011) and in second cycle $r5$ (bin: 101, ADD: 101). Therby, the multiplexer SEG0 has to change from 011 to 101. For this transition,
Bit0: $0 \rightarrow 1$:
Because of the delay, first $r4$ ($a1$) then $r5$ ($b0$) goes on wire c.
Because of the delay, first $r6$ ($a0$) then $r7$ ($b1$) goes on wire d.
Bit1: $1 \rightarrow 0$:

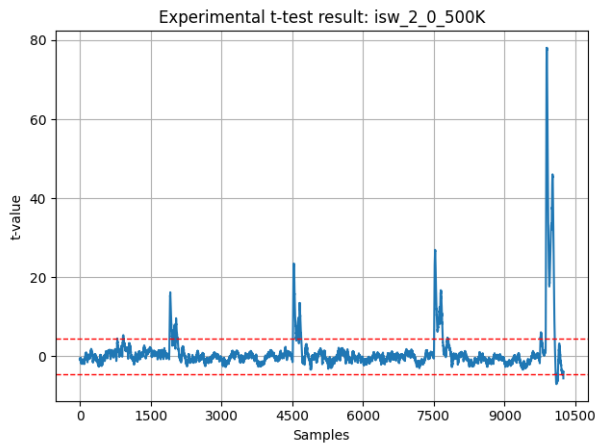


Figure 5.23: Experimental T-test result of isw-2-without-b-0 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Because of the delay, first the value of the wire d ($r7 = b1$) then the value of the wire c ($r5 = b0$) goes on the wire f.
As a result, on the wire f, input b ($b0 \oplus b1$) will leak.

Listing 5.12: Switching delay leakage (in $a0 \times b0$) isw-2-without-b-0

```

# r0 = &a,
# r1 = &b,
# r2 = rnd,
# r3 = &table
# r4 = a1,
# r5 = b0,
# r6 = a0,
# r7 = b1,

  ldrb r0, [r3, r6]   @ r0 = table[a0]
  ldrb r1, [r3, r5]   @ r1 = table[b0]

```

To remove this tiny leakage (without inserting an extra instruction), we can change the order of instructions between Lines 30 and 48 in a way that there will not be a leakage on read bus. For example if we just change the order of the two ldrb in Listing 5.13, there will be a leakage on shares of b the read bus, as share of b are loading in succession.
Figure 5.24 shows the t-test result on of isw-2-1 for 800K traces.

Listing 5.13: Changing the order of instructions for removing switching delay leakage (in $a0 \times b0$) isw-2-without-b-0

```

@@@@@ Loading arguments @@@@@@@@@@@@
  ldrb r6, [r0, #0] @ r6 = a0;
  ldrb r5, [r1, #0] @ r5 = b0;
  ldrb r3, [r0, #4] @ r3 = a1;
  ldrb r7, [r1, #4] @ r7 = b1;
  ldrb r2, [r2, #0] @ r2 = rnd;


@@@@@@@@@ Computing c0 = (a0 * b0) + rnd

@@@@ r0 = &a, r1 = &b, r2 = rnd, r4 = &table
@@@@ r3 = a1, r5 = b0, r6 = a0, r7 = b1,
@@@@ r8 = &c, r9 = &table, r11 = 256
@@@@@@@@@@@
```

Wrong guess about the first three leakages and their solutions:

At first, the sources of leakages in Figure 5.8, in particular cycles: 16, 17, 37, 38, 61, 62, where assumed to be related to the read bus. So that, the clearing the read bus was used as the solution like it was mentioned in Section **??** (Listing 5.14).

This removes the first three leakages for 50k traces.

Figure 5.25 shows the experimental result of applying these patches for 50K traces. But, for more traces the first two leakages again appeared. Figures 5.26 and 5.27 show these leakages, notice: in these traces the write bus was cleared, so there is no leakage around cycles 80, 81.

Listing 5.14: Patching the read bus leakages in isw-2-without-b-0 (loading table of shares)

```
ldrb r0, [r3, r6]   @ r3 = &table, r6 = a0, r0 = table[a0]

  ldrb r1, [r3, #0]   @ clearing the read bus, r1 = table[0]= 0

ldrb r1, [r3, r5]   @ r3 = &table, r5 = b0, r1 = table[b0]
@@@
```

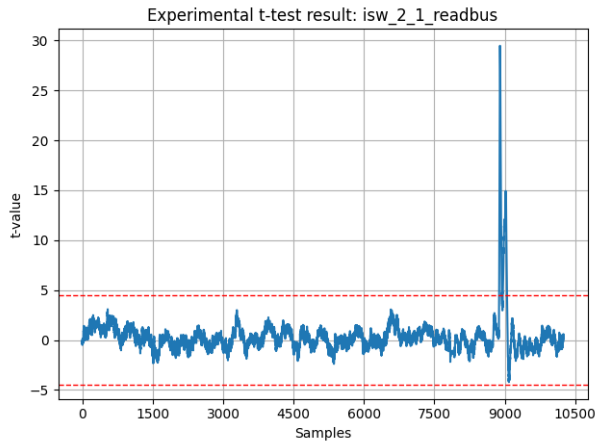


Figure 5.25: Experimental T-test result of isw-2-without-b-0 - patching the read bus (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

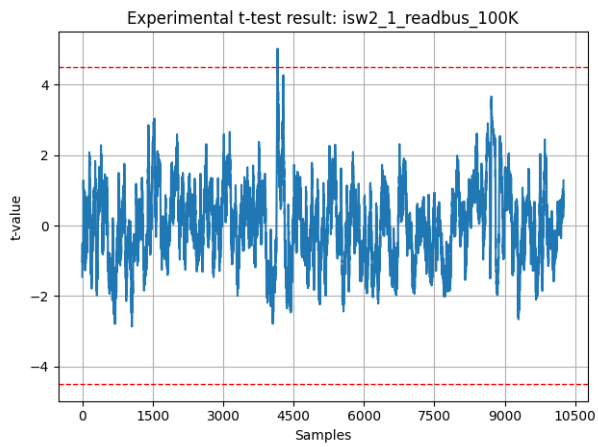


Figure 5.26: Experimental T-test result of isw-2-without-b-0 - patching the read bus (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

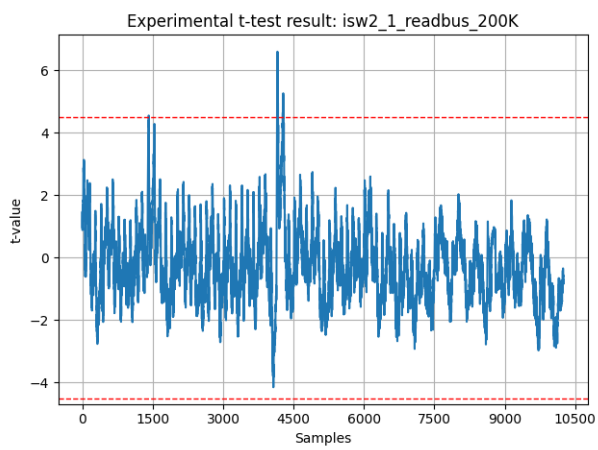


Figure 5.27: Experimental T-test result of isw-2-without-b-0 - patching the read bus (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Finally, Figure 5.28 indicates the t-test result of conducting the both patches.

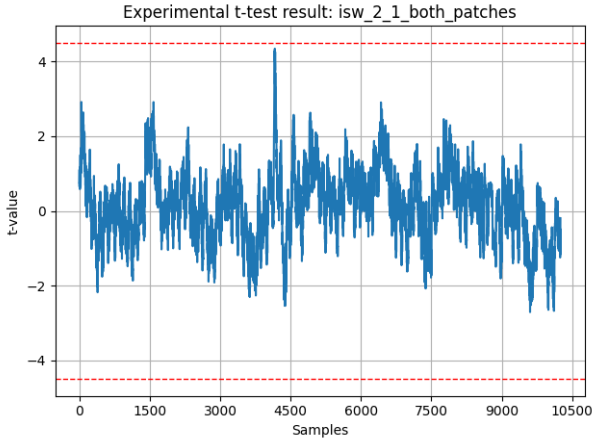


Figure 5.28: Experimental T-test result of isw-2-without-b-0 - both patches (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Description about the register tree:

If we consider the multiplexer tree like the below figure, just for the low registers:

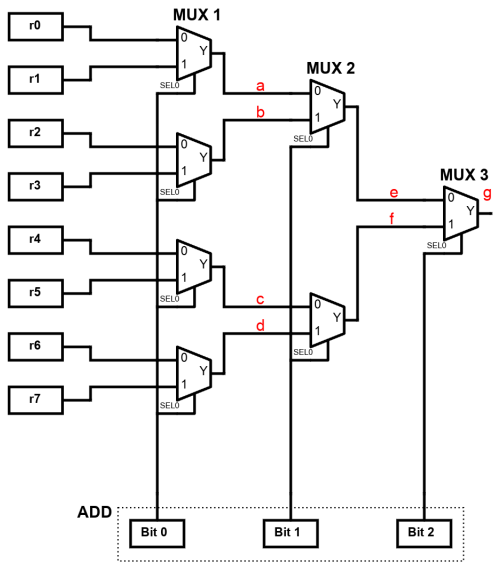


Figure 5.29: Multiplexer tree

For selecting $r0$, the selecting signal is 000 (in the figure ADD: Bit0: 0, Bit1: 0, Bit2: 0).

As Bit0 is zero, the outputs of multiplexers in MUX 1 are:

r0 goes on wire a: $r_0 \rightarrow wire - a$

and so on:

$r_2 \rightarrow wire - b$

$r_4 \rightarrow wire - c$ (here $r4$ is loaded)

$r_6 \rightarrow wire - d$

As Bit0 is zero, the outputs of multiplexers in MUX 2 are:

and so on:

the value on $wire - a(r_0) \rightarrow wire - e$

the value on $wire - c(r_4) \rightarrow wire - f$

And finally, the value on $wire - e(r_0) \rightarrow wire - g$, which shows that r_0 is selected.

6. ISW-3- no function/label call

The ISW-3 multiplication (three shares) is shown in Algorithm 4 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 4 ISW-3

Require: shares $a_0, a_1, a_2, b_0, b_1, b_2, rnd_0, rnd_1, rnd_2$

Ensure: c_0, c_1, c_2

1: $c_0 \leftarrow (a_0.b_0) \oplus rnd0 \oplus rnd1$

2: $c_1 \leftarrow [((a_0.b_1) \oplus rnd0) \oplus (a_1.b_0)] \oplus (a_1.b_1) \oplus rnd2$

3: $c_2 \leftarrow [((a_0.b_2) \oplus rnd1) \oplus (a_2.b_0)] \oplus [((a_1.b_2) \oplus rnd2) \oplus (a_2.b_1)] \oplus (a_2.b_2)$

In the implementation of ISW-3, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

6.1 ISW-3-V0: basic code

Listing 6.1 demonstrates the first implementation of ISW-3 based gfmul-s (ISW-3-V0). The instructions in Listing 6.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 6.1: ISW-3-V0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @r8 = &c

@@@@@ Saving r0 (the address of the input a)
  mov  r9, r0        @ r9 = &a

@@@@@ Saving r1 (the address of the input b)
  mov  r10, r1       @ r10 = &b


  ldr  r3, =table    @ r3 = &taable, The address of the "table", for ai*bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r11, r3       @ r11 = &table

nop
nop
nop
nop

@@@@@ Loading arguments @@@@@@@@@@@@
  ldrb r4, [r0, #0]  @ r4 = a0; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
  ldrb r5, [r1, #0]  @ r5 = b0; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
  ldrb r6, [r0, #4]  @ r6 = a1; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
  ldrb r7, [r1, #4]  @ r7 = b1; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table


@@@@@@@@@@ Computing: ai*bi

@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
@@@@ r4 = a0, r5 = b0, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@@
```

```
  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
  adds r1, #6         @ r1 = 256
  adds r0, r1         @ r0 = s + 256

  ldrb r1, [r3, r0]   @ r1 = s = table[s+256]

  # @@@@ Checking if a0 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r4         @ r0 = -a0 hex(256-dec(a0))

  @@@ a0 in r4 is not needed any more, using r4
  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r4, #32        @ r4 = #32
  asrs r0, r4         @ r0 = -a0 >> 32

  ands r5, r0         @ r5 = tmp = b0 & (-a0 >> 32)
  negs r5, r5         @ r5 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r5, r4         @ r5 = -tmp >> 32

  ands r5, r1         @ r5 = s & (-tmp >> 32) = (a0*b0)


@@@@ r0 = -a0 >> 32, r1 = table[s+256], r2 = &rnd, r3 = &table
@@@@ r4 = #32(&c), r5 = (a0*b0), r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  adds r0, r1          @ r0 = s + 256

  ldrb r1, [r3, r0]    @ r1 = s = table[s+256]


  # @@@@ Checking if a1 = 0 or b1 = 0, return 0; without conditional branch
  negs r0, r6          @ r0 = -a1

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r3, #32
  asrs r0, r3          @ r0 = -a1 >> 32

  ands r7, r0          @ r7 = tmp = b1 & (-a1 >> 32)
  negs r7, r7          @ r7 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r7, r3          @ r7 = -tmp >> 32

  ands r7, r1          @ r7 = s & (-tmp >> 32) = (a1*b1)



@@@@ r0 = -a1 >> 32, r1 = table[s+256], r2 = &rnd, r3 = 32
@@@@ r4 = a0, r5 = (a0*b0), r6 = a1, r7 = (a1*b1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
  adds r1, #6         @ r1 = 256
  adds r0, r1         @ r0 = s + 256

  ldrb r1, [r4, r0]   @ r1 = s = table[s+256]

  # @@@@ Checking if a1 = 0 or b1 = 0, return 0; without conditional branch
  negs r0, r6         @ r0 = -a2

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r4, #32
  asrs r0, r4         @ r0 = -a2 >> 32

  ands r3, r0         @ r3 = tmp = b2 & (-a2 >> 32)
  negs r3, r3         @ r3 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r3, r4         @ r3 = -tmp >> 32

  ands r3, r1         @ r3 = s & (-tmp >> 32)

  mov  r4, r8         @ r4 = &c
  strb r3, [r4, #8]   @ Storing (a2*b2) in c2, r3 = (a2*b2)
@@@@@@@@@@ Computing: ci

@@@@ r0 = -a2 >> 32, r1 = table[s+256], r2 = &rnd, r3 = (a2*b2),
@@@@ r4 = 32(&c), r5 = (a0*b0), r6 = a2, r7 = (a1*b1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@
```

```
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  eors r4, r0   @ r4 = (rnd0+(a0*b1))


@@@@ r0 = (a0*b1), r1 = table[s+256], r2 = &rnd, r3 = #32
@@@@ r4 = (rnd0+(a0*b1)), r5 = a0, r6 = a0 >> 32, r7 = (a1*b1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r6          @ r1 = -tmp >> 32

  ands r1, r5          @ r1 = s & (-tmp >> 32) = (a1*b0)



@@@@ r0 = -a1 >> 32, r1 = (a1*b0), r2 = &rnd, r3 = a1
@@@@ r4 = (rnd0+(a0*b1)), r5 = table[s+256], r6 = 32 , r7 = (a1*b1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  ldrb r1, [r2, #4]   @ r1 = rnd1
  ldrb r5, [r7, #8]   @ r5 = b2

  ldrb r0, [r4, r6]   @ r0 = table[a0]
  ldrb r7, [r4, r5]   @ r7 = table[b2]

  adds r0, r7         @ r0 = s = table[a0] + table[b2]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r7, #250       @ r7 = 250, 256 is not possible, so it is split: 250 + 6
  adds r7, #6         @ r7 = 256
  adds r0, r7         @ r0 = s + 256

  ldrb r7, [r4, r0]   @ r7 = s = table[s+256]

  # @@@@ Checking if a0 = 0 or b2 = 0, return 0; without conditional branch
  negs r0, r6         @ r0 = -a0

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r4, #32
  asrs r0, r4         @ r0 = -a0 >> 32

  ands r5, r0         @ r5 = tmp = b2 & (-a0 >> 32)
  negs r5, r5         @ r5 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r5, r4         @ r5 = -tmp >> 32

  ands r5, r7         @ r5 = s & (-tmp >> 32) = (a0*b2)


@@@
```

```

@@@
```

```
@@@
```

```
  asrs r2, r4          @ r2 = -a1 >> 32

  ands r1, r2          @ r1 = tmp = b2 & (-a1 >> 32)
  negs r1, r1          @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r4          @ r1 = -tmp >> 32

  ands r1, r5          @ r1 = s & (-tmp >> 32) = (a1*b2)

  # mov  r4, r8          @ r = &c
  # strb r1, [r4, #11]

# # computing (rnd2+(a1*b2))

  eors r6, r1          @ r6 = (rnd2+(a1*b2))


@@@
```

```
  ldrb r3, [r4, r1]   @ r3 = table[b1]

  adds r2, r3         @ r2 = s = table[a2] + table[b1]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r3, #250       @ r3 = 250, 256 is not possible, so it is split: 250 + 6
  adds r3, #6         @ r3 = 256
  adds r2, r3         @ r2 = s + 256

  ldrb r3, [r4, r2]   @ r3 = s = table[s+256]

  # @@@@ Checking if a2 = 0 or b1 = 0, return 0; without conditional branch
  negs r2, r7         @ r2 = -a2

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r4, #32
  asrs r2, r4         @ r2 = -a2 >> 32

  ands r1, r2         @ r1 = tmp = b1 & (-a2 >> 32)
  negs r1, r1         @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r4         @ r1 = -tmp >> 32

  ands r1, r3         @ r1 = s & (-tmp >> 32) = (a2*b1)

  @@@@ c2 = [(rnd1+(a0*b2))+(a2*b0)]+[(rnd2+(a1*b2))+(a2*b1)]+(a2*b2)

  mov  r4, r8         @ r4 = &c
  eors r6, r1         @ r6 = [(rnd2+(a1*b2))+(a2*b1)]
  eors r0, r6         @ r0 = [(rnd1+(a0*b2))+(a2*b0)]+[(rnd2+(a1*b2))+(a2*b1)]
  eors r5, r0         @ r5 = c2 = [(rnd1+(a0*b2))+(a2*b0)]+[(rnd2+(a1*b2))+(a2*b1)]+(a2*b2)

  strb r5, [r4, #8]   @ r5 = c2


nop
nop
nop
nop
```

6.1.1 Experimental result:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $150\mu s$
Each clock is sampled with 125 points
Number of samples = 250×125 = 31250 samples
The bytes of shares (a, b, c are not in a word)
Figure 6.1 shows the recorded 31250 S before raising-edge of the trigger.

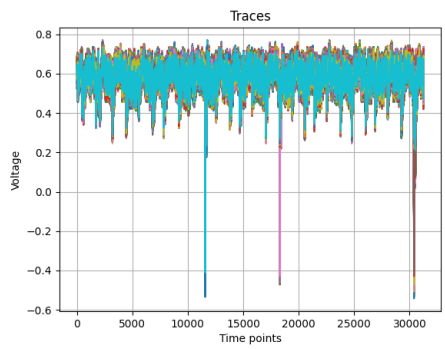


Figure 6.1: Traces isw-3-without-b-0 (Setup: 250MSa/s, 31250S, 4ns, 2MHz)

6.1.1.1 Signal to Noise Ratio (SNR):

Figure 6.2 shows the SNR for all the intermediate values. It can be concluded that the first 1875 (15×125) and samples do not have effect on the SNR, so, they can be removed.
Number of samples = $(250 - 15) \times 125 = 235 \times 125 = 29375$ samples

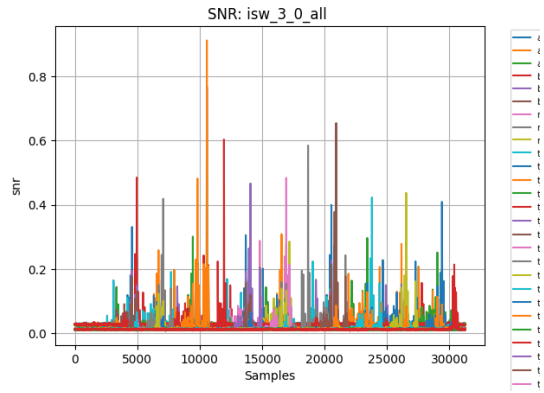


Figure 6.2: SNR isw-3-without-b-0 (Setup: 250MSa/s, 31250S, 4ns, 2MHz)

6.1.1.2 t-test:

$$t - val = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \tag{6.1}$$

First-order univariate t-test:

$$t - val = \frac{\mu_f - \mu_r}{\sqrt{\frac{s_f^2}{n_f} + \frac{s_r^2}{n_r}}} \tag{6.2}$$

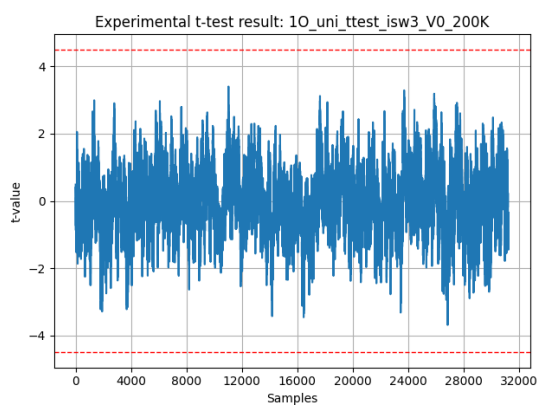


Figure 6.3: First-order univariate ttest isw-3-without-b-0 (Setup: 250MSa/s, 31250S, 4ns, 2MHz)

Second-order univariate t-test:

The traces t are preprocessed by mean-free squared:

$$T = (t - \mu_t)^2 \tag{6.3}$$

$$t - val = \frac{\mu_F - \mu_R}{\sqrt{\frac{s_F^2}{n_f} + \frac{s_R^2}{n_r}}} \tag{6.4}$$

Second-order multivariate t-test:

Finding leakages via splitting t-test in to three t-test s:

In this section we find the leakages caused by 2 shares out of 3 shares using the split-t-test .

1) (sh_0, sh_1)
Listing 6.2 shows the the leakages found by **uEMO**. As it is shown in Figure 6.4, these leakages also exist in real traces (cycles 128, 129, almost 134 from 240 recorded cycles (240×125).

Listing 6.2: Leakages reported by **uELMO** for isw-3-0-sh0-sh1

```
Cycle 121: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x20001F98]=0xBC
, Term=Glitchy decoding port 0 XOR current port 0, Tvalue=319.4491241884423
Cycle 121: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x20001F98]=0xBC
, Term=Glitchy decoding port 0 XOR previous port 0, Tvalue=319.4491241884423
Cycle 122: Instr=Execute:eors r1,r4
Decode:eors r7,r1
Memory: idle
, Term=Decoding port 0 HD, Tvalue=319.4491241884423
Cycle 122: Instr=Execute:eors r1,r4
Decode:eors r7,r1
Memory: idle
, Term=Glitchy decoding port 0 XOR previous port 0, Tvalue=319.4491241884423
Cycle 123: Instr=Execute:eors r7,r1
Decode:eors r0,r7
Memory: idle
, Term=Pipeline Reg 1 HD, Tvalue=319.4491241884423
Cycle 123: Instr=Execute:eors r7,r1
Decode:eors r0,r7
```

```
Memory: idle
, Term=Decoding port 1 HD, Tvalue=319.4491241884423
Cycle 123: Instr=Execute:eors r7,r1
Decode:eors r0,r7
Memory: idle
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=319.4491241884423
Cycle 128: Instr=Execute: mov r7,r10
Decode: mov r4,r11
Memory: idle
, Term=Decoding port 0 HD, Tvalue=319.4491241884423
Cycle 128: Instr=Execute: mov r7,r10
Decode: mov r4,r11
Memory: idle
, Term=Glitchy decoding port 0 XOR previous port 0, Tvalue=319.4491241884423
```

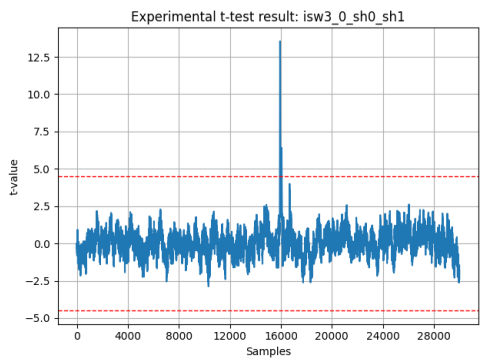


Figure 6.4: t-test_isw3_0_sh0_sh1 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

Reasons and solutions of/for leakages:
1) Leakage in cycles 128 and 129 are related to the two eors instructions in Lines 274 and 275 in Listing 6.1 (listed in Listing 6.3).
This leakage stems from first operands (which are loaded into the same port) in the two eors instructions:
$r1 = a_1.b_0$ and $r7 = a_1.b_1 \rightarrow a_1.b_0 \oplus a_1.b_1 = a_1.(b_0 \oplus b_1)$.

It can be concluded if the power consumption of this point combined with the power consumption of $a_1.b_2$ (in pre-processing step), then b will leak.

Listing 6.3: Source of the first leakage in isw3-0-sh0-sh1

```
1
2
3  # r0 = -a1 >> 32
4  # r1 = (a1*b0)
5  # r2 = &rnd
6  # r3 = a1
7  # r4 = (rnd0+(a0*b1))
8  # r5 = table[s+256]
9  # r6 = 32
10 # r7 = (a1*b1),
11
12   ldrb r0, [r2, #8]    @ r0 = rnd2
13   eors r1, r4          @ r1 = [(rnd0+(a0*b1))+(a1*b0)]
14   eors r7, r1          @ r7 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1)
15
16
```

In order to avoid this leakage, we can change the order of instructions in Listing6.3 in such a way that the two eors instructions be not consecutive.
Figure 6.5 shows the t-test result after applying the patch.

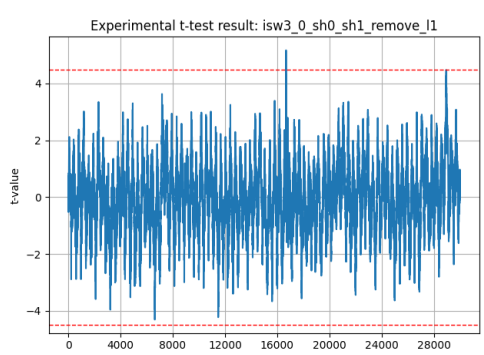


Figure 6.5: t-test_isw3_0_sh0_sh1_patching the first leakage (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

2) Leakage in cycle 134 are related to the two mov instructions in Lines 297 and 298 in Listing 6.1 (listed in Listing 6.5).

Listing 6.4: Patching the first leakage in isw3-0-sh0-sh1

```
# r0 = -a1 >> 32
# r1 = (a1*b0)
# r2 = &rnd
# r3 = a1
# r4 = (rnd0+(a0*b1))
# r5 = table[s+256]
# r6 = 32
# r7 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1),

  eors r1, r4          @ r1 = [(rnd0+(a0*b1))+(a1*b0)]
  ldrb r0, [r2, #8]    @ r0 = rnd2
  eors r7, r1          @ r7 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1)
```

This leakage stems from first operands (which are loaded into the same port) in the two mov instructions:

$r7 = [(rnd_0 \oplus (a_0.b_1)) \oplus (a_1.b_0)] \oplus (a_1.b_1)$

$r4 = rnd_0 \oplus (a_0.b_1)$

$r7 \oplus r4 = [[(rnd_0 \oplus (a_0.b_1)) \oplus (a_1.b_0)] \oplus (a_1.b_1)] \oplus [rnd_0 \oplus (a_0.b_1)]$

$= (rnd_0 \oplus rnd_0) \oplus ((a_0.b_1) \oplus (a_0.b_1)) \oplus((a_1.b_0) \oplus (a_1.b_1))$

$(a_1.b_0) \oplus (a_1.b_1) = a_1.(b_0 \oplus b_1)$

It can be concluded if the power consumption of this point combined with the power consumption of $a_1.b_2$ (in pre-processing step), then input b will leak.

In order to avoid this leakage, we can change the order of instructions in Listing 6.6 in such a way that the two eors instructions be not consecutive.

Figure 6.6 shows the t-test result after applying the patch.

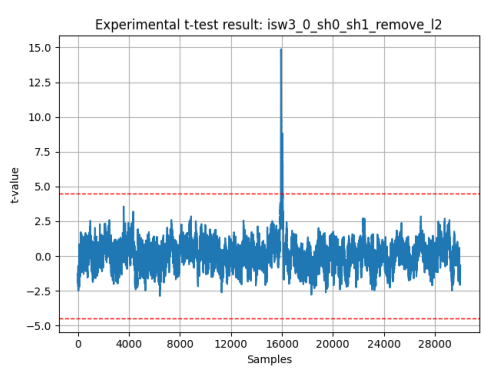


Figure 6.6: t-test_isw3_0_sh0_sh1_patching the second leakage (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

Figure 6.7 shows the t-test result after applying both patches.

Listing 6.5: Source of the second leakage in isw3-0-sh0-sh1

```

# r0 = c1
# r1 = [(rnd0+(a0*b1))+(a1*b0)]
# r2 = &rnd
# r3 = a1
# r4 = (rnd0+(a0*b1))
# r5 = table[s+256]
# r6 = 32
# r7 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1),
# r8 = &c
# r9 = &a
# r10 = &b
# r11 = &table

  mov  r0, r9          @ r0 = &a
  mov  r7, r10         @ r7 = &b
  mov  r4, r11         @ r4 = &table

```

Listing 6.6: Patching the second leakage in isw3-0-sh0-sh1

```

# r0 = c1
# r1 = [(rnd0+(a0*b1))+(a1*b0)]
# r2 = &rnd
# r3 = a1
# r4 = (rnd0+(a0*b1))
# r5 = table[s+256]
# r6 = 32
# r7 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1),
# r8 = &c
# r9 = &a
# r10 = &b
# r11 = &table

  mov  r7, r10         @ r7 = &b
  mov  r0, r9          @ r0 = &a
  mov  r4, r11         @ r4 = &table

```

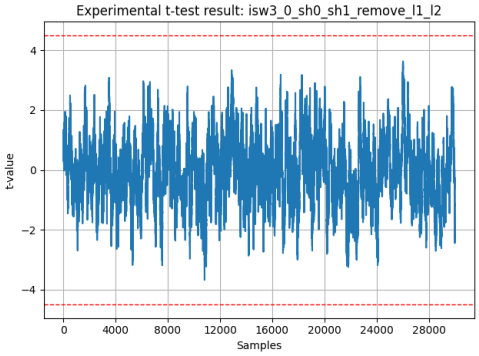


Figure 6.7: t-test_isw3_0_sh0_sh1_patching both leakages (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

2) (sh_0, sh_2)

Listing 6.7 shows the the leakages found by **uEMO**, the source of the leakage are listed in Listing 6.8. As it is shown in Figure 6.8, these leakages do not exist in real traces (up to 100K traces.)

Listing 6.7: Leakages reported by **uELMO** for isw-3-0-sh0-sh2

```
Cycle 71: Instr=Execute: ldrb r1,[r2,#0x4]
Decode:eors r5,r4
Memory: load [0x20001F90]=0xA3
, Term=Decoding port 0 HD, Tvalue=inf
Cycle 71: Instr=Execute: ldrb r1,[r2,#0x4]
Decode:eors r5,r4
Memory: load [0x20001F90]=0xA3
, Term=Glitchy decoding port 0 XOR previous port 0, Tvalue=inf
Cycle 72: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x20001F94]=0xBE
, Term=Glitchy decoding port 0 XOR current port 0, Tvalue=inf
Cycle 72: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x20001F94]=0xBE
, Term=Glitchy decoding port 0 XOR previous port 0, Tvalue=inf
```

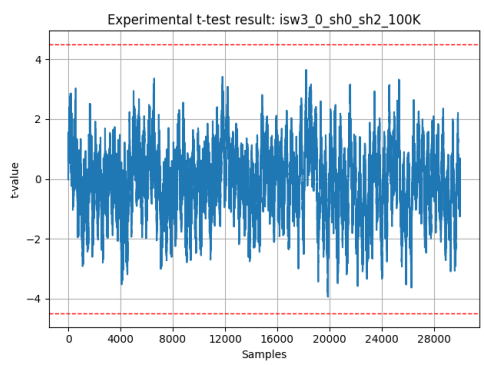


Figure 6.8: t-test_isw3_0_sh0_sh2 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

But why **uELMO** reports the leakage???
The first two operand: $r1 = a_2.b_2$ and $r5 = a_0.b_0$.
Different shares from the same family are using in sequence.

Listing 6.8: Source of the leakage reported by uELMO in isw3-0-sh0-sh2

```
# r0 = -a2 >> 32
# r1 = table[s+256], s = table[a2] + table[b2] => r1 = a2*b2
# r2 = &rnd
# r3 = (a2*b2),
# r4 = 32(&c)
# r5 = (a0*b0)
# r6 = a2
# r7 = (a1*b1),


  ldrb r4, [r2, #0]   @ r4 = rnd0
  ldrb r1, [r2, #4]   @ r1 = rnd1
  eors r5, r4         @ r5 = (a0*b0)+rnd0
```

Listing 6.9 shows how to patch this leakage by reordering the instructions.

Listing 6.9: Source of the leakage reported by uELMO in isw3-0-sh0-sh2

```
# r0 = -a2 >> 32
# r1 = table[s+256], s = table[a2] + table[b2] => r1 = a2*b2
# r2 = &rnd
# r3 = (a2*b2),
# r4 = 32(&c)
# r5 = (a0*b0)
# r6 = a2
# r7 = (a1*b1),

  ldrb r1, [r2, #4]   @ r1 = rnd1
  ldrb r4, [r2, #0]   @ r4 = rnd0
  eors r5, r4         @ r5 = (a0*b0)+rnd0
```

3) (sh_1, sh_2)

Listing 6.10 shows the the leakages found by **uEMO**. As it is shown in Figure 6.12, these leakages also exist in real traces (cycles 51, 206 and 207 from 240 recorded cycles (240×125).
But **uELMO** does not detect the leakages in cycles 82 and 83.

Listing 6.10: Leakages reported by **uELMO** for isw-3-0-sh1-sh2

```
Cycle 52: Instr=Execute: ldrb r0,[r4,r6]
Decode:ldrb r1,[r4,r3]
Memory: load [0x20001FB8]=0x63
, Term=Reg 6 HD, Tvalue=inf
Cycle 197: Instr=Execute: ldrb r1,[r3,#0x4]
Decode:ldrb r2,[r4,r7]
Memory: load [0x20001FA8]=0xCA
, Term=Memory data HD, Tvalue=6.320348900277683
Cycle 197: Instr=Execute: ldrb r1,[r3,#0x4]
Decode:ldrb r2,[r4,r7]
Memory: load [0x20001FA8]=0xCA
, Term=Memory read buffer HD, Tvalue=308.5660668904282
Cycle 198: Instr=Execute: ldrb r2,[r4,r7]
Decode:ldrb r3,[r4,r1]
Memory: load [0x20001FB4]=0x63
, Term=Reg 5 HD, Tvalue=308.5660668904282
```

By applying the splitting t-test to real traces, we identified several leakages at cycles 51, 206, and 207 out of the 240 recorded cycles (each cycle comprising 240×125 data points). As illustrated in Figure 6.12.

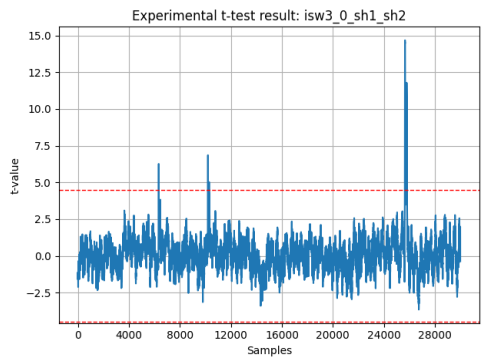


Figure 6.9: t-test_isw3_0_sh1_sh2, 50K traces(Setup: 250MSa/s, 30KS, 4ns, 2MHz)

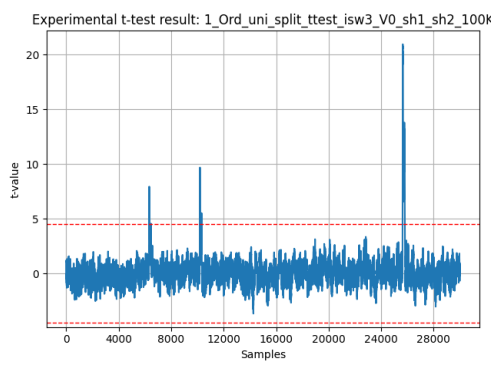


Figure 6.10: t-test_isw3_0_sh1_sh2, 100K traces (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

Reasons and solutions of/for leakages:

1) Leakage in cycle 51 is related to the instructions in lines 25 and 118 in Listing 6.1 (listed in Listing 6.13).

The leakage is because of overwriting the register: HD between the previous value of a register and and its new written value. $r6 = a1 \rightarrow a2$

It can be concluded if the power consumption of this point combined with the power consumption of a_0 (in pre-processing step), then input a will leak.

In order to avoid this leakage, we can clear register $r6$ (eors r6, r6), as it is shown in Listing 6.12

Figure 6.11 shows the t-test result after applying the patch.

Listing 6.11: Source of Reg r6 HD leakage in isw3-0-sh1-sh2

```
ldrb r6, [r0, #4]   @ r6 = a1

      A sequence of instructions

        mov  r0, r9         @ r0 = &a
        mov  r1, r10        @ r1 = &b
        mov  r4, r11        @ r4 = &table

ldrb r6, [r0, #8]   @ r6 = a2
```

Listing 6.12: Patching the Reg r6 HD leakage in isw3-0-sh1-sh2

```
ldrb r6, [r0, #4]   @ r6 = a1

      A sequence of instructions

eors r6, r6   @ =Reg 6 HD, a1 ---> a2

        mov  r0, r9         @ r0 = &a
        mov  r1, r10        @ r1 = &b
        mov  r4, r11        @ r4 = &table

ldrb r6, [r0, #8]   @ r6 = a2
```

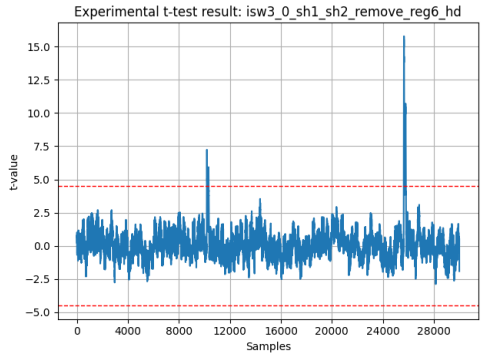


Figure 6.11: t-test_isw3_0_sh0_sh1_patching the Reg HD leakage (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

By applying the splitting t-test to real traces, we identified several leakages at cycles 51, 206, and 207 out of the 240 recorded cycles (each cycle comprising 240×125 data points). As illustrated in Figure 6.12. As the leakages in cycles 206 and 207 are greater than 12, we consider them for the rest of the experiments.

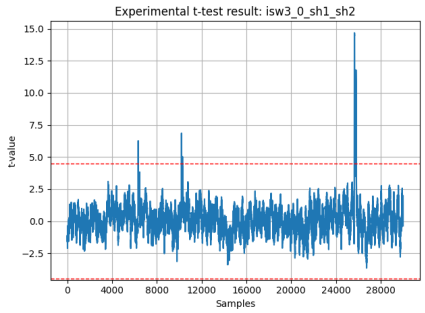


Figure 6.12: t-test_isw3_0_sh1_sh2 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

Reasons and solutions of/for leakages:

Leakage in cycles 206 and 207 are related to the instructions in lines 423, 463 and 475 in Listing 6.1 (listed in Listing 6.13). The leakages are because of several resaons, some are mentioned as follow:
1) overwriting the register: HD between the previous valueand new written value of a register.
2) HW in the reading bus
$r5 = table[table[a_1] + table[b_2] + 256] = a_1.b_2$ and $r5 = a_2.b_2$
$Reg5 - HD = a_1.b_2 \oplus a_2.b_2 = b_2.(a_1 \oplus a_2)$

This leakage can be appear for number of traces more than 3K. As it can be seen in the Figure 6.13

It can be concluded if the power consumption of this point combined with the power consumption of $a_0.b_2$ (in pre-processing step), then the input a will leak.

Checking on pre-processing traces:

Now, we want to find how many traces are needed that this leakage can be detected using a first-order multivariate t-test, where the traces are pre-processed using central production as the combination function.
In other words, we need to combine the power consumption at the point $(a_1 \oplus a_2)$ with the power consumption of a_0 in the pre-processing step. Then, we determine how many traces are required to detect this leakage.
Given the length of the traces and the fact that each cycle is sampled at 125 points, performing the pre-processing step for all possible combinations is time and memory consuming.
Therefore, we only consider the cycle (cy-1) wich contains the maximum SNR for a_0 (Figure 6.14), and the cycle (cy-2: cycle 206) where the $a_1 \oplus a_2$ operation occurs. With this approach, each trace includes 250 samples. Since we aim to combine the sample points in cy1 (a_0) with those in cy2 ($a_1 \oplus a_2$) (not the combinations of sample points within the same cycle), the new pre-processed trace contains $125 \times 125 = 15625$ samples. However, this number of samples is still time and memory consuming.
To reduce the complexity (125 points) in cy1 (a_0), we consider only the points where the SNR is greater than 0.2. From the 125 points in cy1, we select the points: $[40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52]$. Now, the new pre-processed trace contains $13 \times$

Listing 6.13: Source of the second leakage in isw3-0-sh1-sh2

