A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures

Julien Proy, Karine Heydemann, Alexandre Berzati, Fabien Majeric, Albert Cohen

A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures

Julien Proy, Karine Heydemann, Alexandre Berzati, Fabien Majeric, Albert Cohen

A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures – A Secure Software Perspective

Julien Proy
INVIA
Meyreuil, France
julien.proy@invia.fr

Karine Heydemann
Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6
Paris, France
karine.heydemann@lip6.fr

Alexandre Berzati
INVIA
Meyreuil, France
alexandre.berzati@invia.fr

Fabien Majéric
Thalès
La Ciotat, France
fabien.majeric@thalesgroup.com

Albert Cohen
Inria and DI, École Normale Supérieure[†]
Paris, France
albert.cohen@inria.fr

ABSTRACT

In the area of physical attacks, system-on-chip (SoC) designs have not received the same level of attention as simpler micro-controllers. We try to model the behavior of secure software running on a superscalar out-of-order microprocessor typical of more complex SoC, in the presence of electromagnetic (EM) pulses. We first show that it is possible, in a black box approach, to corrupt the loop iteration count of both original and hardened versions of two sensitive loops. We propose a characterization methodology based on very simple codes, to understand and classify the fault effects at the level of the instruction set architecture (ISA). The resulting classification includes the well established instruction skip and register corruption models, as well as new effects specific to more complex processors, such as operand substitution, multiple correlated register corruptions, advanced control-flow hijacking, and combinations of all reported effects. This diversity and complexity of effects can lead to powerful attacks. The proposed methodology and fault classification at ISA level is a first step towards a more complete characterization. It is also a tool supporting the designers of software and hardware countermeasures.

CCS CONCEPTS

• **Security and privacy → Hardware attacks and countermeasures**; *Software and application security*; • **Computer systems organization →** *Embedded and cyber-physical systems.*

KEYWORDS

electromagnetic pulse injection, fault models, superscalar out-of-order, countermeasures to physical attacks

ACM Reference Format:
Julien Proy, Karine Heydemann, Alexandre Berzati, Fabien Majéric, and Albert Cohen. 2019. A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures – A Secure Software Perspective. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES 2019) (ARES '19), August 26–29, 2019, Canterbury, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3339252.3339253

1 INTRODUCTION

Physical attacks and specifically fault attacks have long been a serious threat in the world of embedded secure devices. Sensitive data are stored in devices like smart cards and passports, designed to resist such physical attacks. Tamper resistance is also part of the certification of security protocols, encryption and authentication codes, boot loaders, etc., and the devices themselves [6]. A variety of techniques, such as electromagnetic (EM) pulses, power or clock glitches, laser beams, have been applied to a variety of devices [28].

[†]With Google AI at the time of publication.

In an era of ubiquitous smart devices, higher integration enables complex, higher performance system-on-chip (SoC) architectures to take over simpler micro-controllers in a wide variety of secure applications. Until recently, microprocessor architecture and SoC complexity were considered strong deterrents for thwarting fault attacks [26]. However, recent papers showed that fault attacks are also effective on such devices [15, 23, 25], prompting the attention of security researchers to complex microarchitectures.

EM fault injection does not require specific chip preparation and offers an interesting trade-off between affordability and precision. It has been proven effective on simple microarchitectures [8, 16] and features relatively well understood fault models [11, 16]. SoCs and complex architectures have not received the same level of attention yet. A handful of recent studies targeting complex SoC reported feasible exploitation [15]. However, the achievable effects remain poorly understood; characterizing the effects of EM pulses on complex microarchitectures is a necessary step to enable the design of effective software and hardware countermeasures.

This paper analyzes the behavior of secure software running on a superscalar out-of-order microprocessor in the presence of EM pulses. Following a black box approach, we successfully corrupt the loop iteration count of vanilla as well as hardened versions of two sensitive loops running on an ARM Cortex-A9 processor. We propose a step-by-step methodology using very simple codes to understand and classify the fault effects at the level of the instruction set architecture (ISA).[1] The well established instruction skip and register corruption models are part of this classification, together with new effects specific to more complex processors. The latter include operand substitution, multiple correlated register corruptions, advanced control-flow hijacking, and combinations of all reported effects. This diversity and complexity of effects can lead to powerful attacks. The proposed ISA-level classification and analysis on secure software is a first step towards a more complete characterization of EM-induced faults.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes the hardware and software setup. Serving as a motivation for the whole study, Section 4 conducts a sensitivity analysis of vanilla and hardened versions of representative loops. Section 5 presents a step-by-step methodology and derives an initial fault model classification. We further extend these models to interpret the faults observed on the sensitive loops in Section 6, wrapping up a first characterization. Section 7 concludes with directions towards a more complete characterization.

[1]The ISA is the lowest level available to a programmer implementing secure applications.

2 RELATED WORK

Since the pioneering exploit of a CRT-RSA implementation [4], fault attacks targeting cryptographic applications have been under the spotlight of security research and new studies are published every year [28]. Such fault attacks are either described at an algorithmic level [1, 8] or target a specific implementation [5, 9]. They rely on the attacker's ability to inject and exploit fault effects on a sensitive application: e.g., a skipped instruction or corrupted variable. As a result, most studies considered the injection or exploitation dimensions, while the characterization of possible fault effects received less attention. This is particularly true of modeling efforts targeting the hardware/software interface.

