Fault Injection Characterization on modern CPUs
From the ISA to the Micro-Architecture

Thomas Trouchkine[1], Guillaume Bouffard[1,2][0000−0002−2046−369X], and Jessy Clédière[3]

[1] National Cybersecurity Agency of France (ANSSI),
51, boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France.
thomas.trouchkine@ssi.gouv.fr
[2] Information Security Group, École Normale Supérieure
46 rue d'Ulm, 75230 Paris Cedex 05, France
guillaume.bouffard@ens.fr
[3] CEA, LETI, MINATEC Campus, 38054 Grenoble, France
jessy.clediere@cea.fr

Abstract. Recently, several Fault Attacks (FAs) which target modern Central Processing Units (CPUs) have emerged. These attacks are studied from a practical point of view and, due to the modern CPUs complexity, the underlying fault effect is usually unknown.
In this article, we focus on the characterization of a perturbation (the fault model) on modern CPU. For that, we introduce the first approach to characterize the fault model on modern CPU from the Instruction Set Architecture (ISA) level to the micro-architectural level. This fault model helps at determining which micro-architecture elements are disrupted and how. Our fault model aims at finding original attack paths and design efficient countermeasures. To confront our approach to real modern CPUs, we apply our approach on ARM and x86 architectures CPUs, mainly on the BCM2837 and an Intel Core i3.

1 Introduction

Nowadays, mobile devices are widely used. They are based on high performance System on Chips (SoCs) which embed performance oriented Central Processing Units (CPUs). With all their optimizations, these modern CPUs have shown flaws in their security [8, 11].

Since 2015, several Fault Attacks (FAs) on modern CPUs have been presented, some are new and some others already applied on Micro-Controller Units (MCUs) CPUs [12]. These attacks are very practical and, due to the complexity of modern CPUs, the underlying fault effect is usually unknown. The fault effect knowledge is mandatory for building efficient countermeasures and evaluating the impact of an attack. Therefore, we think that fault characterization on modern CPUs is an important work for the future.

Many fault model characterizations have been done on MCUs but only few on modern CPUs. For determining the fault model on such targets and for making it reproducible, we propose a characterization method and we show its applicability on modern CPUs, based on ARM or x86 cores. The proposed method is inspired by all the works done on MCUs and integrates the different approaches introduced in these different works.

This article is organized as follows. The section 1 presents the background about fault model characterization, our motivations and the modern CPUs specificities. Section 2 introduces a model for CPUs on which we base our method, section 3 describes our method and section 4 presents its application with experimental results. The section 5 concludes and opens on future works.

1.1 Related Works

Practical fault attacks emerged on 2002 with an optical fault attack [23]. Since then, several practical attacks have been applied on cryptographic implementations [20, 21, 22] or on secure softwares [4, 5, 29]. These attacks aim at breaking the implemented algorithm and therefore focus on obtaining a precise fault. Therefore, only few information is given about the characterization process.

The seminal work on the fault model characterization of a CPU was published in 2011 and focus on the clock glitch effects on an ATMega163 MCU [3]. During the next years, many works have been done on the fault model characterization of MCUs CPU and memory [6, 9, 10, 13, 14, 15, 19, 31]. These works gave useful information about the fault effect on CPUs micro-architecture. This knowledge helped in building countermeasures. However, in 2015 and 2016, several software countermeasures have been shown to be ineffective against certain class of FAs [16, 30].

Since 2015, researchers had started to focus on modern CPUs. Their works aim at breaking complex security features like secure boot [26, 27], Trusted Execution Environments (TEEs) [24] or kernel security mechanisms [25, 28] *via* fault injection. All these works focus on the mobile devices security area, where modern CPUs run a complex Operating System (OS) like Linux, Android or iOS.

These articles focus on the attacks practicality and do not present any methodology about the fault characterization. In 2019, Proy *et al.* [18] propose the first fault characterization work on ARM Cortex-A9 based CPU for evaluating their countermeasures against FAs. This work is a first step for fault characterization on modern CPUs. However, the applied method is not clearly described. The authors realize several classical tests to determine how the program execution is modified by the fault. They mainly focus on the Instruction Set Architecture (ISA) layer whereas we propose to determine micro-architectural effects from the ISA fault model.

1.2 Motivations

Regarding the state of the art presented in section 1.1, we think that a fault characterization method on modern CPUs is needed to design efficient countermeasures. Also, these systems are widely used in mobile devices which tend to be integrated almost everywhere in the future and used for critical usage as banking, healthcare, *etc*, enforcing their need in security and reliability.

