# TRAITOR: A Low-Cost Evaluation Platform for Multifault Injection

Ludovic Claudepierre, Pierre-Yves Péneau, Damien Hardy, Erven Rohou

# TRAITOR: A Low-Cost Evaluation Platform for Multifault Injection

### Ludovic Claudepierre
Univ Rennes, Inria, CNRS, IRISA
first.last@irisa.fr

### Damien Hardy
Univ Rennes, Inria, CNRS, IRISA
first.last@irisa.fr

### Pierre-Yves Péneau
Univ Rennes, Inria, CNRS, IRISA
first.last@inria.fr

### Erven Rohou
Univ Rennes, Inria, CNRS, IRISA
first.last@inria.fr

## ABSTRACT

Fault injection is a well-known method to physically attack embedded systems, microcontrollers in particular. It aims to find and exploit vulnerabilities in the hardware to induce malfunction in the software and eventually bypass software security or retrieve sensitive information. We propose a low-cost platform called TRAITOR inducing faults with clock glitches with the capacity to inject numerous and precise bursts of faults. From an evaluation point of view, this platform allows easier and cheaper investigations over complex attacks than costly EMI benches or laser probes.

## KEYWORDS

Multifault injection, Clock glitch, Instruction skip, Physical attack

## 1 INTRODUCTION

Different fault injection techniques have been studied in the literature, such as using laser beams [13], exploiting electromagnetic injection (EMI) [10] or causing clock or voltage glitches [1]. These techniques have been widely covered to inject a single or few faults to hijack control flow or change a register value [3]. In this paper, we present a low-cost evaluation platform to precisely inject numerous faults by clock glitch.

Our previous results [4] using EMI attacks show that EMI has an impact on the clock signal of the microcontroller. This situation motivates us to develop a new experimental platform called TRAITOR (meaning **TRA**nsportable gl**I**tch a**T**tack platf**OR**m) based on clock glitch injection. With a low-cost FPGA, we produce our own faulty clock signal to replace the one of the targeted board. Such a technique avoids the desynchronization encountered with EMI due to several factors such as clock jitters, impedance mismatch, and propagation losses. This improves the efficiency of the attack and makes the injection more controllable and easily reproducible. During our

experiments, we were able to reproduce the same attack more than 1000 times with identical results. The platform also provides the ability to inject consecutive faults and to realize complex attacks inducing several bursts of faults. All these advantages make the platform a really interesting tool *i)* to observe the resilience of a system to multifaults injections, *ii)* for investigating the possibilities from an attacker point of view and *iii)* for evaluating new software based countermeasures. As a contribution, we present TRAITOR, an experimental platform that can inject numerous and precise faults by using clock glitches. An attack on the PIN code double verification illustrates how it works.

The paper is structured as follows: Section 2 details previous related work. Section 3 describes the experimental platform. Experiments are presented in Section 4. We discuss our results in Section 5 and conclude in Section 6.

## 2 RELATED WORK

Faults can be injected to corrupt instructions, to corrupt data, or both. This work focuses on instruction corruption, and more specifically, the capability to skip the execution of one or several instructions at run time.

A generally assumed fault model considers an attacker with the ability to inject one or few faults to skip a single instruction or a set of specific instructions [5, 7, 8]. Such a fault model has been widely explored in the literature through different injection methods such as EMI [10], laser [13], clock [1] and voltage glitches [15].

No matter which injection method is used, they can generate different faults on the chip, leading to the skip of the instruction. Such a fault could be: *i)* changing the bits of the opcode and substituting the executed instruction by another one without side effect (effectively considered as a nop) [10]; *ii)* modify the Program Counter (PC) [14]; *iii)* prevent the refill of the instruction buffer [12].

In a general manner, single fault with laser or using EMI are most of the time sufficient for a successful attack. Nonetheless, the cost of the setup could be high and the success rate is sometimes low (30 % for our EMI bench). This low success rate is due to several physical parameters such as the location of the probe on the chips, the clock jitters, the efficiency of the injection system.