Moro et al. proposed a first ISA-level characterization of EM fault injection on a 32-bit ARM Cortex-M3 processor [16]. Observing that EM fault injections can alter transfers from the Flash memory, they model these effects as load value corruption or instruction replacement at ISA level. Also, 25% of these instruction replacements happen to be equivalent to skipping an instruction. Riviere et al. reported that due to the presence of an instruction cache or prefetch buffer, some instructions (previously fetched) can be replayed as a consequence of EM fault injection on an ARM Cortex-M4 processor [22]. Yuce et al. [27] targeted a 7-stage-pipeline 32-bit LEON3 processor using voltage glitch injection, which can lead to the corruption of up to 5 consecutive instructions—each one in a different pipeline stage.

Dureuil et al. [11] proposed to infer a probabilistic fault model from a series of iterative fault injection campaigns running dedicated test codes. This model is the basis for the robustness evaluation of an application using a simulator. While the accuracy of the inferred probabilistic fault model is highly dependent on the first set of experiments, this work highlights the importance of crafting adequate test codes to analyse fault effects.

Kelly et al. proposed a methodology to characterize a fault model from laser injections on an 8-bit AVR microcontroller [13]. They performed a whole chip scan running test codes consisting of a single instruction from different classes. They mapped every instruction (class) to one or more sensitive areas on the chip and associated it with different observable effects. A deeper investigation revealed that laser beam injection on a selected sensitive area enables to skip an instruction. This methodology shows the importance to use specific test codes to determine fault effects at ISA level, yet the authors did not propose a classification of all observed effects.

The injection of faults on more complex SoC has recently raised the interest of security researchers. Attacks targeting an ARM Cortex-A9 platform—typical of early smartphones—attempted priviledge escalation in Linux using power glitches [23]

and to bypass secure boot protections using laser injection [25]. Majeric et al. [15] sucessfully injected EM-induced faults targeting a hardware AES implementation on a similar SoC. These early studies demonstrate the feasibility to target a specific SoC location without altering the behavior of the full chip. All these works report on the feasibility of exploiting fault injections, but their effects at ISA level are not yet understood. The characterization of fault effects on complex microarchitectures is yet to come, in particular the effects of EM fault injection.

3 EXPERIMENTAL SETUP

This section details the fault injection setup including the targeted hardware device and the software environment supporting the attack campaigns and analyses.

3.1 Device under test

All attacks have been realized on a widespread SoC typical of automotive and Internet of Things (IoT) applications. It consists of a dual-core 32-bit ARM Cortex-A9 MPCore on CMOS 40nm technology implementing the ARMv7-a ISA and clocked at up to 1 GHz. From experiments, we measured a performance around 80M instructions per second.

The following features of the Cortex-A9 microarchitecture stand out from previous studies of simpler in-order processors such as secure element micro-controllers [16, 27]:

- 8-stage variable-latency pipeline;
- superscalar, dual-issue instruction decoder;
- register renaming with out-of-order write-back;
- branch predictor (with a branch target buffer);
- 64B loop buffer bypassing the instruction cache.

The memory hierarchy features separate level 1 instruction and data caches of 32KB each, a unified level 2 cache of 256KB, along with several external memory interfaces (DDR3, NAND-Flash and NOR-Flash). The SoC is embedded in a development board suitable for instrumentation on the test bench, including a serial port (UART) and GPIO signals.

3.2 Attack setup

Electromagnetic fields (EM) are a suitable physical quantity to observe or disturb a running processor. We used the EM injection bench presented in Figure 1. It is composed of an automated XY-table where the target board is placed, a generator capable of injecting pulses from 6ns to 150ns and up to ±400V, an oscilloscope, an EM probe and an EM injector. The EM probe is a coil of copper used to monitor and time the moment of the injection very precisely, and to observe the effects. The EM injector is also a coil of copper, reeled around a piece of ferrite to focus the EM field. The board is connected through a serial port to a control PC to communicate data for analysis. Every component is connected to the control PC which synchronizes the necessary actions after an initial GPIO trigger.

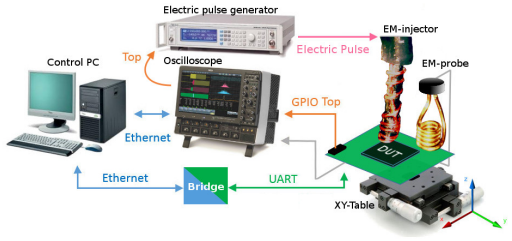

Figure 1: Diagram of the automated EM injection platform

Multiple parameters must be set when injecting a fault such as spatial location, temporal location, injection voltage, pulse polarity and duration. The parameters space is too large to be exhaustively tested. One has to focus on most meaningful parameters [22]. Our goal was to obtain a successful setup (spatial location, voltage and duration of injection) that injects exploitable faults with high probability. Following a trial and error scheme [11], we selected the sensitive location highlighted in Figure 2, a duration of 6ns and a pulse voltage of 310V. Lower voltage reduces the occurrence of successful faults while higher voltage mutes the SoC more frequently. Overall, the *timing offset* to inject the fault after the initial GPIO trigger is the only parameter varying during our experiments. It lets us target different instructions as the processor runs a given test code. In the following, an *attack campaign* refers to series of experiments on the same test code, each experiment consisting of a single EM pulse, and varying the timing offset of injection.