```

# r2 = table[a1] + table[b2] + 256, r4 = &table, r5 = 256
  ldrb r5, [r4, r2]   @ r5 = table[s+256] = a1*b2
        A sequence of instructions
# r0 = [(rnd1+(a0*b2))+(a2*b0)]
# r1 = (a1*b2), r2 = -a1 >> 32, r3 = a1, r4 = 32
# r5 = table[s+256], s = table[a1] + table[b2] => r5 = a1*b2
# r6 = (rnd2+(a1*b2)), r7 = a2,
# @@@@ a2*b1
  # mov  r2, r8         @ r2 = &c  @ (a2 *b2) is in [&c + #8]
  # mov  r4, r11        @ r4 = &table
  # mov  r3, r10        @ r3 = &b
  ldrb r5, [r2, #8]   @ r5 = (a2*b2)
  ldrb r1, [r3, #4]   @ r1 = b1
  # ldrb r2, [r4, r7]   @ r2 = table[a2]
  # ldrb r3, [r4, r1]   @ r3 = table[b1]
  # adds r2, r3         @ r2 = s = table[a2] + table[b1]

```

125 = 1625 samples, a manageable amount.

We then perform the pre-processing and conduct the t-test on the pre-processed traces. Our goal is to determine the minimum number of traces required to detect the leakage:

1) We start with a certain number of traces.

2) We perform the pre-processing, followed by the t-test.

3) If we detect the leakage, we stop; if not, we add more traces and repeat the procedure from step 2 until the leakage is detected.

In our experiment, the leakage was detected after analyzing 1,600,000 traces, with a t-value of 4.64137 (Figure **??**). The figure displays 13 distinct patterns, each representing a combination of 13 points of interest (POIs) from the first cycle with all 125 points in the second cycle.

(a) 3000 traces (b) 5000 traces

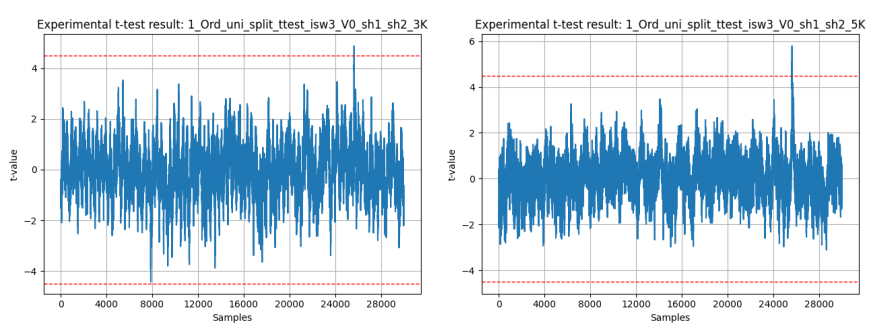


Figure 6.13: split-t-test isw-3-V0-(Setup: 250MSa/s, 30KS, 4ns, 2MHz)

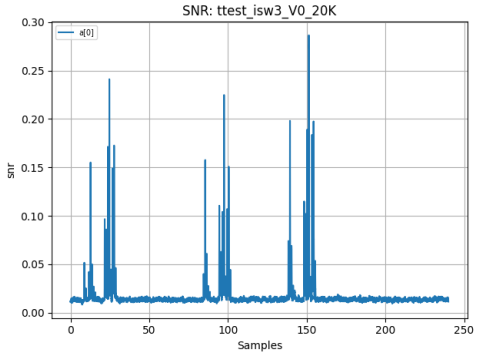


Figure 6.14: SNR_isw3_V0, cy 151 contains high SNR(Setup: 250MSa/s, 30KS, 4ns, 2MHz)

In order to avoid this leakage, we can clear register $r5$ (eors r5, r5), and clear the read bus (ldrb r1, [r4, #0])

In addition, we should avoid manipulating different shares from the same family in sequence. Lines 469 and 470:
$r5 = a_2.b_2 \rightarrow r1 = b_1$ (b_1 and b_2)

Also, in Line 470, $r1$ with value of $a_1.b_2$ is overwriting with $b1$. So that, $r1$ is overwriting with different shares from the same family:
$r1 = a_1.b_2 \rightarrow b1$ (b_1 and b_2).
For avoiding this, we clear register $r1$ (eors r1, r1).

Listing 6.14 shows all of patches for the second leakage.
Figure 6.15 shows the t-test result after applying the patches.

Listing 6.14: Patching the second leakage in isw3-0-sh1-sh2

```
1
2  # r2 = table[a1] + table[b2] + 256
3  # r4 = &table
4  # r5 = 256
5    ldrb r5, [r4, r2]   @ r5 = s = table[s+256]  = a1*b2
6          A sequence of instructions
7  # r0 = [(rnd1+(a0*b2))+(a2*b0)]
8  # r1 = (a1*b2)
9  # r2 = -a1 >> 32
10 # r3 = a1
11 # r4 = 32
12 # r5 = table[s+256], s = table[a1] + table[b2] => r5 = a1*b2
13 # r6 = (rnd2+(a1*b2))
14 # r7 = a2,
15 @@@@ a2*b1
16   eors r5, r5         @ r5 = 0, Clearing r5
17   eors r1, r1         @ r1 = 0, Clearing r1
18   mov  r2, r8         @ r2 = &c  @ (a2 *b2) is in [&c + #8]
19   mov  r4, r11        @ r4 = &table
20   mov  r3, r10        @ r3 = &b
21   # ldrb r5, [r4, r2] @ r5 = s = table[s+256] = a1*b2
22   ldrb r1, [r4, #0]   @ r1 = 0, Clearing the read bus
23   ldrb r5, [r2, #8]   @ r5 = (a2*b2)
24   ldrb r2, [r4, r7]   @ r2 = table[a2]
25   ldrb r1, [r3, #4]   @ r1 = b1
26   ldrb r3, [r4, r1]   @ r3 = table[b1]
27   adds r2, r3         @ r2 = s = table[a2] + table[b1]
```

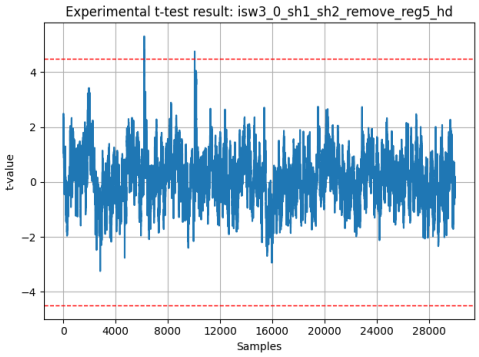


Figure 6.15: t-test_isw3_0_sh0_sh1_patching the Reg 5 HD leakage (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

3) The leakages in cycles 82 and 83 which were not detected by **uELMO**.
This leakage is related to Lines 181 and 182 in Listing 6.1 listed in Listing 6.15.

Regardless the detected leakage, in instructions mov r3, r11 and ldrb r5, [r6, #0], different shares from the same family (a_2 and a_0) were used in sequence. We should avoid using/manipulating different shares from the same family in succession.

Listing 6.15: Source of the third leakage in isw3-0-sh1-sh2

```
# r0 = &c
# r1 = c0
# r2 = &rnd
# r3 = (a2*b2)
# r4 = rnd0
# r5 = (a0*b0)+rnd0
# r6 = a2
# r7 = (a1*b1)

  mov  r6, r9          @ r6 = &a
  mov  r1, r10         @ r1 = &b
  mov  r3, r11         @ r3 = &table
  ldrb r5, [r6, #0]    @ r5 = a0
```

Description about the register tree:
As all codes are based on THUMB-16, just Low registers are considered in the figure.
From paper [5], Table 1:
In Decoding stage of ldrb Rd, [Rn, Rm] instruction:
Port1 ← Rd
Port2 ← Rn
Port3 ← Rm
In Decoding stage of mov Rd, Rm instruction:
Port1 ← Rd
Port2 ← Rn
and From paper [6], the register tree (Figure 6.16).
Regarding Listing 6.15,
According [5]: in the decode of the mov, register $r3$ (address in binary: 011) is imported into $Port1$.
Then in the next cycle, register $r5$ (address in binary: 101) (from ldrb) must be imported into $Port1$.
According [6]:
First cycle: For $r3$, SEL0 signals (ADD: Bit0: 1, Bit1: 1, Bit2: 0) the multiplexers will be 110, so that $r3$ (=$a_2.b_2$) goes on the wires b, e and g.
And also, so that $r7$ (=$a_1.b_1$) goes on the wires d and f.
So that, the values on the wires are:
a: $\&b$
b: $a_2.b_2$
c: $(a_0.b_0) \oplus rnd_0$

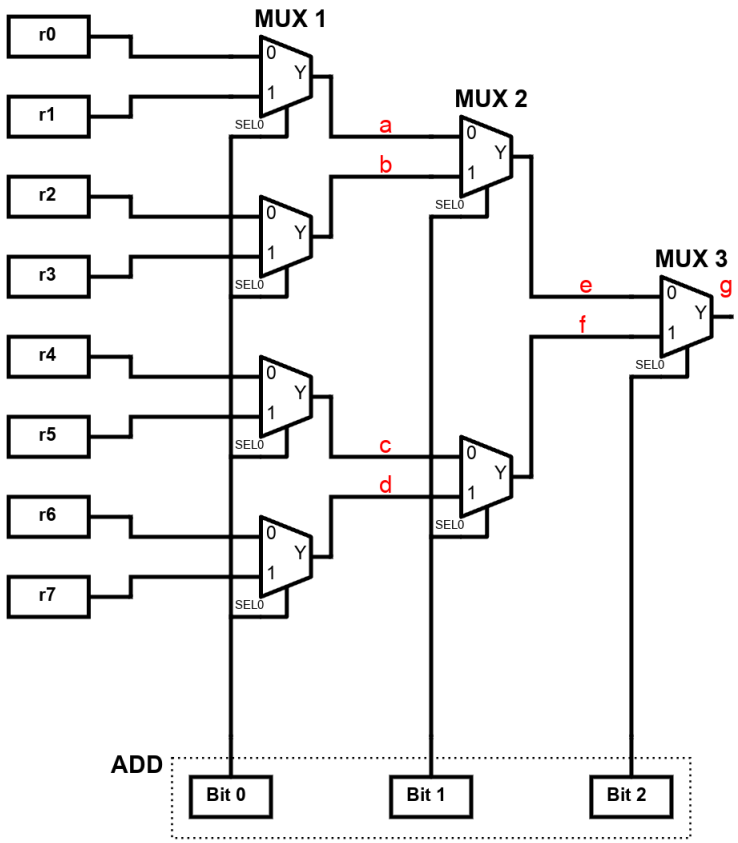


Figure 6.16: Multiplexer tree

d: $a_1.b_1$
e: $a_2.b_2$
f: $a_1.b_1$
g: $a_2.b_2$
Second cycle: For $r5$, SEL0 signals (ADD: Bit0: 1, Bit1: 0, Bit2: 1) the multiplexers will switch to 101, so that:

Ines's idea: There is a delay in switching values in multiplexer values (Bit_i), and the leakage occurs on the wires of the multiplexer tree (wires: a-g in Figure 6.16, not in decoding ports or pipeline registers:
Bit0: $1 \rightarrow 1$: Does not change
Bit1: $1 \rightarrow 0$:
Bit2: $0 \rightarrow 1$:
As already the values of wires d and f is $r7 = a_1.b_1$, Because of the delay, first $r7 = a_1.b_1$ goes on the wire g and then $r5$. As the previous value of wire g is $r3 = a_2.b_2$, there is a leakage.
As a result, on the wire g, $a_1.b_1 \oplus a_2.b_2$ will leak.

For avoiding this leakage, we reorder the instruction as it is shown in Listing 6.16. Figure 6.17 shows the t-test result after applying the patch.

Listing 6.16: Patching the third leakage in isw3-0-sh1-sh2

```
# r0 = &c
# r1 = c0
# r2 = &rnd
# r3 = (a2*b2)
# r4 = rnd0
# r5 = (a0*b0)+rnd0
# r6 = a2
# r7 = (a1*b1)

  mov  r3, r11        @ r3 = &table
  mov  r6, r9         @ r6 = &a
  mov  r1, r10        @ r1 = &b
  ldrb r5, [r6, #0]   @ r5 = a0
```

Figure 6.17: t-test_isw3_0_sh0_sh1_patching the third leakage (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

Figure 6.18 shows the t-test result after removing all leakages in sh1 and sh2.

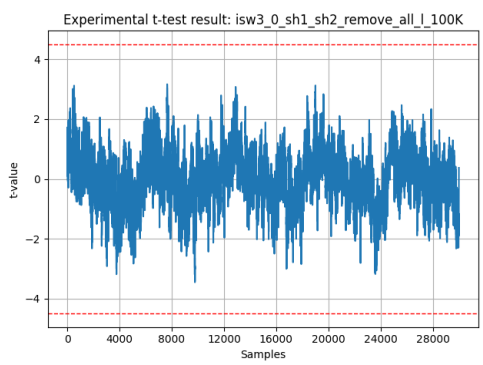


Figure 6.18: t-test_isw3_0_sh0_sh1_patching the third leakage (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

Now, we conduct out all discussed patches aboved on isw3-0, and we call it isw3-1. The Figures 6.19-6.21 show the results of the t-test on isw3-1.

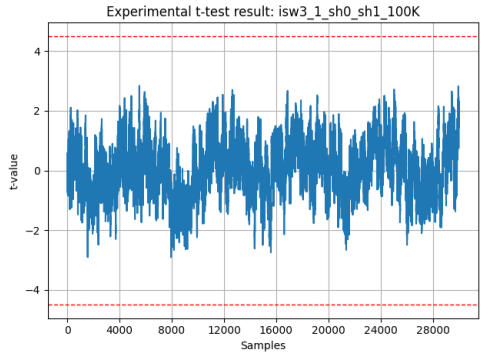


Figure 6.19: t-test_isw3_1_sh0_sh1 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

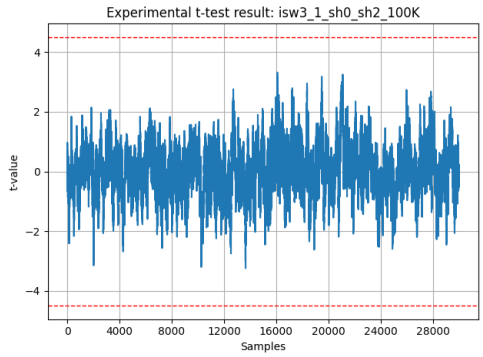


Figure 6.20: t-test_isw3_1_sh0_sh2 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

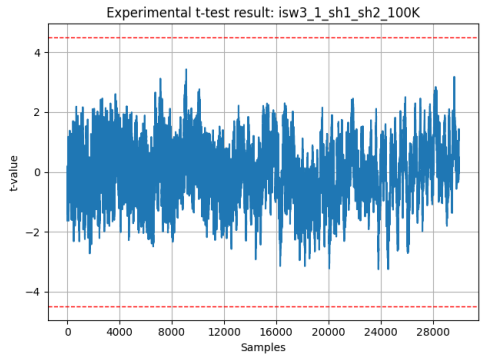


Figure 6.21: t-test_isw3_1_sh1_sh2 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

Finding possible leakages via the code walk-through/ hand tracing :

In this section, we will walk through the code(isw3-1 in Listing 6.17) line by line and find the leakages which would likely happen.

Listing 6.17: ISW-3-V1

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @r8 = &c

@@@@@ Saving r0 (the address of the input a)
  mov  r9, r0        @ r9 = &a

@@@@@ Saving r1 (the address of the input b)
  mov  r10, r1       @ r10 = &b


  ldr  r3, =table    @ r3 = &taable, The address of the "table", for ai*bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r11, r3       @ r11 = &table

nop
nop
nop
nop


@@@@@ Loading arguments @@@@@@@@@@@@
  ldrb r4, [r0, #0]  @ r4 = a0; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
  ldrb r5, [r1, #0]  @ r5 = b0; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
  ldrb r6, [r0, #4]  @ r6 = a1; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
  ldrb r7, [r1, #4]  @ r7 = b1; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table



@@@@@@@@@@ Computing: ai*bi

@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
@@@@ r4 = a0, r5 = b0, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@@
```

```
  adds r0, r1         @ r0 = s = table[a0] + table[b0]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
  adds r1, #6         @ r1 = 256
  adds r0, r1         @ r0 = s + 256

  ldrb r1, [r3, r0]   @ r1 = s = table[s+256]


  # @@@@ Checking if a0 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r4         @ r0 = -a0 hex(256-dec(a0))

  @@@ a0 in r4 is not needed any more, using r4
  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r4, #32        @ r4 = #32
  asrs r0, r4         @ r0 = -a0 >> 32

  ands r5, r0         @ r5 = tmp = b0 & (-a0 >> 32)
  negs r5, r5         @ r5 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r5, r4         @ r5 = -tmp >> 32

  ands r5, r1         @ r5 = s & (-tmp >> 32) = (a0*b0)


@@@@ r0 = -a0 >> 32, r1 = table[s+256], r2 = &rnd, r3 = &table
@@@@ r4 = #32(&c), r5 = (a0*b0), r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
  adds r1, #6         @ r1 = 256
  adds r0, r1         @ r0 = s + 256

  ldrb r1, [r3, r0]   @ r1 = s = table[s+256]



  # @@@@ Checking if a1 = 0 or b1 = 0, return 0; without conditional branch
  negs r0, r6         @ r0 = -a1

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r3, #32
  asrs r0, r3         @ r0 = -a1 >> 32

  ands r7, r0         @ r7 = tmp = b1 & (-a1 >> 32)
  negs r7, r7         @ r7 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r7, r3         @ r7 = -tmp >> 32

  ands r7, r1         @ r7 = s & (-tmp >> 32) = (a1*b1)



@@@@ r0 = -a1 >> 32, r1 = table[s+256], r2 = &rnd, r3 = 32
@@@@ r4 = a0, r5 = (a0*b0), r6 = a1, r7 = (a1*b1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  ldrb r6, [r0, #8]   @ r6 = a2
  ldrb r3, [r1, #8]   @ r3 = b2

  ldrb r0, [r4, r6]   @ r0 = table[a2]
  ldrb r1, [r4, r3]   @ r1 = table[b2]


  adds r0, r1         @ r0 = s = table[a2] + table[b2]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
  adds r1, #6         @ r1 = 256
  adds r0, r1         @ r0 = s + 256

  ldrb r1, [r4, r0]   @ r1 = s = table[s+256]

  # @@@@ Checking if a1 = 0 or b1 = 0, return 0; without conditional branch
  negs r0, r6         @ r0 = -a2

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r4, #32
  asrs r0, r4         @ r0 = -a2 >> 32

  ands r3, r0         @ r3 = tmp = b2 & (-a2 >> 32)
  negs r3, r3         @ r3 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r3, r4         @ r3 = -tmp >> 32

  ands r3, r1         @ r3 = s & (-tmp >> 32)

  mov  r4, r8         @ r4 = &c
  strb r3, [r4, #8]   @ Storing (a2*b2) in c2, r3 = (a2*b2)
@@@@@@@@@@ Computing: ci

@@@@ r0 = -a2 >> 32, r1 = table[s+256], r2 = &rnd, r3 = (a2*b2),
@@@@ r4 = 32(&c), r5 = (a0*b0), r6 = a2, r7 = (a1*b1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@
```

```
172
173 ## Leakage sh0-sh2, found in uELMO
174
175   # @ first operands in the last two inst @ r1: t[a2+b2], r5:a0b0: leaks: a0b0+a2b2
176   # ldrb r4, [r2, #0]   @ r4 = rnd0
177   # ldrb r1, [r2, #4]   @ r1 = rnd1
178
179   ldrb r1, [r2, #4]   @ r1 = rnd1
180   ldrb r4, [r2, #0]   @ r4 = rnd0
181
182
183   eors r5, r4         @ r5 = (a0*b0)+rnd0
184   eors r1, r5         @ r1 = c0 = (a0*b0)+rnd0+rnd1
185
186   mov  r0, r8         @ r0 = &c
187   strb r1, [r0, #0]   @ Storing c0, r1 = c0 = (a0*b0)+rnd0+rnd1
188
189
190 @@@@ r0 = &c, r1 = c0, r2 = &rnd, r3 = (a2*b2),
191 @@@@ r4 = rnd0, r5 = (a0*b0)+rnd0, r6 = a2, r7 = (a1*b1),
192 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
193 @@@@
```

```
215
216
217     ldrb r6, [r3, r5]   @ r6 = table[a0]
218     ldrb r1, [r3, r0]   @ r1 = table[b1]
219
220     adds r6, r1         @ r6 = s = table[a0] + table[b1]
221
222     @@@@ Get the antilog: s = table[s+256]: g+s
223     # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
224     # @ movs rd, <#imm> --> #imm: 0-255
225     movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
226     adds r1, #6         @ r1 = 256
227     adds r6, r1         @ r6 = s + 256
228
229     ldrb r1, [r3, r6]   @ r1 = s = table[s+256]
230
231     # @@@@ Checking if a0 = 0 or b1 = 0, return 0; without conditional branch
232     negs r6, r5         @ r6 = -a0
233
234     # @ asrs rd, <#imm> --> #imm: 0-31
235     movs r3, #32
236     asrs r6, r3         @ r6 = -a0 >> 32
237
238     ands r0, r6         @ r0 = tmp = b1 & (-a0 >> 32)
239     negs r0, r0         @ r0 = -tmp
240
241     # @ asrs rd, <#imm> --> #imm: 0-31
242     asrs r0, r3         @ r0 = -tmp >> 32
243
244     ands r0, r1         @ r0 = s & (-tmp >> 32)  = (a0*b1)
245
246     eors r4, r0   @ r4 = (rnd0+(a0*b1))
247
248
249 @@@@ r0 = (a0*b1), r1 = table[s+256], r2 = &rnd, r3 = #32
250 @@@@ r4 = (rnd0+(a0*b1)), r5 = a0, r6 = a0 >> 32, r7 = (a1*b1),
251 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
252 @@@@
```

```
  mov  r0, r9          @ r0 = &a
  mov  r5, r10         @ r5 = &b
  mov  r6, r11         @ r6 = &table

  ldrb r3, [r0, #4]    @ r3 = a1
  ldrb r1, [r5, #0]    @ r1 = b0

  ldrb r0, [r6, r3]    @ r0 = table[a1]
  ldrb r5, [r6, r1]    @ r5 = table[b0]

  adds r0, r5          @ r0 = s = table[a1] + table[b0]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r5, #250        @ r5 = 250, 256 is not possible, so it is split: 250 + 6
  adds r5, #6          @ r5 = 256
  adds r0, r5          @ r0 = s + 256

  ldrb r5, [r6, r0]    @ r5 = s = table[s+256]

  # @@@@ Checking if a1 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r3          @ r0 = -a1

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r6, #32
  asrs r0, r6          @ r0 = -a1 >> 32

  ands r1, r0          @ r1 = tmp = b0 & (-a1 >> 32)
  negs r1, r1          @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r6          @ r1 = -tmp >> 32

  ands r1, r5          @ r1 = s & (-tmp >> 32) = (a1*b0)



@@@@ r0 = -a1 >> 32, r1 = (a1*b0), r2 = &rnd, r3 = a1
@@@@ r4 = (rnd0+(a0*b1)), r5 = table[s+256], r6 = 32 , r7 = (a1*b1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
@@@@ c1 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1)+rnd2

## Leakage sh0-sh1
  # @ first operands in two eors = r1:a1b0 --> r7:a1b1 , leaks:b0+b1
  # ldrb r0, [r2, #8]   @ r0 = rnd2
  # eors r1, r4         @ r1 = [(rnd0+(a0*b1))+(a1*b0)]
  # eors r7, r1         @ r7 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1)

  eors r1, r4         @ r1 = [(rnd0+(a0*b1))+(a1*b0)]
  ldrb r0, [r2, #8]   @ r0 = rnd2
  eors r7, r1         @ r7 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1)



  eors r0, r7         @ r0 = c1 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1)+rnd2

  mov  r6, r8

  strb r0, [r6, #4]   @ r0 = c1



@@@@ r0 = c1, r1 = [(rnd0+(a0*b1))+(a1*b0)], r2 = &rnd, r3 = a1
@@@@ r4 = (rnd0+(a0*b1)), r5 = table[s+256], r6 = 32 , r7 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@ mov  r4, r8         @ r4 = &c
@@@@ strb r3, [r4, #8]   @ r3 = (a2*b2)
@@@@ (a2 *b2) is in [&c + #8]
@@@@
```

```
344
345
346     ldrb r6, [r0, #0]   @ r6 = a0
347     ldrb r1, [r2, #4]   @ r1 = rnd1
348     ldrb r5, [r7, #8]   @ r5 = b2
349
350     ldrb r0, [r4, r6]   @ r0 = table[a0]
351     ldrb r7, [r4, r5]   @ r7 = table[b2]
352
353     adds r0, r7         @ r0 = s = table[a0] + table[b2]
354
355     @@@@ Get the antilog: s = table[s+256]: g+s
356     # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
357     # @ movs rd, <#imm> --> #imm: 0-255
358     movs r7, #250       @ r7 = 250, 256 is not possible, so it is split: 250 + 6
359     adds r7, #6         @ r7 = 256
360     adds r0, r7         @ r0 = s + 256
361
362     ldrb r7, [r4, r0]   @ r7 = s = table[s+256]
363
364     # @@@@ Checking if a0 = 0 or b2 = 0, return 0; without conditional branch
365     negs r0, r6         @ r0 = -a0
366
367     # @ asrs rd, <#imm> --> #imm: 0-31
368     movs r4, #32
369     asrs r0, r4         @ r0 = -a0 >> 32
370
371     ands r5, r0         @ r5 = tmp = b2 & (-a0 >> 32)
372     negs r5, r5         @ r5 = -tmp
373
374     # @ asrs rd, <#imm> --> #imm: 0-31
375     asrs r5, r4         @ r5 = -tmp >> 32
376
377     ands r5, r7         @ r5 = s & (-tmp >> 32) = (a0*b2)
378
379
380
381 @@@@
```

```

@@@@ r0 = -a0 >> 32, r1 = (rnd1+(a0*b2)), r2 = &rnd, r3 = a1
@@@@ r4 = 32, r5 = (a0*b2), r6 = a0 , r7 = table[s+256],
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table

@@@@
```

```
  negs r0, r0          @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r4          @ r0 = -tmp >> 32

  ands r0, r6          @ r0 = s & (-tmp >> 32) = (a2*b0)

@@@
```

```
  # @@@@ Checking if a1 = 0 or b2 = 0, return 0; without conditional branch
  negs r2, r3         @ r2 = -a1

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r4, #32
  asrs r2, r4         @ r2 = -a1 >> 32

  ands r1, r2         @ r1 = tmp = b2 & (-a1 >> 32)
  negs r1, r1         @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r4         @ r1 = -tmp >> 32

  ands r1, r5         @ r1 = s & (-tmp >> 32) = (a1*b2)

  # mov  r4, r8         @ r = &c
  # strb r1, [r4, #11]

# # computing (rnd2+(a1*b2))

  eors r6, r1         @ r6 = (rnd2+(a1*b2))


@@@
```

```

# mov  r2, r8          @ r2 = &c  @ (a2 *b2) is in [&c + #8]
# mov  r4, r11         @ r4 = &table
# mov  r3, r10         @ r3 = &b

# ldrb r5, [r2, #8]   @ r5 = (a2*b2)
# ldrb r1, [r3, #4]   @ r1 = b1

# ldrb r2, [r4, r7]   @ r2 = table[a2]
# ldrb r3, [r4, r1]   @ r3 = table[b1]

eors r5, r5          @ r5 = 0, Clearing r5
eors r1, r1          @ r1 = 0, Clearing r1
mov  r2, r8          @ r2 = &c  @ (a2 *b2) is in [&c + #8]
mov  r4, r11         @ r4 = &table
mov  r3, r10         @ r3 = &b
# ldrb r5, [r4, r2] @ r5 = s = table[s+256] = a1*b2
ldrb r1, [r4, #0]   @ r1 = 0, Clearing the read bus
ldrb r5, [r2, #8]   @ r5 = (a2*b2)
ldrb r2, [r4, r7]   @ r2 = table[a2]
ldrb r1, [r3, #4]   @ r1 = b1
ldrb r3, [r4, r1]   @ r3 = table[b1]


adds r2, r3          @ r2 = s = table[a2] + table[b1]

@@@@ Get the antilog: s = table[s+256]: g+s
# @ adds rd, <#imm> --> #imm: 3 bits, 0-7
# @ movs rd, <#imm> --> #imm: 0-255
movs r3, #250        @ r3 = 250, 256 is not possible, so it is split: 250 + 6
adds r3, #6          @ r3 = 256
adds r2, r3          @ r2 = s + 256

ldrb r3, [r4, r2]   @ r3 = s = table[s+256]

# @@@@ Checking if a2 = 0 or b1 = 0, return 0; without conditional branch
negs r2, r7          @ r2 = -a2

# @ asrs rd, <#imm> --> #imm: 0-31
movs r4, #32
asrs r2, r4          @ r2 = -a2 >> 32

ands r1, r2          @ r1 = tmp = b1 & (-a2 >> 32)
```

```
  negs r1, r1          @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r4          @ r1 = -tmp >> 32

  ands r1, r3          @ r1 = s & (-tmp >> 32) = (a2*b1)

  @@@@ c2 = [(rnd1+(a0*b2))+(a2*b0)]+[(rnd2+(a1*b2))+(a2*b1)]+(a2*b2)

  mov  r4, r8          @ r4 = &c
  eors r6, r1          @ r6 = [(rnd2+(a1*b2))+(a2*b1)]
  eors r0, r6          @ r0 = [(rnd1+(a0*b2))+(a2*b0)]+[(rnd2+(a1*b2))+(a2*b1)]
  eors r5, r0          @ r5 = c2 = [(rnd1+(a0*b2))+(a2*b0)]+[(rnd2+(a1*b2))+(a2*b1)]+(a2*b2)

  strb r5, [r4, #8]    @ r5 = c2


nop
nop
nop
nop
```

1) First possible leakage: P_L1
Lines 53 and 80 in Listing 6.17.
The read bus:
$a_0.b_0 \rightarrow a_1$
Patch: Clearing the read bus

Listing 6.18: Source of P_L1 in isw3-1

```
# r0 = table[a0] + table[b0] + 256
# r3 = &table
# r1 = 256
  ldrb r1, [r3, r0]   @ r1 = s = table[s+256] = a0*b0

        A sequence of instructions

@@@@ a1*b1

  ldrb r0, [r3, r6]   @ r0 = table[a1]
```

2) Second possible leakage: P_L2
Lines 70 and 80 in Listing 6.17.
Manipulating different shares from the same family in sequence:
$a_0.b_0$ and a_1
Patch: Adding some instruction/s between the two instructions that manipulate unrelated value/s.

Listing 6.19: Source of P_L2 in isw3-1

```

# r0 = -a0 >> 32
# r1 = table[s+256] = a0*b0
# r2 = &rnd
# r3 = &table
# r4 = #32
# r5 = -tmp >> 32
# r6 = a1
# r7 = b1

  ands r5, r1         @ r5 = s & (-tmp >> 32) = (a0*b0)

@@@@ a1*b1

  ldrb r0, [r3, r6]   @ r0 = table[a1]
```

3) P_L3
Line 81 in Listing 6.17.
Overwriting the register:

$r1 = a_0.b_0 \rightarrow b_1$
Patch: Clearing the register

Listing 6.20: Source of P_L3 in isw3-1

```

# r0 = -a0 >> 32
# r1 = table[s+256] = a0*b0
# r2 = &rnd
# r3 = &table
# r4 = #32
# r5 = a0*b0
# r6 = a1
# r7 = b1
@@@@ a1*b1

  ldrb r1, [r3, r7]   @ r1 = table[b1]
```

Listing 6.21 shows the patches for the three leakages mentioned above.

Listing 6.21: Patches for P_L1 - P_L3 in isw3-1

```

# ands r5, r1          @ r5 = s & (-tmp >> 32) = (a0*b0)

    eors r1, r1          @ r1 = 0, Clearing reg 1
    ldrb r4, [r3, #0]    @ r4 = table[0] = 0, clearing the read bus

# ldrb r0, [r3, r6]    @ r0 = table[a1]
```

4) P_L4
Lines 92 and 129 in Listing 6.17.
The read bus:
$a_1.b_1 \rightarrow a_2$
Patch: Clearing the read bus

Listing 6.22: Source of P_L4 in isw3-1

```

 ldrb r1, [r3, r0]   @ r1 = s = table[s+256] = a1*b1 (the read bus)

       A sequence of instructions

  ldrb r6, [r0, #8]   @ r6 = a2

```

5) P_L5
Lines 126- 129 in Listing 6.17.
Manipulating different shares from the same family in sequence:
$a_1.b_1 \rightarrow a_0 \rightarrow a_2$
Patch: Adding some instruction/s between the two instructions that manipulate unrelated value/s.

Listing 6.23: Source of P_L5 in isw3-1

```
# r1 = table[s+256]=a1*b1
# r4 = a0
# r6 = 0

  mov  r1, r10        @ r1 = &b
  mov  r4, r11        @ r4 = &table

  ldrb r6, [r0, #8]   @ r6 = a2

```

For avoiding the leakages P_L4 and P_L5, we can change instructions between Lines 122-133 in Listing 6.17 (Listing 6.24) to Listing 6.25.
We use ldrb r6, [r4, #0] @ r6 = table[0] = 0 for clearing the read bus and also instead of eors r6, r6 for clearing r6.
In addition, we change the order of some instructions for avoiding: 1) Stall 2) Manipulating different shares of the same family.
What happens if the compiler optimizes the code???
For example: The compiler removes ldrb r6, [r4, #0] @ r6 = 0, as after this instruction, there is ldrb r6, [r0, #8] @ r6 = a2.
Regarding:
1) As the compiler does not know that $table[0] = 0$.
2) $r3$ contains 32, and it does not have any effect on the result.
3) ands between zero with any number is zero.
In this case (optimization), for clearing r6 an also for clearing the read bus, instead of ldrb r6, [r4, #0] @ r6 = 0, we can use:
ldrb r3, [r4, #0] @ r3 = 0
ands r6, r3 @ r6 = 0
Can we say: the compiler can replaces eors r6, r6 with movs r6, #0?? Because if it is yes, port2 will hold its previous value. Should we be careful about using eors for clearing the registers???

Listing 6.24: Source of P_L4 and P_L5 in isw3-1

```

 ldrb r1, [r3, r0]   @ r1 = s = table[s+256] = a1*b1 (the read bus)

        A sequence of instructions

# r0 = -a1 >> 32
# r1 = table[s+256]=a1*b1
# r2 = &rnd
# r3 = 32
# r4 = a0
# r5 = (a0*b0)
# r6 = a1
# r7 = (a1*b1)

  eors r6, r6         @ =Reg 6 HD, a1 ---> a2

  mov  r0, r9         @ r0 = &a
  mov  r1, r10        @ r1 = &b
  mov  r4, r11        @ r4 = &table

  ldrb r6, [r0, #8]   @ r6 = a2
  ldrb r3, [r1, #8]   @ r3 = b2

  ldrb r0, [r4, r6]   @ r0 = table[a2]
  ldrb r1, [r4, r3]   @ r1 = table[b2]

```

Listing 6.25: Patching P_L4 and P_L5 in isw3-1

```

 ldrb r1, [r3, r0]   @ r1 = s = table[s+256] = a1*b1 (the read bus)

        A sequence of instructions

# r0 = -a1 >> 32
# r1 = table[s+256]=a1*b1
# r2 = &rnd
# r3 = 32
# r4 = a0
# r5 = (a0*b0)
# r6 = a1
# r7 = (a1*b1)

# eors r6, r6         @ pre_r6 = a1

  mov  r1, r10        @ r1 = &b
  mov  r4, r11        @ r4 = &table
  mov  r0, r9         @ r0 = &a
  ldrb r6, [r4, #0]   @ r6 = table[0] = 0, it also clear r6, no need to have eors r6, r6
  ldrb r3, [r1, #8]   @ r3 = b2
  ldrb r6, [r0, #8]   @ r6 = a2
  ldrb r1, [r4, r3]   @ r1 = table[b2]
  ldrb r0, [r4, r6]   @ r0 = table[a2]

```

4) P_L6
Lines 229 and 262 in Listing 6.17.
The read bus:
$a_0.b_1 \rightarrow a_1$
Patch: Clearing the read bus

Listing 6.26: Source of P_L6 in isw3-1

```
1
2  drb r1, [r3, r6]    @ r1 = s = table[s+256] = a0*b1 (the read bus)
3
4        A sequence of instructions
5
6   ldrb r3, [r0, #4]   @ r3 = a1
7
```

3) P_L7
Line 81 in Listing 6.17.
Overwriting the register:
$r1 = a_0.b_1 \rightarrow b_0$
Patch: Clearing the register
In addition, regarding the previous value of $r1$, here different shares of the same family are manipulated in sequence:
a_1, b_0 and a_0, b_1
Patch: Clearing the register
For avoiding the leakages P_L6 and P_L7, we can change instructions between Lines 258-263 in Listing 6.17 (Listing 6.28) to Listing 6.29.
We use eors r1, r1 for clearing r1 and also, ldrb r1, [r6, #0] @ r1 = table[0] = 0 for clearing the read bus.
In addition, we change the order of some instructions for avoiding: 1) Stall 2) Manipulating different shares of the same family.
What happens if the compiler optimizes the code???
For example: The compiler changes eors r1, r1 to movs r1, #0,
or
The compiler removes lldrb r1, [r6, #0] @ r1 = table[0] = 0, as after this insteuction, there is ldrb r1, [r5, #0] @ r1 = b0.
We can do like Listing 6.30.

Listing 6.27: Source of P_L7 in isw3-1

```

# r0 = a0*b1
# r1 = table[s+256] = a0*b1
# r2 = &rnd
# r3 = #32
# r4 = rnd0+(a0*b1)
# r5 = a0
# r6 = r6 = a0 >> 32
# r7 = (a1*b1)
@@@@ a1*b0

  mov  r0, r9          @ r0 = &a
  mov  r5, r10         @ r5 = &b
  mov  r6, r11         @ r6 = &table

  ldrb r3, [r0, #4]    @ r3 = a1

  ldrb r1, [r5, #0]    @ r1 = b0, overwriting reg
```

Listing 6.28: Source of P_L6 and P_L7 in isw3-1

```

 ldrb r1, [r3, r6]     @ r1 = s = table[s+256] = a0*b1 (the read bus) (the read bus)

        A sequence of instructions

# r0 = a0*b1
# r1 = table[s+256] = a0*b1
# r2 = &rnd
# r3 = #32
# r4 = rnd0+(a0*b1)
# r5 = a0
# r6 = r6 = a0 >> 32
# r7 = (a1*b1)
@@@@ a1*b0

  mov  r0, r9          @ r0 = &a
  mov  r5, r10         @ r5 = &b
  mov  r6, r11         @ r6 = &table

  ldrb r3, [r0, #4]    @ r3 = a1
  ldrb r1, [r5, #0]    @ r1 = b0

```

Listing 6.29: Patching P_L6 and P_L7 in isw3-1

```

 ldrb r1, [r3, r0]   @ r1 = s = table[s+256] = a1*b1 (the read bus)

        A sequence of instructions

# r0 = a0*b1
# r1 = table[s+256] = a0*b1
# r2 = &rnd
# r3 = #32
# r4 = rnd0+(a0*b1)
# r5 = a0
# r6 = r6 = a0 >> 32
# r7 = (a1*b1)
@@@@ a1*b0

  mov  r6, r11        @ r6 = &table
  mov  r0, r9         @ r0 = &a

  eors r1, r1         @ r1 = 0, Clearing r1

  mov  r5, r10        @ r5 = &b

  ldrb r1, [r6, #0]   @ r1 = table[0] = 0, Clearing the read bus

  ldrb r3, [r0, #4]   @ r3 = a1
  ldrb r1, [r5, #0]   @ r1 = b0

```

Listing 6.30: Patching P_L6 and P_L7 in isw3-1 in case the compiler OPT

```

 ldrb r1, [r3, r0]   @ r1 = s = table[s+256] = a1*b1 (the read bus)

        A sequence of instructions

# r0 = a0*b1
# r1 = table[s+256] = a0*b1
# r2 = &rnd
# r3 = #32
# r4 = rnd0+(a0*b1)
# r5 = a0
# r6 = r6 = a0 >> 32
# r7 = (a1*b1)
@@@@ a1*b0

  mov  r6, r11        @ r6 = &table
  mov  r0, r9         @ r0 = &a

  eors r5, r5         @ r5 = 0

  ldrb r3, [r6, #0]   @ r3 = table[0] = 0, Clearing the read bus
  ands r1, r3         @ r1 = 0, Clearing r1

  mov  r5, r10        @ r5 = &b

  ldrb r3, [r0, #4]   @ r3 = a1
  ldrb r1, [r5, #0]   @ r1 = b0

```

8) P_L8
Line 292 and 348 in Listing 6.17.
Overwriting the register:
$r5 = a_1.b_0 \rightarrow b_2$
Patch: Clearing the register

Listing 6.31: Source of P_L8 in isw3-1

```
1
2  # r5 = table[table[a1] + table[b0]=256] = a1*b0
3  @@@@ a1*b0
4    ands r1, r5          @ r1 = s & (-tmp >> 32) = (a1*b0)
5
6   A sequence of instructions
7
8    ldrb r5, [r7, #8]    @ r5 = b2
```

Listing 6.32 shows the patch the leakage mentioned above.

Listing 6.32: Patch for P_L8 in isw3-1

```
1
2    ands r1, r5          @ r1 = s & (-tmp >> 32) = (a1*b0)
3
4          eors r5, r5        @ r5 = 0
5
6   A sequence of instructions
7
8    ldrb r5, [r7, #8]    @ r5 = b2
```

9) P_L9
Line 377 and 402 in Listing 6.17.
Overwriting the register:
$r7 = a_0.b_2 \rightarrow a_2$
Also in Line 402, the read bus leakage can happen.
Patch: Clearing the register

Listing 6.34 shows the patch the leakage mentioned above.
10) P_L10
Regarding the Lines 163, 187, 319 and 573 in Listing 6.17.
It can be concluded if (for example) the power consumption of $c_0 \oplus c_1$ (in the write bus) combined with the power consumption of c_2 (in pre-processing step), then the output $c = a.b$ will leak.
Patch: Clearing the write bus

Listing 6.36 shows the patch the leakage mentioned above.
Figure 6.22 demonstrates t-test results after patching all of the possible leakages.

Listing 6.33: Source of P_L9 in isw3-1

```

# r7 = table[table[a0] + table[b2]=256] = a0*b2
@@@@ a0*b2
  ands r5, r7          @ r5 = s & (-tmp >> 32) = (a0*b2)

 A sequence of instructions

  ldrb r7, [r5, #8]    @ r7 = a2
```

Listing 6.34: Patch for P_L9 in isw3-1

```

  ands r5, r7          @ r5 = s & (-tmp >> 32) = (a0*b2)
            eors r7, r7        @ r7 = 0
 A sequence of instructions

  ldrb r7, [r5, #8]    @ r7 = a2
```

Listing 6.35: Source of P_L10 in isw3-1

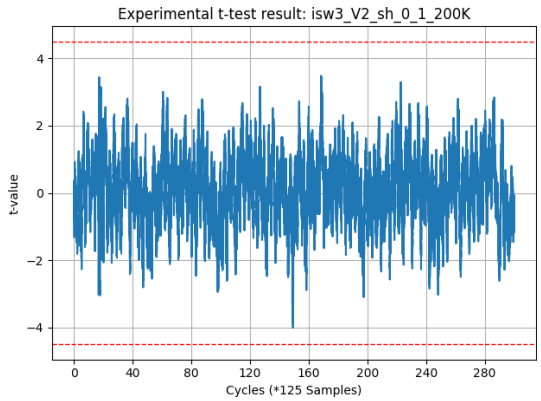
```
# c0 = (a0*b0)+rnd0+rnd1
# c1 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1)+rnd2
# c2 = [(rnd1+(a0*b2))+(a2*b0)]+[(rnd2+(a1*b2))+(a2*b1)]+(a2*b2)
  strb r3, [r4, #8]    @ Storing (a2*b2) in c2, r3 = (a2*b2)
        A sequence of instructions
  strb r1, [r0, #0]    @ Storing c0, r1 = c0 = (a0*b0)+rnd0+rnd1
        A sequence of instructions
  strb r0, [r6, #4]    @ Storing c1, r0 = c1
        A sequence of instructions
  strb r5, [r4, #8]    @ Storing c2, r5 = c2

```

Figure 6.23 demonstrates t-test results after patching all of the possible leakages up to 1 M traces.

