# Fault attacks on two software countermeasures

Nicolas Moro*†, Karine Heydemann†, Amine Dehbaoui‡, Bruno Robisson*, Emmanuelle Encrenaz†
*Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA), F-13541 Gardanne, France - nicolas.moro@cea.fr
†Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005 Paris, France - karine.heydemann@lip6.fr
‡SERMA Technologies, CESTI, F-33615 Pessac, France - a.dehbaoui@serma.com

***Abstract*—Injection of transient faults can be used as a way to attack embedded systems. On embedded processors such as microcontrollers, several studies showed that such a transient fault injection could corrupt either the data loads from the memory or the assembly instructions executed by the circuit. Some countermeasure schemes which rely on temporal redundancy have been proposed to handle this issue. Among them, several schemes add this redundancy at assembly instruction level. In this paper, we perform a practical evaluation for two of those countermeasure schemes by using a pulsed electromagnetic fault injection process on a 32-bit microcontroller.**

## I. Introduction

In this paper, we experimentally evaluate the robustness of two software countermeasure schemes against fault injection on embedded programs. Both countermeasures are designed at assembly code level and rely on providing some replacement sequences to strengthen some sensitive instructions. To perform this evaluation, we use a pulsed electromagnetic fault injection technique. This fault injection technique has turned out to be an effective way to inject transient faults in a circuit's computation [3]. Figure 1 shows an architectural view of the electromagnetic fault injection platform used in this paper. It is similar to the one presented in [4].

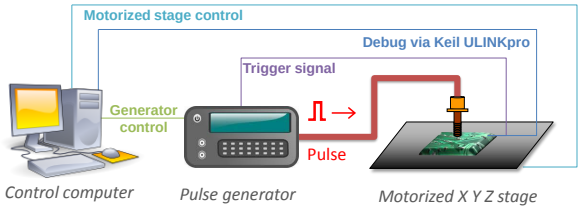


Fig. 1. Electromagnetic fault injection bench

The chosen target is an up-to-date 32-bit microcontroller designed in a CMOS 130 nm technology based on the ARM Cortex-M3 processor [5]. Its operating frequency is set to 56 MHz without any cache memory, and no prefetch buffer is activated. Cortex-M3 processors use a Harvard architecture and run the ARM Thumb-2 instruction set [6], which contains both 16-bit and 32-bit instructions. Some 16-bit instructions can be forced to a 32-bit encoding by using a specific syntax.

## II. Evaluation of the countermeasures

### A. Evaluation approach

We need to define a relevant metric to evaluate the efficiency of the countermeasures. Since the countermeasure sequences add some instructions, the time to execute a full countermeasure sequence becomes longer than the time to execute the initial instruction. Thus, the number of vulnerable points, *i.e.* the injection times for which a fault injection attempt is successful, may also increase. Comparing the percentage of faulty outputs could appear to be a solution to compare two data sets with different numbers of measurements. Nevertheless, we assume that the most meaningful metric for such a comparison is the number of faults that have been obtained on the destination register. The countermeasure is really effective if it can overcome the fact that some new vulnerable points are added and if it can decrease this number of vulnerable points. Thus, comparing the number of faulty outputs is probably the most relevant metric for an attacker since it indicates the number of potential vulnerabilities on an embedded code.

### B. Fault tolerance countermeasure

This countermeasure aims at providing a fault-tolerant replacement sequence for most of the instruction of an instruction set [1]. Such a countermeasure has been designed to be tolerant to any single instruction skip and does not provide any protection to the data flow. An example of the use of this countermeasure is given in Listing 1.

Listing 1. Fault detection countermeasure for a `bl <function>` instruction

```
1 adr  r1, <return_label>
2 adr  r1, <return_label>
3 add  lr, r1, #1  ; Thumb mode requires the
4 add  lr, r1, #1  ; last bit of LR to be set
5 b    <function>
6 b    <function>
7 return_label:
```

We performed some fault injection experiments on four codes: a `bl` instruction without countermeasure (100 ns time interval by steps of 200 ps), a `bl.w` instruction with forced 32-bit encoding without countermeasure (100 ns), the replacement sequence presented in Listing 1 (400 ns) and this replacement sequence with forced 32-bit encoding (400 ns). Several values were used for the pulse voltage, from −210 V to −170 V and from 120 V to 150 V by steps of 5 V. The target circuit crashes for voltages over 150 V. In this experiment, the subroutine that is called only modifies `r0`. Thus, we analyze the number of faults in `r0` at the end of the experiment to evaluate the efficiency of the countermeasure. The *faults on any register* curves correspond to an output in which at least one register contains a faulty value.