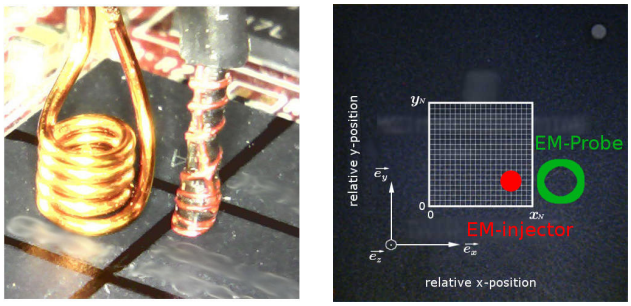

Figure 2: EM probe and EM injector (left) as well as their positioning over the SoC surface (right); the axis represents the SoC package

3.3 Software setup

We derive the most appropriate instant to inject an EM disturbance from the EM probe measurements. First of all, to

ease the detection of the targeted code's pattern in the EM trace, we inserted a sequence of 200 nops before and after the targeted region in the binary code. These sequences exhibit much lower EM emissions that are easily detected on the trace. Figure 3 shows the EM trace of an execution of a loop where the pulse has been injected at the fifth iteration. In Figure 3, we can observe the different loop iterations (small peaks) as well as nop sequences (flat parts), and the fault injection itself (high oversized and truncated peak).

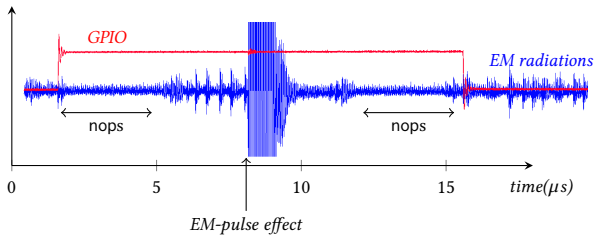

Figure 3: EM radiation emanating from the chip while injecting a fault during the execution of a loop

Inserting nop sequences and coupling those with a GPIO trigger help to determine an interval of maximal timing offsets covering a significant part of the running trace of the target code. E.g., multiple iterations for a loop. An attack campaign steps through that interval, injecting one pulse at a precise timing offset; we selected steps of 5ns to be shorter than CPU clock period. Besides, the execution of any test code ends with a sequence of instructions to store the contents of general-purpose registers `r0` to `r13` and to copy its operating memory areas into an *output buffer*. This output buffer is sent to the PC for analysis.

A dry run without any injection allows to collect reference values in the output buffer. Every subsequent run is subject to an EM fault injection. Comparing the output buffer with reference values is called *result analysis* in the following. We made the following implementation choices to facilitate result analysis:

- the general-purpose registers are initialized with distinct low Hamming weight values, to help narrowing down the analysis in case of (multiple) register corruption;
- similarly, the contents of operating memory areas is initialized with distinct "remarkable" values;
- the test codes are developed in C and compiled with clang/llvm version 6.0 at optimization level -O2; the generated assembly code has been manually rewritten to use all available registers (register allocators typically aims to spare registers) to better track the fault effects on intermediate computations until the end of the attacked code region.

For every fault injection, different types of results can be observed. They are classified and assigned to different groups among:

- *no fault*: the output buffer content exactly matches the reference one: the fault has no visible effect;
- *successful fault*: the output buffer contents does not match the reference one;
- *mute*—the board did not reply to the command; its status is undefined (the board needs a hard reset).

For each successful fault, the received values may be analyzed to help identify what happened at ISA level.

4 PRELIMINARY FAULT SENSITIVITY ANALYSIS

Attacks in the cryptography [9, 12] and systems [17] literature often target loops, aiming for early or deferred loop exit. We would first like to assess the practicality of such control flow disruption on sensitive loops.

4.1 Loop benchmarks

We selected two loops for their representativity of sensitive code and small enough to make the analysis of fault effects tractable:

- the *memcpy*-like function in Listing 1 is typical of firmware updates subject to buffer overflow attacks [17]; instead of copying, values in the source and destination buffers are added to ease result analysis;
- the *memcmp*-like function with early exit in Listing 2 resembles authentication schemes such as PIN verification [10]; again, to facilitate analysis, the outcome of the comparison is stored in a destination buffer at every iteration.

```
for(k=0; k<n; k++) {
  dst[k] += src[k];
}
```

Listing 1: Simple loop

```
for (k=0; k<n; k++) {
   dst[k] = (src1[k] != src2[k]);
   if (dst[k])
     break;
}
```

Listing 2: Multiple exit loop

The second loop is structurally more complex than the first one: it has two different exits. Moreover, its early exit condition depends on a comparison between data read from memory whereas the loop exit of the first loop only depends on a monotonically increasing counter.