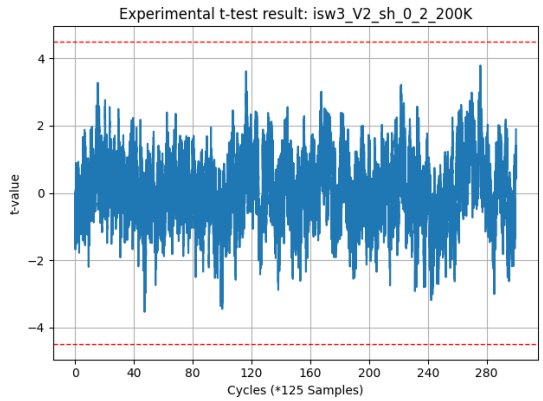
Listing 6.36: Patch for P_L10 in isw3-1

```
1
2  # c0 = (a0*b0)+rnd0+rnd1
3  # c1 = [(rnd0+(a0*b1))+(a1*b0)]+(a1*b1)+rnd2
4  # c2 = [(rnd1+(a0*b2))+(a2*b0)]+[(rnd2+(a1*b2))+(a2*b1)]+(a2*b2)
5
6    strb r3, [r4, #8]   @ Storing (a2*b2) in c2, r3 = (a2*b2)
7          A sequence of instructions
8    strb r1, [r0, #0]   @ Storing c0, r1 = c0 = (a0*b0)+rnd0+rnd1
9          A sequence of instructions
10   strb r0, [r6, #4]   @ Storing c1, r0 = c1
11         A sequence of instructions
12     strb r4, [r4, #8] @ Clearing the write bus
13   strb r5, [r4, #8]   @ Storing c2, r5 = c2
```

(a) Splitting-t-test (sh_0, sh_1)



(b) Splitting-t-test (sh_0, sh_2)



(c) Splitting-t-test (sh_1, sh_2)

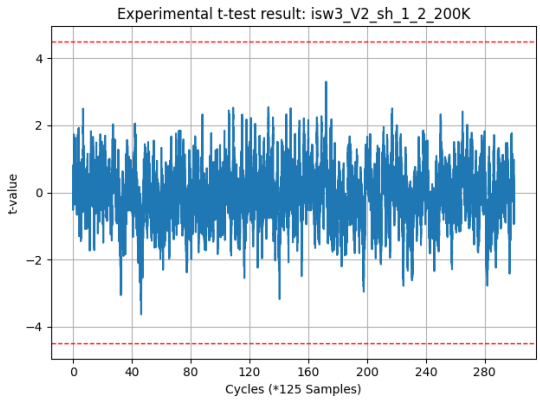
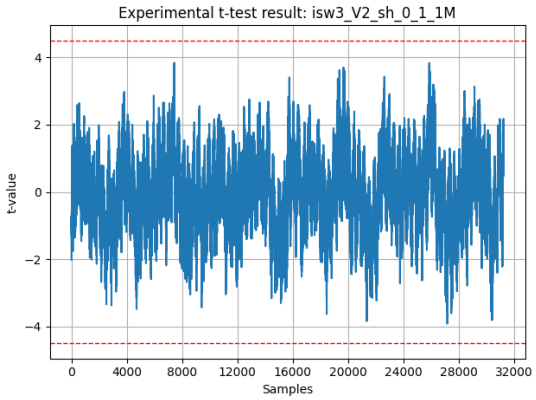
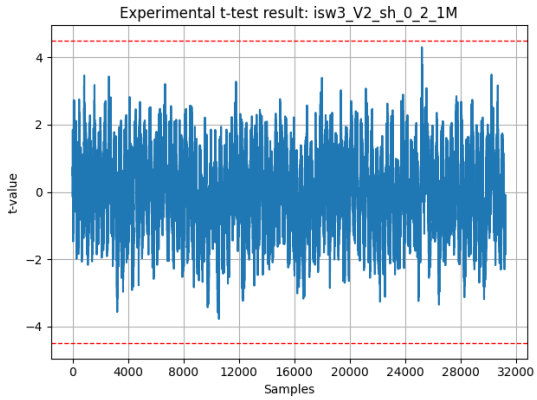


Figure 6.22: t-test isw-3-V2 (Setup: 250MSa/s, 31250S, 4ns, 2MHz)

(a) Splitting-t-test (sh_0, sh_1)



(b) Splitting-t-test (sh_0, sh_2)



(c) Splitting-t-test (sh_1, sh_2)

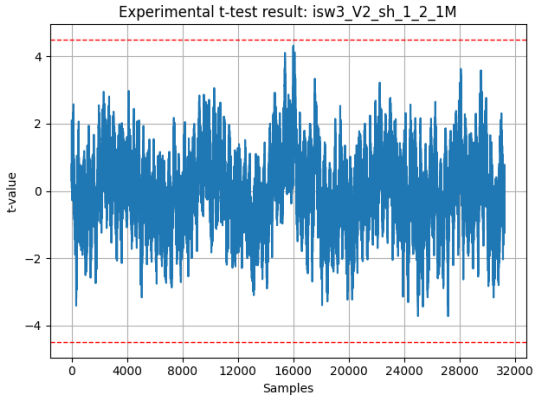


Figure 6.23: t-test isw-3-V2-1M (Setup: 250MSa/s, 31250S, 4ns, 2MHz)

Note:

Most of leakages caused by Overwriting the register rx (as well as the read bus leakage), happen in:

ldrb rx, [&table, table[ai] + table[bj]] @ rx = table[s+256], where s = [&table, table[ai] + table[bj]], rx = ai*bj

ands ry, rx @ ry = s & (-tmp » 32) = (ai*bj)

The two above instructions result in:
$rx = ry = a_i.b_j$ (where $a_i, b_j \neq 0$)
This means, two registers contain the value of $a_i.b_j$.

For avoiding this, it is better instead of using:

ands ry, rx @ ry = s & (-tmp » 32) = (ai*bj)
we use:

ands rx, ry @ rx = s & (-tmp » 32) = (ai*bj)
With this approach we can save several cycles (at most 9 cycles, as there are 9 multiplications in ISW3).
But if we change it, we need to reapply the leakage detection step from the first.

7. BBPP

BBPP multiplication is introduced in [1]. Algorithm 5 represents this scheme for $d + 1$ shares.

Algorithm 5 BBPP

Require: shares $(\alpha_{i,j} = a_i.b_j)_{0 \leq i,j \leq d}$ such as $a = \oplus_{i=0}^{d} a_i$, $b = \oplus_{i=0}^{d} b_i$

Ensure: $c_0, ..., c_d$ such that $c = \oplus_{i=0}^{d} c_i = a.b$

return $c_0, ..., c_d$

8. BBPP-2- no function/label call

The BBPP multiplication for two shares (BBPP-2) is illustrated in Algorithm 6 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 6 BBPP-2

Require: shares a_0, a_1, b_0, b_1, rnd

Ensure: c_0, c_1

1: $c_0 \leftarrow (a_0.b_0) \oplus rnd \oplus (a_0.b_1) \oplus (a_1.b_0)$

2: $c_1 \leftarrow (a_1.b_1) \oplus rnd$

In the implementation of BBPP-2, for the GF multiplications $(a_i.b_j)$, the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

8.1 BBPP-2-no-function-call-0: basic code

Listing 8.1 demonstrates the first implementation of BBPP-2 based gfmul-s (BBPP-2-no-function-call-0). The instructions in Listing 8.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 8.1: BBPP-2-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3         @ r8 = &c


@@@@@ Saving r2 (the address of the rnd)
  mov  r10, r2        @ r10 = &rnd


@@@@ Saving the value 256 in r11. This decreases one cycle.
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r4, #250       @ r4 = 250, 256 is not possible, so it is split: 250+6
  adds r4, #6         @ r4 = 256
  mov  r11, r4        @ r11 = 256


  ldr  r4, =table     @ r4 = &table, The address of the "table", for ai*bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r9, r4         @ r9 = &table


nop
nop
nop
nop



@@@@ r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
@@@@ r8 = &c, r9 = &table, r11 = 256
@@@@@
```

```
@@@@ r0 = &a, r1 = &b, r2 = rnd, r4 = &table
@@@@ r3 = a1, r5 = b0, r6 = a0, r7 = b1,
@@@@ r8 = &c, r9 = &table, r10 = &rnd, r11 = 256
@@@@
```

```
@@@@ a0*b1

  mov  r2, r9          @ r2 = &table
  ldrb r0, [r2, r6]    @ r0 = table[a0]
  ldrb r4, [r2, r7]    @ r4 = table[b1]

  adds r0, r4          @ r0 = s = table[a0]+table[b1]

  @@@@ Get the antilog: s = table[s+256]: g+s
  mov  r4, r11         @ r4 = 256

  adds r0, r4          @ r0 = s+256

  ldrb r4, [r2, r0]    @ r4 = s = table[s+256]

  # @@@@ Checking if a0 = 0 or b1 = 0, return 0; without conditional branch
  negs r0, r6          @ r0 = -a0 hex(256-dec(a0))

  # @ asrs rd, <#imm> --> #imm: 0-31
  @ a0 in r6 is not needed anymore, r6 can be used
  movs r6, #32         @ r6 = #32
  asrs r0, r6          @ r0 = -a0 >> 32

  ands r0, r7          @ r0 = tmp = b1 & (-a0 >> 32)
  negs r0, r0          @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r6          @ r0 = -tmp >> 32

# @ using the same register, r4 = table[s+256]
  ands r4, r0          @ r4 = s & (-tmp >> 32) = (a0*b1)

  @@@@ (a0*b0)+rnd+(a0*b1)
  eors r1, r4          @ r2 = (a0*b0)+rnd+(a0*b1)

@ c0 = (a0*b0)+rnd+(a0*b1)+(a1*b0)
@@@@ r0 = (a0*b1), r1 = (a0*b0)+rnd+(a0*b1), r2 = &table, r3 = a1
@@@@ r4 = (a0*b1), r5 = b0, r6 = 32, r7 = b1,
@@@@ r8 = &c, r9 = &table, r10 = &rnd, r11 = 256
@@@@
```

```
  ldrb r0, [r2, r3]   @ r0 = table[a1]
  ldrb r4, [r2, r5]   @ r4 = table[b0]

  adds r0, r4         @ r0 = s = table[a1]+table[b0]

  @@@@ Get the antilog: s = table[s+256]: g+s
  mov  r4, r11        @ r4 = 256

  adds r0, r4         @ r0 = s+256

  ldrb r4, [r2, r0]   @ r4 = s = table[s+256]

  # @@@@ Checking if a1 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r3         @ r0 = -a1 hex(256-dec(a1))

  # @ asrs rd, <#imm> --> #imm: 0-31
  # movs r6, #32        @ r6 = #32
  asrs r0, r6         @ r0 = -a1 >> 32

# b0 in r5 is not needed anymore
  ands r5, r0         @ r5 = tmp = b0 & (-a1 >> 32)

  negs r5, r5         @ r5 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r5, r6         @ r5 = -tmp >> 32

# @ using the same register, r4 = table[s+256]
  ands r4, r5         @ r4 = s & (-tmp >> 32) = (a1*b0)

  @@@@ c0 = (a0*b0)+rnd+(a0*b1)+(a1*b0)

  eors r1, r4         @ r1 = ((a0*b1)+rnd)+(a1*b0)

  mov  r4, r8         @ r4 = &c, also clearing a1*b0 from r4
  strb r1, [r4, #0]   @ Storing c0, r1 = c0

  mov  r1, r10        @ r1 = &rnd


@@@@@@@@@ Computing c1 = (a1*b1)+rnd
@@@@ r0 = -a1 >> 32, r1 = &rnd, r2 = &table, r3 = a1
@@@@ r4 = &c, r5 = -tmp >> 32, r6 = 32, r7 = b1,
```

```
@@@@ r8 = &c, r9 = &table, r10 = &rnd, r11 = 256
@@@@
```

```
215
216     nop
217     nop
218     nop
219     nop
```

8.1.1 Experimental result:

8.1.1.1 Setup:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $82\mu s$
Each clock is sampled with 125 points
Number of samples = 82×125 = 10250 samples
The bytes of shares (a, b, c are not in a word)

8.1.1.2 t-test:

There are no leakages for up to 500K traces (figures 8.1 and 8.2).

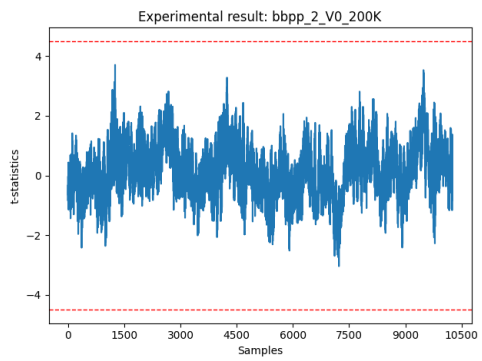


Figure 8.1: t-test_bbp2_V0 (Traces exist) (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

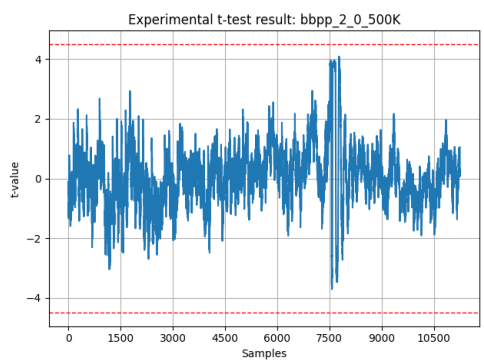


Figure 8.2: t-test_bbpp2_0 (Setup: 250MSa/s, 10250S, 4ns, 2MHz)

Also, we used the leakage simulator uELMO, which reports no leakage.

9. BBPP-OPT-3- no function/label call

The BBPP-3 multiplication (three shares) is shown in Algorithm 7 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 7 BBPP-OPT-3

Require: shares $a_0, a_1, a_2, b_0, b_1, b_2, rnd_0, rnd_1$

Ensure: c_0, c_1, c_2

1: $c_0 \leftarrow (a_0.b_0) \oplus rnd_0 \oplus (a_0.b_2) \oplus (a_2.b_0)$

2: $c_1 \leftarrow (a_1.b_1) \oplus rnd_1 \oplus (a_0.b_1) \oplus (a_1.b_0)$

3: $c_2 \leftarrow (a_2.b_2) \oplus rnd_0 \oplus rnd_1 \oplus (a_1.b_2) \oplus (a_2.b_1)$

In the implementation of BBPP-OPT-3, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

9.1 BBPP-OPT-3-no-function-call-0: basic code

Listing 9.1 demonstrates the first implementation of BBPP-OPT-3 based gfmul-s (BBPP-OPT-3-no-function-call-0). The instructions in Listing 9.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 9.1: BBPP-OPT-3-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @r8 = &c

@@@@@ Saving r0 (the address of the input a)
  mov  r9, r0        @ r9 = &a

@@@@@ Saving r1 (the address of the input b)
  mov  r10, r1       @ r10 = &b


  ldr  r4, =table    @ r4 = &taable, The address of the "table", for ai*bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r11, r4       @ r11 = &table

nop
nop
nop
nop

@@@@@ Loading arguments @@@@@@@@@@@@
  ldrb r3, [r0, #0]  @ r3 = a0; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
  ldrb r5, [r1, #0]  @ r5 = b0; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
  ldrb r6, [r0, #4]  @ r6 = a1; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
  ldrb r7, [r1, #4]  @ r7 = b1; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table

@@@@@@@@@@ Computing: ai*bi

@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = b0, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@@
```

```
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
  adds r1, #6         @ r1 = 256
  adds r0, r1         @ r0 = s + 256

  ldrb r1, [r4, r0]   @ r1 = s = table[s+256]

  # @@@@ Checking if a0 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r3         @ r0 = -a0 hex(256-dec(a0))

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r4, #32        @ r4 = #32
  asrs r0, r4         @ r0 = -a0 >> 32

  ands r0, r5         @ r0 = tmp = b0 & (-a0 >> 32)
  negs r0, r0         @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r4         @ r0 = -tmp >> 32

  ands r1, r0         @ r1 = s & (-tmp >> 32) = (a0*b0)

  mov  r0, r8         @ r0 = &c
  mov  r4, r11        @ r4 = &table    @ here for avoiding stall
  strb r1, [r0, #0]   @ r1 = (a0*b0), Storing (a0*b0) in &c+#0, r1 = (a0*b0)

  eors r1, r1         @ r1 = 0, clearing (a0*b0) @ here to avoid stall

  strb r0, [r0, #4]   @ clearing the write bus

### we need to clear the read bus and also reg1, istead of using two instructions
  ## eors r1, r1         @ r1 = 0, clearing (a0*b0)
  ## ldrb rx, [r4, #0]   @ rx = 0 = table[0],  Clearing the read bus
### just we can use one as:
  # ldrb r1, [r4, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg1
### The one instruction, is not enough, we should use the both instruction,
### because the first operand in one-inst can cause leakage with the next inst
  # ex: r1 and r0, two shares from the same family
    # ldrb r1, [r4, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg5
    # ldrb r0, [r4, r6]   @ r0 = table[a1]
  # eors r1, r1         @ r1 = 0, clearing (a0*b0) @ above to avoid stall
  ldrb r1, [r4, #0]   @ rx = 0 = table[0],  Clearing the read bus
```

```
@@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = b0, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  strb r7, [r7, #8]   @ clearing the write bus

### we need to clear the read bus and also reg1, istead of using two instructions, just we
  # eors r1, r1         @ r1 = 0, clearing (a1*b1)
  # ldrb rx, [r4, #0]   @ rx = 0 = table[0],  Clearing the read bus
  ldrb r1, [r4, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg1

@@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = b0, r6 = 32, r7 = &c,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
172
173   # @ asrs rd, <#imm> --> #imm: 0-31
174   asrs r1, r4         @ r1 = -tmp >> 32
175
176   ands r7, r1         @ r7 = s & (-tmp >> 32) = (a2*b2)
177
178   mov  r1, r8         @ r1 = &c
179   mov  r4, r11        @ r4 = &table   @ here for avoiding stall
180   strb r7, [r1, #8]   @ r7 = (a2*b2), Storing (a2*b2) in c2, r1 = (a2*b2)
181   ## it is not possible to clear the write bus here
182
183 ### we need to clear the read bus and also reg1, istead of using two instructions
184   ## eors r7, r7         @ r7 = 0, clearing (a2*b2)
185   ## ldrb rx, [r4, #0]   @ rx = 0 = table[0],  Clearing the read bus
186 ### just we can use one as:
187   # ldrb r7, [r4, #0]   @ r7 = 0 = table[0],  Clearing the read bus and also reg1
188 ### The one instruction, is not enough, we should use the both instruction,
189 ### because the first operand in one-inst can cause leakage with the next inst
190   # ex: r7 and r1, two shares from the same family
191     # ldrb r7, [r4, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg5
192     # ldrb r1, [r4, r3]   @ r0 = table[a0]
193   eors r7, r7         @ r7 = 0, clearing (a2*b2)
194   ldrb r7, [r4, #0]   @ r7 = 0 = table[0],  Clearing the read bus and also reg5
195
196 @@@@@@@@@@ Computing: ci
197 @@@@ c0 = (a0*b0)+rnd0+(a0*b2)+(a2*b0)
198 @@@@ c1 = (a1*b1)+rnd1+(a0*b1)+(a1*b0)
199 @@@@ c2 = (a2*b2)+rnd0+rnd1+(a1*b2)+(a2*b1)
200
201 @@@@ r0 = b2, r1 = &c, r2 = &rnd, r3 = a0,
202 @@@@ r4 = &table, r5 = b0, r6 = a2, r7 = 0,
203 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
204 @@@@@@@@@@
```

```
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r7, #250       @ r7 = 250, 256 is not possible, so it is split: 250 + 6
  adds r7, #6         @ r7 = 256
  adds r1, r7         @ r1 = s + 256

  ldrb r7, [r4, r1]   @ r7 = s = table[s+256]

  # @@@@ Checking if a0 = 0 or b2 = 0, return 0; without conditional branch
  negs r1, r3         @ r1 = -a0

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r4, #32        @ r4 = 32
  asrs r1, r4         @ r1 = -a0 >> 32

# @ b2 in r0 is not needed anymore, r0 can be used
  ands r0, r1         @ r1 = tmp = b2 & (-a0 >> 32)
  negs r0, r0         @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r4         @ r0 = -tmp >> 32
  mov  r4, r11        @ r4 = &table    @ here for avoiding stall
  ands r7, r0         @ r7 = s & (-tmp >> 32) = (a0*b2)

@@@@ r0 = -tmp >> 32, r1 = -a0 >> 32, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = b0, r6 = a2, r7 = (a0*b2),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  adds r1, r0          @ r1 = s = table[a2] + table[b0]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r0, #250        @ r0 = 250, 256 is not possible, so it is split: 250 + 6
  adds r0, #6          @ r0 = 256
  adds r1, r0          @ r1 = s + 256

  ldrb r0, [r4, r1]    @ r0 = s = table[s+256]

  # @@@@ Checking if a2 = 0 or b0 = 0, return 0; without conditional branch
  negs r1, r6          @ r1 = -a2

  # @ a2 in r6 is not needed anymore, r6 can be used
  # # @ asrs rd, <#imm> --> #imm: 0-31
  movs r6, #32         @ r6 = 32
  asrs r1, r6          @ r1 = -a2 >> 32

  ands r1, r5          @ r1 = tmp = b0 & (-a2 >> 32)
  negs r1, r1          @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r6          @ r1 = -tmp >> 32

  ands r0, r1          @ r0 = s & (-tmp >> 32)  = (a2*b0)

@@@@ r0 = (a2*b0), r1 =  -tmp >> 32, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = b0, r6 = 32, r7 = (a0*b2),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  mov  r7, r10        @ r7 = &b , here for avoiding stall

  strb r0, [r6, #0]   @ r0 = c0, Storing c0, r0 = c0


### we need to clear the read bus and also reg0, istead of using two instructions
  ## eors r0, r0         @ r0 = 0, clearing c0
  ## ldrb rx, [r4, #0]   @ rx = 0 = table[0],  Clearing the read bus
### just we can use one as:
  # ldrb r0, [r4, #0]   @ r0 = 0 = table[0],  Clearing the read bus and also reg1
### The one instruction, is not enough, we should use the both instruction,
### because the first operand in one-inst can cause leakage with the next inst
  # ex: r0 and r0, two shares from the same family
    # ldrb r0, [r4, #0]   @ r0 = 0 = table[0],  Clearing the read bus and also reg5
    # ldrb r6, [r1, #4]   @ r6 = a1
  eors r0, r0         @ r0 = 0, clearing C0 from r0
  ldrb r0, [r4, #0]   @ r0 = 0, clearing c0 and the read bus which contains (a0*b0)


@@@@ r0 = 0, r1 = &a, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = b0, r6 = &c, r7 = &b,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
344
345   # @@@@ Checking if a0 = 0 or b1 = 0, return 0; without conditional branch
346   negs r1, r3          @ r1 = -a0
347
348   # @ a0 in r3 is not needed anymore, r3 can be used
349   # @ asrs rd, <#imm> --> #imm: 0-31
350   movs r3, #32         @ r3 = 32
351   asrs r1, r3          @ r1 = -a0 >> 32
352
353   ands r1, r0          @ r1 = tmp = b1 & (-a0 >> 32)
354   negs r1, r1          @ r1 = -tmp
355
356   # @ asrs rd, <#imm> --> #imm: 0-31
357   asrs r1, r3          @ r1 = -tmp >> 32
358   # mov  r4, r11         @ r4 = &table    @ here for avoiding stall
359   ands r7, r1          @ r7 = s & (-tmp >> 32) = (a0*b1)
360
361
362 @@@@ r0 = b1, r1 = -tmp >> 32, r2 = &rnd, r3 = 32,
363 @@@@ r4 = &table, r5 = b0, r6 = a1, r7 = (a0*b1),
364 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
365 @@@@
```

```
387   adds r3, #6          @ r3 = 256
388   adds r1, r3          @ r1 = s + 256
389
390   ldrb r3, [r4, r1]    @ r3 = s = table[s+256]
391
392   # @@@@ Checking if a1 = 0 or b0 = 0, return 0; without conditional branch
393   negs r1, r6          @ r1 = -a1
394
395   # # @ asrs rd, <#imm> --> #imm: 0-31
396   movs r4, #32         @ r4 = 32
397   asrs r1, r4          @ r1 = -a1 >> 32
398
399   # @ b0 in r5 is not needed anymore, r5 can be used
400   ands r5, r1          @ r5 = tmp = b0 & (-a1 >> 32)
401   negs r5, r5          @ r5 = -tmp
402
403   # @ asrs rd, <#imm> --> #imm: 0-31
404   asrs r5, r4          @ r5 = -tmp >> 32
405   ands r3, r5          @ r3 = s & (-tmp >> 32)  = (a1*b0)
406
407
408 @@@@ r0 = b1, r1 = -a1 >> 32, r2 = &rnd, r3 = (a1*b0),
409 @@@@ r4 =32, r5 = -tmp >> 32, r6 = a1, r7 = (a0*b1),
410 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
411 @@@@
```

```
  ## ldrb rx, [r4, #0]   @ rx = 0 = table[0],  Clearing the read bus
### just we can use one as:
  # ldrb r3, [r4, #0]   @ r3 = 0 = table[0],  Clearing the read bus and also reg1
### The one instruction, is not enough, we should use the both instruction,
### because the first operand in one-inst can cause leakage with the next inst
  # ex: r3 and r3, two shares from the same family
    # ldrb r3, [r4, #0]   @ r = 0 = table[0],  Clearing the read bus and also reg5
    # ldrb r5, [r1, #8]   @ r5 = a2
  eors r3, r3         @ r3 = 0, clearing C1 from r3
  ldrb r3, [r4, #0]   @ r3 = 0, clearing c1 and the read bus which contains (a1*b1)


@@@@ r0 = b1, r1 = &a, r2 = &rnd, r3 = 0,
@@@@ r4 = &table, r5 = &c, r6 = a1, r7 = &b,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
473   asrs r1, r6          @ r1 = -a1 >> 32
474
475   # @ b2 in r5 is not needed anymore, r5 can be used
476   ands r5, r1          @ r1 = tmp = b2 & (-a1 >> 32)
477   negs r5, r5          @ r1 = -tmp
478
479   # @ asrs rd, <#imm> --> #imm: 0-31
480   asrs r5, r6          @ r1 = -tmp >> 32
481   # mov  r4, r11         @ r4 = &table    @ here for avoiding stall
482   ands r7, r5          @ r7 = s & (-tmp >> 32) = (a1*b2)
483
484
485 # @@@@ r0 = b1, r1 = -a1 >> 32, r2 = &rnd, r3 = a2,
486 # @@@@ r4 = &table, r5 = -tmp >> 32, r6 = 32, r7 = (a1*b2),
487 # @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
488 # @@@@
```

```
516   negs r1, r3         @ r1 = -a2
517
518   # # @ asrs rd, <#imm> --> #imm: 0-31
519   # movs r6, #32        @ r6 = 32
520   asrs r1, r6         @ r1 = -a2 >> 32
521
522   # @ b1 in r0 is not needed anymore, r0 can be used
523   ands r0, r1         @ r0 = tmp = b1 & (-a2 >> 32)
524
525   # @ a2 in r3 is not needed anymore, r3 can be used
526   negs r3, r0         @ r3 = -tmp
527
528   # @ asrs rd, <#imm> --> #imm: 0-31
529   asrs r3, r6         @ r3 = -tmp >> 32
530   mov  r0, r8         @ r0 = &c       @ here to avoid stall
531   ands r5, r3         @ r5 = s & (-tmp >> 32)  = (a2*b1)
532
533
534 @@@@ r0 = &c, r1 = -a2 >> 32, r2 = &rnd, r3 = -tmp >> 32,
535 @@@@ r4 = &table, r5 = (a2*b1), r6 = 32, r7 = (a1*b2),
536 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
537 @@@@
```

9.1.1 Experimental result:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $152\mu s$
Each clock is sampled with 125 points
Number of samples = 304×125 = 38000 samples
The bytes of shares (a, b, c are not in a word)

9.1.1.1 Signal to Noise Ratio (SNR):

Figure 9.1 shows the SNR for all the intermediate values. It can be concluded that the first 7Ksamples (56×125) and samples do not have effect on the SNR, so, they can be removed.
Number of samples = $(304 - 56) \times 125 = 248 \times 125 = 31000$ samples.
The peaks between 0-5000 are related to rnd which are captured from the privious executions Figure 9.2.

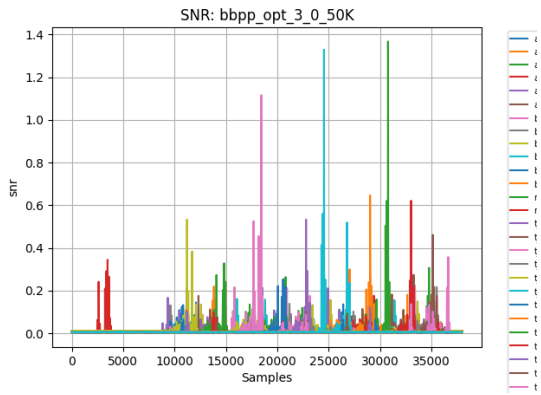


Figure 9.1: SNR bbpp-opt-3-without-b-0 (Setup: 250MSa/s, 38KS, 4ns, 2MHz)

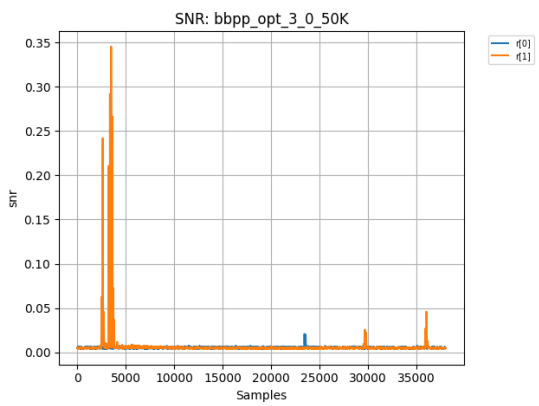


Figure 9.2: SNR bbpp-opt-3-without-b-0 (Setup: 250MSa/s, 38KS, 4ns, 2MHz)

9.1.1.2 t-test:

Finding leakages via splitting t-test in to three t-test s:

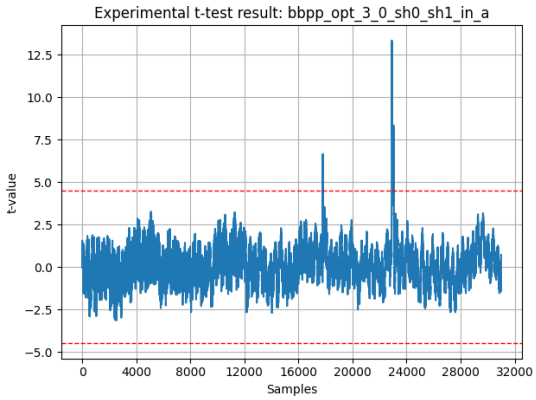
In this section we find the leakages caused by 2 shares out of 3 shares using the split-t-test .
First we use uElmo to find the leakages on V0, and then we remove the leakages and test with uELMO. When we have an implementation that does not shows any leakage in uELMO (V1), then we move to real experiments.
However, for V0 we show the results of the t-test on real traces.
Figures 9.8, 9.9, and 9.10 show the results of the t-test on real traces for both V0 and V1.

1) (sh_0, sh_1)
Listing 9.2 shows the the leakages found by **uEMO**.
in real traces (100K traces):
bbpp_opt_3_0_sh0_sh1_in_a:
Leaky cycles: [143, 184, 185]
Leaky cycles (from trigger): [63, 64, 105]
bbpp_opt_3_0_sh0_sh1_in_b:
Leaky cycles: [158, 173, 176]
Leaky cycles (from trigger): [72, 75, 90]
bbpp_opt_3_0_sh0_sh1:
Leaky cycles: [143, 158, 162, 163, 173, 174, 176, 184, 185, 186, 187, 202]
Leaky cycles (from trigger): [46, 61, 62, 63, 64, 72, 74, 75, 85, 86, 90, 105]

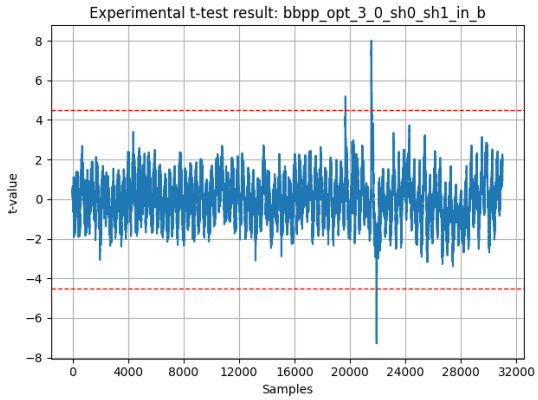
Listing 9.2: Leakages reported by **uELMO** for bbpp-opt-3-0-sh0-sh1

```
Cycle 159: Instr=Execute: ands r3,r5
Decode: mov r5,r8
Memory: idle
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=51.837702329839985
Cycle 167: Instr=Execute:eors r7,r4
Decode: mov r4,r11
Memory: idle
, Term=Pipeline Reg 1 HD, Tvalue=inf
Cycle 169: Instr=Execute:eors r3,r7
Decode: mov r7,r10
Memory: idle
, Term=Pipeline Reg 1 HD, Tvalue=inf
```

(a) Shares-a



(b) Shares-b



(c) Shares-a and Shares-b

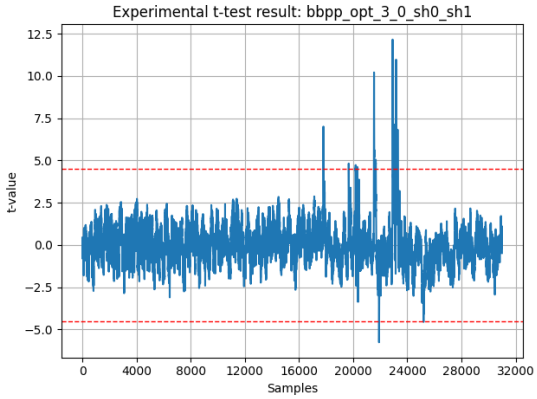


Figure 9.3: t-test bbpp-opt-3-without-b-0-sh0-sh1 (Setup: 250MSa/s, 31KS, 4ns, 2MHz)

Reasons and solutions of/for leakages:

1) Leakages in cycles 62-64 (from trigger) are related to instructions in Lines 419 and 423 in Listing 9.1 (listed in Listing 9.3).

The cause of leakage is mov instruction does not update the pipeline register RS1 in the execution stage according to Table 1 in [5].

Listing 9.3: Source of the first leakage in bbpp-opt-3-0-sh0-sh1

```
1
2
3  # r0 = b1
4  # r1 = rnd1
5  # r2 = &rnd
6  # r3 = (a1*b0)
7  # r4 = (a1*b1)
8  # r5 = -tmp >> 32
9  # r6 = a1
10 # r7 = (a0*b1)
11
12   eors r4, r1          @ r4 = (a1*b1)+rnd1
13   # MOV: here for avoiding stall,
14   # and overwriting ports and pipeline register with
15   # r4=(a1*b1), r7=(a0*b1)
16   mov  r1, r9          @ r1 = &a
17   eors r7, r4          @ r7 = (a1*b1)+rnd1+(a0*b1)
18   # MOV: here for avoiding stall,
19   # and overwriting ports and pipeline register with
20   # r7=(a0*b1), r3=(a1*b0)
21   mov  r4, r11         @ r4 = &table
22   eors r3, r7          @ r3 = c1 = (a1*b1)+rnd1+(a0*b1)+(a1*b0)
23   mov  r7, r10         @ r7 = &b , here for avoiding stall
24
25
```

Regarding Table 9.1, in pipeline register RS1:

$RS1 : a_1.b_1 \rightarrow a_0.b_1 = b_1.(a_0 \oplus a_1)$

It can be concluded if the power consumption of this point combined with the power consumption of $a_2.b_1$ (in pre-processing step), then will a leak. As:

$[b_1.(a_0 \oplus a_1)] \oplus a_2.b_1 = b_1.(a_0 \oplus a_1 \oplus a_2) = b_1.a$

Table 9.1: Leakage stems from mov r1, r9

Instruction		Decoding			Executing	
		Port 1	Port 2	Port 3	RS1	RS2
1	eors r4, r1	$r4 = a_1.b_1$	$r1 = rnd_1$	-	$r4 = a_1.b_1$	$r1 = rnd_1$
2	mov r1, r9	$r1 = rnd_1$	$r9 = ADDa$	-	-	$r9 = ADDa$
3	eors r7, r4	$r7 = a_0.b_1$	$r4 = a_1.b_1 \oplus rnd_1$	-	$r7 = a_0.b_1$	$r4 = a_1.b_1 \oplus rnd_1$

Regarding Table 9.2, in pipeline register RS1:
$RS1 : a_0.b_1 \rightarrow a_1.b_0$
Two shares from the same family are overwriting to the RS1 in sequence (a_0, a_1) and (b_0, b_1).

Table 9.2: Leakage stems from mov r4, r11

Instruction		Decoding			Executing	
		Port 1	Port 2	Port 3	RS1	RS2
1	eors r7, r4	$r7 = a_0.b_1$	$r4 = a_1.b_1 \oplus rnd_1$	-	$r7 = a_0.b_1$	$r4 = a_1.b_1 \oplus rnd_1$
2	mov r4, r11	$r4 = a_1.b_1 \oplus rnd_1$	$r11 = ADDtable$	-	-	$r11 = ADDtable$
3	eors r3, r7	$r3 = a_1.b_0$	$r7 = (a_1.b_1 \oplus rnd_1) \oplus a_0.b_1$	-	$r3 = a_1.b_0$	$r7 = (a_1.b_1 \oplus rnd_1) \oplus a_0.b_1$

In order to avoid this leakage, we have to use instructions that updated pipeline registers.
We use eors, and perform some reordering to avoid the stall. The patch is listed in Listing 9.4.
Figure 9.4 shows the t-test result after applying the patch.

Listing 9.4: Patching the mov-RS1 leakage in bbpp-opt3-0-sh0-sh1

```
1
2
3  # r0 = b1
4  # r1 = -a1 >> 32
5  # r2 = &rnd
6  # r3 = (a1*b0)
7  # r4 = 32
8  # r5 = -tmp >> 32
9  # r6 = a1
10 # r7 = (a0*b1)
11
12   eors r4, r1          @ r4 = (a1*b1)+rnd1
13   eors r1, r1          @ r1 = 0, removing rnd1, updating rs1 as it is not updated in mov
14   eors r7, r4          @ r7 = (a1*b1)+rnd1+(a0*b1)
15   eors r4, r4          @ r4 = 0, removing rnd1, updating rs1 as it is not updated in mov
16   eors r3, r7          @ r3 = c1 = (a1*b1)+rnd1+(a0*b1)+(a1*b0)
17   mov  r1, r9          @ r1 = &a , here for avoiding stall
18   mov  r4, r11         @ r4 = &table , here for avoiding stall
19   mov  r7, r10         @ r7 = &b , here for avoiding stall
20
21
```

2) The leakage is because of Glitchy leakages (**Glitchy register access**):
Leakage in cycles and are related to instructions in Lines 405 and 414 in Listing 9.1 (listed in Listing 9.5).
Regarding Listing 9.5, during the execution of ands r3, r5 instruction, the mov r5, r8 instruction (which was encoded to 0x4645: 0100011001000101) is decoded in the Decoder stage of the pipeline. The 0x4645 (mov) is decoded at first in the ands style (9.5 *a*), then it is decoded according to the mov format (9.5 *b*), so that a glitch will occur here.
From ARM Architecture:

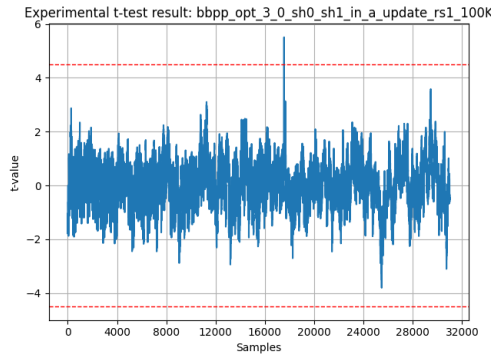


Figure 9.4: t-test_bbpp_opt_3_0_sh0_sh1_patching the first leakage (Setup: 250MSa/s, 31KS, 4ns, 2MHz)

(a) Encoding ANDS instruction

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

`ANDS <Rdn>, <Rm>`	Outside IT block.
`AND<c> <Rdn>, <Rm>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

(b) Encoding MOV (reg) instruction

Encoding T1 ARMv6*, ARMv7 if <Rd> and <Rm> both from R0-R7
ARMv4T, ARMv5T*, ARMv6*, ARMv7 otherwise

`MOV<c> <Rd>, <Rm>`	If <Rd> is the PC, must be outside or last in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Figure 9.5: Encoding format of ANDS and MOV (reg) instruction (ARM Architecture)

Assuming for INSRT Rd, Rm, Rd and Rm are loaded into $port1$ and $port2$, respectively. Regarding it, in the decoding stage of ands r3, r5, $port2$ is loaded with $r5$ ($r5 = -tmp >> 32$), now, when mov r5, r8 is decoded in the ands format, the content of the register with the address bits 5-3 (0x4645: 0100011001000101, which is 000 = 000 and means register $r0$) is loaded into $port2$.
$port1 = r3(a_1.b_0) \rightarrow r5 = 01$
$port2 = r5(01) \rightarrow r0 = b1$
$port1 \otimes port2 = r3 \otimes r0 = (a_1.b_0) \otimes b1$
However, this is not same with what uELMO reported
Cycle 159: Instr=Execute: ands r3,r5 Decode: mov r5,r8 Memory: idle , Term=Glitchy decoding port 1 XOR previous port 1

In order to avoid this leakage, we can add an extra instruction like eors r4, r4 between the two instructions that the two instructions ands and mov be not consecutive (Listing9.6).