This low success rate is not an issue when the attacker looks for a single fault that permanently affects the chip or the running algorithm. Nevertheless, except with laser, the fault injection changes the program at run time and has temporary effects.

There are few papers that mention large fault capability. Bozzato *et al.* [2] use voltage glitches to fault up to 100k instructions using
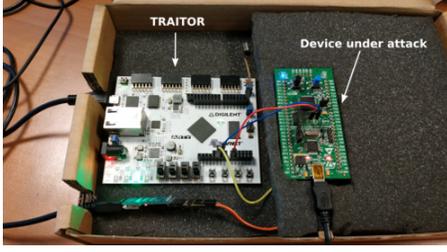
Figure 1: TRAITOR platform.

genetic algorithms to search for optimal parameters with good repeatability. Dutertre *et al.* [6] propose a burst to block the fetch and skip up to 300 instructions with laser injection, and Menu *et al.* [9] use EMI to induce a fault that avoids 6 instructions. EMI and laser have a moderate repeatability but require complex, expensive equipment. Conversely, TRAITOR can inject several bursts of faults in a precise and reliable manner for an approximate price of $130 at the time of writing.

## 3 TRAITOR: THE PLATFORM

In this section, an overview of TRAITOR is first presented. The faulty clock signal generation is then detailed followed by an explanation on how to configure TRAITOR from a user point of view. Some limitations of TRAITOR are finally discussed. The VHDL source code is publicly available[1].

### 3.1 Overview

TRAITOR is an FPGA platform implemented on a Xilinx Artix-7 35T FPGA. The implementation uses 1 MMCM + PLL, 5 Buffers and 2556 LUTs. Overall, less than 20 % of the available resources of a basic FPGA is used.

TRAITOR generates a clock signal with customizable glitches. The platform, depicted on Figure 1, is an FPGA with a part dedicated to the generation of a faulty clock signal and a memory part where users, through UART, can register the parameters of this signal to achieve the attack. The clock signal of the targeted microcontroller is replaced by the output of TRAITOR. To achieve this, we plug our clock signal in place of the High Speed External (HSE) crystal of the board. A UART connection to the user machine is used to control parameters of the clock glitch detailed in Section 3.2.

We work in *evaluation* mode. This means that a full access to the source code is assumed and a pin of the targeted board is used as a trigger and connected to an input of TRAITOR. This trigger is a timing reference to start the injection.

### 3.2 Generation of the faulty clock signal

In this section, we describe the implementation of TRAITOR inside the FPGA and in particular the intermediate signals generated to build the faulty clock signal used to attack. These signals are represented on the left side of Figure 2 and their generation with combinational logic is detailed on the right side. $clk_{glitch}$ and $clk_{out}$ are considered analog signals, the other ones are digital.

---

[1]https://gitlab.inria.fr/traitor/traitor
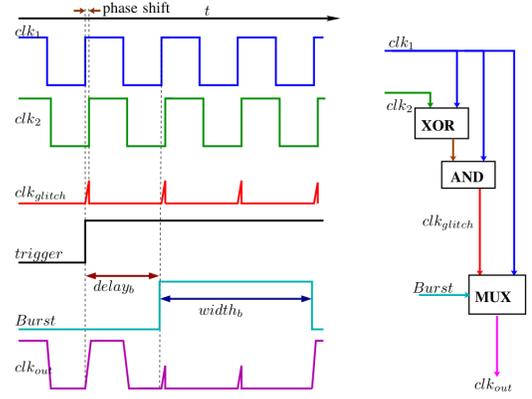


Figure 2: Signals of the TRAITOR platform.

To set up a fault injection, a user has to define three parameters: the delay and the width of each burst, and the amplitude. The delay defines the number of clock cycles to wait before injecting the burst and the width sets the number of cycles it lasts. The amplitude defines the height of the rising edge we allow before cutting down to zero. This faulty rising edge is shown on Figure 2 on the $clk_{glitch}$ signal. The higher the amplitude, the higher the rising edge.