4.2 First fault injection campaign

We conducted an attack campaign on both loops, stepping pulse injections through all instructions executed in a given iteration. Table 1 shows the results.

Table 1: Classification of EM pulse results

Code	No Fault	Mute	Successful faults
loop1	12663 (93.0%)	403 (2.9%)	555 (4.1%)
loop2	14287 (95.0%)	341 (2.4%)	372 (2.6%)

The fraction of successful faults is consistent with existing results on simpler devices [24]. Successful faults can be further divided into two classes: a *harmful fault* led to corrupted output values and taking the wrong exit (incorrect number of iterations or taking the wrong exit in the second loop); a *harmless fault* where output are not nominal but still indicate a correct number of loop iterations and taken exit branch. Table 2 shows the proportion of successful faults that are classified as harmful or harmless: 15% of the successful faults on the first loop break the security property and up to 80% of the successful faults on the second loop. Since the latter involves more registers and instructions in its exit conditions, its attack surface is higher, explaining the higher number of successful faults. These campaigns demonstrate our injection setup's effectiveness on simple loops on a complex microarchitecture.

Table 2: Breakdown of successful faults

Code	Harmful faults	Harmless faults
loop1	87 (15.7%)	468 (84.3%)
loop2	299 (80.4%)	73 (19.6%)

Our first attempt to explain these successful faults is to consider existing fault models, such as instruction skip [16] and register corruption [2, 18, 19]. We analyzed the faulted contents of the output buffer and observed that 8% (resp. 14%) of the faulty behaviors can be explained by an instruction skip (resp. register corruption). The remaining 78% successful faults cannot directly be explained without further investigation.

In practice, sensitive codes are often protected against instruction skip and register corruption [3, 7, 14, 20, 21]. For this reason, the following section studies the feasibility of attacking hardened codes.

4.3 Fault injection campaign on hardened code

We selected two hardening schemes from the literature, one dedicated to loops [20] and a general-purpose scheme [21].

The first scheme replicates the loop exit condition and the slice of instructions involved in its computation. Both conditions are compared at each iteration. Any fault impacting one of the computations is detected and leads to an error handler. This countermeasure is designed to resist under the instruction skip and register corruption fault models.

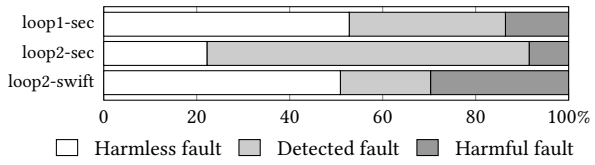

Figure 4: Successful fault breakdown on hardened loops

The second scheme, called *SWIFT*, is based on a duplication of all instructions. This scheme also introduces a signature-based control-flow integrity mechanism: every basic block has its own signature; two variables track signature updates at branches using both original and duplicated predicates. This scheme is designed to detect control-flow hijacking and any register corruption resulting from one *Single Event Upset* [2] where a single bit is flipped.

These countermeasures are designed to be target independent. The loop hardening scheme was validated on a simple micro-controller, the ARM Cortex-M3 [20]; implemented in the llvm middle-end, it is immediately applicable to more complex processors with similar ISA (ARMv7-m vs. ARMv7-a). SWIFT was originally designed for IA64, taking advantage of its instruction predication and wide-issue logic. We adapted it to the ARMv7 assembly language. Since both schemes behave very similarly on the simple control flow of the *loop1* benchmark, we report the SWIFT results on the *loop2* benchmark only.

In the following, *loop1-sec* and *loop2-sec* refer to the application of the loop hardening scheme to both loop benchmarks, whereas *loop2-swift* corresponds to the second loop hardened with SWIFT.

We conducted attack campaigns on these hardened loops. The results include an additional class called *Detected fault*, counting EM pulses resulting into a detected fault with a corrupted output buffer.

Figure 4 shows interesting results. Hardening the code significantly reduces the probability of a successful attack, and this is the case for both countermeasures. The countermeasures do detect some of the faults, this is consistent with our preliminary analysis that some injections led to skipping an instruction or corrupting a register. However, as numerous faults are not detected, these results also suggest that EM pulses may induce complex effects at software level that are not caught by the countermeasures. Further fault characterization is required to better understand the achievable effects visible at ISA level.

5 CHARACTERIZING AND MODELING FAULTS

To characterize the different fault effects, let us step back from loop benchmarks and switch to a set of much simpler (synthetic) codes. The simpler codes are designed to exercise a narrower set of microarchitectural elements (registers, functional units, data path). The attack campaign considers codes of increasing complexity and microarchitectural footprint. This step by step methodology facilitates the isolation of (faulty) effects; it also enables designing subsequent experiments from the previously observed effects.

5.1 Sequence of nop

Without prior knowledge on the effect of electromagnetic injection on this complex SoC, we started from the most basic example consisting of a linear sequence of `nop` instructions.