Listing 9.5: Source of the first leakage in bbpp-opt-3-0-sh0-sh1

```
1
2
3  # r0 = b1, # r1 = -a1 >> 32, # r2 = &rnd, # r3 = (a1*b0)
4  # r4 = 32, # r5 = -tmp >> 32, # r6 = a1, # r7 = (a0*b1)
5
6    ands r3, r5          @ r3 = s & (-tmp >> 32)  = (a1*b0)
7    mov  r5, r8          @ r5 = &c
8
9
```

Listing 9.6: Patching the first leakage in bbpp-0-sh0-sh1

```
1
2
3  # r0 = b1, # r1 = -a1 >> 32, # r2 = &rnd, # r3 = (a1*b0)
4  # r4 = 32, # r5 = -tmp >> 32, # r6 = a1, # r7 = (a0*b1)
5
6    ands r3, r5          @ r3 = s & (-tmp >> 32)  = (a1*b0)
7
8  eors r4, r4     @ r4 = 0  @ add to remove the leakage, r4 is not needed
9
10   mov  r5, r8          @ r5 = &c
11
```

2) (sh_0, sh_2)
Listing 9.7 shows the the leakages found by **uEMO**.
in real traces (100K traces):
bbpp_opt_3_0_sh0_sh2_in_a:
Leaky cycles: []
Leaky cycles (from trigger): []
bbpp_opt_3_0_sh0_sh2_in_b:
Leaky cycles: [85, 134, 135]
Leaky cycles (from trigger): [113, 114, 163]
bbpp_opt_3_0_sh0_sh2:
Leaky cycles: [85, 89, 134, 135, 136, 137]
Leaky cycles (from trigger): [111, 112, 113, 114, 159, 163]

Listing 9.7: Leakages reported by **uELMO** for bbpp-opt-3-0-sh0-sh2

```
Cycle 120: Instr=Execute:eors r0,r7
Decode: mov r7,r10
Memory: idle
, Term=Pipeline Reg 1 HD, Tvalue=inf
```

The leakage related to the instructions listed in Listing 9.8 and is because of HD in RS1 , as is shown in Table 9.3, in pipeline register RS1:
$RS1 : a_0.b_0 \rightarrow a_0.b_2 \rightarrow a_2.b_0$

Listing 9.8: Source of the first leakage in bbpp-opt-3-0-sh0-sh2

```

# r0 = b1, r1 = -a1 >> 32, r2 = &rnd, r3 = (a1*b0),
# r4 =32, r5 = -tmp >> 32, r6 = a1, r7 = (a0*b1),
# r8 = &c, r9 = &a, r10 = &b, r11 = &table
  mov  r5, r8          @ r5 = &c
  ldrb r1, [r2, #4]    @ r1 = rnd1 and also clearing the read bus
  ldrb r4, [r5, #4]    @ r4 = (a1*b1)
  strb r5, [r5, #4]    @ r5 = &c, clearing the write bus which contains (a0*b0)

  eors r4, r1          @ r4 = (a1*b1)^rnd1
  mov  r1, r9          @ r1 = &a , here for avoiding stall
  eors r7, r4          @ r7 = (a1*b1)^rnd1^(a0*b1)
  mov  r4, r11         @ r4 = &table , here for avoiding stall
  eors r3, r7          @ r3 = c1 = (a1*b1)^rnd1^(a0*b1)^(a1*b0)
  mov  r7, r10         @ r7 = &b , here for avoiding stall

  strb r3, [r5, #4]    @ r3 = c1, Storing c1, r3 = c1

```

It can be concluded if the power consumption of this point combined with the power consumption

of $a_0.b_2$ (in pre-processing step), then will b leak.
In Table 9.3, for summary we put:
$r7 = ((a_0.b_0) \oplus rnd_0) \oplus (a_0.b_2) = b_0 \oplus rnd_0 \oplus b_2$

Table 9.3: Leakage stems from eors r0, r7

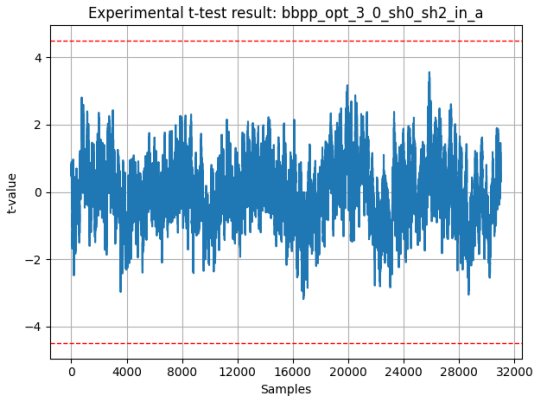
Instruction		Decoding			Executing	
		Port 1	Port 2	Port 3	RS1	RS2
1	eors r4, r1	$r4 = a_0.b_0$	$r1 = rnd_0$	-	$r4 = a_0.b_0$	$r1 = rnd_0$
2	mov r1, r9	$r1 = rnd_0$	$r9 = ADDa$	-	-	$r11 = ADDa$
3	eors r7, r4	$r7 = a_0.b_2$	$r4 = (a_0.b_0) \oplus rnd_0$	-	$r7 = a_0.b_2$	$r4 = (a_0.b_0) \oplus rnd_0$
4	mov r4, r11	$r4 = (a_0.b_0) \oplus rnd_0$	$r11 = ADDtable$	-	-	$r11 = ADDtable$
5	eors r0, r7	$r0 = a_2.b_0$	$r7 = b_0 \oplus rnd_0 \oplus b_2$	-	$r0 = a_2.b_0$	$r7 = b_0 \oplus rnd_0 \oplus b_2$

In order to avoid this leakage, we can add an extra instruction like eors r1, r1 between and also reorder the instructions (Listing9.9).

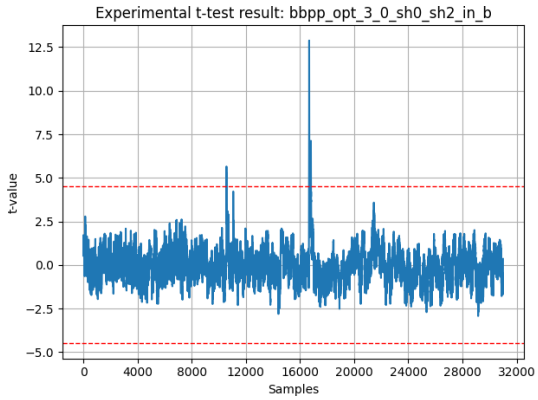
Listing 9.9: Patching the first leakage in bbppopt-3-0-sh0-sh2

```


  eors r4, r1          @ r4 = (a0*b0)^rnd0

eors r1, r1            @ Adding extra instruction, r1 = 0

  eors r7, r4          @ r7 = (a0*b0)^rnd0^(a0*b2)

eors r1, r1            @ Adding extra instruction, r1 = 0

  eors r0, r7          @ r0 = c0 = (a0*b0)^rnd0^(a0*b2)^(a2*b0)
  mov  r7, r10         @ r7 = &b , here for avoiding stall

mov  r4, r11         @ r4 = &table , here for avoiding stall
mov  r1, r9          @ r1 = &a , here for avoiding stall

  strb r0, [r6, #0]    @ r0 = c0, Storing c0, r0 = c0

```

(a) Shares-a



(b) Shares-b



(c) Shares-a and Shares-b

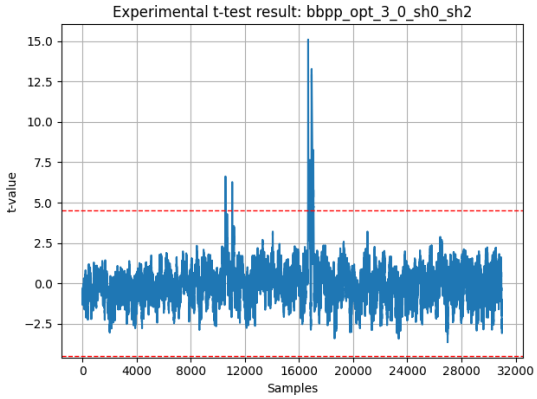


Figure 9.6: t-test bbpp-opt-3-without-b-0-sh0-sh2 (Setup: 250MSa/s, 31KS, 4ns, 2MHz)

3) (sh_1, sh_2)

Listing 9.10 shows the the leakages found by **uEMO**.

in real traces (100K traces):

bbpp_opt_3_0_sh1_sh2_in_a:

Leaky cycles: [193]

Leaky cycles (from trigger): [55]

bbpp_opt_3_0_sh1_sh2_in_b:

Leaky cycles: [205]

Leaky cycles (from trigger): [43]

bbpp_opt_3_0_sh1_sh2:

Leaky cycles: [193, 194, 205, 206, 234, 235, 236]

Leaky cycles (from trigger): [12, 13, 14, 42, 43, 54, 55]

Listing 9.10: Leakages reported by **uELMO** for bbpp-opt-3-0-sh1-sh2

```
Cycle 212: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x20001F9C]=0x4D
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
Cycle 213: Instr=Execute: strb r6,[r0,#0x8]
Decode:eors r4,r1
Memory: idle
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
Cycle 214: Instr=Execute:eors r4,r1
Decode:eors r4,r3
Memory: store [0x20001FA8]=0x20
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
Cycle 215: Instr=Execute:eors r4,r3
Decode:eors r7,r4
Memory: idle
, Term=Decoding port 0 HD, Tvalue=inf
Cycle 215: Instr=Execute:eors r4,r3
Decode:eors r7,r4
Memory: idle
, Term=Glitchy decoding port 0 XOR previous port 0, Tvalue=inf
Cycle 216: Instr=Execute:eors r7,r4
Decode:eors r5,r7
Memory: idle
, Term=Decoding port 0 HD, Tvalue=inf
Cycle 216: Instr=Execute:eors r7,r4
Decode:eors r5,r7
Memory: idle
, Term=Glitchy decoding port 0 XOR previous port 0, Tvalue=inf
Cycle 217: Instr=Execute:eors r5,r7
```

```
Decode:strb r5,[r0,#0x8]
Memory: idle
, Term=Pipeline Reg 1 HD, Tvalue=inf
Cycle 218: Instr=Execute: strb r5,[r0,#0x8]
Decode: mov r8,r8
Memory: idle
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
Cycle 219: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: store [0x20001FA8]=0xE1
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
Cycle 220: Instr=Execute: mov r8,r8
Decode:ldr r4,[PC+#0x6]
Memory: idle
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
Cycle 221: Instr=Execute: ldr r4,[PC+#0x18]
Decode: movs r5,#0x00
Memory: idle
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
Cycle 221: Instr=Execute: ldr r4,[PC+#0x18]
Decode: movs r5,#0x00
Memory: idle
, Term=Decoding port 0 HD, Tvalue=inf
Cycle 221: Instr=Execute: ldr r4,[PC+#0x18]
Decode: movs r5,#0x00
Memory: idle
, Term=Glitchy decoding port 0 XOR previous port 0, Tvalue=inf
Cycle 222: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x8000304]=0xE0000004
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
```

The sources of the leakages are shown in Listing 9.11.
There are 3 kinds of leakages:
1) Neighbouring Reg 4 XOR Reg 5
2) Decoding port 0 HD
3) Pipeline Reg 1 HD
Decoding port 0 HD and Pipeline Reg 1 HD leakages are explained in Table 9.4.
In Table 9.4 for convenience: $rnd_0 : r_0, rnd_1 : r_1$.
Also with FW: forward: paper PROLEAD-SW,: Pipeline Forwarding Effect (PFE).To reduce pipeline stalls in case of data dependenciesforwarding logic allows to access values from later pipeline stages without waiting for thevalues to be written to the destination registers. Gigerl et al. [GPM21] demonstrate on theRISC-V SweRV core that such logic is able to pose a severe security

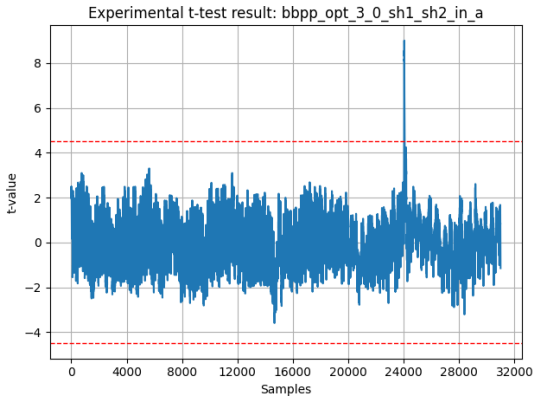
Listing 9.11: Source of leakages in bbpp-opt-3-0-sh1-sh2

```
@@@@ r0 = &c, r1 = -a2 >> 32, r2 = &rnd, r3 = -tmp >> 32,
@@@@ r4 = &table, r5 = (a2*b1), r6 = 32, r7 = (a1*b2),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

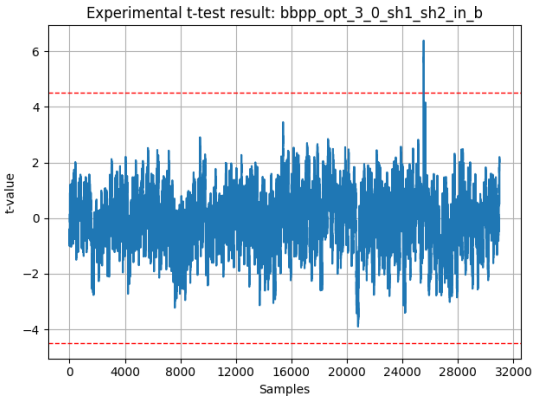
Listing 9.12: Patching the first leakage in bbppopt-3-0-sh1-sh2

```


@ @ ##############################################################################
# swapping r3 and r4
  ldrb r1, [r2, #0]   @ r1 = rnd0 and also clearing the read bus
  ldrb r3, [r0, #8]   @ r3 = (a2*b2)
  ldrb r4, [r2, #4]   @ r4 = rnd1 and also clearing the read bus
  strb r6, [r0, #8]   @ r6 = 32, clearing the write bus which contains (a2*b2)

  eors r3, r1         @ r3 = (a2*b2)^rnd0
eors r6, r6
  eors r3, r4         @ r3 = (a2*b2)^rnd0^rnd1
eors r6, r6
  eors r7, r3         @ r7 = (a2*b2)^rnd0^rnd1^(a1*b2)
eors r6, r6
  eors r5, r7         @ r5 = c2 = (a2*b2)^rnd0^rnd1^(a1*b2)^(a2*b1)
  strb r5, [r0, #8]   @ r5 = c2, Storing c2, r5 = c2

```

(a) Shares-a



(b) Shares-b



(c) Shares-a and Shares-b

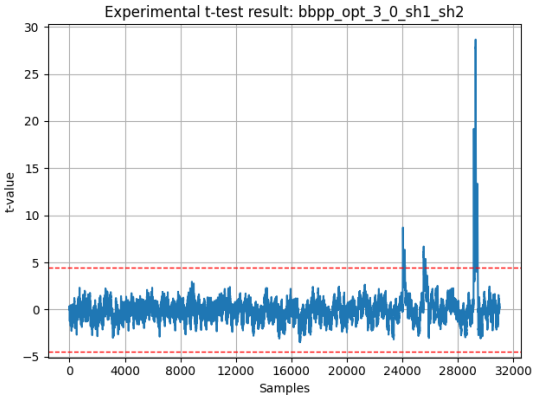
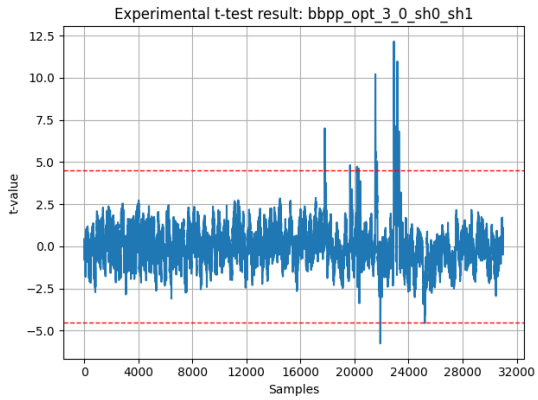


Figure 9.7: t-test bbpp-opt-3-without-b-0-sh1-sh2 (Setup: 250MSa/s, 31KS, 4ns, 2MHz)

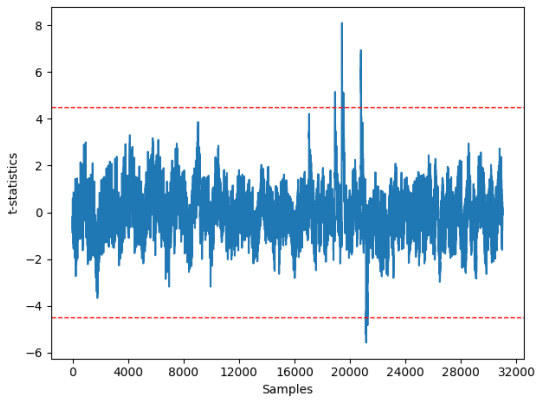
WORK TO DO: We have to remove leakages from V1 and put the results also.

Figures 9.8, 9.9, and 9.10 show the differences between bbpp-opt-3 for V0 and V1 on real traces. V1 is the implementation that all leakages found by uELMO in V0 are removed. From these pictures one can see that despite uELMO does not find any leakage on bbpp-opt-3-V1, but on real traces there are several leakages.

(a) bbpp-opt-3-V0-sh0-sh1-100K



(b) bbpp-opt-3-V1-sh0-sh1-100K



(c) bbpp-opt-3-V1-sh0-sh1-200K

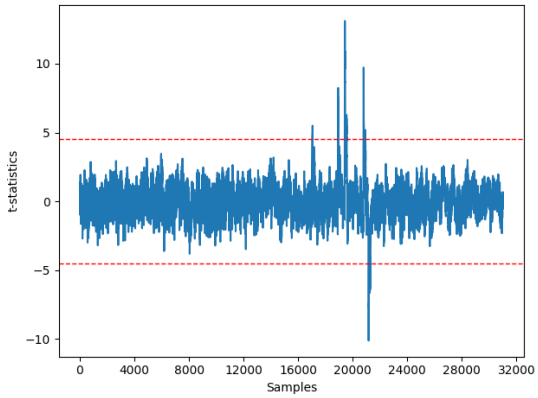
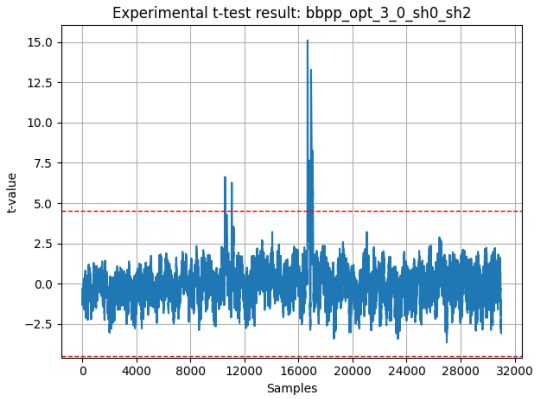
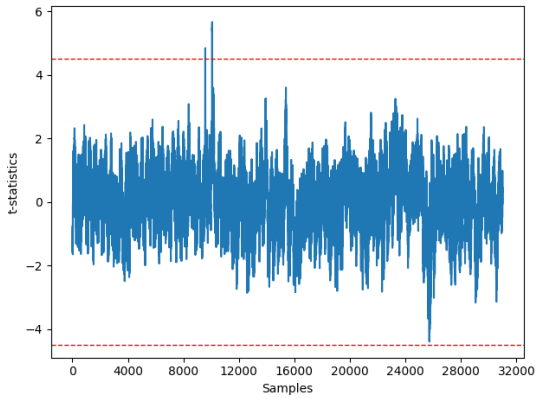


Figure 9.8: t-test bbpp-opt-3-V0-V1-sh0-sh1 (Setup: 250MSa/s, 31KS, 4ns, 2MHz)

(a) bbpp-opt-3-V0-sh0-sh2-100K



(b) bbpp-opt-3-V1-sh0-sh2-100K



(c) bbpp-opt-3-V1-sh0-sh2-200K

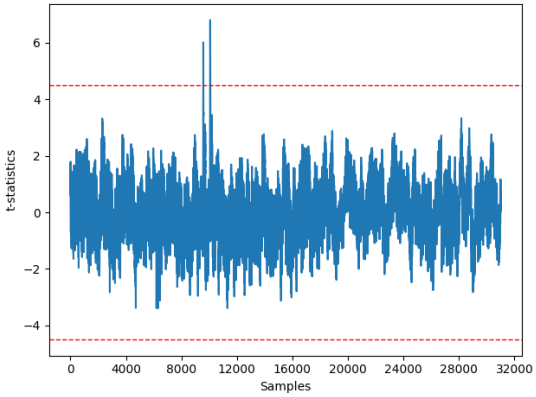
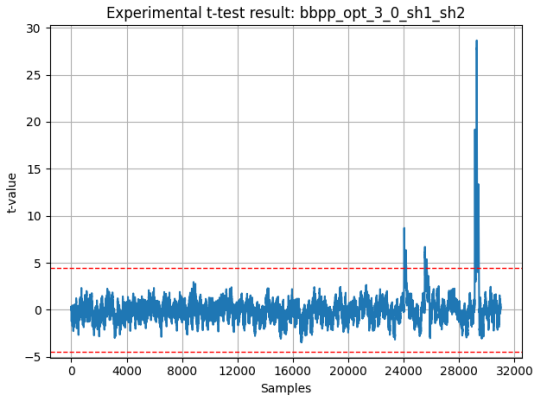
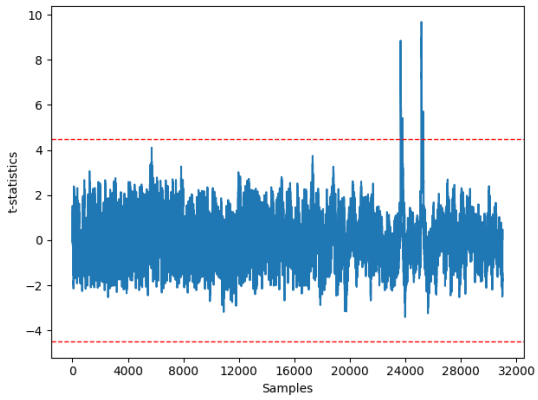


Figure 9.9: t-test bbpp-opt-3-V0-V1-sh0-sh2 (Setup: 250MSa/s, 31KS, 4ns, 2MHz)

(a) bbpp-opt-3-V0-sh1-sh2-100K



(b) bbpp-opt-3-V1-sh1-sh2-100K



(c) bbpp-opt-3-V1-sh1-sh2-200K

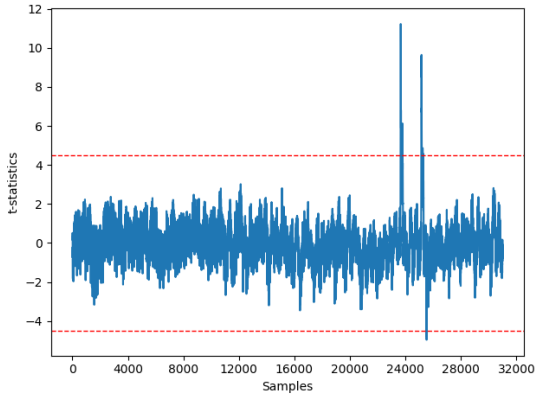


Figure 9.10: t-test bbpp-opt-3-V0-V1-sh1-sh2 (Setup: 250MSa/s, 31KS, 4ns, 2MHz)

10. DOM-indep

DOM-indep multiplication in [8, 9]. Equation 10.1 (Equation 6 in [9] represents this scheme for $0 <= i, j <= d$ ($d + 1$ shares).

$$c_i = (a_i.b_i) \oplus \sum_{j>i}^{d}(a_i.b_j) \oplus rnd_{(i+j(j-1)/2)} \oplus \sum_{j<i}^{d}(a_i.b_j) \oplus rnd_{(j+i(i-1)/2)} \tag{10.1}$$

Algorithm 8 DOM-independent multiplication gadget

Require: shares a_i, b_i $(0 \leq i \leq d)$ such as $a = \oplus_{i=0}^{d} a_i$, $b = \oplus_{i=0}^{d} b_i$
Ensure: $c_0, ..., c_d$ such that $c = \oplus_{i=0}^{d} c_i = a \cdot b$

```
for i = 0 to d do
    u_ii ← (a_i · b_i)
    for j = i + 1 to d do
        r_(i+j(j−1)/2) ←$ {0,1}^n
        u_ij ← (a_i · b_j) ⊕ r_(i+j(j−1)/2)
        u_ji ← (a_j · b_i) ⊕ r_(i+j(j−1)/2)
    end for
end for
for i = 0 to d do
    c_i ← u_i0
    for j = 1 to d do
        c_i ← c_i ⊕ u_ij
    end for
end for
return c_0, ..., c_d
```

11. DOM-indep-2- no function/label call

The DOM-indep multiplication for two shares (DOM-indep-2) is illustrated in Algorithm 9 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 9 DOM-indep-2

Require: shares a_0, a_1, b_0, b_1, rnd

Ensure: c_0, c_1

1: $c_0 \leftarrow (a_0.b_0) \oplus ((a_0.b_1) \oplus rnd)$

2: $c_1 \leftarrow ((a_1.b_0) \oplus rnd) \oplus (a_1.b_1)$

In the implementation of DOM-indep-2, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

11.1 DOM-indep-2-no-function-call-0: basic code

Listing 11.1 demonstrates the first implementation of DOM-indep-2 based gfmul-s (DOM-indep-2-no-function-call-0). The instructions in Listing 11.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 11.1: DOM-indep-2-no-function-call-0

```
 @@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @ r8 = &c


@@@@ Saving the value 256 in r11. This decreases one cycle.
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r4, #250       @ r4 = 250, 256 is not possible, so it is split: 250 + 6
  adds r4, #6         @ r4 = 256
  mov  r11, r4        @ r11 = 256


  ldr  r3, =table    @ r3 = &table, The address of the "table", for ai * bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r9, r3        @ r9 = &table

nop
nop
nop
nop


@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
@@@@ r8 = &c, r9 = &table, r11 = 256
@@@@@
```

```
@@@@ a0 * b0

  ldrb r0, [r3, r6]   @ r0 = table[a0]
  ldrb r1, [r3, r5]   @ r1 = table[b0]

  adds r0, r1         @ r0 = s = table[a0] + table[b0]

  @@@@ Get the antilog: s = table[s+256]: g + s
  mov  r1, r11        @ r1 = 256

  adds r0, r1         @ r0 = s + 256

  ldrb r1, [r3, r0]   @ r1 = s = table[s+256]

  # @@@@ Checking if a0 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r6         @ r0 = -a0 hex(256-dec(a0))

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r3, #32        @ r3 = #32
  asrs r0, r3         @ r0 = -a0 >> 32

  ands r0, r5         @ r0 = tmp = b0 & (-a0 >> 32)
  negs r0, r0         @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r3         @ r0 = -tmp >> 32

  ands r1, r0         @ r1 = s & (-tmp >> 32) = (a0 * b0)
  mov  r10, r1        @ r10 = a0 * b0


@@@@ r0 = -tmp >> 32, r1 = a0 * b0, r2 = rnd, r3 = 32
@@@@ r4 = a1, r5 = b0, r6 = a0, r7 = b1,
@@@@ r8 = &c, r9 = &table, r10 = a0 * b0, r11 = 256
@@@@
```

```
86
87    @@@@ Get the antilog: s = table[s+256]: g + s
88    mov  r3, r11        @ r3 = 256
89
90    adds r0, r3         @ r0 = s + 256
91
92    ldrb r3, [r1, r0]   @ r3 = s = table[s+256]
93
94    # @@@@ Checking if a0 = 0 or b1 = 0, return 0; without conditional branch
95    negs r0, r6         @ r0 = -a0 hex(256-dec(a0))
96
97    # @ asrs rd, <#imm> --> #imm: 0-31
98    @ a0 in r6 is not needed anymore, r6 can be used
99    movs r6, #32        @ r6 = #32
100   asrs r0, r6         @ r0 = -a0 >> 32
101
102   ands r0, r7         @ r0 = tmp = b1 & (-a0 >> 32)
103   negs r0, r0         @ r0 = -tmp
104
105   # @ asrs rd, <#imm> --> #imm: 0-31
106   asrs r0, r6         @ r0 = -tmp >> 32
107
108   ands r0, r3         @ r0 = s & (-tmp >> 32) = (a0 * b1)
109
110   @@@@ (a0 * b1) + rnd
111
112   eors r0, r2         @ r0 = (a0 * b1) + rnd
113
114
115   @@@@ (a0 * b0) + [(a0 * b1) + rnd0]
116   mov  r3, r10        @ r3 = a0 * b0
117
118   # mov  r10, r8        @ r10 = &c  @ deleting a0 * b0
119   eors r0, r3         @ r0 = c0 = (a0 * b0) + [(a0 * b1) + rnd0]
120   mov  r3, r8         @ r3 = &c
121   strb r0, [r3, #0]   @ Storing c0, r0 = c0
122
123 @@@@@@@@@ Computing c1 = [(a1 * b0) + rnd] + (a1 * b1)
124
125 @@@@ r0 = c0, r1 = &table, r2 = rnd, r3 = &c
126 @@@@ r4 = a1, r5 = b0, r6 = 32, r7 = b1,
127 @@@@ r8 = &c, r9 = &table, r10 = &c, r11 = 256
128 @@@@@@@@@
```

```
@@@@ a1 * b0

  ldrb r0, [r1, r4]   @ r0 = table[a1]
  ldrb r3, [r1, r5]   @ r3 = table[b0]

  adds r0, r3         @ r0 = s = table[a1] + table[b0]

  @@@@ Get the antilog: s = table[s+256]: g + s
  mov  r3, r11        @ r3 = 256

  adds r0, r3         @ r0 = s + 256

  ldrb r3, [r1, r0]   @ r3 = s = table[s+256]

  # @@@@ Checking if a1 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r4         @ r0 = -a1 hex(256-dec(a1))

  # @ asrs rd, <#imm> --> #imm: 0-31
  # movs r6, #32        @ r6 = #32
  asrs r0, r6         @ r0 = -a1 >> 32

@ b0 in r5 is not neede any more, r5 can be used
  ands r5, r0         @ r5 = tmp = b0 & (-a1 >> 32)
  negs r5, r5         @ r5 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r5, r6         @ r5 = -tmp >> 32

  ands r5, r3         @ r5 = s & (-tmp >> 32) = (a1 * b0)


  @@@@ (a1 * b0) + rnd
  @ rnd in r2 is not needed any more, r2 can be used

  eors r2, r5         @ r2 = (a1 * b0) + rnd


@@@@ r0 = -a1 >> 32, r1 = &table, r2 = (a1 * b0) + rnd, r3 = table[s+256]
@@@@ r4 = a1, r5 = (a1 * b0), r6 = 32, r7 = b1,
@@@@ r8 = &c, r9 = &table, r10 = &c, r11 = 256
@@@@
```

```
@@@@ a1 * b1

  ldrb r6, [r1, r4]   @ r6 = table[a1]
  ldrb r3, [r1, r7]   @ r3 = table[b1]

  adds r6, r3         @ r6 = s = table[a1] + table[b1]

  @@@@ Get the antilog: s = table[s+256]: g + s
  mov  r0, r11        @ r0 = 256

  adds r6, r0         @ r6 = s + 256

  ldrb r3, [r1, r6]   @ r3 = s = table[s+256]

  # @@@@ Checking if a1 = 0 or b1 = 0, return 0; without conditional branch
  negs r6, r4         @ r6 = -a1 hex(256-dec(a1))

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r5, #32        @ r5 = #32
  asrs r6, r5         @ r6 = -a1 >> 32


  ands r6, r7         @ r6 = tmp = b1 & (-a1 >> 32)
  negs r6, r6         @ r6 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r6, r5         @ r6 = -tmp >> 32

  ands r3, r6         @ r3 = s & (-tmp >> 32) = (a1 * b1)


  @@@@ c1 = [(a1 * b0) + rnd] + (a1 * b1)

  eors r2, r3         @ r2 =  [((a0 * b1) + rnd) + (a1 * b0)] + (a1 * b1)
  mov  r3, r8         @ r3 = &c

  strb r2, [r3, #4]   @ Storing c1, r2 = c1


  nop
  nop
  nop
  nop
```

11.1.1 Experimental result:

11.1.1.1 Setup:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $81.5\mu s$
Each clock is sampled with 125 points
Number of samples = 120×125 = 15000 samples
The bytes of shares (a, b, c are not in a word)
Figure 11.1 shows the recorded 15KS before raising-edge of the trigger.

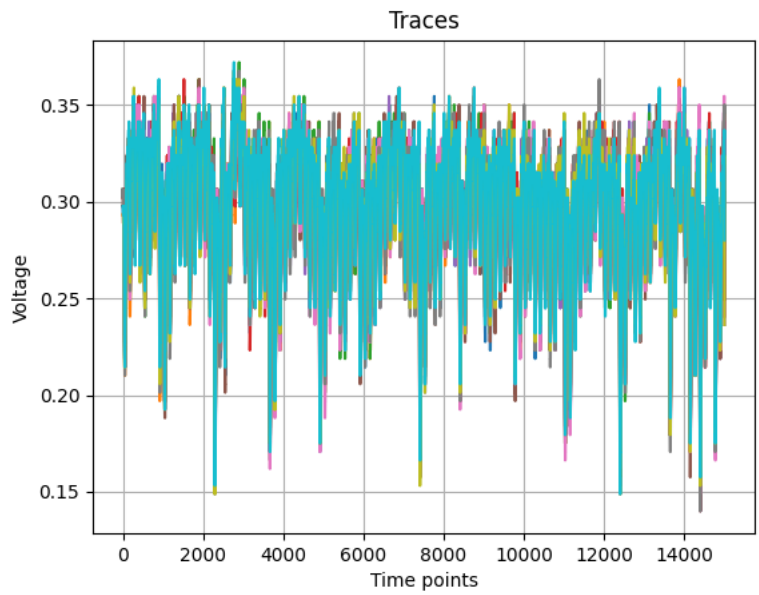


Figure 11.1: Traces dom-indep-2-without-b-0 (Setup: 250MSa/s, 15kS, 2ns, 2MHz)

11.1.1.2 Signal to Noise Ratio (SNR):

Figure 11.2 shows the SNR for all the intermediate values. It can be concluded that the first 3000 (24×125) and also the last 1000 (8×125) samples do not have effect on the SNR, so, they can be removed.
Number of samples = $(120 - 24 - 8) \times 125 = 88 \times 125$ = 11000 samples

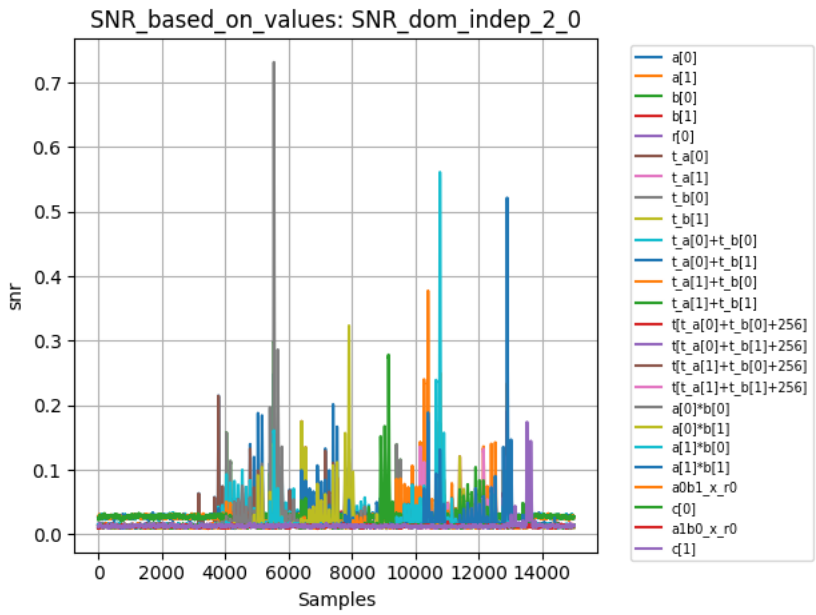


Figure 11.2: SNR dom-indep-2-without-b-0 (Setup: 250MSa/s, 15kS, 2ns, 2MHz)

Figure 11.3 shows the SNR of traces with 11K samples.

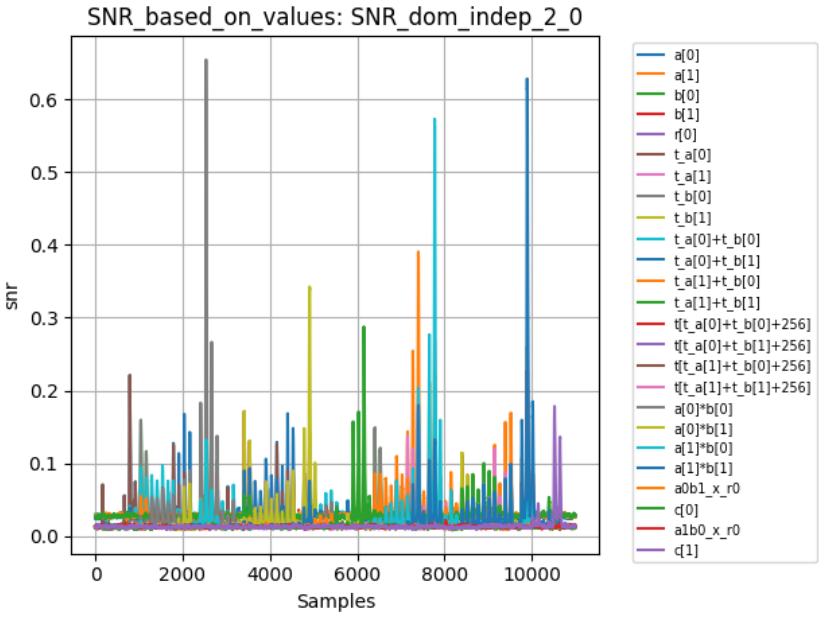


Figure 11.3: SNR dom-indep-2-without-b-0 (Setup: 250MSa/s, 11KS, 2ns, 2MHz)

11.1.1.3 t-test:

Figure 11.5 shows the fixed-versus-random t-test result on both inputs a and b for 50k traces. The leakages occur at cycles: 16, 17, 35, 36, 43, 44, 45, 86 and 87.
61, 99, 116, 121, 208 in DOM-indep-2-no-function-call-0.

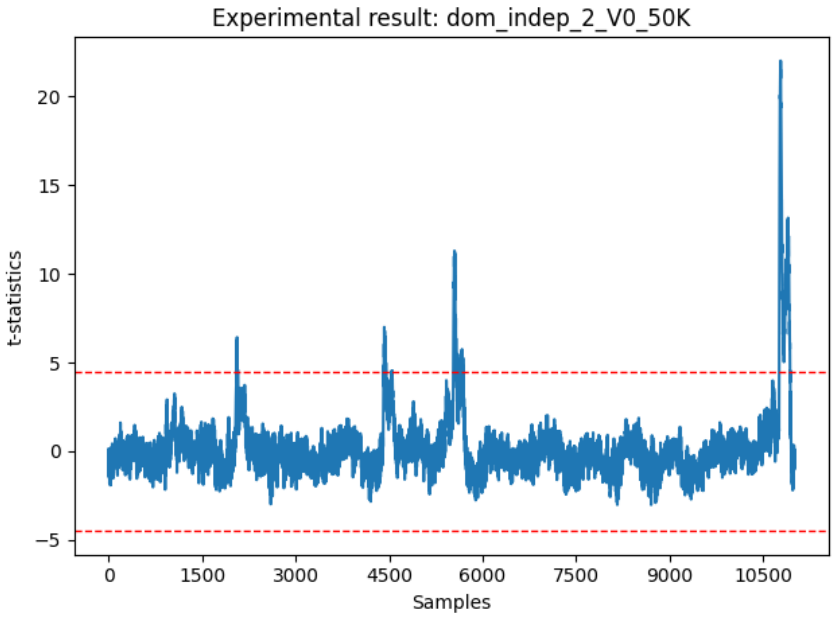
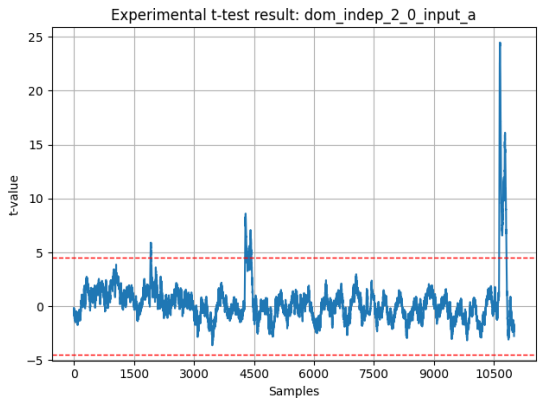
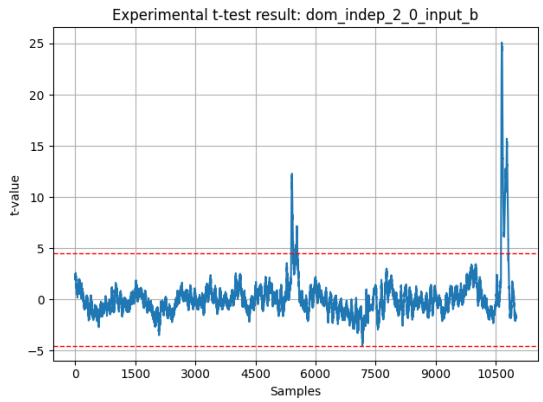


Figure 11.4: ttest dom-indep-2-V0 (traces exist)(Setup: 250MSa/s, 11KS, 2ns, 2MHz)

(a) t-test on input a



(b) t-test on input b



(c) t-test on both inputs a and b

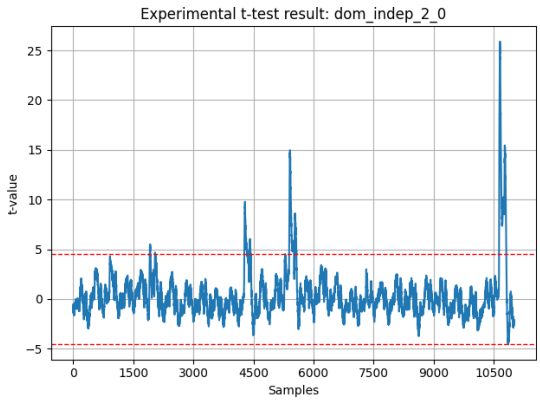


Figure 11.5: t-test dom-indep-2-0-without-b-0-(Setup: 250MSa/s, 11KS, 2ns, 2MHz)

Also, we used the leakage simulator uELMO, which reportes the same leakages (Listing 11.2).

Listing 11.2: Leakages reported by **uELMO** for dom-indep-2-0