The idea of the attack is, during a rising edge, to cut down to zero the clock signal to disturb the internal behavior of the CPU as observed with electromagnetic fault injection [4]. To do so, TRAITOR generates two clock signals $clk_1$ and $clk_2$ with a tunable phase shift. This phase shift controls the amplitude. The faulty clock signal, denoted $clk_{glitch}$, is generated by using the rising time of the analog output. It is obtained by the following logic operation:

$$clk_{glitch} = (clk_1 \oplus clk_2) \ \& \ clk_1 \tag{1}$$

During the time $clk_1$ and $clk_2$ are in different states, the edge signal starts rising and stops when the two signals are again in the same state. The amplitude of the glitch is above low state and below high level. We conserve only the glitch at the rising edge of $clk_1$.

The trigger signal denoted *trigger* is received by TRAITOR and its rising edge is used as reference to count the number of clock cycles to wait before the injection of the bursts. Upon the reception of this signal, the *Burst* signal is generated. It stays at 0 during $delay_b$ the waiting delay before the injection, then switches to 1 during $width_b$ the width of the $b^{th}$ burst. Then, it goes back to 0 until the cycle counter reaches the value $delay_{b+1}$ and then switches to 1 during $width_{b+1}$ cycles and then falls back to 0 and so on. This is described in Formula 2, where $t$ is the current clock cycle and $t_{trig}$ the clock cycle at which *trigger* is received. *Nburst* represents the maximal number of bursts supported by TRAITOR.

$$Burst = \begin{cases} 1 & \text{if } \exists b, b \leq Nburst \ \wedge \\ & t_{trig} + delay_b \leq t \leq t_{trig} + delay_b + width_b \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The output signal of TRAITOR, denoted $clk_{out}$, switches between $clk_1$ and $clk_{glitch}$ when $Burst = 0$ and $Burst = 1$, respectively.

**Table 1: Memory layout of the FPGA**

| Burst number | Address | Parameter | Width (bytes) |
|---|---|---|---|
| All | 0x00 | Amplitude | 1 |
| 0 | 0x01 | Delay | 4 |
| | 0x05 | Width | 4 |
| 1 | 0x09 | Delay | 4 |
| | 0x0D | Width | 4 |
| ... | ... | ... | ... |
| N | 0x01 + 8 * N | Delay | 4 |
| | 0x05 + 8 * N | Width | 4 |

## 3.3 Software configuration of TRAITOR

TRAITOR is configurable through UART and the memory inside the FPGA. The communication protocol between computer and FPGA is the following: *<command> <address> <value>*. Command is either 0 or 1, respectively for read or write; address is a [1-4]-byte field; value is a [1-4]-byte field used with the *write* command. We dedicate specific addresses in the memory to set the value of the amplitude, the delay and the width for all burst.

The memory layout is given in Table 1. Address 0x00 is used to set the amplitude, which is not tunable per burst. For each burst, we use an 8-byte memory range to specify the delay and the width. The first burst uses a 4-byte memory address starting at 0x01 to set the delay to wait before starting the attack. The width of this first burst is specified in the next 4-byte memory range that follows the delay, *i.e.,* from 0x05 to 0x08. Then, all successive bursts use an 8-byte memory range for these two parameters. The second burst (N=1) starts from 0x09 to 0x10, third burst from 0x11 to 0x18, and so on. Burst $N$ is at address 0x01 + 8 x N, starting with $N = 0$. The present version of TRAITOR supports a maximum of 31 bursts.

TRAITOR keeps reading the memory and counting the number of clock cycles until it reaches the delay value of the next burst and acts accordingly with the width value.

## 3.4 Limitations

Our platform has few limitations, categorized as strong or weak.

**Strong limitations.** The possibility to plug an external clock source is a strong limitation to the platform. In this paper, we remove the external crystal to plug TRAITOR. However, a crystal is not always present on a board. It could also be done by using dedicated pins on the board that are configured to receive an external clock. Nonetheless, this feature is not available on all boards.