On a campaign of 3000 runs, we observed 142 *mute*s but no successful fault. It appears that, at ISA level, neither registers nor instructions have been corrupted. This was somewhat surprising to us given the observed register corruption on the loop benchmarks. One may immediately deduce that there is no general instruction replacement going on: any replacement, if it occurs, has no effect on the architectural registers. In particular, this experiment rules out any of thin air instruction replacement on or before the decode stage.

By construction, it is not possible to detect instruction skip on such a code, which brings us to the following experiment.

5.2 Single counter incrementation

Our second test example consists of a sequence of `add r0, r0, #1` instructions. By incrementing only one register, we attempt to correlate corruption effects with instruction operands.

Among all successful faults, only `r0` has been corrupted. This is an important result consistent with the previous experiment on a sequence of `nop`s. It also indicates that the microarchitecture does not implement `nop` as an instruction involving one or more register operands (e.g., `add r0, r0, #0`).

We also analyzed the faulted values observed in `r0`. Most of these correspond to twice the expected (unfaulted) value plus a small negative offset[2]. Occasionally three or four times. This could be explained by the selection of an architectural or bypass logic register instead of the expected constant #1. The offset is proportional to the number of instructions remaining to be executed after the fault affecting `r0`. We could confirm this timing correlation on the EM trace. The proportionality is faithfully observed except in very rare cases where the offset is smaller; this suggests (rare) additional effects such as instruction skip and bit flips. At ISA level, it can be modeled as the replacement of one or more instructions `add r0, r0, #1` by `add r0, r0, r0`. Given the selectivity of the observed values, we can also rule out a functional unit replacement, e.g., `sll r0, r0, #1` would be an alternative replacement but such an explanation would authorize replacements with more arithmetic and logic operations leading to a much wider distribution of observed values.

We then studied the sensitivity to the register operand number. Replacing `r0` with `r5` in the sequence of additions showed that only `r5` was faulted, which is consistent with the previous findings. Yet the faulted values now behave differently: we now observe values consistent with a majority of replacement with `add r5, r5, r0` and a minority with `add r5, r5, r5`! This motivates further study of the influence of the register operand number in the fault effects, which will be the purpose of the following experiment.

So far, 96% of these two experiments can be explained with the replacement of one operand, but the exact nature of this replacement remains obscure at this point. Many microarchitectural effects could explain the observed results: corruption may take place in the registers themselves, in selectors/multiplexers addressing architectural or physical register banks, or the bypass logic. We are also not sure how many registers and/or instructions have been corrupted, but several observations hint at multiple faults, possibly correlated/coupled..

5.3 Multiple counter incrementation

The next experiment runs a sequence of `add rx, rx, Cx` instructions where `Cx` is a different constant for every register, with `x` ranging from 0 to 9. This configuration has four advantages. First of all, since the sequence features additions only, it mitigates complexity. Second, it touches multiple registers, but none of these induces any dependence. Third, the distance between two instructions modifying the same register is 10 instructions, limiting time-dependent effects associated with out-of-order execution or pipelining. Finally, every register is incremented using a different immediate constant (`Cx`). We used prime numbers to reduce chances of collision in corrupted values.

This campaign hints at the number of instructions or registers that may be impacted by a single injection, as well as the potential interaction between simultaneously executed instructions. And our observations indicate that many registers can be corrupted by a single fault. As previously observed, only registers occurring as instruction operands are corrupted, and we also observe that all registers have the same sensitivity to corruption or to be used as a source operand in a replacement (resulting in the corruption of another register). Figure 5 shows the distribution of the number

[2]Small relative to the initial value of `r0`, see Section 3.

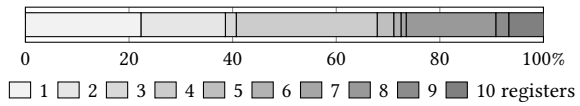

Figure 5: Distribution of the number of corrupted registers per successful injection

of corrupted registers impacted by one single injection. Multiple register corruption is more likely to occur: the average number of corrupted registers per successful fault is 3.8 and all alive registers can be corrupted at once.

The analysis is already delicate, even on such a simple code. Yet some behaviors can still be explained thanks to the setup of initial register values. We observed that some registers received the values expected for other ones. At ISA level, it can be modeled as the replacement of `add ry, ry, Cy` by `add ry, rx, Cx` where `rx + Cx` is the value expected for another register. Some corrupted values can also be explained by an operand replacement. All these observed faults still seem to come from the selection of an architectural or bypass logic register instead of expected operands.

Some corrupted registers also derive from in-flight values (from the register itself or another one) with several flipped bits. Also, some corrupted register contents match precisely the reset of their 16 most-significant bits. The experiment highlights another frequent effect resulting into a double register corruption. Considering the two instructions `add r2, r2, #5` and `add r3, r3, #7`, we observed correlated corruptions of the form

$$\mathit{faulted\_value}(\texttt{r2}) - \mathit{expected\_value}(\texttt{r2}) = -5$$
$$\mathit{faulted\_value}(\texttt{r3}) - \mathit{expected\_value}(\texttt{r3}) = 7$$

These can be modeled by one instruction skip (here `add r2, r2, #5`) and the replay of an instruction (`add r3, r3, #7`). The frequency of this effect targeting different registers with different values rules out another cause such as bit flips. Some groups of corrupted registers do not belong to consecutive instructions in the program order; this is most likely due to superscalar out-of-order execution.