1.3 Contribution

We propose a method that would allow us to characterize fault model on a modern CPU. This method is based on an ISA fault model determination but is oriented to also provide information about the micro-architectural fault effect. Therefore, we have two contributions, a modern CPU model and then a fault characterization method built on this model. The introduced modern CPU model is easily adaptable to match with MCU CPUs. This makes our approach adaptable to any type of CPU matching this model, even most MCU CPUs.

2 Modern CPU modeling

2.1 Modern CPUs specificities

The previous works on fault characterization on MCUs give information about what we can expect from a characterization and how to do it. Unfortunately, modern CPUs are different from MCU CPUs as shown on figure 1. Indeed, they are more complex and embed several cores with optimizations like out-of-order execution, speculative execution, branch prediction, *etc*. They also have multiple levels of caches and a Memory Management Unit (MMU) abstracting their memory.

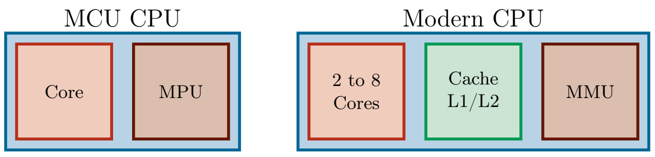

Fig. 1: Micro-controller and modern architectures

Even if their specification is public, another issue with CPUs is that their implementation is not available. Moreover, most of the time, debug tools for these platforms are either only partially open or not available. Therefore, the only way to retrieve information is through the ISA layer. In other words, as we do not have access to the physical layout, we aim at characterizing the fault model at the program level. This is a real issue as for building efficient countermeasures, a software knowledge is not enough, but a micro-architectural fault effect knowledge is also necessary. Therefore, a method that enables to retrieve information on the micro-architectural CPU behavior, based on the ISA fault model, is required.

2.2 Modern CPU model

This section aims at offering a complete and comprehensive description of modern CPUs. We start from the observation that any CPUs can be modeled with three functional elements.

- A pipeline which fetches, decodes and executes instructions.
- The registers where the manipulated data are stored.
- A memory storing the instructions and some data.

Actually, the memory is external to the CPU. However, there is an internal one, called *cache*, where a part of the external memory is copied. The three functional elements are based on Micro-Architectural Blocks (MABs) as introduced in figure 2.

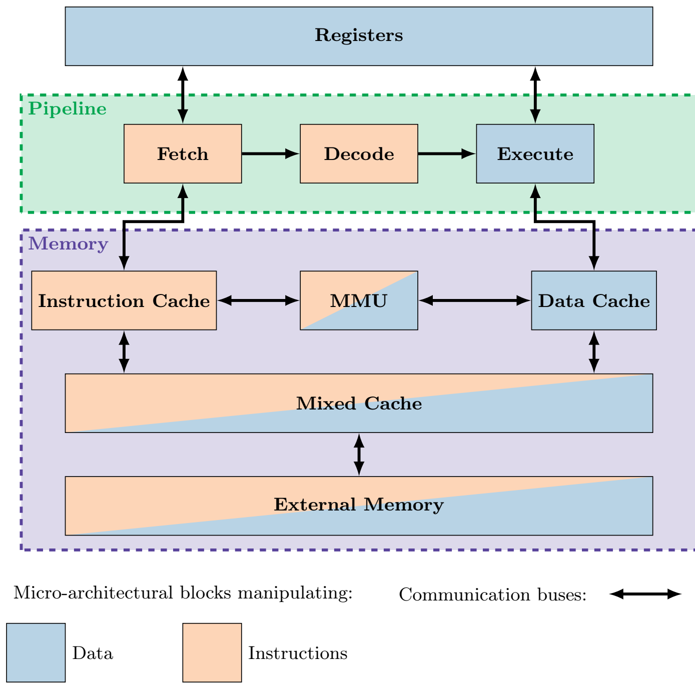

Fig. 2: CPU model

The *pipeline* fetches and decodes the instructions then the *execute stage* realizes the operation. In modern CPUs, these blocks have several optimizations that we do not consider in our model. The memory relies on several cache level and a MMU. Usually, CPUs have a mixed architecture where the data path and the instruction path are separated only at the lowest cache level. The instructions and the data are not differentiated in the high cache level (L2/L3) and the external memory, this is a *Von Neumann* architecture. But, in the lowest level of cache (L1) the instructions and data are separated, this is an *Harvard* architecture. As modern CPUs have both organizations, they are said to have a mixed architecture.