**Weak limitations.** To start the attack, TRAITOR needs a synchronization point. Since we assume evaluation mode, we synchronize by manually inserting a trigger on a specific pin of the board and connected to the FPGA. However, such an access is not mandatory. For a real attack, this could be avoided for example by running non-invasive measures on the power of the board to identify one or several regions that could be sensitive to an attack.

The PLL (Phase-Locked Loop) is a well-known system used to provide a stable clock signal, avoiding fast variation and diminishing jitters. As TRAITOR relies on a brutal modification of the clock signal, the PLL has to be disabled. The PLL is a well-used device that can be found on almost every manufactured chip. It is small

and does not require a lot of resources in terms of power and area. Hence, it is very often activated and acts as a hardware countermeasure to clock-glitches. However, in this paper we propose an evaluation platform and do not consider the attacker point of view. This allows us to disable the PLL directly in the source code of our programs. For a real attack, disabling the PLL could be achieved by using electromagnetic injection. Then, TRAITOR could be used for the rest of the attack. Such a scenario would place the PLL as a strong limitation though.

## 4 TRAITOR: EXPERIMENTS

### 4.1 Setting up the attack

For practical reasons, it is not always possible to put the trigger just before the instructions to skip. For example, considering a scenario where an attacker wants to avoid a compare-and-branch at the end of a loop after $N = 5$ iterations to have this $N$ value in a register and use it afterwards. Such an attack requires to put the trigger outside of this loop. If not, the trigger would be activated at the first iteration and the $N$ value would be 1 instead of 5.

To set up the attack, we follow these steps: *i)* find where we want to skip instructions by looking at the assembly code, *ii)* find where the trigger can be set accordingly and *iii)* determine the number of clock cycles that occur between the trigger and this group of instructions. For a certain number of instructions, this is a fixed value that can be found in the ARM ISA description. Since we are in an evaluation mode, we control the compilation flags and thus, compile without optimizations (-O0 flag). This limits the usage of instructions that have variable execution time.

However, this number of cycles could differ considering an `if-then-else` with a completely different number of instruction for each case. In a scenario with a branch that is taken 50% of the time and not taken 50% of the time, this generates a variable number of clock cycles. To overcome this obstacle, we perform some tests to measure the time between two points in the program. Those tests are mainly done by setting an additional variable $V$ in the program, modify its value at a specific point, then inject faults to check if $V$ is modified. In such a case, we found the number of clock cycles between the trigger and $V$. We move $V$ in another part of the code and restart the process until we reach the targeted region.

### 4.2 Glitch capabilities

For the most common instructions (`add`, `mov`, `push`, `pop`, `ldr`, `cmp`) the fault model often observed is the "skip-by-repeat instruction". A similar fault model has been observed by Rivière *et al.* [12]. In this model, previously fetched instructions are repeated by preventing the fetch buffer to refill. However, we also experienced a true skip without repeat. While the exact reason is not clear despite our investigation, we note that it occurs when we inject faults near conditional branch instructions like `beq` and `bne`.

The STM32F100RB chip is used to present our platform. This board has a 32-bit fetch buffer where two Thumb 16-bit instructions or one 32-bit instruction can be stored. By altering the fetch request, we are able to skip one or two instructions depending on the size of the request. Most of the time, we also repeat one or two instructions depending on what is currently in the fetch buffer. With such a fault attack, counters can be incremented by repeating

```
1  <F>:
2    10500: sub, sp, sp #8
3            ...
4    loop:
5            ...
6    10598: add, r1, r1,#1      ─┐
7    1059c: cmp r1, #10          ├ aligned
8    105a0: bnz   10500 <loop>  ─┘
9    105a4: add sp, sp, 8        ┐ aligned
10   105a8: bx lr                ┘
```

**Listing 1: Aligned instructions that prevent an attack**

add or sub instructions and branches can be skipped. Consequently, the execution of functions can be avoided and dead code can be executed by targeting the branch return instruction. ldr, mov, str and cmp instructions can be repeated or skipped too. Finally, the stack can be manipulated by repeating or skipping push and pop.