5.4 Isolating effects

In the previous attack campaign, multiple operands are often corrupted simultaneously across one or more instructions. The injection pulse duration is really short (6ns) and does not directly explain this behavior. We thus suspected side effects due to pipelining where a single glitch can influence multiple instructions [27]. In an attempt to reduce the number of simultaneous corruptions, we replayed the last campaign inserting one or several `nop` instructions between every addition. The average number of simultaneously corrupted registers effectively went down from 3.8 to 3.2 with 1 `nop`, 1.8 with 2 `nops`, 1.7 with 4 `nops`, and to 1.1 with 50 `nops`. These experiments confirm that a single fault may affect several in-flight instructions.

5.5 Fault models and classification

To summarize our findings, we observed and analyzed different fault effects and highlighted different of their properties: a register not used by the processor has a very low probability of being corrupted; all registers seem to have the same sensitivity despite a fixed EM-pulse spatial position; registers value can be directly corrupted (multiple bit-flips or most-significant half-word reset); all alive registers can be corrupted at once and inserting `nop` instructions between every instruction reduces the number of corrupted registers. Finally, instructions can be corrupted in multiple ways. They can either be skipped (no write, no operation) or replayed, or their operands can be substituted mainly by operands from other instructions being processed.

These are general observations and analyzed behaviors for the fault injection setup. Before confronting and applying these findings to complex test codes, we further classify the results of a single injection according to the type of effect observable at ISA level:

- *Instruction skip*: one instruction is skipped.
- *Register most-significant half-word (mshw) reset*: the corrupted value corresponds to a reset of the 16 most-significant bits of the expected value.
- *Register corruption*: the corrupted value is either derived from another existing one (bit flips) or does not seem correlated to any in-flight value.
- *Source operand substitution*: an immediate value or a source register of an instruction is not the expected one; the corrupted value usually comes from the immediate or source register operand of another instruction.

In most cases, results cannot be explained directly by a single one of the four previous groups. Instead, a combination of these effects is needed, often with strongly coupled *composite effects*:

- Combined skip of an instruction and replay of a previously executed one.
- Combined register corruptions leading to correlated corrupted values.
- Repeated occurrences of the same fault effects, such as instruction skip or register corruption with or without correlated values.
- Sometimes, *mixed faults* that correspond to two or more distinct of the above, without any apparent correlation.

Moreover, the instructions impacted in these fault effect do not necessary form consecutive intervals due to out-of-order execution. Also, the occurrence rate of all these fault patterns are not equal. The precise effect of one injection is

known to be difficult to predict. However, we observed several times some faulty outputs with the same values. Hence, the probability to reproduce some faults is far from negligible.

Based on these analysis and classification, we can revisit the yet unexplained results from more complex loop examples. This is the purpose of the next section.

6 LOOPING BACK

A major difference between the previous test codes and the more complex loops resides in the instruction mix. In addition to arithmetic operations, loops contain memory accesses (load/store) and branches, and access more diverse data coming from memory. Also, the results are more difficult to analyze due to the propagation of the fault effects until the contents of the output buffer can be retrieved.

6.1 Loop analysis w.r.t. fault classification

To analyze results from successful faults on loop test codes, we first considered the fault effect classification established in the previous section. As a faulted contents of the output buffer can sometimes be explained with more than one single ISA-level effect, we prioritized the models in reverse order of their (apparent) complexity: we first look for a skipped instruction, and subsequently for operand substitution, register corruption, composite faults, and eventually mixed models. We managed to classify a significant part of the results, but some effects remain unexplained when considering these fault models only.

Some values loaded from memory areas have been corrupted while source memory remains unaltered. These values follow two patterns: either similar to alive register contents with several flipped bits or values apparently not correlated to any available values but repeated over injections. The latter ones can be explained by values from other memory areas and thus be related to an operand substitution in the load instruction. The former pattern can be modeled by a register corruption as already observed. However, we cannot ensure that it does not come from a corruption of the memory transfer itself. Moreover, we encountered this case several times highlighting a sensitivity of load instruction. We call this a *load corruption* effect.

The second of these new effects is much more original and is best understood at the level of the control-flow graph (CFG) of the function. Some injections on hardened versions can only be explained by a jump from the end of a basic block (linear sequence of instructions without a branch) *directly to the beginning of another one*. When the target block is an illegal destination block in the CFG, we name this effect a *magic edge*. We did not observe any random jumps to the middle of basic blocks. This behavior hints at the corruption of the branch target buffer or another branch prediction mechanism, rather than a more direct effect on the program counter that would have led to a greater diversity of target addresses (i.e., one would have observed jumps into the middle of basic blocks).

The occurrence of such a magic edge may induce an early exit that bypasses all the control blocks of a loop-centric countermeasure. It is not detected by SWIFT either, although it is meant to protect against a very wide range of control-flow hijacking patterns. Overall, this new effect provides a powerful attack vehicle, although it seems difficult at this point to precisely control the target of a magic edge.