Physically, a core corresponds to the registers, the pipeline, the MMU and the cache. The CPU is composed of one or more cores, but in the end, its behavior corresponds to this model.

This model is usually used for fault characterization on MCUs as all fault models are explained by a MAB perturbation presented in figure 2. As most of the MCUs have only one core in their CPU, this model fits them well. The question is to know whether this model is still relevant for a multi-core optimized CPU. We will show that, on average, it is enough for determining more than 80% of the fault effects.

3 Fault effect analysis on CPU

During a Fault Injection (FI), one or several CPU MABs are disturbed. As they can all be perturbed during a fault injection, the full fault effect characterization can be a complicated process. However, according to the previous works, in most cases, the fault affects only a single MAB [9, 19]. We actually verified this assumption on modern CPUs. Under this simplified paradigm, the fault characterization problem aims at determining which MAB is faulted and how.

To reach our objective, the proposed method consists in realizing a fault during the test program execution and in determining the micro-architectural fault that can explain the observed misbehavior. An underlying assumption is that the fault affects the same MABs in the same way independently of the executed program. This has been experimentally verified; depending on the processor state some new effects can appear, but a set of usual effects remains.

3.1 Determining the faulted element

The method general idea is to apply a top-down approach. We start by determining whether the fault affects the registers, the pipeline or the memory. Once we know which element is affected, we determine which of its MABs is faulted.

To achieve this, we rely on the available registers observation and the executed instructions knowledge. The way they are faulted gives information about the faulted element. To discriminate which element is faulted, we repeatedly execute the same instruction as introduced in listing 1.1 (for ARM) and listing 1.2 (for x86) on a known state CPU.

Listing 1.1: `mov r0, r0` (ARM)

```
mov r0, r0 // Several times
```

Listing 1.2: `mov rax, rax` (x86)

```
mov rax, rax // Several times
```

These instructions are given as examples but have two important properties. First, they do not fetch any data from the memory, which means that a fault in the memory can only affect the instructions, which simplifies the analysis. Secondly, the instructions do nothing and are therefore semantically equivalent to `nop`. This is helpful since a modification of the registers state can only be caused by a fault[4] and its effect is not drowned within a complex program.

Disturbing the program execution will give a distribution of faulted values in the registers. The next step consists in determining whether these faulted values come from a fault on the manipulated data or on the instructions. Indeed, the execution of the n^{th} program instruction by the CPU can be modeled such as in (1):

$$s_{n+1} = ins_n(s_n), \tag{1}$$

where s_{n+1} is the CPU state after the execution of the n^{th} instruction ins_n. The CPU state corresponds to all its registers and is usually named the data. An instruction is composed of three elements: an opcode encoding the operation to do, a reference to the destination register and reference(s) to the operand(s). These operands can be registers or immediate values. Depending on the architecture, the encoding of this information may vary but they are always present.

When there is a fault during an instruction execution, we assume here that it either applies on the data or on the instruction. We experimentally verified this assumption. Therefore, the faulted instruction execution can be modeled such as in (2).

$$\tilde{s_{n+1}} = \tilde{ins}(\tilde{s_n}), \tag{2}$$

where $\tilde{x}$ denotes the faulted representation of x. From this representation, we can define the fault model f_{data} on the data as introduced in (3), and the fault model f_{ins} on the instruction as presented in (4).

$$\tilde{s_n} = f_{data}(s_n), \tag{3}$$

$$\tilde{ins} = f_{ins}(ins). \tag{4}$$

These fault models can have different descriptions to match with the different underlying fault causes. The data fault types and their corresponding MABs are presented in table 1.

Based on the figure 2 and table 1, it is possible, from these fault types, to determine which MABs have been faulted. In the case of a register corruption, it is straightforward that the registers are faulted. If there is a memory corruption, the cache storing the data or the data bus is faulted. In the the *bad fetch* case, either the cache has loaded the wrong data or the MMU has failed the address translation.

[4] This assumption must be carefully studied as some registers like the Program Counter (PC) are always modified independently of the executed instruction.

Faulted element	Data				
Fault type	Register corruption	Memory corruption		Bad fetch	
Faulted MAB	Registers	Cache	Data bus	Cache	MMU

Table 1: Data fault models

For the instructions, the fault types, presented in table 2, are corruption and bad fetch.