The main advantage of the TRAITOR platform is the possibility to produce numerous glitches during each execution. It gives the possibility to investigate the security over more complex attacks than single faults.

However, we have to take into account the size of instructions for our attacks. Due to the fact that most of them are 16-bit Thumb instructions, they are often aligned by groups of two. Unless it is a 32-bit instruction, avoiding a specific instruction is not possible. Thus, skipping the first instruction of a group of two will repeat the second one, and *vice-versa*. During the setup of the attack, we have to take into account this side effect. If the instruction before or after the one we want to skip is vital for the attack and cannot be avoided, the attack is impossible on this part of the code.

Listing 1 gives such an example. In this case, we want to skip the final branch of a loop in function F, and return to the previous function without any other modification, especially the stack pointer (SP). One can see that the branch is located before the add instruction that restores SP. If those two instructions are aligned, we cannot avoid the branch since it would also skip the add. This would result in an incorrect stack pointer value when executing the bx lr to return to the previous function.

## 4.3 Illustration with the single PIN verification

The PIN code single verification has been successfully attacked with a single fault injection for many years. In the algorithm presented in Listing 2, a correct pin code is signaled by a green LED and an incorrect one is signaled by a blue LED. The idea of the attack is to hijack the normal control flow when entering a wrong PIN by skipping the first if() test where check_result() is called. This is highlighted in red on Listing 2. By doing this, the Program Counter goes to the part of the program where the PIN is assumed correct, even with an incorrect input. Note that the call of the check_result() function does not need to be skipped. The only critical instructions are those who act according to the value returned by this function.

The assembly code of the single verification PIN code is presented in Listing 3. The if-then-else is expressed by the two instructions cmp and beq, respectively at address 80004d4 and 80004d6. In this use case, if the input value is not correct, the branch to 80004ec is taken. If not, we continue with the next instruction after beq.

The trigger signal, not shown here, is put before the if(). The TRAITOR setup is the following: 70 cycles of delay, then a 1-clock

```
1  if (check_result(result)) {
2    state = 1; // Correct PIN
3    Green_LED_on();
4  } else {
5    state = 0; // Wrong PIN
6    Blue_LED_on();
7  }
```

**Listing 2: PIN code Single check algorithm**

```
1  80004ce:   bl  80007c8 <check_result>   ─┐
2  80004d2:   mov r3, r0                    │
3  80004d4:   cmp r3, #0                    ├ if
4  80004d6:   beq.n 80004ec <main+0x22c>   ─┘
5  80004d8:   ldr r3, [pc, #236]           ─┐
6  80004da:   movs  r2, #1        ; state = 1│
7  80004dc:   str r2, [r3, #0]              │
8  80004de:   movs  r2, #1                  │
9  80004e0:   mov.w r1, #512                ├ then
10 80004e4:   ldr r0, [pc, #180]            │
11 80004e6:   bl  8002cb2 <HAL_GPIO_WritePin>│
12 80004ea:   b.n 80004fe <main+0x23e>     ─┘
13 80004ec:   ldr r3, [pc, #216]           ─┐
14 80004ee:   movs  r2, #0        ; state = 0│
15 80004f0:   str r2, [r3, #0]              │
16 80004f2:   movs  r2, #1                  ├ else
17 80004f4:   mov.w r1, #256                │
18 80004f8:   ldr r0, [pc, #160]            │
19 80004fa:   bl  8002cb2 <HAL_GPIO_WritePin>─┘
```

**Listing 3: Assembly code of the single verification code PIN. Strikethrough instructions are skipped.**

cycle duration for the glitch. With such a configuration, we are able to bypass the comparison and go into the block where the PIN is assumed correct even though it is not. Since we attack a conditional branch, we do not observe a repeat in this case.

One could argue that the attack works because the compiler decides to put the then case after the beq instruction. However, in the opposite case, our attack targets the check_result() function to always return 1. This has been tested with success.