```
Cycle 24: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x200001F2]=0x85
, Term=Glitchy decoding port 1 XOR current port 1, Tvalue=inf
Cycle 24: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x200001F2]=0x85
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 25: Instr=Execute: negs r0,r6
Decode: movs r3,#0x20
Memory: idle
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 41: Instr=Execute: stall by LDR
Decode: stall
```

```
Memory: load [0x200001F2]=0x85
, Term=Glitchy decoding port 1 XOR current port 1, Tvalue=inf
Cycle 41: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x200001F2]=0x85
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 42: Instr=Execute: negs r0,r6
Decode: movs r6,#0x20
Memory: idle
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 50: Instr=Execute: mov r3,r10
Decode:eors r0,r3
Memory: idle
, Term=Decoding port 1 HD, Tvalue=inf
Cycle 51: Instr=Execute:eors r0,r3
Decode: mov r3,r8
Memory: idle
, Term=Reg 3 HD, Tvalue=inf
Cycle 87: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: store [0x20001FBC]=0x6B
, Term=Memory write buffer HD, Tvalue=inf
Cycle 88: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: idle
, Term=Memory write buffer delayed HD, Tvalue=inf
```

Reasons and solutions:

1) Glitchy leakages: Lines 61, 99. As it was mentioned in: **Glitchy register access**.

Figure 11.6 shows the result of applying one of its solutions $replacing r3 and r4$ for 50K traces.

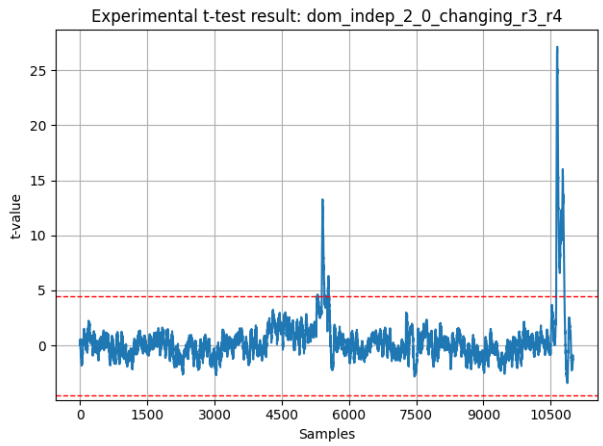


Figure 11.6: Experimental T-test result of dom-indep-2-0-without-b-0 - patch: changing r3 and r4 (Setup: 250MSa/s, 11KS, 2ns, 2MHz)

After changing $r3$ and $r4$, uELMO reportes (Listing 11.4, Lines 9 and 13) about Neighbouring effects [11] on Lines 187, 190 and 192 in DOM-indep-2-no-function-call-0. After replacing the contents of $r3$ and $r4$ with each other, these lines are as it is shown in Listing 11.3..

As it is written in Listing 11.3., registers $r4$ and $r5$ which contain $table[table[a1]+table[b1]+256]$ and $(a1 * b0)$, respectively, so that it can relive information about b. In order to avoid this leakage, one possible way is removing the context of $r5$ before using it in movs r5, #32. Putting eors r5, r5 after eors r2, r5 (Line 164) is an option.

Listing 11.3: New lines after changing r3 and r4

```
1
2   # r3 = a1 (before: r3 = table[table[a1] + table[b1] + 256])
3   # r4 = table[table[a1] + table[b1] + 256] (before: r4 = a1)
4   # r5 = (a1*b0)
5   # r6 = table[a1] + table[b1] + 256
6
7   negs r6, r3          @ r6 = -a1 hex(256-dec(a1))
8
9   # @ asrs rd, <#imm> --> #imm: 0-31
10  movs r5, #32         @ r5 = #32
11  asrs r6, r5          @ r6 = -a1 >> 32
12
```

Listing 11.4: Leakages reported by **uELMO** for dom-indep-2-0_changing_r3_r4

```
Cycle 50: Instr=Execute: mov r4,r10
Decode:eors r0,r4
Memory: idle
, Term=Decoding port 1 HD, Tvalue=inf
Cycle 51: Instr=Execute:eors r0,r4
Decode: mov r4,r8
Memory: idle
, Term=Reg 4 HD, Tvalue=inf
Cycle 77: Instr=Execute: negs r6,r3
Decode: movs r5,#0x20
Memory: idle
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
Cycle 78: Instr=Execute: movs r5,#0x20
Decode: asrs r6,r5
Memory: idle
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=inf
Cycle 87: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: store [0x20001FBC]=0xA3
, Term=Memory write buffer HD, Tvalue=inf
Cycle 88: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: idle
, Term=Memory write buffer delayed HD, Tvalue=inf
```

Listing 11.5: Leakages reported by **uELMO** for dom-indep-2-0_changing_r3_r4_and_clearing_r5

```
Cycle 50: Instr=Execute: mov r4,r10
Decode:eors r0,r4
Memory: idle
, Term=Decoding port 1 HD, Tvalue=inf
Cycle 51: Instr=Execute:eors r0,r4
Decode: mov r4,r8
Memory: idle
, Term=Reg 4 HD, Tvalue=inf
Cycle 88: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: store [0x20001FBC]=0x5F
, Term=Memory write buffer HD, Tvalue=inf
Cycle 89: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: idle
, Term=Memory write buffer delayed HD, Tvalue=inf
```

2) The write bus (str instruction): Lines 121 and 208 in Listing 11.1 made this type of leakage. Solution is clearing the write bus after storing $c0$ (Listing 11.6). Figure 11.7 shows the result of applying this patch for 50K traces.

Listing 11.6: Patching the write bus leakage in dom-indep-2-without-b-0 (storing shares of c)

```
strb r0, [r3, #0]   @ Storing c0, r0 = c0

  A sequence of instructions

  strb r3, [r3, #4]   @ Clearing the write bus, r1 = &c

strb r2, [r3, #4]   @ Storing c1, r2 = c1

```

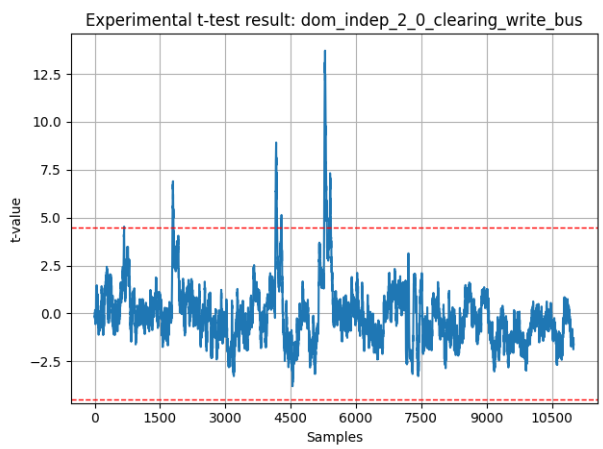


Figure 11.7: Experimental T-test result of dom-indep-2-0-without-b-0 - patching the write bus (Setup: 250MSa/s, 11KS, 2ns, 2MHz)

3) Register HD leakage between the previous value of a register and and its new written value. Rewriting register causes a leakage in Line 116. As, $r3 = table[table[a0] + table[b1] + 256]$, and then with movs r3, r10, $r3$ becomes $(a0 \times b0)$, so that, b will leak.
For removing this leakage, the previous content of $r3$ has to be cleared. Thus, after computing $a0 \times b0$ in ands r0, r3 (Line 108), putting eors r3, r3 will clean the $r3$. Figure 11.8 shows the result of applying the mentioned patch for 100K traces.

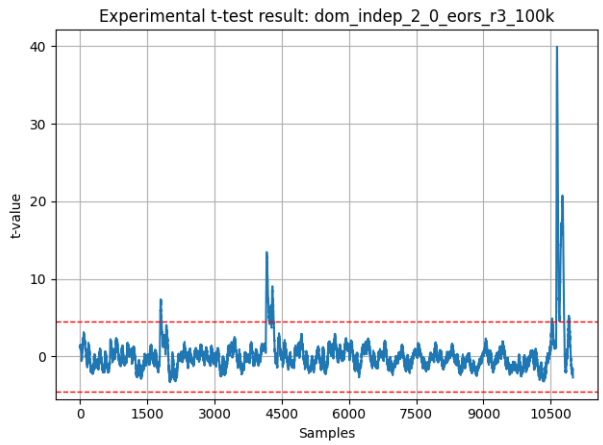


Figure 11.8: Experimental T-test result of dom-indep-2-0-without-b-0 - patching the register HD (Setup: 250MSa/s, 11KS, 2ns, 2MHz)

Less leaky implementation of dom-indep-2:
By applying the below patches:
1) Replacing the contents of $r3$ and $r4$ with each other
2) Clearing the write bus
3) Removing the Register HD
4) Removing the Neighboring effects
With this implementation, uELMO, reportes no leakages.
Figures 11.9, 11.11 and 11.12 show the t-test result for 50K, 100K, 500K traces.

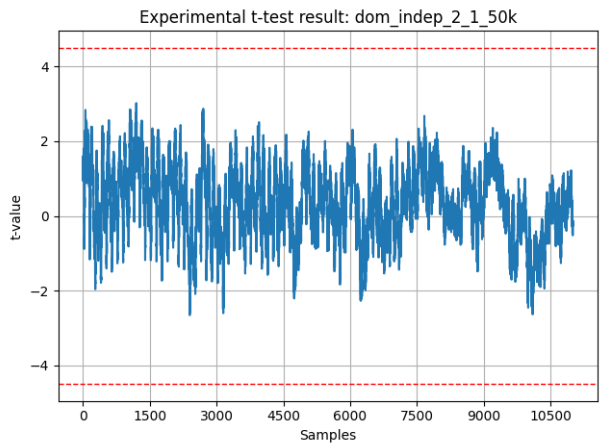


Figure 11.9: Experimental T-test result of dom-indep-2-1-without-b (Setup: 250MSa/s, 11KS, 2ns, 2MHz)

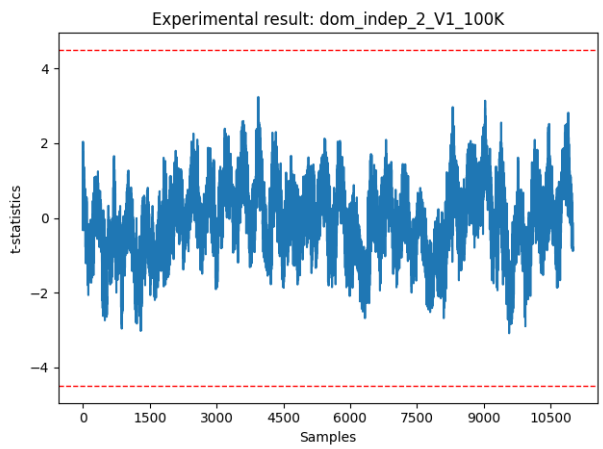


Figure 11.10: Experimental T-test result of dom-indep-2-V1 (traces exist) (Setup: 250MSa/s, 11KS, 2ns, 2MHz)

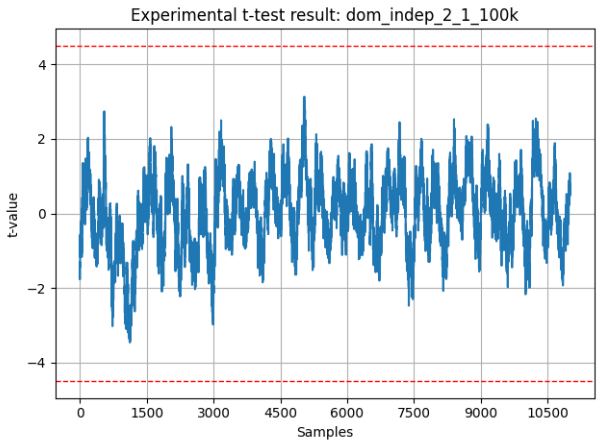


Figure 11.11: Experimental T-test result of dom-indep-2-1-without-b (Setup: 250MSa/s, 11KS, 2ns, 2MHz)

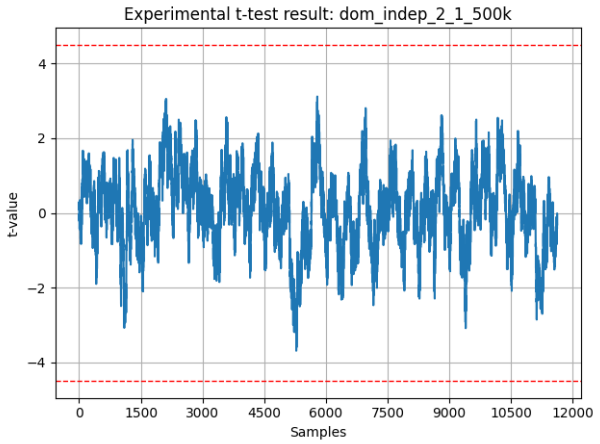


Figure 11.12: Experimental T-test result of dom-indep-2-1-without-b (Setup: 250MSa/s, 11KS, 2ns, 2MHz)

12. DOM-indep-3- no function/label call

The DOM-indep-3 multiplication (three shares) is shown in Algorithm 10 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 10 DOM-indep-3

Require: shares $a_0, a_1, a_2, b_0, b_1, b_2, rnd_0, rnd_1, rnd_2$

Ensure: c_0, c_1, c_2

1: $c_0 \leftarrow (a_0.b_0) \oplus ((a_0.b_1) \oplus rnd_0) \oplus ((a_0.b_2) \oplus rnd_1)$

2: $c_1 \leftarrow ((a_1.b_0) \oplus rnd_0) \oplus (a_1.b_1) \oplus ((a_1.b_2) \oplus rnd_2)$

3: $c_2 \leftarrow ((a_2.b_0) \oplus rnd_1) \oplus ((a_2.b_1) \oplus rnd_2) \oplus (a_2.b_2)$

In the implementation of DOM-indep-3, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

12.1 DOM-indep-3-no-function-call-0: basic code

Listing 12.1 demonstrates the first implementation of DOM-indep-3 based gfmul-s (DOM-indep-3-no-function-call-0). The instructions in Listing 12.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 12.1: DOM-indep-3-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @r8 = &c

@@@@@ Saving r0 (the address of the input a)
  mov  r9, r0        @ r9 = &a

@@@@@ Saving r1 (the address of the input b)
  mov  r10, r1       @ r10 = &b


  ldr  r3, =table    @ r3 = &taable, The address of the "table", for ai*bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r11, r3       @ r11 = &table

nop
nop
nop
nop


@@@@@ Loading arguments @@@@@@@@@@@@
  ldrb r4, [r0, #0]  @ r4 = a0; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
  ldrb r5, [r1, #0]  @ r5 = b0; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
  ldrb r6, [r0, #4]  @ r6 = a1; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
  ldrb r7, [r1, #4]  @ r7 = b1; r0 = &a, r1 = &b, r2 = &rnd, r3 = &table



@@@@@@@@@@ Computing: ai*bi

@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
@@@@ r4 = a0, r5 = b0, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@@
```

```

  adds r0, r1         @ r0 = s = table[a0] + table[b0]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
  adds r1, #6         @ r1 = 256
  adds r0, r1         @ r0 = s + 256

  ldrb r1, [r3, r0]   @ r1 = s = table[s+256]

  # @@@@ Checking if a0 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r4         @ r0 = -a0 hex(256-dec(a0))

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r3, #32        @ r3 = #32
  asrs r0, r3         @ r0 = -a0 >> 32

  ands r5, r0         @ r5 = tmp = b0 & (-a0 >> 32)
  negs r5, r5         @ r5 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r5, r3         @ r5 = -tmp >> 32

  ands r1, r5         @ r1 = s & (-tmp >> 32) = (a0*b0)

  mov  r0, r8         @ r0 = &c
  mov  r3, r11        @ r3 = &table
  strb r1, [r0, #0]   @ r1 = (a0*b0), Storing (a0*b0) in &c+#0, r1 = (a0*b0)
  strb r0, [r0, #4]   @ clearing the write bus

### we need to clear the read bus and also reg1, instead of using two instructions, just we
  # eors r1, r1         @ r1 = 0, clearing (a0*b0)
  # ldrb rx, [r3, #0]   @ rx = 0 = table[0],  Clearing the read bus
  ldrb r1, [r3, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg1



@@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = table
@@@@ r4 = a0, r5 = -tmp >> 32, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
@@@@ a1*b1

  ldrb r5, [r3, r6]   @ r5 = table[a1]
  ldrb r1, [r3, r7]   @ r1 = table[b1]

  adds r5, r1         @ r5 = s = table[a1] + table[b1]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
  adds r1, #6         @ r1 = 256
  adds r5, r1         @ r5 = s + 256

  ldrb r1, [r3, r5]   @ r1 = s = table[s+256]


  # @@@@ Checking if a1 = 0 or b1 = 0, return 0; without conditional branch
  negs r5, r6         @ r5 = -a1

  @ a1 in r6 is not needed anymore, r6 can be used
  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r6, #32
  asrs r5, r6         @ r5 = -a1 >> 32

  ands r5, r7         @ r5 = tmp = b1 & (-a1 >> 32)
  negs r5, r5         @ r5 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r5, r6         @ r5 = -tmp >> 32

  ands r1, r5         @ r1 = s & (-tmp >> 32) = (a1*b1)

  # mov  r0, r8         @ r0 = &c
  # mov  r3, r11        @ r3 = &table
  strb r1, [r0, #4]   @ r1 = (a1*b1), Storing (a1*b1) in &c+#4, r1 = (a1*b1)
  strb r0, [r0, #8]   @ clearing the write bus

### we need to clear the read bus and also reg1, istead of using two instructions, just we
  # eors r1, r1         @ r1 = 0, clearing (a1*b1)
  # ldrb rx, [r3, #0]   @ rx = 0 = table[0],  Clearing the read bus
  ldrb r1, [r3, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg1
```

```

@@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = &table
@@@@ r4 = a0, r5 = -tmp >> 32, r6 = 32, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```

  mov  r1, r8          @ r1 = &c
  strb r5, [r1, #8]    @ r5 = (a2*b2), Storing (a2*b2) in c2, r1 = (a2*b2)
  ## it is not possible to clear the write bus here
### we need to clear the read bus and also reg1, istead of using two instructions, just we
  # eors r5, r5          @ r5 = 0, clearing (a2*b2)
  # ldrb rx, [r3, #0]    @ rx = 0 = table[0],  Clearing the read bus
  ldrb r5, [r3, #0]    @ r5 = 0 = table[0],  Clearing the read bus and also reg5

@@@@@@@@@@ Computing: ci
@@@@ c0 = (a0*b0)+[(a0*b1)+rnd0]+[(a0*b2)+rnd1]
@@@@ c1 = [(a1*b0)+rnd0]+(a1*b1)+[(a1*b2)+rnd2]
@@@@ c2 = [(a2*b0)+rnd1]+[(a2*b1)+rnd2]+(a2*b2)


@@@@ r0 = b2, r1 = &c, r2 = &rnd, r3 = &table,
@@@@ r4 = a0, r5 = 0, r6 = 32, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@
```

```
  movs r6, #32
  asrs r1, r6          @ r1 = -a0 >> 32

  @ b1 in r7 is not needed anymore, r7 can be used
  ands r7, r1          @ r7 = tmp = b1 & (-a0 >> 32)
  negs r7, r7          @ r7 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r7, r6          @ r7 = -tmp >> 32

  ands r5, r7          @ r7 = s & (-tmp >> 32)  = (a0*b1)

  eors r5, r3          @ r5 = (a0*b1)+rnd0


@@@@ r0 = b2, r1 = -a0 >> 32, r2 = &rnd, r3 = rnd0,
@@@@ r4 = a0, r5 = (a0*b1)+rnd0, r6 = 32, r7 = -tmp >> 32,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@@@@ c0 = (a0*b0)+[(a0*b1)+rnd0]+[(a0*b2)+rnd1]
@@@@ (a0*b2)+rnd1
@@@@ a0*b2

  mov  r3, r11         @ r3 = &table
  ldrb r7, [r3, r4]    @ r7 = table[a0]
  ldrb r6, [r3, r0]    @ r6 = table[b2]

  adds r7, r6          @ r7 = s = table[a0] + table[b2]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r6, #250        @ r6 = 250, 256 is not possible, so it is split: 250 + 6
  adds r6, #6          @ r6 = 256
  adds r7, r6          @ r7 = s + 256

  ldrb r6, [r3, r7]    @ r6 = s = table[s+256]

  ldrb r7, [r2, #4]    @ r7 = rnd1

  # @@@@ Checking if a0 = 0 or b2 = 0, return 0; without conditional branch
  # negs r1, r4          @ r1 = -a0
```

```
258
259     # # @ asrs rd, <#imm> --> #imm: 0-31
260     # movs r6, #32
261     # asrs r1, r6          @ r1 = -a0 >> 32
262     ### From previous computation: r1 = -a0 >> 32
263     @ b2 in r0 is not needed anymore, r0 can be used
264     ands r0, r1          @ r0 = tmp = b2 & (-a0 >> 32)
265
266     movs r4, #32         @ r4 = 32     @ for avoiding stall
267
268     negs r0, r0          @ r0 = -tmp
269
270     # @ asrs rd, <#imm> --> #imm: 0-31
271     asrs r0, r4          @ r0 = -tmp >> 32
272
273     ands r6, r0          @ r6 = s & (-tmp >> 32)  = (a0*b2)
274
275     mov  r1, r8          @ r1 = &c   @ here for avoiding stall
276
277     eors r6, r7          @ r6 = (a0*b2)+rnd1
278
279
280   @@@@ r0 = -tmp >> 32, r1 = (-a0 >> 32), r2 = &rnd, r3 = &table,
281   @@@@ r4 = 32, r5 = (a0*b1)+rnd0, r6 = (a0*b2)+rnd1, r7 = rnd1,
282   @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
283   @@@@
```

```
@@@@ r0 = 0, r1 = &c, r2 = &rnd, r3 = &table,
@@@@ r4 = 32, r5 = &a, r6 = &b, r7 = 0,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  asrs r0, r4          @ r0 = -tmp >> 32

  ands r5, r0          @ r0 = s & (-tmp >> 32)  = (a1*b0)

  eors r5, r3          @ r5 = (a1*b0)+rnd0



@@@@ r0 = -tmp >> 32, r1 = -a1 >> 32, r2 = &rnd, r3 = rnd0,
@@@@ r4 = 32, r5 = (a1*b0)+rnd0, r6 = &b, r7 = a1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  # asrs r1, r6         @ r1 = -a1 >> 32
  ### From previous computation: r1 = -a1 >> 32
  @ b2 in r0 is not needed anymore, r0 can be used
  ands r0, r1         @ r0 = tmp = b2 & (-a1 >> 32)

  movs r6, #32        @ r6 = 32     @ for avoiding stall

  negs r0, r0         @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r6         @ r0 = -tmp >> 32

  ands r7, r0         @ r6 = s & (-tmp >> 32)  = (a1*b2)

  mov  r1, r8         @ r1 = &c @ here for avoiding stall


  eors r7, r4         @ r6 = (a1*b2)+rnd2


@@@@ r0 = -tmp >> 32, r1 = (-a1 >> 32), r2 = &rnd, r3 = &table,
@@@@ r4 = rnd2, r5 = (a1*b0)+rnd0, r6 = 32, r7 = (a1*b2)+rnd2,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
@@@@ r0 = 0, r1 = &c, r2 = &rnd, r3 = &table,
@@@@ r4 = 0, r5 = &a, r6 = 32, r7 = &b,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
473
474   eors r5, r3          @ r5 = (a2*b0)+rnd1
475
476
477 @@@@ r0 = -tmp >> 32, r1 = -a2 >> 32, r2 = &rnd, r3 = rnd1,
478 @@@@ r4 = a2, r5 = (a2*b0)+rnd1, r6 = 32, r7 = &b,
479 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
480 @@@@
```

```
516
517   movs r7, #32        @ r7 = 32     @ here for avoiding stall
518
519   negs r2, r0         @ r2 = -tmp   @ &rnd is not needed any more
520
521   # @ asrs rd, <#imm> --> #imm: 0-31
522   asrs r2, r7         @ r2 = -tmp >> 32
523
524   ands r4, r2         @ r4 = s & (-tmp >> 32)  = (a2*b1)
525
526   mov  r1, r8         @ r1 = &c  @ here for avoiding stall
527
528   eors r4, r6         @ r4 = (a2*b1)+rnd2
529
530
531 @@@@ r0 =  b1 & (-a2 >> 32), r1 = -a2 >> 32, r2 = -tmp >> 32, r3 = &table,
532 @@@@ r4 = (a2*b1)+rnd2, r5 = (a2*b0)+rnd1, r6 = rnd2, r7 = 32,
533 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
534 @@@@
```

12.1.1 Experimental result:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $150\mu s$
Each clock is sampled with 125 points
Number of samples = 240×125 = 30000 samples
The bytes of shares (a, b, c are not in a word)
Figure 12.1 shows the recorded 30KS before raising-edge of the trigger.

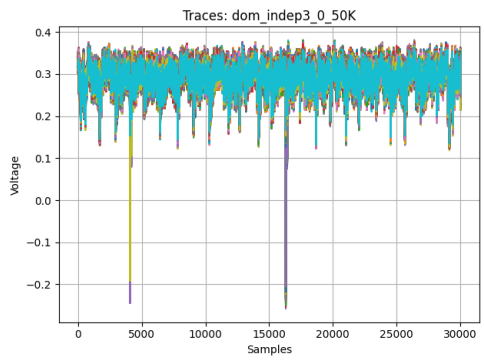


Figure 12.1: Traces DOM-indep-3-without-b-0 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

12.1.1.1 Signal to Noise Ratio (SNR):

12.1.1.2 t-test:

First-order univariate t-test:

Up to 1M traces, no leakage.

$$t - val = \frac{\mu_f - \mu_r}{\sqrt{\frac{s_f^2}{n_f} + \frac{s_r^2}{n_r}}} \tag{12.1}$$

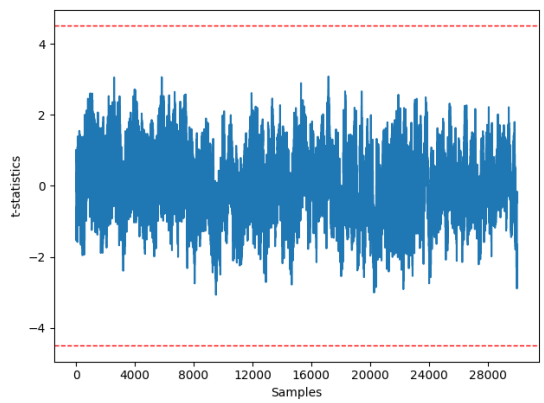


Figure 12.2: First-order univariate ttest dom-indep-3-v0 (Setup: 250MSa/s, 31250S, 4ns, 2MHz, 1M traces)

Second-order univariate t-test:

The traces t are preprocessed by mean-free squared:

$$T = (t - \mu_t)^2 \tag{12.2}$$

$$t - val = \frac{\mu_F - \mu_R}{\sqrt{\frac{s_F^2}{n_f} + \frac{s_R^2}{n_r}}} \tag{12.3}$$

Second-order multivariate t-test:

Finding leakages via splitting t-test in to three t-test s:

Finding leakages via splitting t-test in to three t-test s:

In this section we find the leakages caused by 2 shares out of 3 shares using the split-t-test .

1) (sh_0, sh_1)

Listing 12.2 shows the the leakages found by **uEMO**.

As it is shown in Figure 12.3, there are some leakages in real traces (cycles 33, 34, 35, 45, 46 from 240 recorded cycles (240×125) up to 100K traces).

Listing 12.2: Leakages reported by **uELMO** for DOM-indep-3-0-sh0-sh1

```
Cycle 44: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x200001B4]=0xAC
, Term=Glitchy decoding port 1 XOR current port 1, Tvalue=inf
Cycle 44: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x200001B4]=0xAC
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 45: Instr=Execute: negs r5,r6

Decode: movs r6,#0x20
Memory: idle
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 46: Instr=Execute: movs r6,#0x20
Decode: asrs r5,r6
Memory: idle
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=42.39207180279218
Cycle 47: Instr=Execute: asrs r5,r6
Decode: ands r5,r7
Memory: idle
, Term=LSB Neighbouring Reg 4 XOR Reg 5, Tvalue=42.39207180279218
```

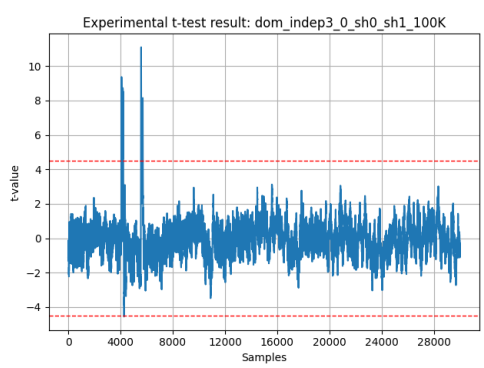


Figure 12.3: t-test_dom_indep3_0_sh0_sh1 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

Reasons and solutions of/for leakages:
1) Leakages in cycle 33, 34 and 35 are related to the instructions in
offset and register based address leakage.
lines 78 and 89 in Listing 12.1 (Lines 10 and 11 in Listing 12.3).
See file experiments/DOM_INDEP_3_sh0_sh1_3ldr_leakage.txt

Listing 12.3: Source of the first leakage in dom-indep-3-0-sh1-sh2

```
# r0 = table[a0] + table[b0]
# r1 = a0*b0
# r2 = &rnd
# r3 = &table
# r4 = a0
# r5 = -tmp>> 32
# r6 = a1
# r7 = b1

  ldrb r1, [r3, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg1
  ldrb r5, [r3, r6]   @ r5 = table[a1]
```

2) Leakages in cycle 46 and 45 are are related to the instructions in
lines 105, 109 and 110 in Listing 12.1 (listed in Listing 12.4).
a) The first reason is because of Glitchy leakages (**Glitchy register access**):
In movs r6, #32, because of the glitch first $r4$ then $r6$ are loaded:
$r4 = a_0, r6 = a_1 \rightarrow a_0 \oplus a_1$
b) The second leakage is (reported by **uELMO** Neighbouring effects [11] on Line 105,
Or As in real traces till now we have not faced Neighbouring effect leakages, we can say it is because of Manipulating different shares from the same family in sequence:
$r4 = a_0, r5 = table[a_1] + table[b_1] + 256, \rightarrow a_0 \oplus a_1$

Listing 12.4: Source of the second leakage in dom-indep-3-0-sh0-sh1

```
# r4 = a0
# r5 = table[a1] + table[b1] + 256
# r6 = a1

  negs r5, r6          @ r5 = -a1
  movs r6, #32
  ands r5, r7          @ r5 = tmp = b1 & (-a1 >> 32)
```

2) (sh_0, sh_2)

Listing 12.5 shows the the leakages found by **uEMO**, the source of the leakage are listed in Listing **??**. As it is shown in Figure 12.4, there are some leakages in real traces (cycles 71 and 72 from 240 recorded cycles (240×125) up to 100K traces).

Listing 12.5: Leakages reported by **uELMO** for DOM-indep-3-0-sh0-sh2

```
Cycle 68: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x20000281]=0x2B
, Term=Glitchy decoding port 1 XOR current port 1, Tvalue=inf
Cycle 68: Instr=Execute: stall by LDR
Decode: stall
Memory: load [0x20000281]=0x2B
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
Cycle 69: Instr=Execute: negs r1,r6

Decode: movs r6,#0x20
Memory: idle
, Term=Glitchy decoding port 1 XOR previous port 1, Tvalue=inf
```

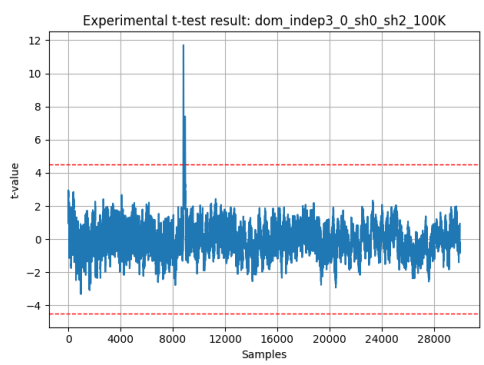


Figure 12.4: t-test_dom_indep3_0_sh0_sh2 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

Reasons and solutions of/for leakages:
1) Leakages in cycle 71 and 72 are related to the instructions in lines 158 and 162 in Listing 12.1 (Listing 12.6).
The leakage is because of Glitchy leakages (**Glitchy register access**):
In movs r6, #32, because of the glitch first $r4$ then $r6$ are loaded:
$r4 = a_0, r6 = a_2 \rightarrow a_0 \oplus a_2$

Listing 12.6: Source of the first leakage in dom-indep-3-0-sh0-sh1

```
# r4 = a0
# r6 = a2

 negs r1, r6          @ r1 = -a2
 movs r6, #32
```

3) (sh_1, sh_2)

uEMO did not report any leakage.

Figure 12.5 also indicates no leakage for $sh1$ and $sh2$.

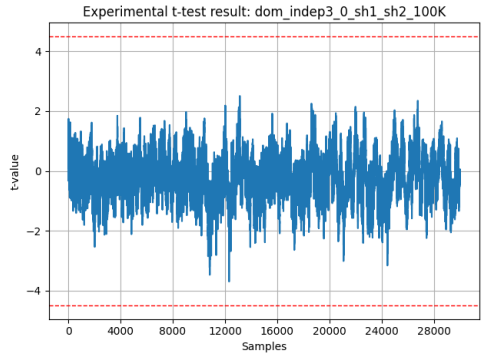
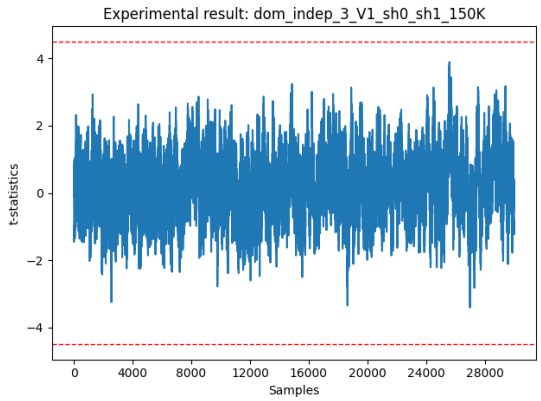


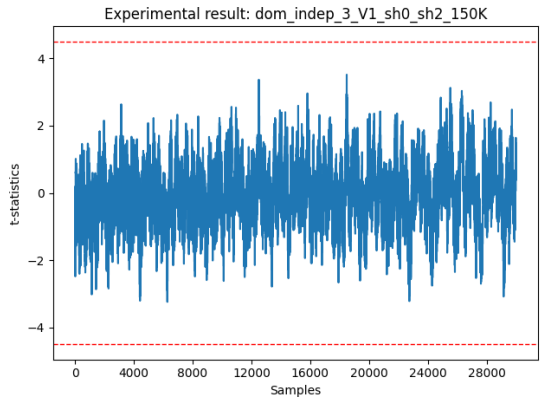
Figure 12.5: t-test_dom_indep3_0_sh1_sh2 (Setup: 250MSa/s, 30KS, 4ns, 2MHz)

dom-indep-3-V1

(a) t-test on (sh_0, sh_1)



(b) t-test on (sh_0, sh_2)



(c) t-test on (sh_1, sh_2)

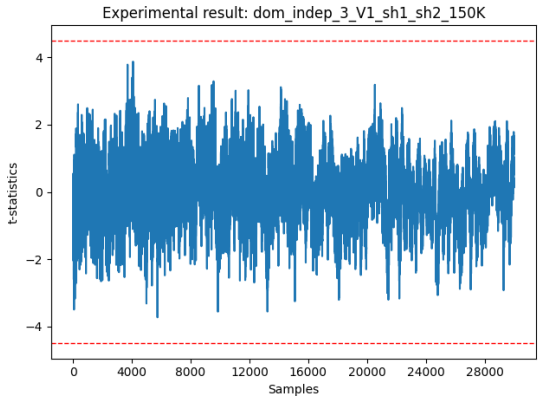


Figure 12.6: t-test_dom_indep_3_V1_two_shares(Setup: 250MSa/s, 56750S, 2ns, 2MHz)

13. HPC1

HPC1 multiplication is introduced in [2]. Algorithm 13.1 represents this scheme for $d + 1$ shares.