Faulted element	Instruction				
Fault type	Corruption			Bad fetch	
Faulted MAB	Pipeline	Cache	Bus	Cache	MMU

Table 2: Instruction fault models

If an instruction corruption is observed, the fault affects either one of the pipeline MABs or the cache or the instruction bus. In the case of a bad fetch, either the instruction cache has loaded the wrong instruction or the address translation has failed.

Regarding the test code presented in listing 1.2 and listing 1.1, the data fault models *memory corruption* and *bad fetch* cannot appear as there is no data fetched from the memory. Therefore, we can focus on the remaining fault models and this is enough for determining which element among the registers, the pipeline or the memory has been faulted.

4 Experimental analysis

This section aims at applying the approach introduced in section 2. We present the experimental protocol and the corresponding results on two targets, a BCM2837 from a Raspberry Pi 3 model B board and an Intel Core i3 from a classical computer.

4.1 BCM2837

Now that we introduced a method which determines the affected element, we decide to apply our approach on an experimental work. The presented work comes from an attack campaign realized on a BCM2837 SoC from a Raspberry Pi 3 model B board. The tested code is the repetition of the `orr r3, r3` instruction and the observed registers initial values are presented in table 3. These values

Register	Initial value	Register	Initial value
r0	0x80000001	r5	0x04000020
r1	0x40000002	r6	0x02000040
r2	0x20000004	r7	0x01000080
r3	0x10000008	r8	0x00800100
r4	0x08000010	r9	0x00400200

Table 3: Observed registers initial values

are chosen to be identifiable and hard to compute from each others with simple operations (`or`, `xor`, *etc*).

This setup has been disturbed using ElectroMagnetic Pulse (EMP). The obtained faulted values are presented with their probability of appearance in figure 3.

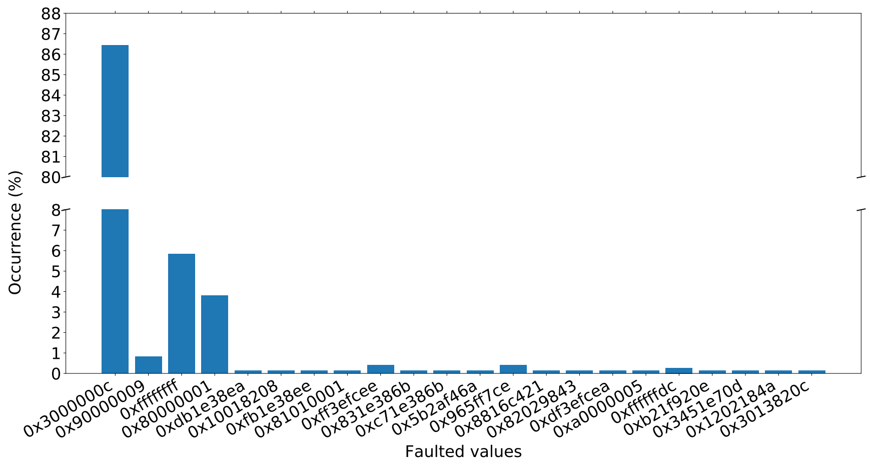

Fig. 3: Faulted values distribution with their occurrence probability obtained from an ElectroMagnetic Fault Injection (EMFI) campaign on a BCM2837.

Several values appear with different probabilities, however there are always some outstanding values that are frequently obtained. Here, these values are `0x3000000c` (86.45%), `0xffffffff` (5.83%) and `0x80000001` (3.79%). We ignore the other values as the latter ones are sufficient for demonstrating the method relevance.

Register corruption analysis. According to the method presented in section 2, we want to check if the perturbation corrupted the registers. To do so, we need to know the registers initial content. In our experiment, the only faulted registers are `r0`, `r1` and `r3`.

Giving the faulted and the initial values, it is possible to determine the fault model on the registers. The fault model we consider for register corruption is the masking fault model, this defines the fault as a logical mask applied to the initial value. In other words, the random variable f_{data} is one of the following functions[5]:

$$f_{\texttt{xor},e} : x \to x \texttt{ xor } e,$$

$$f_{\texttt{and},e} : x \to x \texttt{ and } e,$$

$$f_{\texttt{or},e} : x \to x \texttt{ or } e,$$

with e the error viewed as a logical word with the same size as x.

As several fault models can explain the obtained faulted value, we consider that a fault model is relevant if it explains the obtained faulted value for at least two different experiments. In our context, the only observed *register corruption* fault model is when the faulted value is `0xffffffff` and the corresponding function is $f_{\texttt{or},\texttt{0xffffffff}}$. This fault model appears in around 5% of the cases.