## 4.4 Attack on double PIN verification

As a countermeasure, a double verification has been proposed as illustrated in Listing 4. After the first test, a second test is realized. To be authenticated, the two tests have to pass. If the two tests have different results, the user is not authenticated. Moreover, a single fault attack can be detected. If the two tests have different results, both LEDs are switched on.

As TRAITOR is able to perform multiple fault injection, this double verification whose assembly code is presented in Listing 5 can be attacked by repeating twice in the same run the technique used for the single verification. We setup the platform to inject a first burst of 1 cycle after waiting 70 cycles, then a second burst after waiting 164 cycles. With such a configuration, we target instructions at 8000554, 8000556 for the first if(), then instructions at 8000576 and 8000578 for the second if(). Thus, these two tests always exit with a state = 1 value, which indicates a correct PIN. Moreover, the attack remains undetected since the blue LED is never switched on. Finally, this attack generalizes to $N$ verifications with $N$ injections.

```
1   if (check_result(result)) {
2     state = 0; // Assumed correct. Double check
3     Green_LED_on();
4     if (check_result(result)) {
5       state = 1 // Correct
6     } else {
7       state = 2; // Wrong after correct. Attack detected!
8       Blue_LED_on();
9     }
10  } else {
11    state = 3; // Assumed wrong. Double check
12    Blue_LED_on();
13    if (check_result(result)) {
14      state = 4; // Correct after wrong. Attack detected!
15      Green_LED_on();
16    } else {
17      state = 5; //Wrong
18    }
19  }
```

**Listing 4: PIN code Double verification algorithm**

```
1   800054e:   bl   80007a0 <check_result>
2   8000552:   mov r3, r0
3   8000554:   cmp r3, #0                          ┐ if
4   8000556:   beq.n 80005d4 <main+0x314>          ┘
5   8000558:   movs  r2, #1                        ┐
6   800055a:   mov.w r1, #512
7   800055e:   ldr r0, [pc, #64]
8   8000560:   bl   8002c8a <HAL_GPIO_WritePin>
9   8000564:   ldr r3, [pc, #100]
10  8000566:   movs  r2, #0        ; state = 0
11  8000568:   str r2, [r3, #0]
12  800056a:   ldr r3, [pc, #92]
13  800056c:   ldr r3, [r3, #0]
14  800056e:   mov r0, r3
15  8000570:   bl   80007a0 <check_result>
16  8000574:   mov r3, r0                          ┐ nested if
17  8000576:   cmp r3, #0                          │
18  8000578:   beq.n 8000582 <main+0x2c2>          ┘       then
19  800057a:   ldr r3, [pc, #80]                   ┐ nested then
20  800057c:   movs  r2, #1     ; state = 1        │
21  800057e:   str r2, [r3, #0]                    ┘
22  8000580:   b.n 8000610 <main+0x350>            ┐
23  8000582:   ldr r3, [pc, #72]
24  8000584:   movs  r2, #2     ; state = 2        │
25  8000586:   str r2, [r3, #0]
26  8000588:   movs  r2, #1                        ┐ nested else
27  800058a:   mov.w r1, #256
28  800058e:   ldr r0, [pc, #16]
29  8000590:   bl   8002c8a <HAL_GPIO_WritePin>    ┘
30  8000594:   b.n 8000610 <main+0x350>            ┘
31             ...
```

**Listing 5: Assembly code of the double verification code PIN. Strikethrough instructions are skipped.**

## 4.5 Large burst and multiple burst injection

One of the goals of TRAITOR is to inject multiple bursts and also long bursts in a reproducible manner. In this section, we give two examples of such use cases.