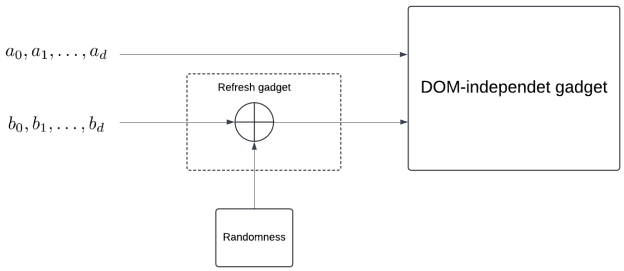


Figure 13.1: HPC1

14. HPC1-OPT-2

The HPC1-OPT-2 multiplication for two shares (HPC1-OPT-2) is illustrated in Algorithm 11 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 11 HPC1-OPT-2

Require: shares $a_0, a_1, b_0, b_1, r_0, r_1$

Ensure: c_0, c_1

1: $REFb_0 = b_0 \oplus r_0$

2: $REFb_1 = b_1 \oplus r_0$

3: $c_0 \leftarrow (a_0.REFb_0) \oplus ((a_0.REFb_1) \oplus r_1)$

4: $c_1 \leftarrow ((a_1.REFb_0) \oplus r_1) \oplus (a_1.REFb_1)$

In the implementation of HPC1-OPT-2, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

14.1 HPC1-OPT-2-no-function-call-0: basic code

Listing 14.1 demonstrates the first implementation of HPC1-OPT-2 based gfmul-s (HPC1-OPT-2-no-function-call-0). The instructions in Listing 14.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 14.1: HPC1-OPT-2-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @ r8 = &c


@@@@ Saving the value 256 in r11. This decreases one cycle.
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r4, #250       @ r4 = 250, 256 is not possible, so it is split: 250 + 6
  adds r4, #6         @ r4 = 256
  mov  r11, r4        @ r11 = 256


  ldr  r3, =table    @ r3 = &table, The address of the "table", for ai * bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r9, r3        @ r9 = &table


nop
nop
nop
nop



@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &table
@@@@ r8 = &c, r9 = &table, r11 = 256
@@@@@
```

```
@@@@@ Loading arguments @@@@@@@@@@@@
  ldrb r5, [r0, #4] @ r5 = a1;    r0 = &a, r1 = &b, r2 = rnd1
  ldrb r2, [r2, #4] @ r2 = rnd1;  r0 = &a, r1 = &b
  ldrb r7, [r0, #0] @ r7 = a0;    r0 = &a, r1 = &b, r2 = &rnd

@@@@@@@@@ Computing c0 = (a0 * ref_b0) + [(a0 * ref_b1) + rnd0]

@@@@ r0 = &a, r1 = &b, r2 = rnd1, r3 = &table
@@@@ r4 = ref_b0, r5 = a1, r6 = ref_b1, r7 = a0,
@@@@ r8 = &c, r9 = &table, r11 = 256
@@@@@@@@@@@
```

```
@@@@ r4 = ref_b0, r5 = a1, r6 = ref_b1, r7 = a0,
@@@@ r8 = &c, r9 = &table, r10 = a0 * ref_b0, r11 = 256
@@@@
```

```
  mov  r3, r8          @ r3 = &c
  strb r0, [r3, #0]    @ Storing c0, r0 = c0
@@@@@@@@@ Computing c1 = [(a1 * ref_b0) + rnd1] + (a1 * ref_b1)

@@@@ r0 = c0, r1 = &table, r2 = rnd1, r3 = &c
@@@@ r4 = ref_b0, r5 = a1, r6 = ref_b1, r7 = 32,
@@@@ r8 = &c, r9 = &table, r10 = &c, r11 = 256
@@@@@@@@@
```

```
  eors r2, r4         @ r2 = (a1 * ref_b0) + rnd1


@@@@ r0 = -a1 >> 32, r1 = &table, r2 = (a1 * ref_b0) + rnd1, r3 = table[s+256]
@@@@ r4 = (a1 * ref_b0), r5 = a1, r6 = ref_b1, r7 = 32,
@@@@ r8 = &c, r9 = &table, r10 = &c, r11 = 256
@@@@
```

```
strb r2, [r3, #4]   @ Storing c1, r2 = c1


nop
nop
nop
nop
```

14.1.1 Experimental result:

14.1.1.1 Setup:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $86\mu s$
Each clock is sampled with 125 points
Number of samples = 120×125 = 15000 samples
The bytes of shares (a, b, c are not in a word)
Figure 14.1 shows the recorded 15KS before raising-edge of the trigger.

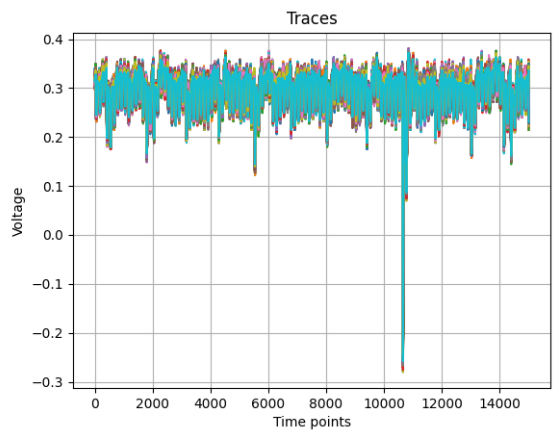


Figure 14.1: Traces hpc1-opt-2-without-b-0 (Setup: 250MSa/s, 15kS, 2ns, 2MHz)

14.1.1.2 Signal to Noise Ratio (SNR):

Figure 14.2 shows the SNR for all the intermediate values. It can be concluded that the first 2500 (20×125) and also the last 1000 (8×125) samples do not have effect on the SNR, so, they can be removed.
Number of samples = $(120 - 20 - 8) \times 125 = 92 \times 125$ = 11500 samples

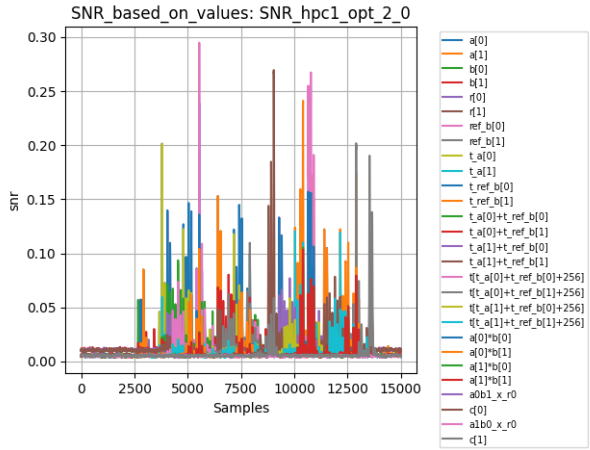


Figure 14.2: SNR hpc-1-opt-2-without-b-0 (Setup: 250MSa/s, 15kS, 2ns, 2MHz)

Figure 14.3 shows the SNR of traces with 11.5K samples.

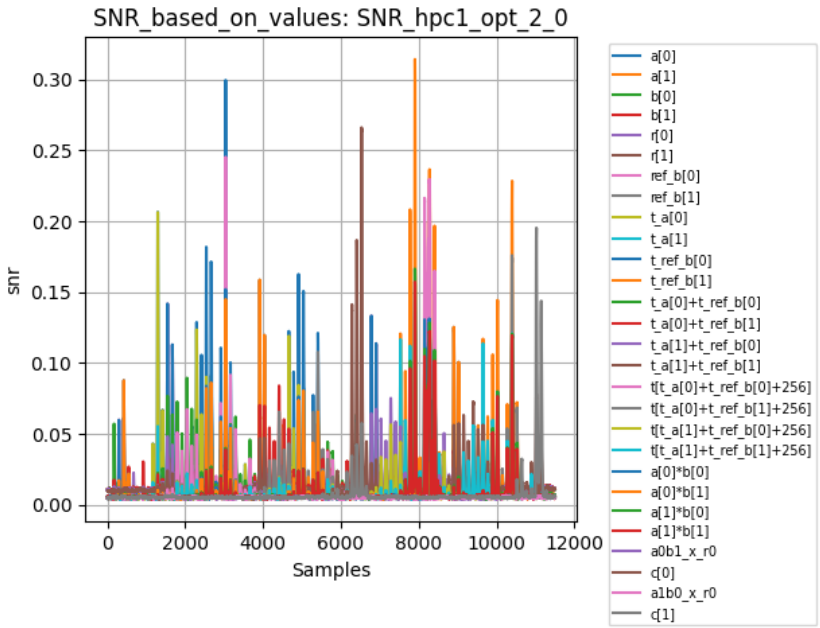


Figure 14.3: SNR hpc-1-opt-2-without-b-0 (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

14.1.1.3 t-test:

Figure 14.5 shows the fixed-versus-random t-test results on both inputs a and b for 100K traces. The leakages occur at cycles: 4-7, 10, 11, 19, 38, 46-48, 90 and 91.
35, 36, 127, 130, 215
in HPC1-OPT-2-no-function-call-0.

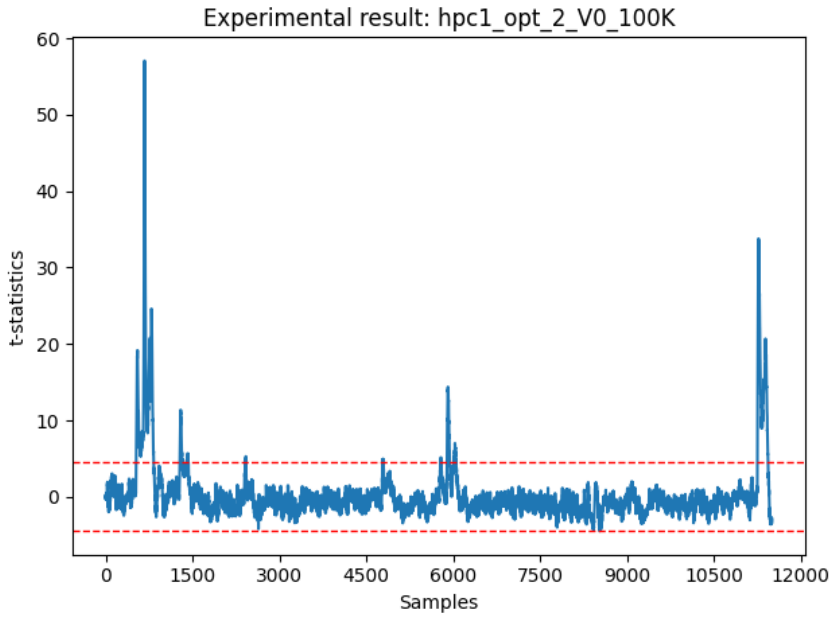
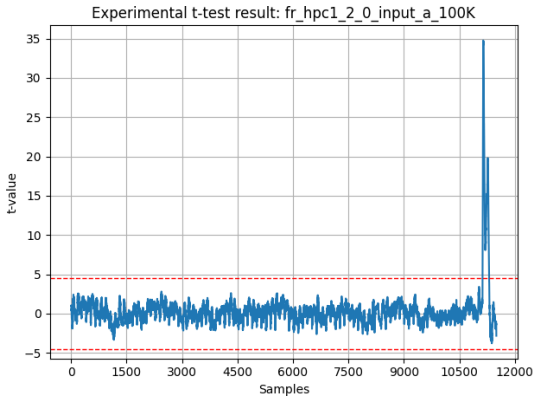
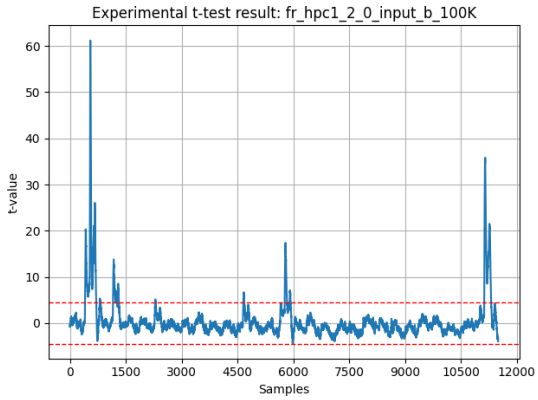


Figure 14.4: SNR hpc-1-opt-2-V0 (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

(a) t-test on input a



(b) t-test on input b



(c) t-test on both inputs a and b

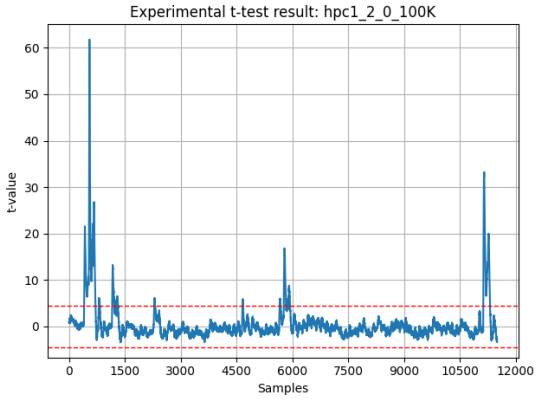


Figure 14.5: t-test hpc1-opt-2-0-without-b-0-(Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

Also, the leakage simulator uELMO, which reports the some leakages which are listed in Listing 14.2.

Listing 14.2: Leakages reported by **uELMO** for hpc1-opt-2-0

```
Cycle 16: Instr=Execute:eors r4,r7
Decode:eors r6,r7
Memory: idle
, Term=Decoding port 0 HD, Tvalue=inf
Cycle 16: Instr=Execute:eors r4,r7
Decode:eors r6,r7
Memory: idle
, Term=Glitchy decoding port 0 XOR previous port 0, Tvalue=inf
Cycle 17: Instr=Execute:eors r6,r7
Decode:ldrb r5,[r0,#0x4]
Memory: idle
, Term=Pipeline Reg 1 HD, Tvalue=inf
Cycle 17: Instr=Execute:eors r6,r7
Decode:ldrb r5,[r0,#0x4]
Memory: idle
, Term=ALU output HD, Tvalue=inf
Cycle 55: Instr=Execute: mov r3,r10
Decode:eors r0,r3
Memory: idle
, Term=Decoding port 1 HD, Tvalue=inf
Cycle 56: Instr=Execute:eors r0,r3
Decode: mov r3,r8
Memory: idle
, Term=Reg 3 HD, Tvalue=inf
Cycle 93: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: store [0x20001FBC]=0xE7
, Term=Memory write buffer HD, Tvalue=inf
Cycle 94: Instr=Execute: mov r8,r8
Decode: mov r8,r8
Memory: idle
, Term=Memory write buffer delayed HD, Tvalue=inf
```

Reasons and solutions:

1) Leakages cycles 4-7, are correspond to the Optimized Refresh Mask of b (Lines 33, 35, 36, Listed in 14.3)

Listing 14.3: Leakages related to Refreshing shares of b

```
1
2   ldrb r6, [r1, #4] @ r6 = b1;    r0 = &a, r1 = &b, r2 = rnd1
3   eors r4, r7       @ r4 = ref_b0  @ ref_b0 = b0 + rnd0
4   eors r6, r7       @ r6 = ref_b1  @ ref_b1 = b1 + rnd0
5
6
```

In these thre instructions, $port1$ is filled as: $b1 \rightarrow b0 \rightarrow b1$, which causes leaking b.
For avoiding this leakage, we mix the Optimized Refresh Mask part and Loading arguments in order to refuse to filling the $ports$ with shares of same inputs, as it is shown in Listing 14.4

Listing 14.4: Mixing Loading arguments and Optimized Refresh Mask

```
1
2  @@@@@ Loading arguments and Optimized Refresh Mask:
3
4    ldrb r4, [r1, #0] @ r4 = b0;    r0 = &a, r1 = &b, r2 = rnd1
5    ldrb r7, [r2, #0] @ r7 = rnd0;  r0 = &a, r1 = &b, r2 = &rnd
6
7    eors r4, r7       @ r4 = ref_b0  @ ref_b0 = b0 + rnd0
8
9    ldrb r5, [r0, #4] @ r5 = a1;    r0 = &a, r1 = &b, r2 = rnd1
10   ldrb r6, [r1, #4] @ r6 = b1;    r0 = &a, r1 = &b, r2 = rnd1
11   ldrb r2, [r2, #4] @ r2 = rnd1;  r0 = &a, r1 = &b
12
13   eors r6, r7       @ r6 = ref_b1  @ ref_b1 = b1 + rnd0
14
15   ldrb r7, [r0, #0] @ r7 = a0;    r0 = &a, r1 = &b, r2 = &rnd
16
```

Figure 14.6 shows the result of applying this patch for 50K traces.

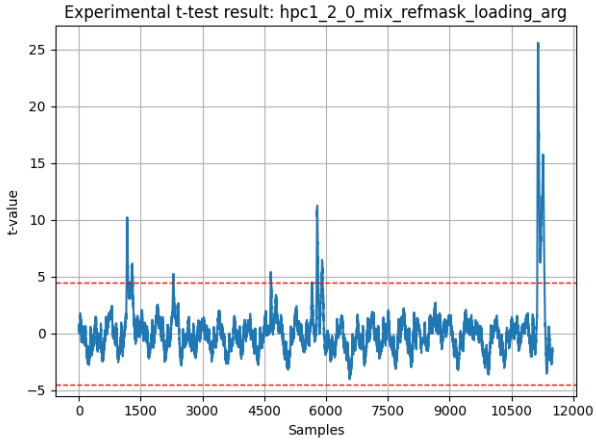


Figure 14.6: Experimental T-test result of hpc1-opt-2-0-without-b-0 - patch: mixing refmask and loadining args (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

2) The write bus (str instruction): (leakages in cycles 90 and 91) Lines 130 and 215 in Listing 14.1 made this type of leakage. Solution is clearing the write bus after storing $c0$ (Listing 14.5). Figure 14.7 shows the result of applying this patch for 50K traces.

Listing 14.5: Patching the write bus leakage in hpc1-opt-2-without-b-0 (storing shares of c)

```
strb r0, [r3, #0]   @ Storing c0, r0 = c0

  A sequence of instructions

  strb r3, [r3, #4]   @ Clearing the write bus, r3 = &c

strb r2, [r3, #4]   @ Storing c1, r2 = c1

```

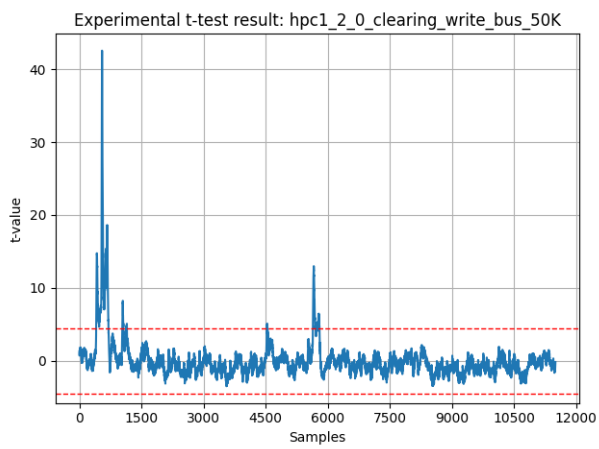


Figure 14.7: Experimental T-test result of hpc1-opt-2-0-without-b-0 - patching the write bus (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

3) Register HD leakage between the previous value of a register and and its new written value. (leakages in cycles 46-48) Rewriting register causes a leakage in Line 127. As, $r3 = table[table[a_0]+table[REFb_1]+256]$, and then with mov r3, r10, $r3$ becomes $(a_0 \times REFb_0)$, so that, b will leak. For removing this leakage, the previous content of $r3$ has to be cleared.
Thus, after computing $a_0 \times REFb_0$ in ands r0, r3 (Line 120), putting eors r3, r3 will clean the $r3$. Figure 14.8 shows the result of applying the mentioned patch for 100K traces.

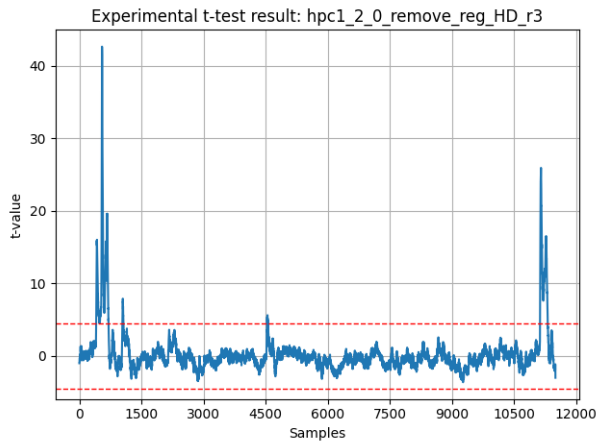


Figure 14.8: Experimental T-test result of hpc1-opt-2-0-without-b-0 - patching the register HD (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

4) Two LDRB leakage: leakages in cycles 10-12. Line 56 and 57 (Listed in Listing 14.6). The leakage is related to the two ldrb instructions.
Delay on switching values in SEL0 signal in multiplexer tree (Figure 14.9):

Listing 14.6: Leakages related to two ldrb instructions, delay on switching register tree

```
1
2   # r0 = &a,
3   # r1 = &b,
4   # r2 = &rnd
5   # r3 = &table
6   # r4 = ref-b0
7   # r5 = a1
8   # r6 = ref-b1
9   # r7 = a0
10
11   ldrb r0, [r3, r7]    @ r0 = table[a0]
12   ldrb r1, [r3, r4]    @ r1 = table[ref_b0]
13
14
```

Description about the register tree:
As all codes are based on THUMB-16, just Low registers are considered in the figure.

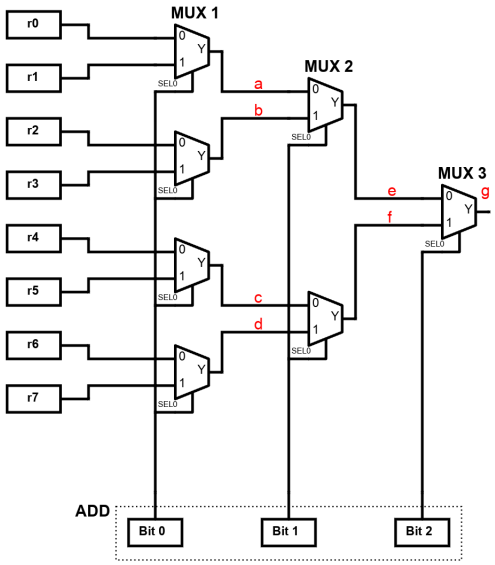


Figure 14.9: Multiplexer tree

From paper [5], Table 1:
In Decoding stage of ldrb Rd, [Rn, Rm] instruction:
Port1 ← Rd
Port2 ← Rn
Port3 ← Rm

and From paper [6], register tree (Figure 14.9).
Regarding Listing 14.6,
According [5]: in the decode of the first ldrb, register $r7$ (address in binary: 111) is imported into $Port3$. Then in the next cycle, register $r4$ (address in binary: 001) (from the second ldrb) must be imported into $Port3$.
First cycle: According [6]: For $r7$, SEL0 signals (ADD: Bit0: 1, Bit1: 1, Bit2: 1) the multiplexers will be 111, so that $r7$ (=$a0$) goes on the wires d, f and g.
Second cycle: According [6]: For $r4$, SEL0 signals (ADD: Bit0: 0, Bit1: 0, Bit2: 1) the multiplexers will switch to 001, so that:

There is a delay in switching values in multiplexer values (Bit_i), and the leakage occurs on the wires of the multiplexer tree (wires: a-g in Figure 14.9, not in decoding ports or pipeline registers:
Bit0: $1 \rightarrow 0$:
Because of the delay, first $r5$ ($a1$) then $r4$ ($ref - b0 = b0 \oplus rnd0$) goes on wire c.
Because of the delay, first $r7$ ($a0$) then $r6$ ($ref_b1 = b1 \oplus rnd0$) goes on wire d.
Bit1: $1 \rightarrow 0$:
Because of the delay, first the value of the wire c ($r4 = ref - b0 = b0 \oplus rnd0$) then the value of the wire d ($r6 = ref - b1 = b1 \oplus rnd0$) goes on the wire f.
As a result, on the wire f, input b ($ref - b0 \oplus ref - b1 = (b0 \oplus rnd0) \oplus (b1 \oplus rnd0) = b0 \oplus b1 = b$) will leak.

There are 4 ways to remove this leakage as listed in Listing 5.6, 5.7, 5.8 and 5.9.
1) Changing the order of the 2 LDRB instructions (delay on switching wires) (Listing 5.6, Figure 5.15)
For removing this leakage (without inserting an extra instruction), we change the order of the last two instructions (Listing 14.7. With this way, decreasing one clock cycle, as in ldrb r7, [r0, #0] and ldrb r0, [r3, r7] there is a STALL regarding $r7$.

Listing 14.7: Removing leakages related to two LDRBs

```

ldrb r7, [r0, #0]   @ r7 = a0;    r0 = &a, r1 = &b, r2 = &rnd
ldrb r1, [r3, r4]   @ r1 = table[ref_b0]
ldrb r0, [r3, r7]   @ r0 = table[a0]


```

Figure 14.10 shows the result of applying the mentioned patch for 100K traces.

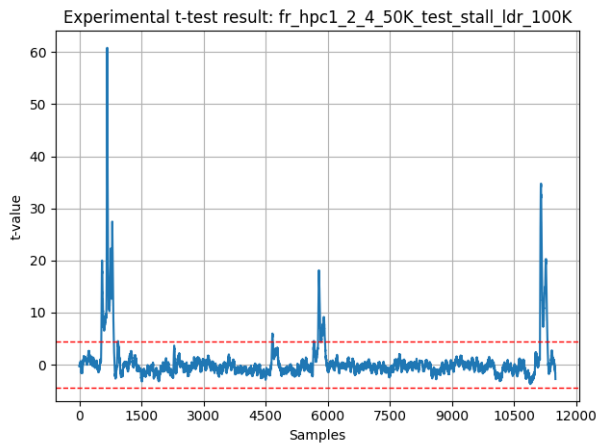


Figure 14.10: Experimental T-test result of hpc1-opt-2-0-without-b-0 - patching the two LDRB instructions (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

5) Leakage in cycle 19: Lines 69 and 72
Leakage: Delay on switching values on wires of the multiplexer tree plus glitchy leakage

Listing 14.8: Mixed glitchy and delay leakage

```
1
2  # r0 = table[a0] + table[ref_b0] + 256,
3  # r1 = table[table[a0] + table[ref_b0] + 256],
4  # r2 = rnd1,
5  # r3 = &table,
6  # r4 = ref_b0,
7  # r5 = a1,
8  # r6 = ref_b1,
9  # r7 = a0
10
11   negs r0, r7          @ r0 = -a0 hex(256-dec(a0))
12
13   # @ asrs rd, <#imm> --> #imm: 0-31
14   movs r3, #32         @ r3 = #32
15
```

The effect of the glitchy leakage:
During the execution of negs r0, r7 instruction, the movs r3, #32 instruction (which was encoded to 0x2320: 0010001100100000) is decoded in the Decoder stage of the pipeline. The 0x2320 (movs) is decoded at first in the negs style (5.9 a), then it is decoded according to the movs format (5.9 b), so that a glitch will occur here.
Assuming for INSRT Rd, Rm, Rd and Rm are loaded into $port1$ and $port2$, respectively. Regarding it, in the decoding stage of negs, port 1 is loaded with $r7$ ($r7 = a0$), now, when the movs is decoded in the negs format , the content of register of the bits 5-3 (0x2320: 0010 0011 0010 0000, which is 100 = 4 and means register $r4$ ($= ref - b0$)) is loaded into port 1. There is no leakage in decoding port1 as $a0 \rightarrow ref - b0$.

Regarding it, multiplexer tree needs to first select $r7$ (binary representation:111) with SEL0 signals (ADD: Bit0: 1, Bit1: 1, Bit2: 1) and then $r4$ (binary representation:100) with SEL0 signals (ADD: Bit0: 0, Bit1: 0, Bit2: 1).

The effect of the switching delay:
In the first case (ADD: 111): $r7$ (= $a0$) goes on the wires d, f, g.
Then, for ADD: 001, the multiplexer SEL0 signals has to change from 111 to 001. Regarding the delay in switching the values:
Bit0: $1 \rightarrow 0$:
Because of the delay, first $r5$ ($a1$) then $r4$ ($ref - b0$) goes on wire c.
Because of the delay, first $r7$ ($a0$) then $r6$ ($ref - b1$) goes on wire d.
Bit1: $1 \rightarrow 0$:
Because of the delay, first the value of the wire c ($r4 = ref - b0$) then the value of the wire d ($r6 = ref - b1$) goes on the wire f.
Thereby, on the wire f, input b ($ref - b0 \oplus ref - b1$) will leak.

For removing this kind of leakage, we can use two approaches as the follow:
1) Instead of using movs rx, #32, using the two instructions: movs rx, #16 and adds rx, #16
With this approach, when the movs is decoded in the negs format, the bits 5-3 would be 010 = 2 and means the content of register $r2$ is loaded into port 1.
2) Through the implementation of HPC1-OPT-2-no-function-call-0, changing the content of $r3$ and $r4$ with each other. This approach does not need extra instruction. Figure 14.12 shows the result of applying the mentioned patch for 100K traces.
But this way causes another leakage in LDRB r0 r1 r5 and LDRB r3 r1 r4.

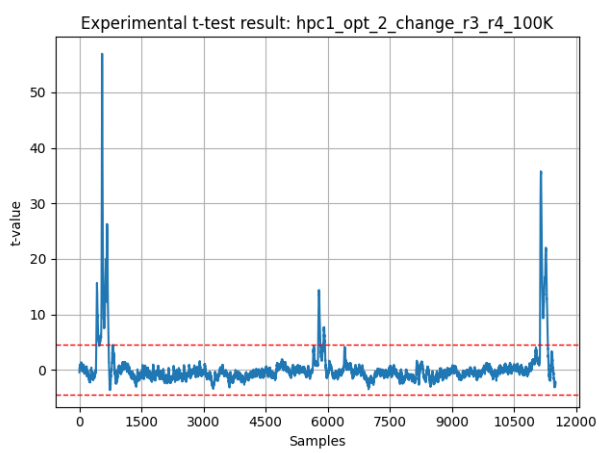


Figure 14.11: Experimental T-test result of hpc1-opt-2-0-without-b-0 - patching the mixed leakage (glitchy and delay on switching wires) (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

6) Leakage in cycle 38: Lines 106 and 111
Same like the previous leakage (ITEM 5)

Listing 14.9: Mixed glitchy and delay leakage

```
1
2  # r0 = table[a0] + table[ref_b1] + 256,
3  # r1 = &table,
4  # r2 = rnd1,
5  # r3 = table[table[a0] + table[ref_b1] + 256],
6  # r4 = ref_b0,
7  # r5 = a1,
8  # r6 = ref_b1,
9  # r7 = a0
10
11   negs r0, r7          @ r0 = -a0 hex(256-dec(a0))
12
13   # @ asrs rd, <#imm> --> #imm: 0-31
14   movs r7, #32         @ r3 = #32
15
```

Figure 14.12 shows the result of applying the mentioned patch for 100K traces.

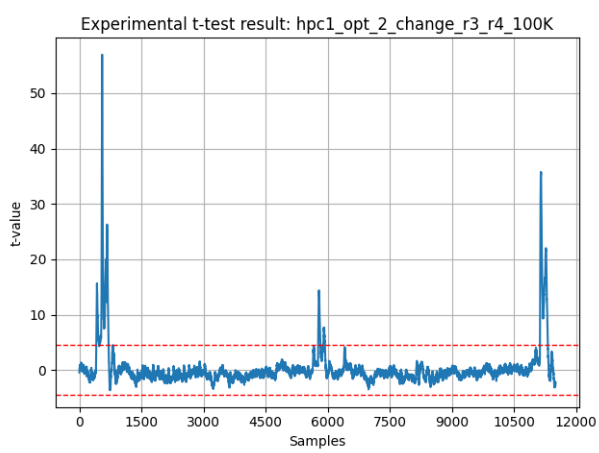


Figure 14.12: Experimental T-test result of hpc1-opt-2-0-without-b-0 - patching the mixed leakage (glitchy and delay on switching wires) (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

Less leaky implementation of hpc1-opt-2:
By applying the below patches:
1) Mixing loading arguments and optimized refresh mask
2) Clearing the write bus
3) Removing the Register HD
4) changing the order of the 2 LDRB instructions (delay on switching wires)
5) Removing the mixed leakage (glitchy and delay on switching wires)
With this implementation, uELMO, reportes no leakages.
Figures 14.13 and 14.15 show the t-test result for 100K and 600K traces.

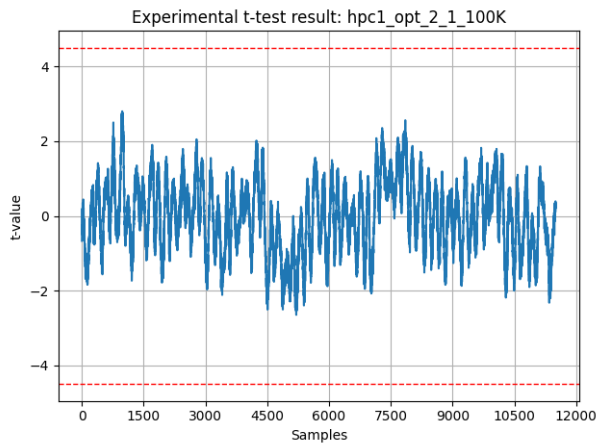


Figure 14.13: Experimental T-test result of hpc1-opt-2-1-without-b (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

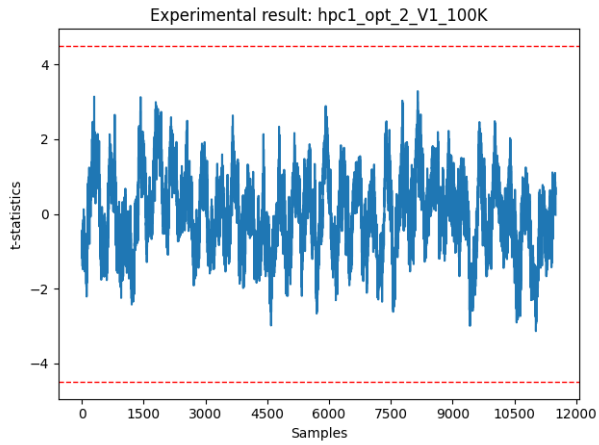


Figure 14.14: Experimental T-test result of hpc1-opt-2-V1 (traces exist) (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

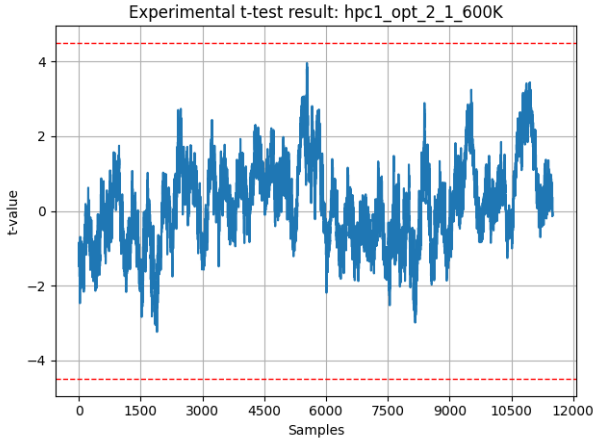


Figure 14.15: Experimental T-test result of hpc1-opt-2-1-without-b (Setup: 250MSa/s, 11.5KS, 2ns, 2MHz)

15. HPC1-OPT-3- no function/label call

The HPC1-OPT-3 multiplication (three shares) is shown in Algorithm 12 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 12 HPC1-OPT-3

Require: shares $a_0, a_1, a_2, b_0, b_1, b_2, r_0 - r_4$

Ensure: c_0, c_1, c_2

1: $REFb_0 = b_0 \oplus r_0$

2: $REFb_1 = b_1 \oplus r_1$

3: $REFb_2 = b_2 \oplus (r_0 \oplus r_1)$

4: $c_0 \leftarrow (a_0.REFb_0) \oplus ((a_0.REFb_1) \oplus r_2) \oplus ((a_0.REFb_2) \oplus r_3)$

5: $c_1 \leftarrow ((a_1.REFb_0) \oplus r_2) \oplus (a_1.REFb_1) \oplus ((a_1.REFb_2) \oplus r_4)$

6: $c_2 \leftarrow ((a_2.REFb_0) \oplus r_3) \oplus ((a_2.REFb_1) \oplus r_4) \oplus (a_2.REFb_2)$

In the implementation of HPC1-OPT-3, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

15.1 HPC1-OPT-3-no-function-call-0: basic code

Listing 15.1 demonstrates the first implementation of HPC1-OPT-3 based gfmul-s (HPC1-OPT-3-no-function-call-0). The instructions in Listing 15.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 15.1: HPC1-OPT-3-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @r8 = &c

@@@@@ Saving r0 (the address of the input a)
  mov  r9, r0        @ r9 = &a

@@@@@ Saving r1 (the address of the input b)
  mov  r10, r1       @ r10 = &b


  ldr  r4, =table    @ r4 = &taable, The address of the "table", for ai*bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r11, r4       @ r11 = &table

nop
nop
nop
nop


@@@@@
```

```
43
44  @@@@ ref_b2
45
46    eors r4, r3          @ r3 = rnd0 ^ rnd1
47    ldrb r6, [r0, #4]    @ r6 = a1; r0 = &a, r1 = &b, r2 = &rnd
48    ldrb r3, [r1, #8]    @ r4 = b2; r0 = &a, r1 = &b, r2 = &rnd
49    eors r3, r4          @ r4 = ref_b2  @ ref_b2 = b2 ^ (rnd0 ^ rnd1)
50    strb r0, [r1, #8]    @ Clearing the write bus and Cleaaring b2 from the memory
51    strb r3, [r1, #8]    @ Storing ref_b2 in the memory
52    ldrb r4, [r0, #0]    @ r4 = a0; r0 = &a, r1 = &b, r2 = &rnd
53
54
55  @@@@@@@@@@ Computing: ai*bi
56
57  @@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = ref_b2
58  @@@@ r4 = a0, r5 = ref_b0, r6 = a1, r7 = ref_b1,
59  @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
60  @@@@@@@@@@
```

```
  ands r5, r0          @ r5 = tmp = ref_b0 & (-a0 >> 32)
  negs r5, r5          @ r5 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r5, r3          @ r5 = -tmp >> 32

  ands r1, r5          @ r1 = s & (-tmp >> 32) = (a0*ref_b0)

  mov  r0, r8          @ r0 = &c
  mov  r3, r11         @ r3 = &table
  strb r1, [r0, #0]    @ r1 = (a0*ref_b0), Storing (a0*ref_b0) in &c+#0, r1 = (a0*ref_b0)
# we need to clear the read bus and also reg1, istead of using two instructions, just we ca
  # eors r1, r1          @ r1 = 0, clearing (a1*ref_b1)
  # ldrb rx, [r3, #0]    @ rx = 0 = table[0],  Clearing the read bus
  ldrb r1, [r3, #0]    @ r1 = 0 = table[0],  Clearing the read bus and also reg1
  strb r0, [r0, #4]    @ clearing the write bus

@@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = &table
@@@@ r4 = a0, r5 = -tmp >> 32, r6 = a1, r7 = ref_b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
129
130   @ a1 in r6 is not needed anymore, r6 can be used
131   # @ asrs rd, <#imm> --> #imm: 0-31
132
133 @ # fake leakage: leaky cycle: [129, 130]
134 @ mov r4, r4
135
136   movs r6, #32
137   asrs r5, r6          @ r5 = -a1 >> 32
138
139   ands r5, r7          @ r5 = tmp = ref_b1 & (-a1 >> 32)
140   negs r5, r5          @ r5 = -tmp
141
142   # @ asrs rd, <#imm> --> #imm: 0-31
143   asrs r5, r6          @ r5 = -tmp >> 32
144
145   ands r1, r5          @ r1 = s & (-tmp >> 32) = (a1*ref_b1)
146
147   # mov  r0, r8          @ r0 = &c
148   # mov  r3, r11         @ r3 = &table
149   strb r1, [r0, #4]    @ r1 = (a1*ref_b1), Storing (a1*ref_b1) in &c+#4, r1 = (a1*ref_b1)
150 ## we need to clear the read bus and also reg1, istead of using two instructions, just we c
151   # eors r1, r1          @ r1 = 0, clearing (a1*ref_b1)
152   # ldrb rx, [r3, #0]    @ rx = 0 = table[0],  Clearing the read bus
153   ldrb r1, [r3, #0]    @ r1 = 0 = table[0],  Clearing the read bus and also reg1
154   strb r0, [r0, #8]    @ clearing the write bus
155
156 @@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = &table
157 @@@@ r4 = a0, r5 = -tmp >> 32, r6 = 32, r7 = ref_b1,
158 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
159 @@@@
```

```
172     ldrb r1, [r3, r6]   @ r1 = table[a2]
173     ldrb r5, [r3, r0]   @ r5 = table[ref_b2]
174
175     adds r1, r5         @ r1 = s = table[a2] + table[ref_b2]
176
177     @@@@ Get the antilog: s = table[s+256]: g+s
178     # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
179     # @ movs rd, <#imm> --> #imm: 0-255
180     movs r5, #250       @ r5 = 250, 256 is not possible, so it is split: 250 + 6
181     adds r5, #6         @ r5 = 256
182     adds r1, r5         @ r1 = s + 256
183
184     ldrb r5, [r3, r1]   @ r5 = s = table[s+256]
185
186     # @@@@ Checking if a2 = 0 or ref_b2 = 0, return 0; without conditional branch
187     negs r1, r6         @ r1 = -a2
188
189     @ a2 in r6 is not needed anymore, r6 can be used
190     # @ asrs rd, <#imm> --> #imm: 0-31
191     movs r6, #32
192     asrs r1, r6         @ r1 = -a2 >> 32
193
194     ands r1, r0         @ r1 = tmp = ref_b2 & (-a2 >> 32)
195     negs r1, r1         @ r1 = -tmp
196
197     # @ asrs rd, <#imm> --> #imm: 0-31
198     asrs r1, r6         @ r1 = -tmp >> 32
199
200     ands r5, r1         @ r5 = s & (-tmp >> 32) = (a2*ref_b2)
201
202     mov  r1, r8         @ r1 = &c
203     strb r5, [r1, #8]   @ r5 = (a2*ref_b2), Storing (a2*ref_b2) in c2, r1 = (a2*ref_b2)
204     ## it is not possible to clear the write bus here
205     eors r5, r5         @ r5 = 0, clearing (a2*ref_b2)
206     ldrb r5, [r3, #0]   @ r5 = 0 = table[0],  Clearing the read bus
207
208 @@@@@@@@@@@ Computing: ci
209 @@@@ c0 = (a0*ref_b0)^[(a0*ref_b1)^rnd2]^[(a0*ref_b2)^rnd3]
210 @@@@ c1 = [(a1*ref_b0)^rnd2]^(a1*ref_b1)^[(a1*ref_b2)^rnd4]
211 @@@@ c2 = [(a2*ref_b0)^rnd3]^[(a2*ref_b1)^rnd4]^(a2*ref_b2)
212
213
214 @@@@ r0 = ref_b2, r1 = &c, r2 = &rnd, r3 = &table,
```

```
@@@@ r4 = a0, r5 = 0, r6 = 32, r7 = ref_b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  mov  r1, r8         @ r1 = &c   @ here for avoiding stall

  eors r6, r7         @ r6 = (a0*ref_b2)^rnd3


@@@@ r0 = -tmp >> 32, r1 = (-a0 >> 32), r2 = &rnd, r3 = &table,
@@@@ r4 = 32, r5 = (a0*ref_b1)^rnd2, r6 = (a0*ref_b2)^rnd3, r7 = rnd3,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  ldrb r0, [r6, #0]   @ r0 = ref_b0

  ldrb r1, [r3, r7]   @ r1 = table[a1]
  ldrb r5, [r3, r0]   @ r5 = table[ref_b0]

  adds r1, r5         @ r1 = s = table[a1] + table[ref_b0]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r5, #250       @ r5 = 250, 256 is not possible, so it is split: 250 + 6
  adds r5, #6         @ r5 = 256
  adds r1, r5         @ r1 = s + 256

  ldrb r5, [r3, r1]   @ r5 = s = table[s+256]
  ldrb r3, [r2, #8]   @ r3 = rnd2

  # @@@@ Checking if a1 = 0 or ref_b0 = 0, return 0; without conditional branch
  negs r1, r7         @ r1 = -a1

  # @ asrs rd, <#imm> --> #imm: 0-31
  # movs r4, #32
  asrs r1, r4         @ r1 = -a1 >> 32

  @ ref_b0 in r0 is not needed anymore, r0 can be used
  ands r0, r1         @ r0 = tmp = ref_b0 & (-a1 >> 32)
  negs r0, r0         @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r4         @ r0 = -tmp >> 32

  ands r5, r0         @ r0 = s & (-tmp >> 32)  = (a1*ref_b0)

  eors r5, r3         @ r5 = (a1*ref_b0)^rnd2

@@@@ r0 = -tmp >> 32, r1 = -a1 >> 32, r2 = &rnd, r3 = rnd2,
@@@@ r4 = 32, r5 = (a1*ref_b0)^rnd2, r6 = &b, r7 = a1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
387
388   mov  r3, r11        @ r3 = &table
389   ldrb r0, [r6, #8]   @ r0 = ref_b2
390
391   ldrb r6, [r3, r7]   @ r6 = table[a1]
392
393  ### From previous computation: r1 = -a1 >> 32, so r7 = a1 is not needed any more
394   ldrb r7, [r3, r0]   @ r7 = table[ref_b2]
395
396   adds r6, r7         @ r6 = s = table[a1] + table[ref_b2]
397
398   @@@@ Get the antilog: s = table[s+256]: g+s
399   # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
400   # @ movs rd, <#imm> --> #imm: 0-255
401   movs r7, #250       @ r7 = 250, 256 is not possible, so it is split: 250 + 6
402   adds r7, #6         @ r7 = 256
403   adds r6, r7         @ r6 = s + 256
404
405   ldrb r7, [r3, r6]   @ r7 = s = table[s+256]
406
407   ldrb r4, [r2, #16]   @ r4 = rnd4
408
409   # @@@@ Checking if a1 = 0 or ref_b2 = 0, return 0; without conditional branch
410   # negs r1, r4         @ r1 = -a1
411
412   # # @ asrs rd, <#imm> --> #imm: 0-31
413   # movs r6, #32
414   # asrs r1, r6         @ r1 = -a1 >> 32
415   ### From previous computation: r1 = -a1 >> 32
416   @ ref_b2 in r0 is not needed anymore, r0 can be used
417   ands r0, r1         @ r0 = tmp = ref_b2 & (-a1 >> 32)
418
419   movs r6, #32        @ r6 = 32     @ for avoiding stall
420
421   negs r0, r0         @ r0 = -tmp
422
423   # @ asrs rd, <#imm> --> #imm: 0-31
424   asrs r0, r6         @ r0 = -tmp >> 32
425
426   ands r7, r0         @ r6 = s & (-tmp >> 32)  = (a1*ref_b2)
427
428   mov  r1, r8         @ r1 = &c @ here for avoiding stall
429
```

```
  eors r7, r4          @ r6 = (a1*ref_b2)^rnd4

@@@@ r0 = -tmp >> 32, r1 = (-a1 >> 32), r2 = &rnd, r3 = &table,
@@@@ r4 = rnd2, r5 = (a1*ref_b0)^rnd2, r6 = 32, r7 = (a1*ref_b2)^rnd4,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
473   ldrb r5, [r3, r0]   @ r5 = table[ref_b0]
474
475   adds r1, r5         @ r1 = s = table[a2] + table[ref_b0]
476
477   @@@@ Get the antilog: s = table[s+256]: g+s
478   # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
479   # @ movs rd, <#imm> --> #imm: 0-255
480   movs r5, #250       @ r5 = 250, 256 is not possible, so it is split: 250 + 6
481   adds r5, #6         @ r5 = 256
482   adds r1, r5         @ r1 = s + 256
483
484   ldrb r5, [r3, r1]   @ r5 = s = table[s+256]
485   ldrb r3, [r2, #12]   @ r3 = rnd3
486
487   # @@@@ Checking if a2 = 0 or ref_b0 = 0, return 0; without conditional branch
488   negs r1, r4         @ r1 = -a2
489
490   # @ asrs rd, <#imm> --> #imm: 0-31
491   # movs r6, #32
492   asrs r1, r6         @ r1 = -a2 >> 32
493
494   @ ref_b0 in r0 is not needed anymore, r0 can be used
495   ands r0, r1         @ r0 = tmp = ref_b0 & (-a2 >> 32)
496   negs r0, r0         @ r0 = -tmp
497
498   # @ asrs rd, <#imm> --> #imm: 0-31
499   asrs r0, r6         @ r0 = -tmp >> 32
500
501   ands r5, r0         @ r0 = s & (-tmp >> 32)  = (a2*ref_b0)
502
503   eors r5, r3         @ r5 = (a2*ref_b0)^rnd3
504
505 @@@@ r0 = -tmp >> 32, r1 = -a2 >> 32, r2 = &rnd, r3 = rnd3,
506 @@@@ r4 = a2, r5 = (a2*ref_b0)^rnd3, r6 = 32, r7 = &b,
507 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
508 @@@@
```

```
516   ldrb r0, [r7, #4]   @ r0 = ref_b1
517
518   ldrb r7, [r3, r4]   @ r7 = table[a2]
519
520  ### From previous computation: r1 = -a2 >> 32, so r4 = a2 is not needed any more
521   ldrb r4, [r3, r0]   @ r4 = table[ref_b1]
522
523   adds r7, r4         @ r7 = s = table[a2] + table[ref_b1]
524
525   @@@@ Get the antilog: s = table[s+256]: g+s
526   # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
527   # @ movs rd, <#imm> --> #imm: 0-255
528   movs r4, #250       @ r4 = 250, 256 is not possible, so it is split: 250 + 6
529   adds r4, #6         @ r4 = 256
530   adds r7, r4         @ r7 = s + 256
531
532   ldrb r4, [r3, r7]   @ r4 = s = table[s+256]
533
534   ldrb r6, [r2, #16]   @ r6 = rnd4
535
536   # @@@@ Checking if a2 = 0 or ref_b1 = 0, return 0; without conditional branch
537   # negs r1, r4         @ r1 = -a2
538
539   # # @ asrs rd, <#imm> --> #imm: 0-31
540   # movs r7, #32
541   # asrs r1, r7         @ r1 = -a2 >> 32
542   ### From previous computation: r1 = -a2 >> 32
543   @ ref_b1 in r0 is not needed anymore, r0 can be used
544   ands r0, r1         @ r0 = tmp = ref_b1 & (-a2 >> 32)
545
546   movs r7, #32        @ r7 = 32     @ here for avoiding stall
547
548   negs r2, r0         @ r2 = -tmp   @ &rnd is not needed any more
549
550   # @ asrs rd, <#imm> --> #imm: 0-31
551   asrs r2, r7         @ r2 = -tmp >> 32
552
553   ands r4, r2         @ r4 = s & (-tmp >> 32)  = (a2*ref_b1)
554
555   mov  r1, r8         @ r1 = &c  @ here for avoiding stall
556
557   eors r4, r6         @ r4 = (a2*ref_b1)^rnd4
558
```

```
@@@@ r0 =  ref_b1 & (-a2 >> 32), r1 = -a2 >> 32, r2 = -tmp >> 32, r3 = &table,
@@@@ r4 = (a2*ref_b1)^rnd4, r5 = (a2*ref_b0)^rnd3, r6 = rnd2, r7 = 32,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

15.1.1 Experimental result:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $162\mu s$
Each clock is sampled with 125 points
Number of samples = 324×125 = 40500 samples

We can discard the first 54 cycles. And just measure 270 cycles.
The bytes of shares (a, b, c are not in a word)

15.1.1.1 Signal to Noise Ratio (SNR):

15.1.1.2 t-test:

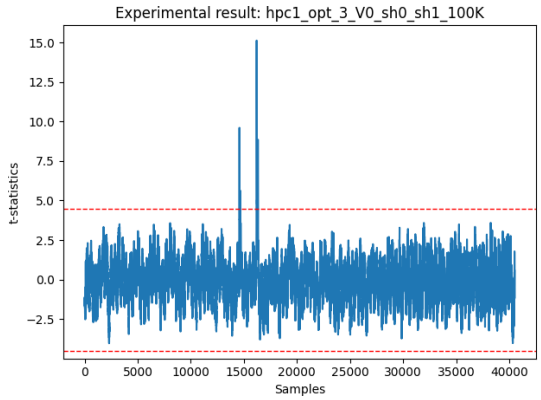
Finding leakages via splitting t-test in to three t-test s:

In this section we find the leakages caused by 2 shares out of 3 shares using the split-t-test.
For hpc1-opt-3-V0 with 100K traces, there are leakages at:
(sh_0, sh_1): cycles: 116, 117, 129, 130
129, 130: movs r6, #32
offset and register based address leakage. Listing 15.2
(sh_0, sh_2): cycles: 154, 155
154, 155: movs r6, #32
(sh_1, sh_2): No leakages
All these leakages are regarding: $r4 = a_0$ and $r6 = a_1$.
And are because of Glichty register access, multiplexer three, that load $r4 = a_0$.

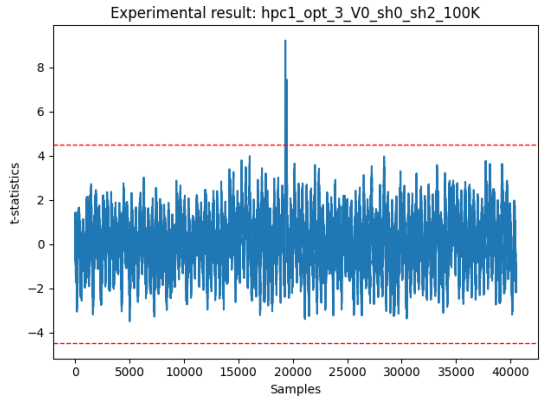
Listing 15.2: offset and registrar base address leakagee