Instruction corruption analysis. As the register corruption analysis was inconsistent for some faulted values, the next step consisted in checking if a faulted instruction can explain them. The idea here is to first determine the instructions that, from the registers initial state, explain the faulted value.

Regarding the faulted values and the registers initial state we observe that `0x3000000c` can be obtained with the `or` between `r2` and `r3` and that `0x80000001` can be obtained by moving the value in `r1` into the faulted register. The corresponding faulted instructions are `orr r3, r2` and `mov r3, r1`.

Because the initial instruction is known, we can determine the fault model f_{ins}. We decided to consider a fault model that modifies the elements (opcode, operands, *etc*) constituting the instruction. The faulted instruction $\widetilde{ins}$ is derived from the initial instruction ins. Determining the fault model consists in determining which part of the instruction was corrupted and how.

In this experiment, the fault model corresponding to the `0x3000000c` faulted value is that the instruction second operand is set to `r2` and correspond to the `orr r3, r2` faulted instruction. This happens in around 85% of the cases and was tested with other instructions. The fault model corresponding to the `0x80000001` faulted value is that the opcode is set to a `mov` and the second operand to `r1`, it happens in around 3.5% of the cases.

Conclusion. During this experiment, the faults may affect the pipeline or the registers. We have determined the faulted element with their corresponding fault model for the three main cases (*i.e.* those with greatest occurrence probability in figure 3). This covers 96.07% of the observed faults and it experimentally validates that the model is relevant for this CPU.

[5] It is possible to consider f_{data} as the combination of these functions with different errors, but this do not change the way to apply our methodology.

4.2 Intel Core i3

After having tested our method on an ARM architecture, we want to test it on an x86 architecture. Therefore, we realized an attack campaign on an Intel Core i3 CPU using the repetition of the `mov rbx, rbx` instruction as a test code.

As the x86 architecture is different from the ARM architecture, the available registers for observation are not the same. Also, the tested architecture is a 64 bits architecture. It appears that these differences do not impact our methodology and we were able to determine the fault model for almost 80% of the cases. The faulted values distribution is presented in figure 4.

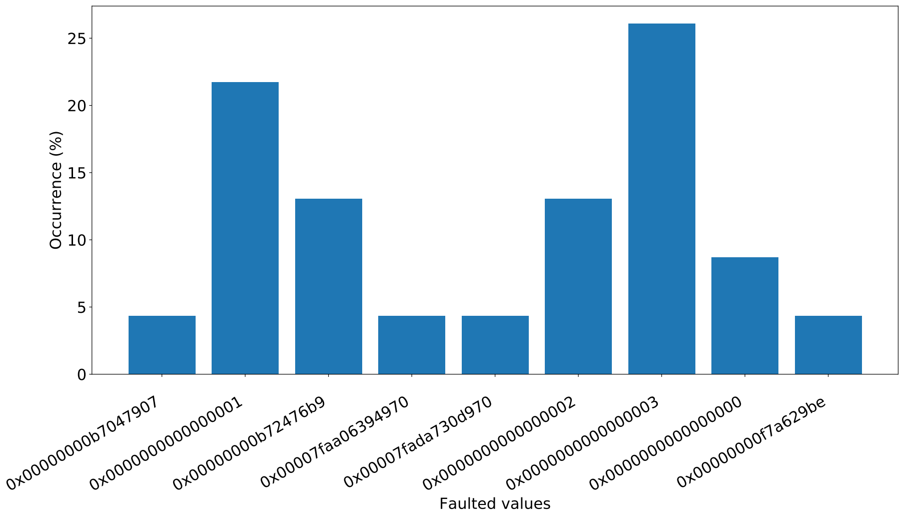

Fig. 4: Faulted values distribution with their occurrence probability obtained from an EMFI campaign on an Intel Core i3.

Using our method, we are able to determine that the faulted register is always `rbx`. There is a register corruption which sets the register to `0x0` in 8.7% of the cases corresponding to the fault model $f_{and,0x0000000000000000}$. In 56.53% of the cases, the faulted value comes from another register, these faulted values are `0x1` (register `rax`), `0x3` (register `rdi`) and `0x00007fXXXXXXXXXX` (register `rci` with a different value for every execution of the tested program). The corresponding fault model is to set the instruction second operand to either `0x0`, `0x2` or `0x5`.

The last identified fault model is for the faulted value `0x2` and corresponds to the logical `AND` between `rbx` and `r11`. This happens in 13% of the cases and corresponds to set the opcode to `0x21` and the second operand to `0xb`.