**Large burst injection.** Listing 6 presents a function, asm_add(), that accumulates the sum from 0 to 100 into register R4, then moves R4 to R0 and returns. A nominal result should be 5050. We successfully test different burst sizes on this function to modify the final value in R4. For example, we execute add R4,#1/add R4,#2, then inject a burst of 50 cycles. This burst repeats these two instructions for 50 cycles before returning to a normal state. The final value

```
1   <asm_add>:
2     mov R4, #0
3     add R4, #1
4     add R4, #2
5     add R4, #3     ┐
6     add R4, #4     │
7       ...          │ burst here, repeat add #1 add #2
8     add R4, #93    │
9     add R4, #94    │
10    add R4, #95    ┘
11    add R4, #96
12      ...
13    add R4, #100
14    mov R0, R4
15    bx lr
```

**Listing 6: Long addition with modified result. Trigger is just before calling this function.**

```
1   <asm_sub>:
2     movw R4, 0xEA3C ; Put 125500 in R4
3     movt R4, 0x0001
4     sub R4, #1
5     sub R4, #2
6     sub R4, #3     ┐
7     sub R4, #4     │
8     sub R4, #5     │
9     sub R4, #6     │ burst #1, repeat sub #1 sub #2
10    sub R4, #7     │
11    sub R4, #8     ┘
12    sub R4, #9
13    sub R4, #10
14    sub R4, #11
15    sub R4, #12
16    sub R4, #13    ┐
17    sub R4, #14    │
18    sub R4, #15    │
19    sub R4, #16    │ burst #2, repeat sub #11 sub #12
20    sub R4, #17    │
21    sub R4, #18    ┘
22    sub R4, #19
23      ...
24    sub R4, #102
25    sub R4, #103   ┐
26    sub R4, #104   │
27    sub R4, #105   │
28    sub R4, #106   │ burst #3, repeat sub #101 sub #102
29    sub R4, #107   │
30    sub R4, #108   ┘
31    sub R4, #109
32      ...
```

**Listing 7: Long subtraction attacked with multiple burst. Trigger is just before calling this function.**

is 3750, which is the expected value. We successfully inject bursts up to 92 cycles before obtaining an unexpected result. 94, 96, 98 and 100 cycles gives numbers we cannot explain yet and are under investigations. We also observe that injecting a burst of 102 cycles sets the value of R4 to its original value before entering into the asm_add() function. In our case, it is a special address used to control a LED on the board. Finally, any burst width of more than 102 cycles causes a hardfault and the CPU goes into a special handler.

**Multiple burst injection.** One feature of the platform is the possibility to inject multiple bursts and to combine them with the multiple width. In this use case, we take control of a long subtraction depicted in Listing 7. We first set the initial value to 125500, then do the subtraction from 0 to 255. We inject three bursts of 5 cycles each to alter the execution and obtained the expected value

when replacing these 18 instructions.

The key aspect of these simple use cases is that they are easily reproducible. We ran those experiments more than 1000 times and we consistently obtain the same result. We can flash the board multiple time, or use different boards of the same model, results do not vary. Conversely to EMI, we have a full control of the clock of the board since we **are** the clock through the FPGA. Such a platform with multiple bursts and long bursts with a high level of reproducibility could be considered for more advanced attacks.

## 5 DISCUSSION

### 5.1 EM injection vs Clock glitch

EM injection is commonly preferred to clock glitch because it is completely non-invasive. Nevertheless, the equipment to perform EMI is expensive and it can be hard to achieve a high success rate of injection. Clock glitch is a much cheaper way to attack but requires access to the clock tree (most of the time, the crystal). Furthermore, a lot of countermeasures exist and are implemented. The most common is the PLL that controls the clock signal, avoids any disruption on it and makes us able to choose the clock frequency. Thus, using a platform like TRAITOR needs an inactive PLL and an access to the crystal to deliver a faulty clock signal.

### 5.2 Possibilities with multifault attack

The previous application shows the possibilities of the multifault capability of the TRAITOR platform. If we consider the possibility to attack any instruction and combine the different effects of each injection, TRAITOR could lead to complex attacks. We show the theoretical possibilities of such a complex multifault attack by using a simulator [11]. The fault model used in this paper is the virtual-nop, *i.e.,* an instruction can be skipped without side effect. Performing this kind of complex attack can be seen as a way to rewrite an existing program at run-time. In our previous work [11] we call this paradigm "NOP-Oriented Programming". We show how this fault model can lead to Control Flow Graph (CFG) hijacking, alterations in the number of loop iterations (do fewer or more iterations) and registers/memory rewriting. We also demonstrated the Turing-completeness of this fault model.