6.2 Fault classification on loops

Let us now classify all the successful faults observed during the different attack campaigns on loop benchmarks.

As the hardened loops leave some faults undetected, we applied a costly combination of both countermeasures to the *loop2* example from Listing 2. We first applied loop scheme from [20] and hardened the resulting code using SWIFT scheme. We refer to this version as *loop2-sec+swift*. We also ran an attack campaign on this version.

Figure 6 shows the distribution of successful faults according to all the fault models established in this paper, for all loop benchmarks. Note that the ratio of instruction skip may be slightly skewed (in excess) due to the priority we assigned to this model in the classification methodology (cf. Section 6.1). Also, the single faults are displayed first for every fault model, before any composite faults. Note that 15% of the successful faults remain unexplained. Further investigation is needed, involving additional attack campaigns and analyses.

The results also show that all fault models have been observed in every attack campaign. The proportion of a given fault model varies widely across the test codes, but composite effects are always far from negligible.

Let us now focus on the classification of the fault effects that manage to bypass the software countermeasures of the hardened loops, by considering only harmful (undetected) faults. Figure 7 selects some of the effects listed in Figure 6, detailing the breakdown of successful faults.

We can observe that most of the single faults (from Figure 6) are either detected or harmless. The harmful single faults, magic edge excluded, are most often due to an operand substitution (destination register) that leads to a double register corruption that bypasses the detection mechanisms.

Harmful faults are mainly due to composite faults or magic edges. However, composite faults affecting differently an original computation and its duplicated version are detected: 48% of the composite faults of Figure 6 are no more present

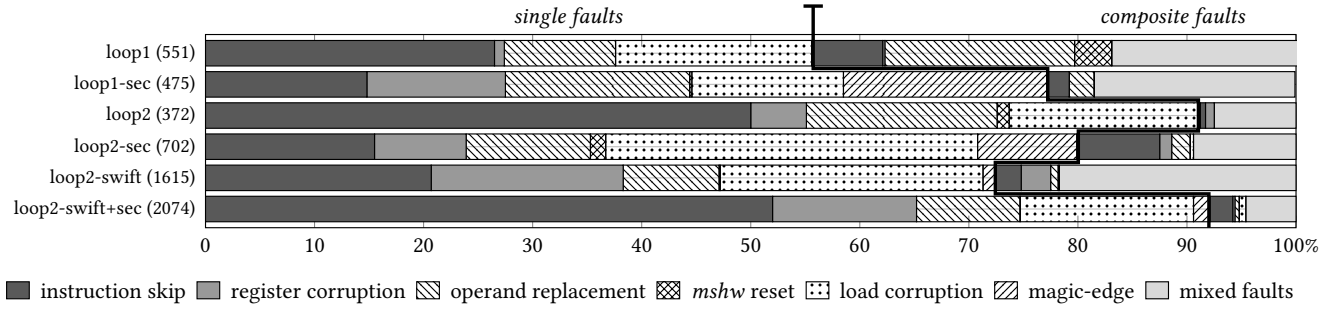

Figure 6: Fault distribution according to the different models and the multiplicity of faults for each loop code (with the total number of *successful faults* for each code)

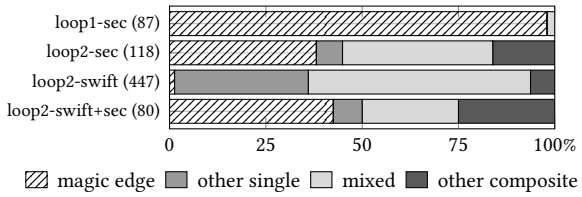

Figure 7: Distribution of effects leading to a harmful fault (with the total number of harmful faults for each code)

in Figure 7. The remaining composite faults bypass the countermeasures and represent a large fraction of the undetected faults.

Magic edges represent a real threat for all the hardened version. A single magic edge cannot be detected by the first (loop-centric) countermeasure. For the hardened versions of *loop2*, around 33% of mixed faults are a combination of a magic edge and another corruption. In particular, stacking the two countermeasures (version *loop2-swift+sec*) is not sufficient to entirely protect against magic edge and composite effects.

Also, code size may grow exponentially with the stacking of countermeasures: ×3 for *loop2-sec* and *loop2-swift* and ×11 for *loop2-swift+sec*. This advocates for further investigation of fault effects on complex microarchitectures to provide a more complete characterization. It would enable the deployment of protection schemes intrinsically resistant to multiple faults. A mix of software and hardware countermeasures seems particularly attractive to mitigate overhead.

7 CONCLUSION

We first established the vulnerability of loop-based applications to electromagnetic (EM) fault injections targeting an out-of-order superscalar processor. Following a black-box approach and relying on widely available equipment, we demonstrated the exploitation of this vulnerability by locally disrupting the control flow of hardened versions of sensitive loops. We also demonstrated that state-of-the-art software countermeasures could reduce the probability of success of such attacks, but fail to achieve levels of protection comparable with running the same hardened loops on simpler micro-controllers.