The remaining faults (21.78%) could not be determined with our method. However, the observed faulted values seems to correspond to values manipulated by the Linux OS layer. However, this investigation is out of the scope of this work and therefore not further explained.

Conclusion. With these results, we demonstrated that our method is reliable independently of the target architecture. However, on targets implementing optimizations (like the Intel Core i3), this approach is not exhaustive.

The analysis presented in section 4.1 and section 4.2 enable to model the fault at the Instruction Set Architecture (ISA) level. In other words, we can use this model to explain how the program execution is affected by our faults. With this knowledge it is possible to build some software countermeasures. But, as explained in section 1.1, software countermeasures may become irrelevant because the faulted MAB is not clearly identified.

4.3 Determining the faulted MAB

After having determined which CPU element (*cf.* table 1 and table 2) is faulted, it is interesting to check which of its MAB is affected. Usually, this determination is done using debug tools. However, as we mentioned, these tools are, most of the time, not available on our targets. In this section, we then present how we can determine which MAB is faulted using different test programs.

Pipeline characterization. As presented in figure 2 the pipeline has three main functions. The fetch function is mainly linked to the memory system. Then, for the pipeline characterization, it is more relevant to consider only the instruction decoding and its execution.

The first step consists in determining whether the fault affects the instruction before it was decoded or not. If the fault appears before the decoding, then either the instruction bus or the decoding are faulted. In the other case, the fault targets the execute stage. To determine if the instruction is faulted before its decoding, we check if the fault perturbs similarly instructions with different encoding. The proposed method can be applied on every encoded part of the instruction.

As this is instruction targeted part dependent, we present an example with the fault campaign realized on the BCM2837 where we were able to fault the instruction second operand. To determine if the fault appears before the instruction decoding, we aim at faulting an instruction which encodes a different information where the second operand is usually encoded.

The determined fault model is that the instruction second operand is set to `r2`. According to the ARM instruction encoding, this corresponds to set the eleven instruction Least Significant Bits (LSBs) to `0x002`. If the fault corrupts the instruction before its decoding, then the fault effect must be independent of the information encoded on these bits. In the tested case (`mov r3, r3`), these bits encode a register. Therefore, we can fault another instruction which uses the same bits to encode another information, an immediate value for example. If the fault corrupts the instruction and the obtained immediate value is `0x02`, then we conclude that the instruction has been perturbed before its decoding. Otherwise, we conclude that the instruction is faulted during its execution. Listing 1.3 introduces a code example for realizing this test.

This test code is a bit different from the previous ones as it uses several types of instructions. However, thanks to our own analysis reported in section 4.1 (or section 4.2), the ISA fault model is here assumed to be known and it can hence be applied to anticipate the fault effects at this level. As we intent to fault the `mov r3, #0x03` instruction, we will repeat the first three instructions to be sure to fault only one of them. These instructions are needed as the program must terminate as soon as a fault is detected. Otherwise, the next `mov r3, #0x03` will overwrite the fault.

Listing 1.3: Immediate value test code (ARM)

```
        mov r3, #0x03 // -+
        cmp r3, #0x02 //  | Several times
        be fault      // -+
        mov r9, #0x55
        b end
fault:  mov r9, 0xaa
end:    nop
```

The ISA fault model determined in section 4.1 (or section 4.2) implies that if the first instruction second operand is set to `0x02`, then a fault is obtained. If it occurs on the second instruction, then there is no error since the second operand is already set to `0x02`. Eventually, if it occurs on the third instruction, then we observe a modification of the instruction offset. Moreover, this offset can be manipulated using `nop`[6] instructions to have its eleven LSBs set to `0x002`. In this case, the fault has no impact.

This example was done with a fault model that modifies the second operand. However, some other instructions parts may be affected, such as the opcode. In this case, the demonstrated test can still be done but with different instructions. This requires the Instruction Set Format (ISF) knowledge implemented by the target. For instance, the ARM ISF is available on their website[7].

Memory characterization. The memory relies on three main elements: the buses, the MMU and the cache. If an instruction corruption is detected and has occurred before the decoding stage, two MABs can have been faulted: the buses or the cache. Distinguishing between these two cases is difficult but it is possible to determine if it happens on the mixed cache and buses or on the dedicated one. Indeed, as the highest level of cache memory is both dedicated to the data and the instructions, a fault targeting this part of the memory subsystem corrupts the instruction and the data similarly. If not, then the dedicated part of the

[6] Using `mov r2, r2` to be fault resistant for instance.