```

  ldrb r1, [r3, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg1
  strb r0, [r0, #4]   @ clearing the write bus

@@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = &table
@@@@ r4 = a0, r5 = -tmp >> 32, r6 = a1, r7 = ref_b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

(a) t-test on (sh_0, sh_1)



(b) t-test on (sh_0, sh_2)



(c) t-test on (sh_1, sh_2)

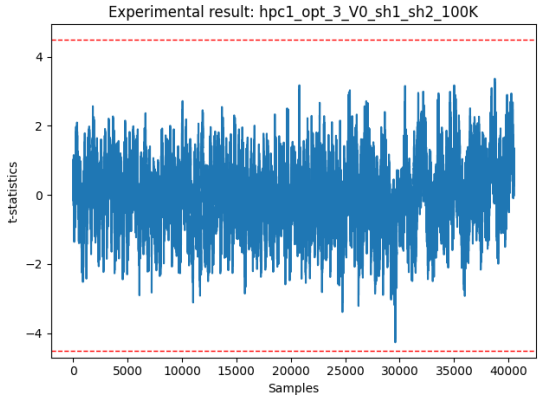
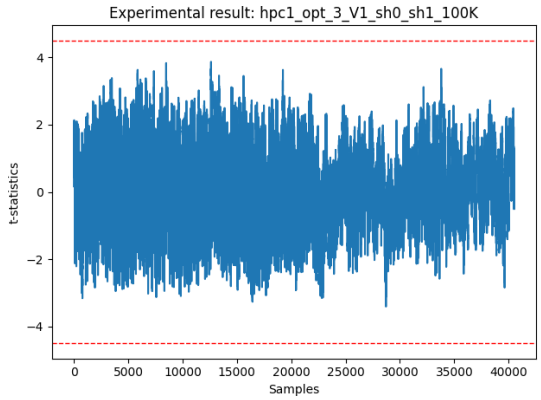
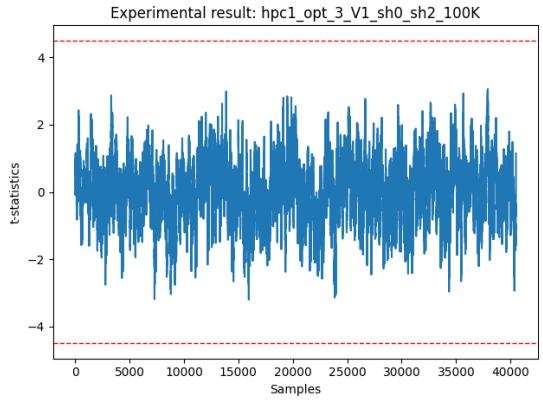


Figure 15.1: t-test_hpc1_opt3_V0_two_shares(Setup: 250MSa/s, 40500S, 2ns, 2MHz)

(a) t-test on (sh_0, sh_1)



(b) t-test on (sh_0, sh_2)



(c) t-test on (sh_1, sh_2)

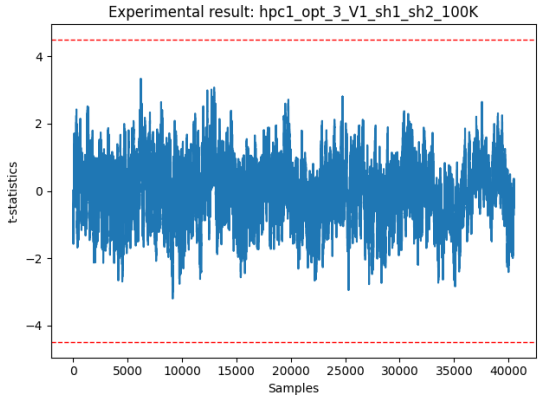


Figure 15.2: t-test_hpc1_opt3_V1_two_shares(Setup: 250MSa/s, 40500S, 2ns, 2MHz)

16. PINI1

PINI1 multiplication is introduced in [4]. Algorithm 13 represents this scheme for $d + 1$ shares.

Algorithm 13 PINI1

Require: shares $a_i, b_i (0 \leq i \leq d)$ such as $a = \oplus_{i=0}^{d} a_i$, $b = \oplus_{i=0}^{d} b_i$

Ensure: $c_0, ..., c_d$ such that $c = \oplus_{i=0}^{d} c_i = a.b$ **return** $c_0, ..., c_d$

17. PINI1-2

The PINI1-2 multiplication for two shares (PINI1-2) is illustrated in Algorithm 14 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 14 PINI1-2

Require: shares a_0, a_1, b_0, b_1, rnd

Ensure: c_0, c_1

1: $c_0 \leftarrow (a_0.b_0) \oplus ((a_0 \oplus 1).rnd) \oplus (a_0.(b_1 \oplus rnd))$

2: $c_1 \leftarrow (a_1.b_1) \oplus ((a_1 \oplus 1).rnd) \oplus (a_1.(b_0 \oplus rnd))$

In the implementation of PINI1-2, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

17.1 PINI1-2-no-function-call-0: basic code

Listing 17.1 demonstrates the first implementation of PINI1-2 based gfmul-s (PINI1-2-no-function-call-0). The instructions in Listing 17.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 17.1: PINI1-2-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @r8 = &c

@@@@@ Saving r0 (the address of the input a)
  mov  r9, r0        @ r9 = &a

@@@@@ Saving r1 (the address of the input b)
  mov  r10, r1       @ r10 = &b


@@@@ Saving the value 256 in r11. This decreases one cycle.
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r3, #250       @ r3 = 250, 256 is not possible, so it is split: 250 + 6
  adds r3, #6         @ r3 = 256
  mov  r11, r3        @ r11 = 256

@@@@@ Saving the address of the "table"
  ldr  r4, =table    @ r4 = &table, The address of the "table", for ai*bj multiplications


nop
nop
nop
nop

@@@@@@@@@@ Computing: c0 = (a0*b0)+[(a0+1)*rnd]+[a0*(b1+rnd)]

@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = 256
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = 256
@@@@@@@@@@
```

```
@@@@ c1 = (a1*b1)+[(a1+1)*rnd]+[a1*(b0+rnd)]

@@@@ r0 = &a, r1 = &b, r2 = rnd, r3 = 256
@@@@ r4 = &table, r5 = b0, r6 = a0, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = 256
@@@@
```

```
  eors r1, r1          @ r1 = 0, Clearing (a0*b0) from r1

@@@@ r0 = 1, r1 = 0, r2 = rnd, r3 = 32
@@@@ r4 = &table, r5 = &c, r6 = a0, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = 256
@@@@
```

```
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = 256
@@@@
```

```
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = 256
@@@@
```

```
  # @@@@ Checking if a1 = 0 or b1 = 0, return 0; without conditional branch
  negs r0, r6          @ r0 = -a1 hex(256-dec(a1))

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r5, #32         @ r5 = #32
  asrs r0, r5          @ r0 = -a1 >> 32

  # @@@@ b1 is not used in the rest computation of c1
  ands r7, r0          @ r7 = tmp = b1 & (-a1 >> 32)
  negs r7, r7          @ r7 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r7, r5          @ r7 = -tmp >> 32

  ands r7, r1          @ r7 = s & (-tmp >> 32) = (a1*b1)

  eors r1, r1          @ r1 = 0, clearing table[s+256]


@@@@ r0 = -a1 >> 32, r1 = 0, r2 = rnd, r3 = 256
@@@@ r4 = &table, r5 = 32, r6 = a1, r7 = a1*b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = 256
@@@@
```

```
  # @@@@ Checking if a1 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r6          @ r0 = -(a1+1) hex(256-dec((a1+1)))

  # @ asrs rd, <#imm> --> #imm: 0-31
  # movs r5, #32         @ r5 = #32
  asrs r0, r5          @ r0 = -(a1+1) >> 32

  ands r0, r2          @ r0 = tmp = rnd & (-(a1+1) >> 32)
  negs r0, r0          @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r5          @ r0 = -tmp >> 32

  ands r1, r0          @ r1 = s & (-tmp >> 32) = ((a1+1)*rnd)
  # mov  r3, r8          @ r3 = &c
  # strb r1, [r3, #4]    @ Storing ((a1+1)*rnd) in &c+#0
  # eors r1, r1          @ r1 = 0, Clearing ((a1+1)*rnd)


@@@@ r0 = -tmp >> 32, r1 = ((a1+1)*rnd), r2 = rnd, r3 = 256
@@@@ r4 = &table, r5 = 32, r6 = a1+1, r7 = (a1*b1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = 256
@@@@
```

```
#   # mov  r3, r11          @ r3 = 256

  adds r0, r3          @ r0 = s + 256

  ldrb r2, [r4, r0]    @ r2 = s = table[s+256]

  # @@@@ Checking if a1 = 0 or b0 = 0, return 0; without conditional branch
  negs r0, r6          @ r0 = -a1 hex(256-dec(a1))

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r3, #32         @ r3 = #32
  asrs r0, r3          @ r0 = -a1 >> 32

  ands r0, r5          @ r0 = tmp = (b0+rnd) & (-a1 >> 32)
  negs r0, r0          @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r3          @ r0 = -tmp >> 32

  ands r2, r0          @ r2 = s & (-tmp >> 32) = (a1 *(b0+rnd))


@@@@ r0 = -tmp >> 32, r1 = ((a1+1)*rnd), r2 = (a1 *(b0+rnd)), r3 = 32
@@@@ r4 = &table, r5 = b0+rnd, r6 = a1, r7 = (a1*b1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = 256
@@@@
```

17.1.1 Experimental result:

17.1.1.1 Setup:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $107\mu s$
Each clock is sampled with 125 points
Number of samples = 214×125 = 26750 samples

We can remove the first 54 cycles, $214 - 54 = 160$.
The bytes of shares (a, b, c are not in a word).

17.1.1.2 t-test:

Figure 17.1 shows the fixed-versus-random t-test results on both inputs a and b for 100K traces that there is no leakage.
No leakages were found by **uELMO**

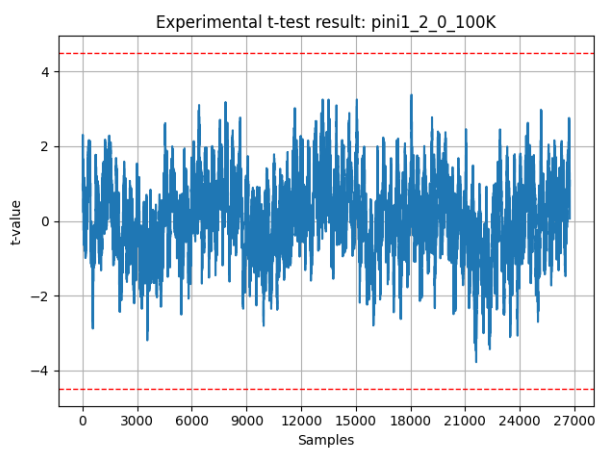


Figure 17.1: Experimental T-test result of pini1-2-0-without-b-0 (Setup: 250MSa/s, 26750KS, 2ns, 2MHz)

But for 300K traces there are leakages at cycles: With 160 cycles captured (20K samples):
Leaky cycles: [86, 87, 88]
Leaky cycles (from trigger): [72, 73, 74]
This leakage is because of:
Read bus: ldrb r5, [r0, #0], the context of the read bus will be $a_0.b_0$, then after several instructions, at ldrb r6, [r0, #4], a_1 is loaded and so that the context of the read bus will be a_1:
Read bus: $a_0.b_0 \rightarrow a_1$
The sources of this type of leakage are pointed in Listing 17.2.

Listing 17.2: Source of the leakage in pini1-2 for 300K traces

```
  ldrb r5, [r0, #0]   @ r5 = a0*b0

  eors r5, r3         @ r5 = c0 = (a0*b0)^[(a0^1)*rnd]^[a0*(b1^rnd)]
  mov  r3, r11        @ r3 = 256, for clearing [(a0^1)*rnd]^[a0*(b1^rnd)], and avoiding sta
  strb r5, [r0, #0]   @ r5 = c0, Storing c0
  strb r1, [r0, #4]   @ r1 = 0, Clearing the write bus
  eors r5, r5         @ r5 = 0, learing c0

@@@@ (a1*b1)

  mov  r0, r9       @ r0 = &a
  eors r7, r2       @ r7 = b1

  ldrb r6, [r0, #4] @ r6 = a1;

```

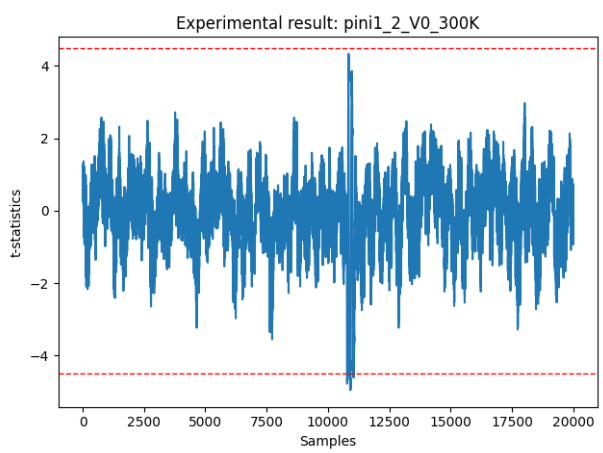


Figure 17.2: Experimental T-test result of pini1-2-V0 (Setup: 250MSa/s, 20KS, 2ns, 2MHz)

Patch: clearing the read bus Listing 17.3, using ldrb r1, [r4, #0].
Up to 500K traces (214 cycles) there is no leakage for pini1-2-V1, Figure 17.3.

Listing 17.3: Patching the leakage in pini1-2 for 300K traces

```

  ldrb r5, [r0, #0]   @ r5 = a0*b0

  ldrb r1, [r4, #0]   @ r1 = 0, Clearing the read bus, table[0] = 0, as it contains (a0*b0)


  eors r5, r3         @ r5 = c0 = (a0*b0)^[(a0^1)*rnd]^[a0*(b1^rnd)]
  mov  r3, r11        @ r3 = 256, for clearing [(a0^1)*rnd]^[a0*(b1^rnd)], and avoiding sta
  strb r5, [r0, #0]   @ r5 = c0, Storing c0
  strb r1, [r0, #4]   @ r1 = 0, Clearing the write bus
  eors r5, r5         @ r5 = 0, learing c0

@@@@ (a1*b1)

  mov  r0, r9       @ r0 = &a
  eors r7, r2       @ r7 = b1

  ldrb r6, [r0, #4] @ r6 = a1;

```

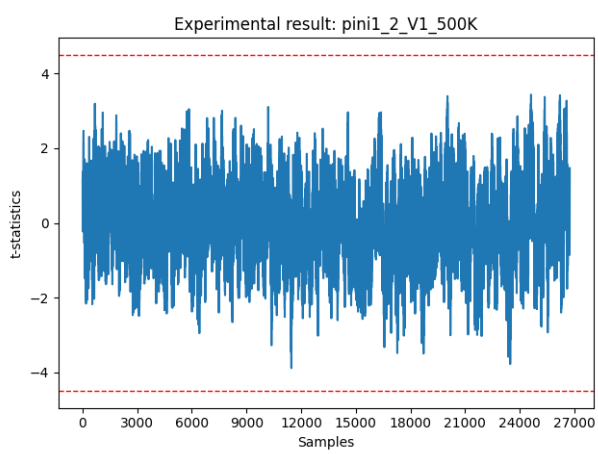


Figure 17.3: Experimental T-test result of pini1-2-V1 (Setup: 250MSa/s, 26750KS, 2ns, 2MHz)

18. PINI1-3- no function/label call

The PINI1-3 multiplication (three shares) is shown in Algorithm 15 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 15 PINI1-3

Require: shares $a_0, a_1, a_2, b_0, b_1, b_2, r_0 - r_3$

Ensure: c_0, c_1, c_2

1: $c_0 \leftarrow (a_0.b_0) \oplus [((a_0 \oplus 1).r_0) \oplus (a_0.(b_1 \oplus r_0))] \oplus [((a_0 \oplus 1).r_1) \oplus (a_0.(b_2 \oplus r_1))]$

2: $c_1 \leftarrow (a_1.b_1) \oplus [((a_1 \oplus 1).r_0) \oplus (a_1.(b_0 \oplus r_0))] \oplus [((a_1 \oplus 1).r_2) \oplus (a_1.(b_2 \oplus r_2))]$

3: $c_2 \leftarrow (a_2.b_2) \oplus [((a_2 \oplus 1).r_2) \oplus (a_2.(b_0 \oplus r_2))] \oplus [((a_2 \oplus 1).r_2) \oplus (a_2.(b_1 \oplus r_2))]$

In the implementation of PINI1-3, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

18.1 PINI1-3-no-function-call-0: basic code

Listing 18.1 demonstrates the first implementation of PINI1-3 based gfmul-s (PINI1-3-no-function-call-0). The instructions in Listing 18.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 18.1: PINI1-3-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @r8 = &c

@@@@@ Saving r0 (the address of the input a)
  mov  r9, r0        @ r9 = &a

@@@@@ Saving r1 (the address of the input b)
  mov  r10, r1       @ r10 = &b


  ldr  r4, =table    @ r4 = &taable, The address of the "table", for ai*bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r11, r4       @ r11 = &table

nop
nop
nop
nop

@@@@@ Loading arguments @@@@@@@@@@@@
  ldrb r3, [r0, #0]  @ r3 = a0; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
  ldrb r5, [r1, #0]  @ r5 = b0; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
  ldrb r6, [r0, #4]  @ r6 = a1; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
  ldrb r7, [r1, #4]  @ r7 = b1; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table

@@@@@@@@@@ Computing: ai*bi

@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = b0, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@@
```

```
43    # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
44    # @ movs rd, <#imm> --> #imm: 0-255
45    movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250 + 6
46    adds r1, #6         @ r1 = 256
47    adds r0, r1         @ r0 = s + 256
48
49    ldrb r1, [r4, r0]   @ r1 = s = table[s+256]
50
51    # @@@@ Checking if a0 = 0 or b0 = 0, return 0; without conditional branch
52    negs r0, r3         @ r0 = -a0 hex(256-dec(a0))
53
54    # @ asrs rd, <#imm> --> #imm: 0-31
55    movs r4, #32        @ r4 = #32
56    asrs r0, r4         @ r0 = -a0 >> 32
57
58    ands r5, r0         @ r5 = tmp = b0 & (-a0 >> 32)
59    negs r5, r5         @ r5 = -tmp
60
61    # @ asrs rd, <#imm> --> #imm: 0-31
62    asrs r5, r4         @ r5 = -tmp >> 32
63
64    ands r1, r5         @ r1 = s & (-tmp >> 32) = (a0*b0)
65
66    mov  r0, r8         @ r0 = &c
67    mov  r4, r11        @ r4 = &table
68    strb r1, [r0, #0]   @ r1 = (a0*b0), Storing (a0*b0) in &c+#0, r1 = (a0*b0)
69    strb r0, [r0, #4]   @ clearing the write bus
70
71  ### we need to clear the read bus and also reg1, istead of using two instructions
72    ## eors r1, r1         @ r1 = 0, clearing (a0*b0)
73    ## ldrb rx, [r4, #0]   @ rx = 0 = table[0],  Clearing the read bus
74  ### just we can use one as:
75    # ldrb r1, [r4, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg1
76  ### The one instruction, is not enough, we should use the both instruction,
77  ### because the first operand in one-inst can cause leakage with the next inst
78    # ex: r1 and r5, two shares from the same family
79      # ldrb r1, [r4, #0]   @ r5 = 0 = table[0],  Clearing the read bus and also reg5
80      # ldrb r5, [r4, r6]   @ r5 = table[a1]
81
82    eors r1, r1         @ r1 = 0, clearing (a0*b0)
83    ldrb r1, [r4, #0]   @ rx = 0 = table[0],  Clearing the read bus
84
85  @@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = a0,
```

```
@@@@ r4 = &table, r5 = -tmp >> 32, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  # eors r1, r1          @ r1 = 0, clearing (a1*b1)
  # ldrb rx, [r4, #0]    @ rx = 0 = table[0],  Clearing the read bus
  ldrb r1, [r4, #0]    @ r1 = 0 = table[0],  Clearing the read bus and also reg1

@@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = -tmp >> 32, r6 = 32, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  asrs r1, r6          @ r1 = -tmp >> 32

  ands r5, r1          @ r5 = s & (-tmp >> 32) = (a2*b2)

  mov  r1, r8          @ r1 = &c
  strb r5, [r1, #8]    @ r5 = (a2*b2), Storing (a2*b2) in c2, r1 = (a2*b2)

  @ b2 in r0 is not needed anymore, r0 can be used, or be cleared
  @ in a0+1, we need #1
  movs r0, #1          @ r0 = 1

  ## it is not possible to clear the write bus here
### we need to clear the read bus and also reg1, istead of using two instructions
  ## eors r5, r5          @ r5 = 0, clearing (a2*b2)
  ## ldrb rx, [r4, #0]    @ rx = 0 = table[0],  Clearing the read bus
### just we can use one as:
  # ldrb r5, [r4, #0]    @ r5 = 0 = table[0],  Clearing the read bus and also reg5
### The one instruction, is not enough, we should use the both instruction,
### because the first operand in one-inst can cause leakage with the next inst
  # ex: r5 and r3, two shares from the same family
    # ldrb r5, [r4, #0]    @ r5 = 0 = table[0],  Clearing the read bus and also reg5
    # eors r3, r0          @ r3 = a0+1

  eors r5, r5          @ r5 = 0, clearing (a2*b2)
  ldrb r5, [r4, r0]    @ r5 = 0 = table[r0=1],  Clearing the read bus and also reg5

@@@@@@@@@@ Computing: ci
@@@@ c0 = (a0*b0)+[((a0+1)*rnd0)+(a0*(b1+rnd0))]+[((a0+1)*rnd1)+(a0*(b2+rnd1))]
@@@@ c1 = (a1*b1)+[((a1+1)*rnd0)+(a1*(b0+rnd0))]+[((a1+1)*rnd2)+(a1*(b2+rnd2))]
@@@@ c2 = (a2*b2)+[((a2+1)*rnd1)+(a2*(b0+rnd1))]+[((a2+1)*rnd2)+(a2*(b1+rnd2))]


@@@@ r0 = 1, r1 = &c, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = 0, r6 = 32, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@
```

```
  adds r1, r5          @ r1 = s = table[a0+1] + table[rnd0]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r5, #250        @ r5 = 250, 256 is not possible, so it is split: 250 + 6
  adds r5, #6          @ r5 = 256
  adds r1, r5          @ r1 = s + 256

  ldrb r5, [r4, r1]    @ r5 = s = table[s+256]

  # @@@@ Checking if (a0+1) = 0 or rnd0 = 0, return 0; without conditional branch
  negs r1, r3          @ r1 = -(a0+1)

  # @ asrs rd, <#imm> --> #imm: 0-31
  # movs r6, #32
  asrs r1, r6          @ r1 = -(a0+1) >> 32

  ands r1, r0          @ r1 = tmp = rnd0 & (-(a0+1) >> 32)
  negs r1, r1          @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r6          @ r1 = -tmp >> 32

  ands r5, r1          @ r1 = s & (-tmp >> 32) = ((a0+1)*rnd0)


@@@@ r0 = rnd0, r1 = -tmp >> 32, r2 = &rnd, r3 = a0+1 ,
@@@@ r4 = &table, r5 = ((a0+1)*rnd0), r6 = 32, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r6, #250       @ r6 = 250, 256 is not possible, so it is split: 250 + 6
  adds r6, #6         @ r6 = 256
  adds r1, r6         @ r1 = s + 256

  ldrb r6, [r4, r1]   @ r6 = s = table[s+256]

  # @@@@ Checking if a0 = 0 or (b1+rnd0) = 0, return 0; without conditional branch
  negs r1, r3         @ r1 = -a0

  # @@ rnd0 is not needed any more
  # # @ asrs rd, <#imm> --> #imm: 0-31
  movs r0, #32        @ r4 = 32
  asrs r1, r0         @ r1 = -a0 >> 32

  @ (b1+rnd0) in r7 is not needed anymore, r7 can be used
  ands r7, r1         @ r7 = tmp = (b1+rnd0) & (-a0 >> 32)
  negs r7, r7         @ r7 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r7, r0         @ r7 = -tmp >> 32

  ands r6, r7         @ r6 = s & (-tmp >> 32)  = (a0*(b1+rnd0))

  @ in a0+1, we need #1
  movs r1, #1
@@@@ [((a0+1)*rnd0)+(a0*(b1+rnd0))]
  eors r5, r6         @ r5 = [((a0+1)*rnd0)+(a0*(b1+rnd0))]
  eors r6, r6

@@@@ r0 = 32, r1 = 1, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = [((a0+1)*rnd0)+(a0*(b1+rnd0))], r6 = 0, r7 = -tmp >> 32,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  adds r6, r7         @ r6 = s = table[a0+1] + table[rnd1]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r7, #250       @ r7 = 250, 256 is not possible, so it is split: 250 + 6
  adds r7, #6         @ r7 = 256
  adds r6, r7         @ r6 = s + 256

  ldrb r7, [r4, r6]   @ r7 = s = table[s+256]

  # @@@@ Checking if (a0+1) = 0 or rnd1 = 0, return 0; without conditional branch
  negs r6, r3         @ r6 = -(a0+1)

  # @ asrs rd, <#imm> --> #imm: 0-31
  # movs r0, #32
  asrs r6, r0         @ r6 = -(a0+1) >> 32

  ands r6, r1         @ r6 = tmp = rnd1 & (-(a0+1) >> 32)
  negs r6, r6         @ r6 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r6, r0         @ r6 = -tmp >> 32

  ands r7, r6         @ r6 = s & (-tmp >> 32) = ((a0+1)*rnd1)

@@@@ r0 = 32, r1 = rnd1, r2 = &rnd, r3 = a0+1 ,
@@@@ r4 = &table, r5 = [((a0+1)*rnd0)+(a0*(b1+rnd0))], r6 = -tmp >> 32, r7 = ((a0+1)*rnd1),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  adds r1, r6         @ r1 = s = table[a0] + table[b2+rnd1]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r6, #250       @ r6 = 250, 256 is not possible, so it is split: 250 + 6
  adds r6, #6         @ r6 = 256
  adds r1, r6         @ r1 = s + 256

  ldrb r6, [r4, r1]   @ r6 = s = table[s+256]

  # @@@@ Checking if a0 = 0 or (b2+rnd1) = 0, return 0; without conditional branch
  negs r1, r3         @ r1 = -a0

  # @@ rnd0 is not needed any more
  # # @ asrs rd, <#imm> --> #imm: 0-31
  # a0 in r3 is not needed anymore
  movs r3, #32        @ r3 = 32
  asrs r1, r3         @ r1 = -a0 >> 32

  @ (b2+rnd1) in r0 is not needed anymore, r0 can be used
  ands r0, r1         @ r0 = tmp = (b2+rnd1) & (-a0 >> 32)
  negs r0, r0         @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r3         @ r0 = -tmp >> 32

  ands r6, r0         @ r6 = s & (-tmp >> 32)  = (a0*(b2+rnd1))

@@@@ [((a0+1)*rnd1)+(a0*(b2+rnd1))]
  eors r7, r6         @ r7 = [((a0+1)*rnd1)+(a0*(b2+rnd1))]

@@@@ r0 = -tmp >> 32, r1 = -a0 >> 32, r2 = &rnd, r3 = 32,
@@@@ r4 = &table, r5 = [((a0+1)*rnd0)+(a0*(b1+rnd0))],
@@@@ r6 = (a0*(b2+rnd1)), r7 = [((a0+1)*rnd1)+(a0*(b2+rnd1))],
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  strb r3, [r6, #0]   @ r3 = 0, clearing the write bus which contains (a2*b2)

  eors r1, r5         @ r1 = (a0*b0)+[((a0+1)*rnd0)+(a0*(b1+rnd0))]
  mov  r5, r10        @ r5 = &b  @ here for avoiding stall
  eors r1, r7         @ r1 = c0
  mov  r7, r9         @ r7 = &a  @ here for avoiding stall

  strb r1, [r6, #0]   @ r1 = c0, Storing c0, r0 = c0
  eors r1, r1         @ r1 = 0, clearing C0 from r1
# here the ldrb is not needed as first rnd0 is loaded, so it clear the read bus
#  ldrb r1, [r4, #0]   @ r1 = 0, clearing c0 and the read bus which contains (a0*b0)


@@@@ r0 = -tmp >> 32, r1 = 0, r2 = &rnd, r3 = 0,
@@@@ r4 = &table, r5 = &b, r6 = &c, r7 = &a,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  movs r6, #32
  asrs r1, r6          @ r1 = -(a1+1) >> 32

  ands r1, r0          @ r1 = tmp = rnd0 & (-(a1+1) >> 32)
  negs r1, r1          @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r6          @ r1 = -tmp >> 32

  ands r5, r1          @ r1 = s & (-tmp >> 32) = ((a1+1)*rnd0)

@@@@ r0 = rnd0, r1 = -tmp >> 32, r2 = &rnd, r3 = a1+1 ,
@@@@ r4 = &table, r5 = ((a1+1)*rnd0), r6 = 32, r7 = &b,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  @ (b0+rnd0) in r7 is not needed anymore, r7 can be used
  ands r7, r1         @ r7 = tmp = (b0+rnd0) & (-a1 >> 32)
  negs r7, r7         @ r7 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r7, r0         @ r7 = -tmp >> 32

  ands r6, r7         @ r6 = s & (-tmp >> 32)  = (a1*(b0+rnd0))


  @ in a1+1, we need #1
  movs r1, #1
@@@@ [((a1+1)*rnd0)+(a1*(b0+rnd0))]
  eors r5, r6         @ r5 = [((a1+1)*rnd0)+(a1*(b0+rnd0))]
  eors r6, r6

@@@@ r0 = 32, r1 = 1, r2 = &rnd, r3 = a1,
@@@@ r4 = &table, r5 = [((a1+1)*rnd0)+(a1*(b0+rnd0))], r6 = 0, r7 = -tmp >> 32,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  # @ asrs rd, <#imm> --> #imm: 0-31
  # movs r0, #32
  asrs r6, r0         @ r6 = -(a1+1) >> 32

  ands r6, r1         @ r6 = tmp = rnd2 & (-(a1+1) >> 32)
  negs r6, r6         @ r6 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r6, r0         @ r6 = -tmp >> 32

  ands r7, r6         @ r6 = s & (-tmp >> 32) = ((a1+1)*rnd2)

@@@@ r0 = 32, r1 = rnd2, r2 = &rnd, r3 = a1+1 ,
@@@@ r4 = &table, r5 = [((a1+1)*rnd0)+(a1*(b1+rnd0))], r6 = -tmp >> 32, r7 = ((a1+1)*rnd2),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
559   # @@ rnd0 is not needed any more
560   # # @ asrs rd, <#imm> --> #imm: 0-31
561   # a1 in r3 is not needed anymore
562   movs r3, #32        @ r3 = 32
563   asrs r1, r3         @ r1 = -a1 >> 32
564
565   @ (b2+rnd2) in r0 is not needed anymore, r0 can be used
566   ands r0, r1         @ r0 = tmp = (b2+rnd2) & (-a1 >> 32)
567   negs r0, r0         @ r0 = -tmp
568
569   # @ asrs rd, <#imm> --> #imm: 0-31
570   asrs r0, r3         @ r0 = -tmp >> 32
571
572   ands r6, r0         @ r6 = s & (-tmp >> 32)  = (a1*(b2+rnd2))
573
574 @@@@ [((a1+1)*rnd2)+(a1*(b2+rnd2))]
575   eors r7, r6         @ r7 = [((a1+1)*rnd2)+(a1*(b2+rnd2))]
576
577 @@@@ r0 = -tmp >> 32, r1 = -a1 >> 32, r2 = &rnd, r3 = 32,
578 @@@@ r4 = &table, r5 = [((a1+1)*rnd0)+(a1*(b1+rnd0))],
579 @@@@ r6 = (a1*(b2+rnd2)), r7 = [((a1+1)*rnd2)+(a1*(b2+rnd2))],
580 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
581 @@@@
```

```
@@@@ r4 = &table, r5 = &b, r6 = &c, r7 = &a,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
@@@
```

```
  eors r6, r6

@@@@ r0 = 32, r1 = 1, r2 = &rnd, r3 = a2,
@@@@ r4 = &table, r5 = [((a2+1)*rnd1)+(a2*(b0+rnd1))], r6 = 0, r7 = -tmp >> 32,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
@@@
```

```
@@@@ [((a2+1)*rnd2)+(a2*(b1+rnd2))]
  eors r7, r6          @ r7 = [((a2+1)*rnd2)+(a2*(b1+rnd2))]

@@@@ r0 = -tmp >> 32, r1 = r1 = -a2 >> 32, r2 = &rnd, r3 = 32,
@@@@ r4 = &table, r5 = [((a2+1)*rnd1)+(a2*(b1+rnd1))],
@@@@ r6 = (a2*(b1+rnd2)), r7 = [((a2+1)*rnd2)+(a2*(b1+rnd2))],
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

18.1.1 Experimental result:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $227\mu s$
Each clock is sampled with 125 points
Number of samples = 454×125 = 56750 samples

We can discard the first 54 cycles. And just measure 400 cycles.
The bytes of shares (a, b, c are not in a word)

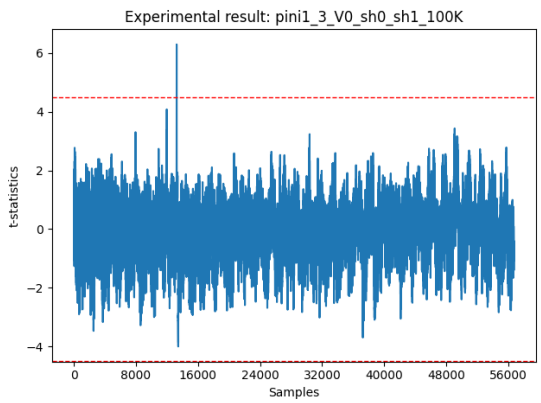
18.1.1.1 Signal to Noise Ratio (SNR):

18.1.1.2 t-test:

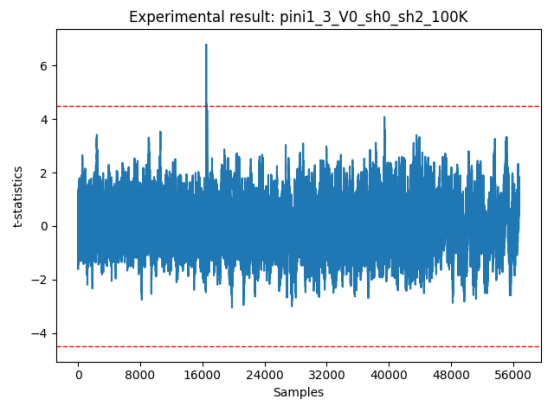
Finding leakages via splitting t-test in to three t-test s:

In this section we find the leakages caused by 2 shares out of 3 shares using the split-t-test .
For pini1-3-V0 with 100K traces, there are leakages at:
(sh_0, sh_1): cycles: 105, 106
(sh_0, sh_2): cycles: 131, 132
(sh_1, sh_2): cycles: 237, 247, 329, 344, 391
All these leakages are regarding: $r4 = a_0$ and $r6 = a_1$.

(a) t-test on (sh_0, sh_1)



(b) t-test on (sh_0, sh_2)



(c) t-test on (sh_1, sh_2)

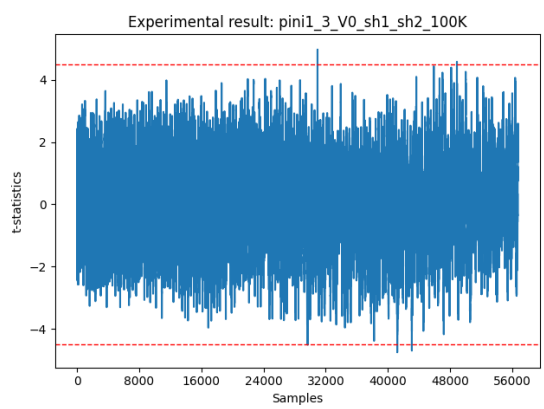


Figure 18.1: t-test_pini1_3_V0_two_shares(Setup: 250MSa/s, 56750S, 2ns, 2MHz)

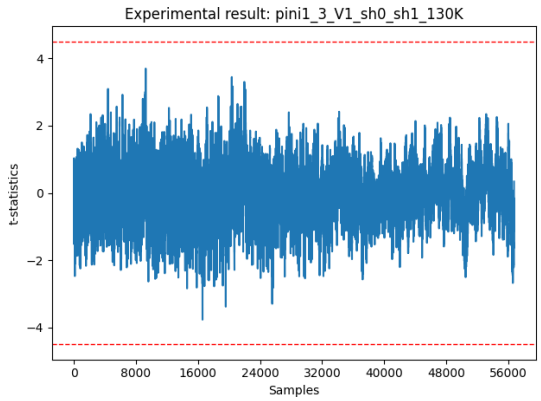
pini1-3-V1

Changing $r3$ and $r4$:

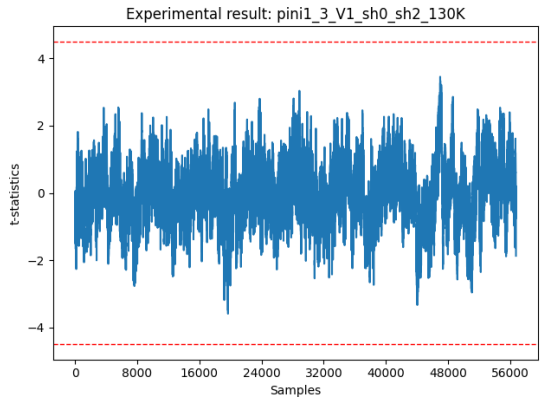
No leakages were found up to 130K traces.

No leakages were found by **uELMO**.

(a) t-test on (sh_0, sh_1)



(b) t-test on (sh_0, sh_2)



(c) t-test on (sh_1, sh_2)

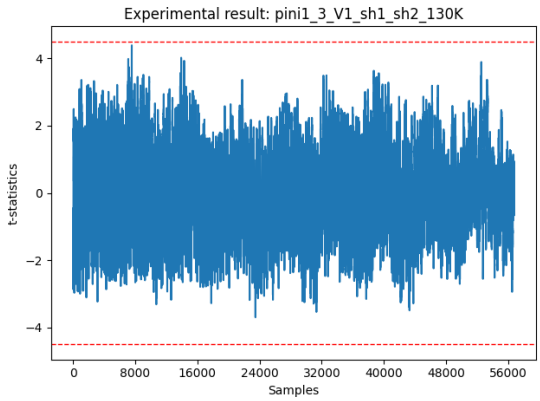


Figure 18.2: t-test_pini1_3_V1_two_shares(Setup: 250MSa/s, 56750S, 2ns, 2MHz)

19. PINI2

PINI2 multiplication is introduced in [3]. Algorithm 16 represents this scheme for $d + 1$ shares.

Algorithm 16 PINI2

Require: shares $a_i, b_i (0 \leq i \leq d)$ such as $a = \oplus_{i=0}^{d} a_i$, $b = \oplus_{i=0}^{d} b_i$

Ensure: $c_0, ..., c_d$ such that $c = \oplus_{i=0}^{d} c_i = a.b$ **return** $c_0, ..., c_d$

20. PINI2-2

The PINI2-2 multiplication for two shares (PINI2-2) is illustrated in Algorithm 17 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 17 PINI2-2

Require: shares $a_0, a_1, b_0, b_1, r_0 - r_2$

Ensure: c_0, c_1

In the implementation of PINI2-2, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

20.1 PINI2-2-no-function-call-0: basic code

Listing 20.1 demonstrates the first implementation of PINI2-2 based gfmul-s (PINI2-2-no-function-call-0). The instructions in Listing 20.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 20.1: PINI2-2-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @r8 = &c

@@@@@ Saving r0 (the address of the input a)
  mov  r9, r0        @ r9 = &a

@@@@@ Saving r1 (the address of the input b)
  mov  r10, r1       @ r10 = &b

@@@@@ Saving r2 (the address of the input rnd)
  mov  r11, r2       @ r11 = &rnd


@@@@ Saving the value 256 in r11. This decreases one cycle.
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r3, #250       @ r3 = 250, 256 is not possible, so it is split: 250 + 6
  adds r3, #6         @ r3 = 256
  # mov  r11, r3        @ r11 = 256

@@@@@ Saving the address of the "table"
  ldr  r4, =table    @ r4 = &table, The address of the "table", for ai*bj multiplications


nop
nop
nop
nop

@@@@@@@@@@ Computing: c0 = (a0*b0)+[(a0+1)*rnd]+[a0*(b1+rnd)]

@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = 256
@@@@ r4 = &table
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &rnd
@@@@@@@@@@
```

```

@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = 256
@@@@ r4 = &table, r5 = b0, r6 = a0,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &rnd
@@@@
```

```
@@@@ r0 = -a0 >> 32, r1 = (a0*b0), r2 = &rnd, r3 = 256
@@@@ r4 = &table, r5 = &c, r6 = a0, r7 = 32,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &rnd
@@@@ rnd: (s0) rnd0, (s1) rnd1, (r01) rnd2
@@@@
```

```
  eors r1, r2          @ r01+(a0*s01)


@@@@ r0 = -tmp >> 32, r1 = r01+(a0*s01), r2 = r01, r3 = 256
@@@@ r4 = &table, r5 = s01, r6 = a0, r7 = 32,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &rnd
@@@@
```

```
  asrs r0, r6          @ r0 = -tmp >> 32

  ands r2, r0          @ r2 = s & (-tmp >> 32) = (a0*(b1+s01))

  eors r1, r2          @ r1 = r01+(a0*s01)+[a0*(b1+s01)]

  eors r2, r2          @ r2 = 0, Clearing (a0*(b1+s01))



@@@@ r0 = -tmp >> 32, r1 = r01+(a0*s01)+[a0*(b1+s01)], r2 = 0, r3 = 256
@@@@ r4 = &table, r5 = s01, r6 = 32, r7 = 0,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &rnd
@@@@
```

```
  ands r7, r5          @ r7 = tmp = s01 & (-(b0) >> 32)
  negs r7, r7          @ r7 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r7, r6          @ r7 = -tmp >> 32

  ands r2, r7          @ r2 = s & (-tmp >> 32) = (b0*s01)

  eors r1, r2          @ r1 = r01+(a0*s01)+[a0*(b1+s01)]+(b0*s01)

  eors r2, r2          @ r2 = 0, clearing (b0*s01)

@@@@ r0 = b0, r1 = r01+(a0*s01)+[a0*(b1+s01)]+(b0*s01), r2 = 0, r3 = 256
@@@@ r4 = &table, r5 = s01, r6 = 32, r7 = -tmp >> 32,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &rnd
@@@@
```

```
  asrs r7, r0          @ r7 = -b0 >> 32

  ands r5, r7          @ r5 = tmp = (a1+s01) & (-b0 >> 32)
  negs r5, r5          @ r5 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r5, r0          @ r5 = -tmp >> 32

  ands r6, r5          @ r6 = s & (-tmp >> 32) = (b0*(a1+s01))

  eors r1, r6          @ r1 = r01+(a0*s01)+[a0*(b1+s01)]+(b0*s01)+(b0*(a1+s01))



@@@@ r0 = 32, r1 = r01+(a0*s01)+[a0*(b1+s01)]+(b0*s01)+(b0*(a1+s01)), r2 = a1, r3 = 256
@@@@ r4 = &table, r5 = -tmp >> 32, r6 = (b0*(a1+s01)), r7 = -b0 >> 32,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &rnd
@@@@
```

```
@@@
```

```
strb r1, [r6, #4]   @ r1 = (a1*b1), Storing (a1*b1) in &c+#0


nop
nop
nop
nop
```