We proposed a step-by-step methodology to characterize the fault effects at the level of the instruction set architecture (ISA), starting from extremely simple yet carefully designed code fragments, and following on with loop benchmarks including loops hardened with software countermeasures. We applied this methodology to identify a range of fault models, some already well-known observed on simpler processors such as instruction skip or register corruption, as well as newer ones, potentially very powerful and specific to complex microarchitectures. Among the latter, we observed and classified correlated effects such as *instruction skip and replay*, *operand substitution* and *magic-edge* control flow hijacking, as well as compositions of such effects.

These results explain the wide vulnerabilities left open by state-of-the-art software countermeasures designed for conventional fault models. In particular, they highlight the fundamental weakness of replication-based software-only protections, aimed at single, localized faults. The correlated and mixed fault models we characterize seem to evade such replication tactics. Our results motivate further research in two directions: (1) broadening the characterization of EM fault injection at ISA level, covering microarchitectures from different vendors taped out with different technological nodes and exploring more die locations, and (2) designing and evaluating hardware or hybrid hardware-software countermeasures capable of detecting multiple fault effects with strong correlations, as well as advanced control flow hijacking.

REFERENCES

[1] C. Aumüller, P. Bier, P. Hofreiter, W. Fischer, and J.-P. Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. Cryptology ePrint Archive, Report 2002/073, 2002.

[2] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[3] T. Barry, D. Couroussé, and B. Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *Workshop on Cryptography and Security in Computing Systems*, CS2, 2016.

[4] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT*, pages 37–51. Springer, 1997.

[5] E. Brier, D. Naccache, P. Q. Nguyen, and M. Tibouchi. Modulus fault attacks against rsa-crt signatures. Cryptology ePrint Archive, Report 2011/388, 2011.

[6] Common Criteria. Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 5.

[7] R. De Keulenaer, J. Maebe, K. De Bosschere, and B. De Sutter. Link-time smart card code hardening. *International Journal of Information Security*, pages 1–20, 2015.

[8] A. Dehbaoui, J. Dutertre, B. Robisson, P. Orsatelli, P. Maurine, and A. Tria. Injection of transient faults using electromagnetic pulses – practical results on a cryptographic system. Cryptology ePrint Archive, Report 2012/123, 2012.

[9] A. Dehbaoui, A. Mirbaha, N. Moro, J. Dutertre, and A. Tria. Electromagnetic glitch on the AES round counter. In *COSADE*, 2013.

[10] L. Dureuil, G. Petiot, M. Potet, T. Le, A. Crohen, and P. de Choudens. FISSC: A fault injection and simulation secure collection. In *SAFECOMP*, 2016.

[11] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière. From code review to fault injection attacks: Filling the gap using fault model inference. In *Smart Card Research and Advanced Applications Conference (CARDIS)*, pages 107–124. Springer, 2016.

[12] T. Espitau, P. Fouque, B. Gérard, and M. Tibouchi. Loop-abort faults on lattice-based signature schemes and key exchange protocols. *IEEE Trans. Computers*, 67(11):1535–1549, 2018.

[13] M. S. Kelly, K. Mayes, and J. F. Walker. Characterising a cpu fault attack model via run-time data analysis. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 79–84, 2017.

[14] J.-F. Lalande, K. Heydemann, and P. BerthomÃĪ. Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In *Computer Security - ESORICS*, volume 8713 of *LNCS*, pages 200–218. Springer, 2014.

[15] F. Majéric, E. Bourbao, and L. Bossuet. Electromagnetic security tests for SoC. In *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 265–268, 2016.

[16] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88, 2013.

[17] S. Nashimoto, N. Homma, Y.-i. Hayashi, J. Takahashi, H. Fuji, and T. Aoki. Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering*, 2016.

[18] S. Ordas, L. Guillaume-Sage, and P. Maurine. EM injection: Fault model and locality. *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 3–13, 2015.

[19] S. Ordas, L. Guillaume-Sage, K. Tobich, J.-M. Dutertre, and P. Maurine. Evidence of a larger em-induced fault model. In *Smart Card Research and Advanced Applications (CARDIS)*, pages 245–259. Springer, 2015.

[20] J. Proy, K. Heydemann, A. Berzati, and A. Cohen. Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Transactions on Architecture and Code Optimization*, 14(4):36, 2017.

[21] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[22] L. Rivière, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage. High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2015.

[23] N. Timmers and C. Mune. Escalating privileges in linux using voltage fault injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8, 2017.

[24] N. Timmers, A. Spruyt, and M. Witteman. Controlling PC on ARM using fault injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35, 2016.

[25] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux. Laser-induced fault injection on smartphone bypassing the secure boot. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 41–48, 2017.

[26] B. Weyl. Secure on-board architecture specification. EVITA project Deliverable D3.2, 2011.

[27] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont. Software fault resistance is futile: Effective single-glitch attacks. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58, 2016.

[28] B. Yuce, P. Schaumont, and M. Witteman. Fault attacks on secure embedded software: Threats, design, and evaluation. *Journal of Hardware and Systems Security*, 2(2):111–130, 2018.