[7] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACCCHGF.html`

memory subsystem is corrupted. For this characterization step, we propose a method we applied on the BCM2837.

The method consists in (1) initializing a page of memory (4 kB), then (2) setting the observed registers values to addresses in this page and fault a code such as introduced in listing 1.4.

This code realizes memory loads and stores at/from the address stored in `r9` to/from the register `r8`. As the memory page is initialized with known values, the expected value in `r8` are known. The ISA fault model built in section 4.1 (or section 4.2) implies that the perturbation should set the second operand to `0x2` and the faulted instructions should become `str r8, [r2]` or `ldr r8, [r2]`.

Listing 1.4: Memory test code (ARM)

```
str r8, [r9] // Several
ldr r8, [r9] //     times
```

An unanticipated fault is that with probability 25%, the faulted value is the `ldr r8, [r9]` instruction encoded value. In this case, the faulted instruction is `ldr r8, [PC]` which corresponds to set the first operand to `0xff`.

For the other faults (74.4% exactly), the observed faulted value is always $b_{ad} + 50$ where b_{ad} the page memory base address. This is the value stored in `r2`. In this case, the faulted instruction is `mov r8, r2`. This is consistent with the previously determined fault model (see section 4.1 and section 4.2). Moreover, the fault does not only modify the second operand but also the opcode, forcing the instruction into a data processing instruction instead of memory loading instruction. As we already tested data processing instructions, we did not see this fault effect. This shows the importance of testing different types of instructions for determining the complete fault effect.

During this experiment, we did not observe faults on the fetched data. Therefore, we conclude that the fault targets the dedicated to the instruction part of the memory subsystems. In figure 2, it corresponds to the instruction cache, its connected buses and the fetch MAB.

Regarding the tested codes presented in listing 1.1, in the case the fault does not fit neither a fault model on the registers nor a fault model on the instructions we conclude that the fault provokes an instruction *bad fetch*. Usually, the corresponding fault models are either instruction skipping [9,21], instruction replay [18,19] or instruction replacement [3,14]. The literature proposes a large panel of fault models on the cache with different characterization methods but only on MCUs (except [18]). As MCUs do not embed MMU, the failed translation of the address fault does not appears simplifying the analysis.

We could characterize the fault on the MMU but only using debug tools. In order to remain in the scope of this work, we consider that if no cache fault model is consistent, the fault affects the MMU.

5 Conclusion and future works

In this paper, we introduced for the first time a general method for characterizing the fault model of perturbations on a CPU and demonstrated its applicability on modern CPUs embedded in a BCM2837 and an Intel Core i3 SoC. This method focuses on determining the faults effect at the ISA level and then at the micro-architectural level using only simple tests programs. As the method works for modern CPUs with many features, we strongly believe that it can been applied on MCUs CPU as well by not considering unimplemented MABs.

This approach gives us a better understanding of faults effect and therefore exploit them or mitigate them. This is a useful tool especially for evaluations where we need to determine the fault model, use it to find attack paths and build efficient countermeasures. As both a knowledge at the ISA level and micro-architectural level are determined using our method, it is possible to build both software and hardware countermeasures.

Based on this result, the future works consist in applying this method to characterize the faults effects on popular systems. Such as mobile devices, determining how these systems can be faulted will help in understanding the impact of physical attacks targeting them and build efficient countermeasures.

Another future work is to improve the CPU model and adapt the method to match with some new optimization mechanisms that are implemented in modern CPUs. Indeed, even if the presented work on the BCM2837 shows that we are able, in more than 95% of the cases, to determine the fault model, on some other targets, like an Intel Core i3, this model able to recover only 80% of the cases. We think that some complex optimization mechanisms that are not considered in our model are involved in the faulty behavior and it is interesting to work on how to model them and to characterize a fault considering them.