20.1.1 Experimental result:

20.1.1.1 Setup:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $109\mu s$
Each clock is sampled with 125 points
Number of samples = 218×125 = 27250 samples
The bytes of shares (a, b, c are not in a word)
Figure **??** shows the recorded 27250S before raising-edge of the trigger.

20.1.1.2 t-test:

Figure 20.1 shows the fixed-versus-random t-test results on both inputs a and b for 100K traces that there is no leakage.
No leakages were found by **uELMO**.

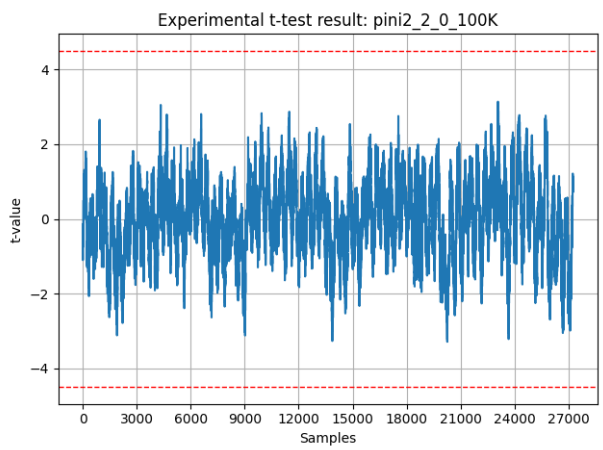


Figure 20.1: Experimental T-test result of pini2-2-0-without-b-0 (Setup: 250MSa/s, 27250S, 2ns, 2MHz)

Also as it is shown in Figure 20.2 there is no leakage for 300K traces.

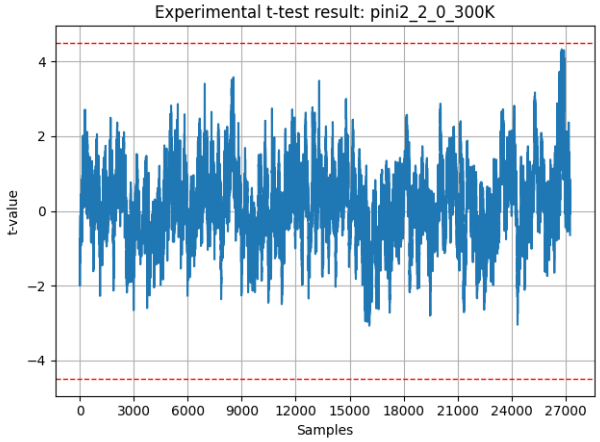


Figure 20.2: Experimental T-test result of pini2-2-0-without-b-0 (Setup: 250MSa/s, 27250S, 2ns, 2MHz)

21. PINI2-3- no function/label call

The PINI2-3 multiplication (three shares) is shown in Algorithm 18 where the inputs/outputs of the gadget are one byte (8-bit value) and also $a_i.b_j$ is the GF multiplication.

Algorithm 18 PINI2-3

Require: shares $a_0, a_1, a_2, b_0, b_1, b_2, r_0 - r_3$

Ensure: c_0, c_1, c_2

1: $s_{01} = s_0 \oplus s_1 = r_0 \oplus r_1$
2: $p^0_{01} = a_0.s_{01}$
3: $p^1_{01} = a_0.(b_1 \oplus s_{01})$
4: $p^2_{01} = b_0.s_{01}$
5: $p^3_{01} = b_0.(a_1 \oplus s_{01})$
6: $s_{02} = s_0 \oplus s_2 = r_0 \oplus r_2$
7: $p^0_{02} = a_0.s_{02}$
8: $p^1_{02} = a_0.(b_2 \oplus s_{02})$
9: $p^2_{02} = b_0.s_{02}$
10: $p^3_{02} = b_0.(a_2 \oplus s_{02})$
11: $s_{12} = s_1 \oplus s_2 = r_1 \oplus r_2$
12: $p^0_{12} = a_1.s_{12}$
13: $p^1_{12} = a_1.(b_2 \oplus s_{12})$
14: $p^2_{12} = b_1.s_{12}$
15: $p^3_{12} = b_1.(a_2 \oplus s_{12})$
16: $c_0 \leftarrow (a_0.b_0) \oplus (r_0 \oplus p^0_{02} \oplus p^1_{02} \oplus p^2_{02} \oplus p^3_{02} \oplus r_4 \oplus p^0_{01} \oplus p^1_{01} \oplus^2_{01} \oplus p^3_{01})$
17: $c_1 \leftarrow [(a_1.b_1) \oplus (r_5 \oplus p^0_{12} \oplus p^1_{12} \oplus p^2_{12} \oplus p^3_{12})] \oplus r_4)$
18: $c_2 \leftarrow ((a_2.b_2) \oplus r_5) \oplus r_3$

In the implementation of PINI2-3, for the GF multiplications ($a_i.b_j$), the gfmul-s based on the sequence of instructions (Listing 2.3) is used.

21.1 PINI2-3-no-function-call-0: basic code

Listing 21.1 demonstrates the first implementation of PINI2-3 based gfmul-s (PINI2-3-no-function-call-0). The instructions in Listing 21.1 is put in the part Gadget_Instructions in .S file (Listing 1.1).

Listing 21.1: PINI2-3-no-function-call-0

```
@@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = &c

@@@@@ Saving r3 (the address of the output)
  mov  r8, r3        @r8 = &c

@@@@@ Saving r0 (the address of the input a)
  mov  r9, r0        @ r9 = &a

@@@@@ Saving r1 (the address of the input b)
  mov  r10, r1       @ r10 = &b

#@@@@@ Saving r2 (the address of the input rnd)
#  mov  r11, r2       @ r11 = &rnd

@@@@@ Saving the address of the "table"
  ldr  r4, =table    @ r4 = &table, The address of the "table", for ai*bj multiplications
@@@@ Reducing the use of LDR instruction: ldr rx,=table @ rx = &table; --> mov rx, r11
  mov  r11, r4       @ r11 = &table

nop
nop
nop
nop


@@@@@ Loading arguments @@@@@@@@@@@@
  ldrb r3, [r0, #0]  @ r3 = a0; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
  ldrb r5, [r1, #0]  @ r5 = b0; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
  ldrb r6, [r0, #4]  @ r6 = a1; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table
  ldrb r7, [r1, #4]  @ r7 = b1; r0 = &a, r1 = &b, r2 = &rnd, r4 = &table

@@@@@@@@@@ Computing: ai*bi

@@@@ r0 = &a, r1 = &b, r2 = &rnd, r3 = a0,
@@@@ r4 = &table, r5 = b0, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@@
```

```
43
44    adds r0, r1          @ r0 = s = table[a0]+table[b0]
45
46    @@@@ Get the antilog: s = table[s+256]: g+s
47    # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
48    # @ movs rd, <#imm> --> #imm: 0-255
49    movs r1, #250        @ r1 = 250, 256 is not possible, so it is split: 250+6
50    adds r1, #6          @ r1 = 256
51    adds r0, r1          @ r0 = s+256
52
53    ldrb r1, [r4, r0]    @ r1 = s = table[s+256]
54
55    # @@@@ Checking if a0 = 0 or b0 = 0, return 0; without conditional branch
56    negs r0, r3          @ r0 = -a0 hex(256-dec(a0))
57
58    # @ asrs rd, <#imm> --> #imm: 0-31
59    movs r4, #32         @ r4 = #32
60    asrs r0, r4          @ r0 = -a0 >> 32
61
62    ands r0, r5          @ r0 = tmp = b0 & (-a0 >> 32)
63    negs r0, r0          @ r0 = -tmp
64
65    # @ asrs rd, <#imm> --> #imm: 0-31
66    asrs r0, r4          @ r0 = -tmp >> 32
67
68    ands r1, r0          @ r1 = s & (-tmp >> 32) = (a0*b0)
69
70    mov  r0, r8          @ r0 = &c
71    @ b0 in r5 is not needed anymore
72    mov  r5, r11         @ r5 = &table    @ here for avoiding stall
73    strb r1, [r0, #0]    @ r1 = (a0*b0), Storing (a0*b0) in &c+#0, r1 = (a0*b0)
74
75    eors r1, r1          @ r1 = 0, clearing (a0*b0) @ here to avoid stall
76
77    strb r0, [r0, #4]    @ clearing the write bus
78
79  ### we need to clear the read bus and also reg1, istead of using two instructions
80    ## eors r1, r1          @ r1 = 0, clearing (a0*b0)
81    ## ldrb rx, [r5, #0]    @ rx = 0 = table[0],  Clearing the read bus
82  ### just we can use one as:
83    # ldrb r1, [r5, #0]    @ r1 = 0 = table[0],  Clearing the read bus and also reg1
84  ### The one instruction, is not enough, we should use the both instruction,
85  ### because the first operand in one-inst can cause leakage with the next inst
```

```
  # ex: r1 and r0, two shares from the same family
    # ldrb r1, [r5, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg5
    # ldrb r0, [r5, r6]   @ r0 = table[a1]
  # eors r1, r1         @ r1 = 0, clearing (a0*b0) @ above to avoid stall
  ldrb r1, [r5, #0]   @ rx = 0 = table[0],  Clearing the read bus

@@@@ r0 = &c, r1 = 0, r2 = &rnd, r3 = a0,
@@@@ r4 = 32, r5 = &table, r6 = a1, r7 = b1,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  ands r1, r7         @ r1 = s & (-tmp >> 32) = (a1*b1)

  mov  r7, r8         @ r7 = &c
  # mov  r5, r11        @ r5 = &table
  strb r1, [r7, #4]   @ r1 = (a1*b1), Storing (a1*b1) in &c+#4, r1 = (a1*b1)
  strb r7, [r7, #8]   @ clearing the write bus

### we need to clear the read bus and also reg1, istead of using two instructions, just we
  # eors r1, r1         @ r1 = 0, clearing (a1*b1)
  # ldrb rx, [r5, #0]   @ rx = 0 = table[0],  Clearing the read bus
  ldrb r1, [r5, #0]   @ r1 = 0 = table[0],  Clearing the read bus and also reg1

@@@@ r0 = -a1 >> 32, r1 = 0, r2 = &rnd, r3 = a0,
@@@@ r4 = 256, r5 = &table, r6 = 32, r7 = &c,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r6, #32         @ r6 = 32
  asrs r1, r6          @ r1 = -a2 >> 32

  ands r1, r0          @ r1 = tmp = b2 & (-a2 >> 32)
  negs r1, r1          @ r1 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r1, r6          @ r1 = -tmp >> 32

  ands r7, r1          @ r7 = s & (-tmp >> 32) = (a2*b2)

  mov  r4, r8          @ r4 = &c
  strb r7, [r4, #8]    @ r7 = (a2*b2), Storing (a2*b2) in c2, r1 = (a2*b2)
  ## it is not possible to clear the write bus here

### we need to clear the read bus and also reg7, istead of using two instructions
  ## eors r7, r7          @ r7 = 0, clearing (a2*b2)
  ## ldrb rx, [r5, #0]    @ rx = 0 = table[0],  Clearing the read bus
### as rnd0 and rnd2 are loaded into r7 and r1, reg7 and the read bus are cleared

##########################################################################################
@@@@@@@@@@ Computing: ci
@@@@ c0 = (a0*b0)+(r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01+p1_01+p2_01+p3_01)
@ c0 = (a0*b0)+[rnd3+(a0*s02)+(a0*(b2+s02))+(b0*s02)+(b0*(a2+s02))+rnd4+(a0*s01)+(a0*(b1+s0
@@@@ c1 = [(a1*b1)+(r12(=rnd5)+p0_12+p1_12+p2_12+p3_12)]+r1(=rnd4)
@@@@ c2 = [(a2*b2)+r12(=rnd5))]+r02(=rnd3)

@@@@ r0 = b2, r1 = -tmp >> 32, r2 = &rnd, r3 = a0,
@@@@ r4 = &c, r5 = &table, r6 = 32, r7 = (a2*b2),
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@@@@@@@
```

```
  ldrb r1, [r2, #8]   @ r1 = rnd2

  eors r7, r1         @ r7 = rnd0+rnd2 = s02
  ldrb r6, [r5, r3]   @ r6 = table[a0]
  ldrb r1, [r5, r7]   @ r1 = table[s02]

  adds r6, r1         @ r6 = s = table[a0]+table[s02]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r1, #250       @ r1 = 250, 256 is not possible, so it is split: 250+6
  adds r1, #6         @ r1 = 256
  adds r6, r1         @ r6 = s+256

  ldrb r1, [r5, r6]   @ r1 = s = table[s+256]
  ldrb r4, [r2, #12]  @ r4 = rnd3   @ it also clears the read bus wich contains (a0*s02)

  # @@@@ Checking if a0 = 0 or s02 = 0, return 0; without conditional branch
  negs r6, r3         @ r6 = -a0 hex(256-dec(a0))

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r5, #32        @ r5 = #32
  asrs r6, r5         @ r6 = -a0 >> 32

  ands r6, r7         @ r6 = tmp = s02 & (-a0 >> 32)
  negs r6, r6         @ r6 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r6, r5         @ r6 = -tmp >> 32

  ands r1, r6         @ r1 = s & (-tmp >> 32) = p0_02= (a0*s02)

  # mov  r5, r11        @ r5 = &table, @ here to avoid stall
  eors r1, r4         @ r1 = rnd3+(a0*s02)=r02(=rnd3)+p0_02

### it is better to clear the read bus, as they are (a0*s02)
### we need to clear the read bus which contains (a0*s02)
  ## but it was done by:
  # ldrb r4, [r2, #12]  @ r4 = rnd3
### no need to clear r4 = rnd3

  mov  r5, r11        @ r5 = &table, @ here to clear the piplines registers and decoding po
```

```
@@@@ r0 = b2, r1 = r02(=rnd3)+p0_02, r2 = &rnd, r3 = a0,
@@@@ r4 = rnd3, r5 = &table, r6 = -tmp >> 32, r7 = s02,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
# (b2+s02) in r0 is not needed anymore
  mov  r0, r10        @ r0 = &b, for clearing r0 = (b2+s02) and for loading b0 later, @ hea

  ands r4, r6         @ r4 = s & (-tmp >> 32) = p1_02= (a0*(b2+s02))

  eors r1, r4         @ r1 = r02(=rnd3)+p0_02+p1_02

### It is better to clear the read bus and also reg4, as they are (a0*(b2+s02)), which seem
  eors r4, r4          @ r4 = 0, clearing (a0*(b2+s02))
  ldrb r4, [r5, #0]    @ rx = 0 = table[0],  Clearing the read bus

@@@@ r0 = &b, r1 = r02(=rnd3)+p0_02+p1_02, r2 = &rnd, r3 = 32,
@@@@ r4 = 0, r5 = &table, r6 = -tmp >> 32, r7 = s02,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  # @ asrs rd, <#imm> --> #imm: 0-31
  # movs r3, #32        @ r3 = #32
  asrs r0, r3         @ r0 = -b0 >> 32

  ands r0, r7         @ r0 = tmp = s02 & (-b0 >> 32)
  negs r0, r0         @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r3         @ r0 = -tmp >> 32

  ands r4, r0         @ r4 = s & (-tmp >> 32) = p2_02= (b0*s02)

  eors r1, r4        @ r1 = r02(=rnd3)+p0_02+p1_02+p2_02

### it is better to clear the read bus and also reg4, as they are (b0*s02), which seems ran
  # eors r4, r4          @ r4 = 0, clearing (b0*s02)
  mov  r4, r9         @ r4 = &a, for loading a2 later, @ hear to avoid stall,

  ldrb r0, [r5, #0]    @ rx = 0 = table[0],  Clearing the read bus

@@@@ r0 = 0, r1 = r02(=rnd3)+p0_02+p1_02+p2_02, r2 = &rnd, r3 = 32,
@@@@ r4 = &a, r5 = &table, r6 = b0, r7 = s02,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  ldrb r4, [r5, r3]   @ r4 = s = table[s+256]

  # @@@@ Checking if b0 = 0 or (a2+s02) = 0, return 0; without conditional branch
  negs r3, r6         @ r3 = -b0 hex(256-dec(b0))

@ b0 in r6 is not needed anymore
  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r6, #32        @ r6 = #32
  asrs r3, r6         @ r3 = -b0 >> 32

  ands r3, r0         @ r3 = tmp = (a2+s02) & (-b0 >> 32)
  negs r3, r3         @ r3 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r3, r6         @ r3 = -tmp >> 32

# (a2+s02) in r0 is not needed anymore

  mov  r0, r9        @ r0 = &a, for clearing r0 = (a2+s02) and for loading b0 later, @ hear

  ands r4, r3         @ r4 = s & (-tmp >> 32) = p3_02= (b0*(a2+s02))

  eors r1, r4         @ r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02

### it is better to clear the read bus and also reg4, as they are (b0*(a2+s02)), which seem
  eors r4, r4          @ r4 = 0, clearing (b0*(a2+s02)

  # not needed, beacuse of the next instruction
  # ldrb r4, [r5, #0]    @ rx = 0 = table[0],  Clearing the read bus

##############################################################################
@@@@ r0 = &a, r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02, r2 = &rnd, r3 = -tmp >> 32,
@@@@ r4 = 0, r5 = &table, r6 = 32, r7 = s02,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
@@@@ p0_01= a0*s01

  ldrb r7, [r2, #0]   @ r7 = rnd0
  ldrb r3, [r0, #0]   @ r3 = a0
  ldrb r0, [r2, #4]   @ r0 = rnd1

  eors r7, r0         @ r7 = rnd0+rnd1 = s01
  ldrb r6, [r5, r3]   @ r6 = table[a0]
  ldrb r0, [r5, r7]   @ r0 = table[s01]


  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r0, #250       @ r0 = 250, 256 is not possible, so it is split: 250+6
  adds r0, #6         @ r0 = 256
  adds r6, r0         @ r6 = s+256

  ldrb r0, [r5, r6]   @ r0 = s = table[s+256]
  ldrb r4, [r2, #16]  @ r4 = rnd4   @ it also clears the read bus wich contains (a0*s01)

  # @@@@ Checking if a0 = 0 or s01 = 0, return 0; without conditional branch
  negs r6, r3         @ r6 = -a0 hex(256-dec(a0))

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r5, #32        @ r5 = #32
  asrs r6, r5         @ r6 = -a0 >> 32

  ands r6, r7         @ r6 = tmp = s01 & (-a0 >> 32)
  negs r6, r6         @ r6 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r6, r5         @ r6 = -tmp >> 32

  ands r0, r6         @ r0 = s & (-tmp >> 32) = p0_01=(a0*s01)

  # mov  r5, r11        @ r5 = &table, @ here to avoid stall
#  @ r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02
  eors r1, r4         @ r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)

  mov  r4, r10        @ r0 = &b @ here for avoidind stall

### we need to clear the read bus which contains (a0*s01)
  ## but it was done by:
```

```
  # ldrb r4, [r2, #12]  @ r4 = rnd3
### clear r0 = p0_01=(a0*s01)
  eors r0, r0         @ r0 = 0

  mov  r5, r11        @ r5 = &table, @ here to clear the piplines registers and decoding po

@@@@ r0 = 0, r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01
@@@@ r2 = &rnd, r3 = a0,
@@@@ r4 = &b, r5 = &table, r6 = -tmp >> 32, r7 = s01,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  ands r6, r0          @ r6 = tmp = (b1+s01) & (-a0 >> 32)
  negs r6, r6          @ r6 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r6, r3          @ r6 = -tmp >> 32

# (b1+s01) in r0 is not needed anymore
  mov  r0, r10         @ r0 = &b, for clearing r0 = (b1+s01) and for loading b0 later, @ hea

  ands r4, r6          @ r4 = s & (-tmp >> 32) = p1_01 = (a0*(b1+s01))

# r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01
  eors r1, r4          @ r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01+p1_01

### It is better to clear the read bus and also reg4, as they are (a0*(b1+s01)), which seem
  eors r4, r4           @ r4 = 0, clearing (a0*(b1+s01))
  ldrb r4, [r5, #0]     @ rx = 0 = table[0],  Clearing the read bus

@@@@ r0 = &b, r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01+p1_01
@@@@ r2 = &rnd, r3 = 32,
@@@@ r4 = 0, r5 = &table, r6 = -tmp >> 32, r7 = s01,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  movs r4, #250       @ r4 = 250, 256 is not possible, so it is split: 250+6
  adds r4, #6         @ r4 = 256
  adds r0, r4         @ r0 = s+256

  ldrb r4, [r5, r0]   @ r4 = s = table[s+256]

  # @@@@ Checking if b0 = 0 or s01 = 0, return 0; without conditional branch
  negs r0, r6         @ r0 = -b0 hex(256-dec(b0))

  # @ asrs rd, <#imm> --> #imm: 0-31
  # movs r3, #32        @ r3 = #32
  asrs r0, r3         @ r0 = -b0 >> 32

  ands r0, r7         @ r0 = tmp = s01 & (-b0 >> 32)
  negs r0, r0         @ r0 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r0, r3         @ r0 = -tmp >> 32

  ands r4, r0         @ r4 = s & (-tmp >> 32) = p2_01= b0*s01

# @ r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01+p1_01
  eors r1, r4        @ r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01+p1_01+p2_01

### it is better to clear the read bus and also reg4, as they are (b0*s01), which seems ran
  # eors r4, r4          @ r4 = 0, clearing (b0*s01)
  mov  r4, r9         @ r4 = &a, for loading a1 later, @ hear to avoid stall,

  ldrb r0, [r5, #0]    @ rx = 0 = table[0],  Clearing the read bus

@@@@ r0 = 0, r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01+p1_01+p2_01,
@@@@ r2 = &rnd, r3 = 32,
@@@@ r4 = &a, r5 = &table, r6 = b0, r7 = s01,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  ldrb r3, [r5, r6]   @ r3 = table[b0]
  ldrb r4, [r5, r7]   @ r4 = table[(a1+s01)]

  adds r3, r4         @ r3 = s = table[b0]+table[(a1+s01)]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r4, #250       @ r4 = 250, 256 is not possible, so it is split: 250+6
  adds r4, #6         @ r4 = 256
  adds r3, r4         @ r3 = s+256

  ldrb r4, [r5, r3]   @ r4 = s = table[s+256]

  # @@@@ Checking if b0 = 0 or (a1+s01) = 0, return 0; without conditional branch
  negs r3, r6         @ r3 = -b0 hex(256-dec(b0))

@ b0 in r6 is not needed anymore
  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r6, #32        @ r6 = #32
  asrs r3, r6         @ r3 = -b0 >> 32

  ands r3, r7         @ r3 = tmp = (a1+s01) & (-b0 >> 32)
  negs r3, r3         @ r3 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r3, r6         @ r3 = -tmp >> 32

# (a1+s01) in r7 is not needed anymore
  mov  r7, r8        @ r7 = &c, for clearing r7 = (a1+s01) and for loading (a0*b0) later, @

  ands r4, r3         @ r4 = s & (-tmp >> 32) = p3_01 = (b0*(a1+s01))

# @ r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01+p1_01+p2_01,
  eors r1, r4         @ r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01+p1_01+p2_01

### it is better to clear the read bus and also reg4, as they are (b0*(a1+s01)), which seem
  eors r4, r4          @ r4 = 0, clearing (b0*(a1+s01)
  ldrb r4, [r5, #0]    @ rx = 0 = table[0],  Clearing the read bus
# ##############################################################################
@@@@ r0 = a1, r1 = r02(=rnd3)+p0_02+p1_02+p2_02+p3_02+r1(=rnd4)+p0_01+p1_01+p2_01+p3_01,
@@@@ r2 = &rnd, r3 = -tmp >> 32,
@@@@ r4 = 0, r5 = &table, r6 = 32, r7 = &c,
```

```
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  adds r6, r1          @ r6 = s+256

  ldrb r1, [r5, r6]    @ r1 = s = table[s+256]
  ldrb r4, [r2, #20]   @ r4 = rnd5   @ it also clears the read bus wich contains (a1*s12)

  # @@@@ Checking if a1 = 0 or s12 = 0, return 0; without conditional branch
  negs r6, r0          @ r6 = -a1 hex(256-dec(a1))

  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r3, #32         @ r3 = #32
  asrs r6, r3          @ r6 = -a1 >> 32

  ands r6, r7          @ r6 = tmp = s12 & (-a1 >> 32)
  negs r6, r6          @ r6 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r6, r3          @ r6 = -tmp >> 32

  ands r1, r6          @ r1 = s & (-tmp >> 32) = p0_12= (a1*s12)

  mov  r6, r10         @ r6 = &b, @ here to avoid stall

  eors r1, r4          @ r1 = rnd5+(a1*s12)=r12(=rnd5)+p0_12

### it is better to clear the read bus, as they are (a1*s12)
### we need to clear the read bus which contains (a1*s12)
  ## but it was done by:
  # ldrb r4, [r2, #20]  @ r4 = rnd5
### no need to clear as it is r4 = rnd5

@@@@ r0 = a1, r1 = r12(=rnd5)+p0_02, r2 = &rnd, r3 = 32,
@@@@ r4 = rnd5, r5 = &table, r6 = &b, r7 = s12,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
  eors r3, r7          @ r3 = b2+s12
  ldrb r6, [r5, r0]    @ r6 = table[a1]
  ldrb r4, [r5, r3]    @ r4 = table[(b2+s12)]

  adds r6, r4          @ r6 = s = table[a1]+table[(b2+s12)]

  @@@@ Get the antilog: s = table[s+256]: g+s
  # @ adds rd, <#imm> --> #imm: 3 bits, 0-7
  # @ movs rd, <#imm> --> #imm: 0-255
  movs r4, #250        @ r4 = 250, 256 is not possible, so it is split: 250+6
  adds r4, #6          @ r4 = 256
  adds r6, r4          @ r6 = s+256

  ldrb r4, [r5, r6]    @ r4 = s = table[s+256]

  # @@@@ Checking if a1 = 0 or (b2+s12) = 0, return 0; without conditional branch
  negs r6, r0          @ r6 = -a1 hex(256-dec(a1))

@ a1 in r0 is not needed anymore
  # @ asrs rd, <#imm> --> #imm: 0-31
  movs r0, #32         @ r0 = #32
  asrs r6, r0          @ r6 = -a1 >> 32

  ands r6, r3          @ r6 = tmp = (b2+s12) & (-a1 >> 32)
  negs r6, r6          @ r6 = -tmp

  # @ asrs rd, <#imm> --> #imm: 0-31
  asrs r6, r0          @ r6 = -tmp >> 32

# @ (b2+s12) in r3 is not needed anymore
  mov  r3, r10         @ r3 = &b, for clearing r3 = (b2+s12) and for loading b1 later, @ hea

  ands r4, r6          @ r4 = s & (-tmp >> 32) = p1_12= (a1*(b2+s12))

  eors r1, r4          @ r1 = r12(=rnd5)+p0_12+p1_12

### It is better to clear the read bus and also reg4, as they are (a1*(b2+s12)), which seem
  eors r4, r4           @ r4 = 0, clearing (a1*(b2+s12))
  ldrb r4, [r5, #0]     @ rx = 0 = table[0],  Clearing the read bus

@@@@ r0 = 32, r1 = r12(=rnd5)+p0_12+p1_12, r2 = &rnd, r3 = &b,
@@@@ r4 = 0, r5 = &table, r6 = -tmp >> 32, r7 = s12,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
```

```
@@@
```

```
  # eors r4, r4          @ r4 = 0, clearing (b1*s12)
  mov  r4, r9         @ r4 = &a, for loading a2 later, @ hear to avoid stall,

  ldrb r3, [r5, #0]    @ rx = 0 = table[0],  Clearing the read bus

@@@@ r0 = 32, r1 = r12(=rnd5)+p0_12+p1_12+p2_12, r2 = &rnd, r3 = 0,
@@@@ r4 = &a, r5 = &table, r6 = b1, r7 = s12,
@@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
@@@@
```

```
860
861 # (a2+s12) in r3 is not needed anymore
862
863   mov  r3, r8        @ r3 = &c, for clearing r3 = (a2+s12) and for loading a1*b1 later, @ h
864
865   ands r4, r0         @ r4 = s & (-tmp >> 32) = p3_12= (b1*(a2+s12))
866
867   eors r1, r4         @ r1 = r12(=rnd5)+p0_12+p1_12+p2_12+p3_12
868
869 ### it is better to clear the read bus and also reg4, as they are (b1*(a2+s12)), which seem
870   eors r4, r4          @ r4 = 0, clearing (b1*(a2+s12)
871
872   # not needed, beacuse of the next instruction
873   # ldrb r4, [r5, #0]    @ rx = 0 = table[0],  Clearing the read bus
874 # ##############################################################################
875 @@@@ r0 = -tmp >> 32, r1 = r12(=rnd5)+p0_12+p1_12+p2_12+p3_12,
876 @@@@ r2 = &rnd, r3 = &c,
877 @@@@ r4 = 0, r5 = &table, r6 = 32, r7 = &c,
878 @@@@ r8 = &c, r9 = &a, r10 = &b, r11 = &table
879 @@@@
```

```
903
904     eors r4, r0          @ r4 = (a2*b2)+(r12(=rnd5)
905
906     strb r6, [r3, #8]    @ r4 = 0, clearing the write bus which contains (a2*b2)
907     ldrb r5, [r2, #12]   @ r5 = rnd3 @ it also clears the read bus wich contains a2*b2
908
909     eors r4, r5          @ r4 = c2
910     strb r4, [r3, #8]    @ r4 = c2, Storing c2, r4 = c2
911
912
913
914
915
916     nop
917     nop
918     nop
919     nop
```

21.1.1 Experimental result:

frequency = $2MHz$
Sampling rate = $250MSa/s$
Period = $228\mu s$
Each clock is sampled with 125 points
Number of samples = 456×125 = $57K$ samples
The bytes of shares (a, b, c are not in a word)

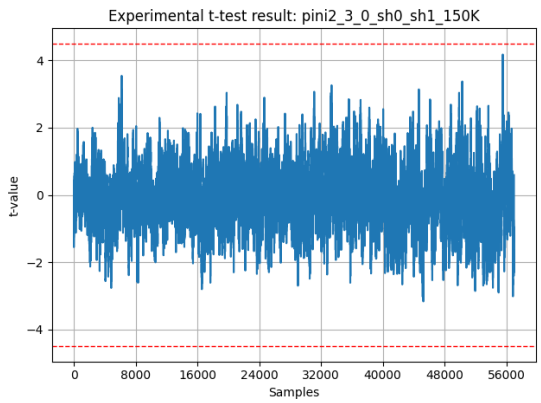
21.1.1.1 Signal to Noise Ratio (SNR):

21.1.1.2 t-test:

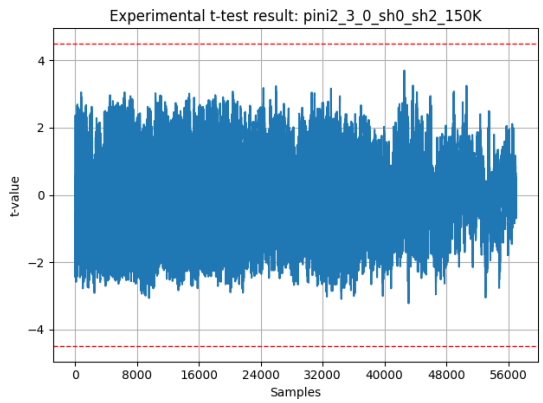
Finding leakages via splitting t-test in to three t-test s:

In this section we find the leakages caused by 2 shares out of 3 shares using the split-t-test .
No leakages were found up to 200K traces.
No leakages were found by **uEMO**.

(a) t-test on (sh_0, sh_1)



(b) t-test on (sh_0, sh_2)



(c) t-test on (sh_1, sh_2)

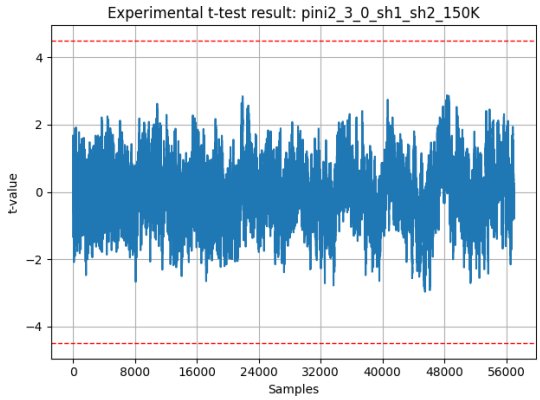
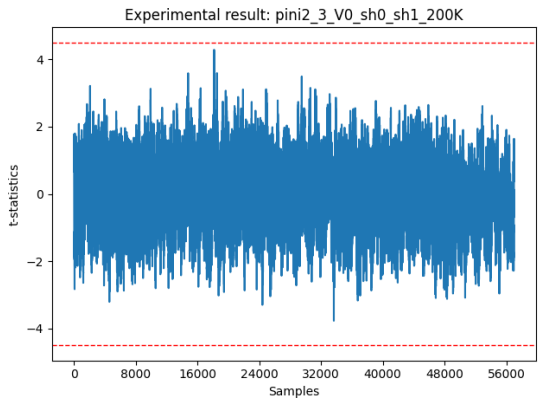
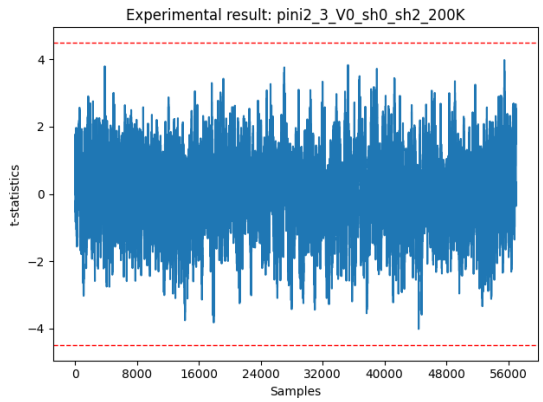


Figure 21.1: t-test_pini2_3_0_two_shares(Setup: 250MSa/s, 57KS, 2ns, 2MHz)

(a) t-test on (sh_0, sh_1)



(b) t-test on (sh_0, sh_2)



(c) t-test on (sh_1, sh_2)

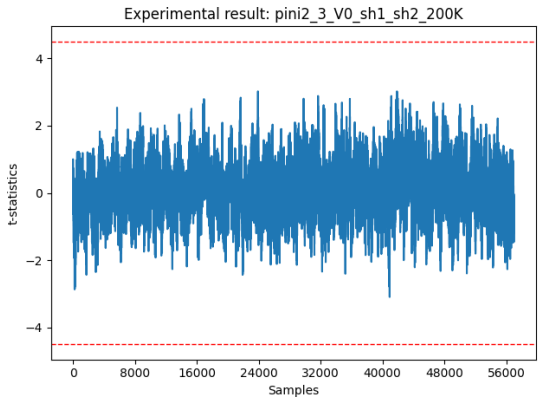


Figure 21.2: t-test_pini2_3_V0_two_shares (Traces exist) (Setup: 250MSa/s, 57KS, 2ns, 2MHz)

Bibliography

[1] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648. Springer, 2016.

[2] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[3] Gaëtan Cassiers and François-Xavier Standaert. Towards globally optimized masking: From low randomness to low noise rate or probe isolating multiplications with reduced randomness and security against horizontal attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):162–198, 2019.

[4] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, 15:2542–2555, 2020.

[5] Si Gao, Elisabeth Oswald, and Dan Page. Reverse engineering the micro-architectural leakage features of a commercial processor. Cryptology ePrint Archive, Report 2021/794, 2021. https://ia.cr/2021/794.

[6] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1469–1468. USENIX Association, 2021.

[7] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In *Advances in Cryptology - EUROCRYPT*, pages 567–597, 2017.

[8] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3, 2016.

[9] Hannes Gross, Stefan Mangard, and Thomas Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, pages 95–112, 2017.

[10] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[11] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.

[12] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer, 2010.