The fault injection results are shown on Fig. 2. We can observe that applying the countermeasure without forced 32-bit encoding does not seem to be efficient. Indeed, this

This work was done while Amine Dehbaoui was with École Nationale Supérieure des Mines de Saint-Étienne, F-13541 Gardanne, France

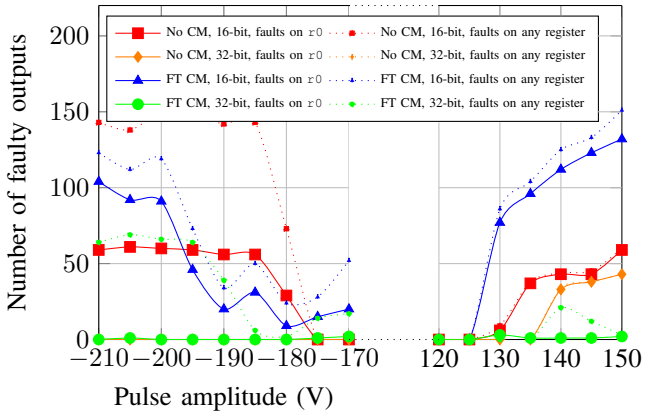


Fig. 2. Fault injection results for the fault tolerance countermeasure

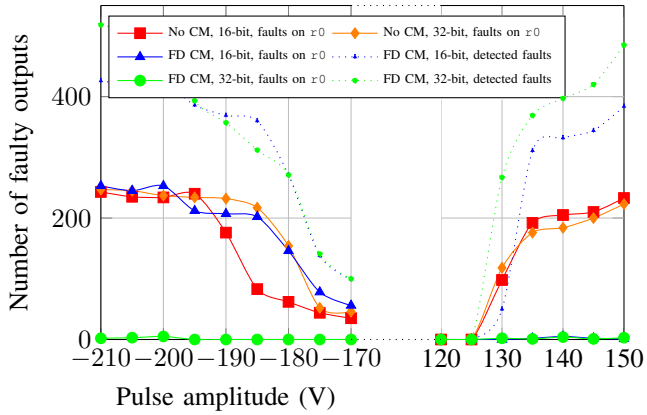


Fig. 3. Fault injection results for the fault detection countermeasure

countermeasure has not been designed to be resistant to two consecutive instruction corruptions. Because of the memory alignment in the experiment, in the replacement sequence the first two `adr` instructions and later the last two `b` instructions are loaded in a single *fetch* stage. Thus, it seems that such double corruption happened. Moreover, we can also observe that no faulty output has been obtained for pulses with a negative voltage on a single 32-bit `bl.w` instruction. Nevertheless, such an instruction could still be faulted by using pulses with a positive voltage. An explanation for such result can be found in the way instructions are encoded. The 16-bit subset of the instruction set is very compact (most of the 16-bit values correspond to one instruction) while the 32-bit subset is very sparse: very few bit flips can change a 16-bit instruction into another instruction, but this assertion is not true for a 32-bit encoding. Finally, for the experiment with a fault tolerance countermeasure and a forced 32-bit encoding, very few faults on `r0` have been obtained. Even if some faults on some other registers can still be obtained, this countermeasure appears to be very effective for both positive and negative glitches. Applying the countermeasure scheme with a forced 32-bit encoding is a necessary condition to guarantee its efficiency.

### C. Fault detection countermeasure

This countermeasure aims at detecting any single fault, including instruction skips, some cases of instruction replacements and faults on the data flow. As an example, the countermeasure for a `ldr` instruction is given in Listing 2. In this code example, a value is loaded from the Flash memory. The address of this value is relative to the program counter.

Listing 2. Fault detection countermeasure for a `ldr` instruction

```
1 ldr   r0, [pc, #40] ; initial load instruction
2 ldr   r1, [pc, #38] ; duplicated load instruction
3 cmp   r0, r1        ; comparison between r0 and r1
4 bne   <error>       ; if r0 != r1, raise an error
```

We performed some fault injection experiments on four codes: a single `ldr` instruction (150 ns time interval by steps of 200 ps), a single `ldr.w` instruction with a forced 32-bit encoding (150 ns), the replacement sequence presented in Listing 2 (300 ns) and the same replacement sequence with a forced 32-bit encoding for every instruction (500 ns).

The fault injection results are presented in Fig. 3. The *detected faults* curves show the number of calls to the `error` subroutine. From a black box approach, we can observe that applying the countermeasure without forced 32-bit encoding creates more vulnerable injection times than the initial single `ldr` instruction and does not bring any security for negative glitches. Nevertheless, this countermeasure seems to work very well for positive glitches. Finally, the countermeasure scheme appears to be very effective with a 32-bit encoding of the instructions.

## III. Conclusion

We have provided a first attempt of practical study of two assembly-level software countermeasure against fault injection attacks. Even if those countermeasures are theoretically secure, it turns out that the level of security they add could be nullified if their implementation on a target platform is not performed in the right way. The fault tolerance countermeasure has been very effective to protect an isolated subroutine call instruction. Thus, it seems such a sensitive instruction can be significantly reinforced against fault attacks. Since this countermeasure has been formally proven resistant to instruction skips, its main limitation appears to be due to its considered fault model which might be too simplistic. Otherwise, the fault detection countermeasure has been designed to protect a smaller set of instructions and it has been very effective on the considered test cases.

## References


[1] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," *Journal of Cryptographic Engineering*, pp. 1–12, 2014.

[2] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented AES," in *Proceedings of WESS 2010*. ACM, 2010.

[3] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria, "Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES," in *Proceedings of FDTC 2012*. IEEE, 2012.

[4] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller," in *Proceedings of FDTC 2013*. IEEE, 2013.

[5] J. Yiu, *The Definitive Guide To The ARM Cortex-M3*, 2009.

[6] ARM, "ARM Architecture Reference Manual - Thumb-2 Supplement," 2005.