References

1. Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016. IEEE Computer Society (2016)
2. Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017. IEEE Computer Society (2017)
3. Balasch, J., Gierlichs, B., Verbauwhede, I.: An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In: Breveglieri et al. [7], pp. 105–114
4. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In: Prouff [17], pp. 297–313
5. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff [17], pp. 283–296
6. Bozzato, C., Focardi, R., Palmarini, F.: Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(2), 199–224 (2019)
7. Breveglieri, L., Guilley, S., Koren, I., Naccache, D., Takahashi, J. (eds.): Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011. IEEE Computer Society (2011)

8. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution pp. 1–19 (2019)
9. Korak, T., Hoefler, M.: On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms. In: Tria, A., Choi, D. (eds.) Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014. pp. 8–17. IEEE Computer Society (2014)
10. Kumar, D.S.V., Beckers, A., Balasch, J., Gierlichs, B., Verbauwhede, I.: An In-Depth and Black-Box Characterization of the Effects of Laser Pulses on ATmega328P. In: Bilgin, B., Fischer, J. (eds.) Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11389, pp. 156–170. Springer (2018)
11. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 973–990. USENIX Association (2018)
12. Majéric, F., Bourbao, E., Bossuet, L.: Electromagnetic security tests for SoC. In: 2016 IEEE International Conference on Electronics, Circuits and Systems, ICECS 2016, Monte Carlo, Monaco, December 11-14, 2016. pp. 265–268. IEEE (2016)
13. Menu, A., Bhasin, S., Dutertre, J., Rigaud, J., Danger, J.: Precise Spatio-Temporal Electromagnetic Fault Injections on Data Transfers. In: Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2019, Atlanta, GA, USA, August 24, 2019. pp. 1–8. IEEE (2019)
14. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In: Fischer, W., Schmidt, J. (eds.) Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013. pp. 77–88. IEEE Computer Society (2013)
15. Obermaier, J., Tatschner, S.: Shedding too much Light on a Microcontroller's Firmware Protection. In: Enck, W., Mulliner, C. (eds.) 11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017. USENIX Association (2017)
16. Patranabis, S., Chakraborty, A., Nguyen, P.H., Mukhopadhyay, D.: A Biased Fault Attack on the Time Redundancy Countermeasure for AES. In: Mangard, S., Poschmann, A.Y. (eds.) Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers. Lecture Notes in Computer Science, vol. 9064, pp. 189–203. Springer (2015)
17. Prouff, E. (ed.): Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7079. Springer (2011)
18. Proy, J., Heydemann, K., Berzati, A., Majéric, F., Cohen, A.: 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, Canterbury, UK, August 26-29, 2019. pp. 7:1–7:10. ACM (2019)

19. Rivière, L., Najm, Z., Rauzy, P., Danger, J., Bringer, J., Sauvage, L.: High precision fault injections on the instruction cache of ARMv7-M architectures. In: IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015. pp. 62–67. IEEE Computer Society (2015)
20. Schmidt, J.M., Hutter, M.: Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results (2007)
21. Schmidt, J., Herbst, C.: A Practical Fault Attack on Square and Multiply. In: Breveglieri, L., Gueron, S., Koren, I., Naccache, D., Seifert, J. (eds.) Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, DC, USA, 10 August 2008. pp. 53–58. IEEE Computer Society (2008)
22. Schmidt, J., Hutter, M., Plos, T.: Optical Fault Attacks on AES: A Threat in Violet. In: Breveglieri, L., Koren, I., Naccache, D., Oswald, E., Seifert, J. (eds.) Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009. pp. 13–22. IEEE Computer Society (2009)
23. Skorobogatov, S.P., Anderson, R.J.: Optical Fault Induction Attacks. In: Jr., B.S.K., Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2523, pp. 2–12. Springer (2002)
24. Tang, A., Sethumadhavan, S., Stolfo, S.J.: CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 1057–1074. USENIX Association (2017)
25. Timmers, N., Mune, C.: Escalating Privileges in Linux Using Voltage Fault Injection. In: Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017 [2], pp. 1–8
26. Timmers, N., Spruyt, A., Witteman, M.: Controlling PC on ARM Using Fault Injection. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016 [1], pp. 25–35
27. Vasselle, A., Thiebeauld, H., Maouhoub, Q., Morisset, A., Ermeneux, S.: Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot. In: Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017 [2], pp. 41–48
28. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., Giuffrida, C.: Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1675–1689. ACM (2016)
29. van Woudenberg, J.G.J., Witteman, M.F., Menarini, F.: Practical Optical Fault Injection on Secure Microcontrollers. In: Breveglieri et al. [7], pp. 91–99
30. Yuce, B., Ghalaty, N.F., Santapuri, H., Deshpande, C., Patrick, C., Schaumont, P.: Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In: Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016 [1], pp. 47–58
31. Yuce, B., Ghalaty, N.F., Schaumont, P.: Improving Fault Attacks on Embedded Software Using RISC Pipeline Characterization. In: Homma, N., Lomné, V. (eds.) Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2015, Saint Malo, France, September 13, 2015. pp. 97–108. IEEE Computer Society (2015)