TRAITOR can be considered as a way to practically realize such an attack with nevertheless some adaptation of the fault model. Indeed, our fault model is not the virtual-nop. We often observe a "skip-by-repeat" behavior on the most common instructions, while a true virtual-nop is experienced with conditional branches. Consequently, making a fault injection on a targeted instruction has often side effects which depend on the previously fetched instructions. Because of that, performing an attack could be much more complicated as these side effects have to be taken into account.

## 6 CONCLUSION

Fault injection commonly considers the possibility of inducing a single fault while some articles started to propose multifault techniques. We believe that multifault and long burst injection

need to be deeply investigated. In order to evaluate the potential of these paradigms, the evaluation platform TRAITOR has been developed. It is based on an FPGA signal replacing the clock of the targeted chip. The faults are performed by clock glitch injection with a reliable control of the injection and a high repeatability. This platform is low-cost, easy to use and can inject numerous bursts of faults. It can be a really powerful tool to be used in evaluation mode to study and develop complex attacks with several bursts of faults and to evaluate new software based countermeasures.

Our ongoing work focuses on reproducing attacks we previously proposed [11] on TRAITOR to control loop iterations, registers and memory. Future work will focus on extracting a precise fault model for TRAITOR with the STM32F100RB board. In addition, exploration of software or hardware countermeasures are considered.

## REFERENCES

[1] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An In-Depth and Black-Box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2011.

[2] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 199–224, 2019.

[3] Sebanjila K. Bukasa, Ronan Lashermes, Jean-Louis Lanet, and Axel Legay. Let's Shock Our IoT's Heart: ARMv7-M Under (Fault) Attacks. In *13th International Conference on Availability, Reliability and Security*, ARES. ACM, 2018.

[4] L. Claudepierre and P. Besnier. Microcontroller Sensitivity to Fault-Injection Induced by Near-Field Electromagnetic Interference. In *Joint International Symposium on Electromagnetic Compatibility, Sapporo and Asia-Pacific International Symposium on Electromagnetic Compatibility (EMC Sapporo/APEMC)*, 2019.

[5] Emmanuelle Dottax, Christophe Giraud, Matthieu Rivain, and Yannick Sierra. On Second-Order Fault Analysis Resistance for CRT-RSA Implementations. In *IFIP International Workshop on Information Security Theory and Practices*, pages 68–83. Springer, 2009.

[6] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. Experimental Analysis of the Laser-Induced Instruction Skip Fault Model. In *Nordic Conference on Secure IT Systems*, pages 221–237. Springer, 2019.

[7] Sho Endo, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. A Multiple-Fault Injection Attack by Adaptive Timing Control Under Black-Box Conditions and a Countermeasure. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2014.

[8] Chong Hee Kim and Jean-Jacques Quisquater. Fault Attacks for CRT Based RSA: New Attacks, New Results and New Countermeasures. In *1st IFIP International Conference on Information Security Theory and Practices: Smart Cards, Mobile and Ubiquitous Computing Systems*, WISTP'07. Springer-Verlag, 2007.

[9] Alexandre Menu, Jean-Max Dutertre, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Luc Danger. Experimental Analysis of the Electromagnetic Instruction Skip Fault Model. In *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–7. IEEE, 2020.

[10] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. IEEE, 2013.

[11] Pierre-Yves Péneau, Ludovic Claudepierre, Damien Hardy, and Erven Rohou. NOP-Oriented Programming: Should we Care? In *European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, September 2020.

[12] Lionel Rivière, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures. In *International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015.

[13] Sergei P Skorobogatov and Ross J Anderson. Optical Fault Induction Attacks. In *Int. workshop on cryptographic hardware and embedded systems*. Springer, 2002.

[14] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM Using Fault Injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2016.

[15] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2016.