20th International Symposium on ElectroMagnetic Compatibility (CEM2020)

EM Injection Vs. Modern CPU Fault Characterization And AES Differential Fault Analysis

T. Trouchkine[1], G. Bouffard[1,2], J. Clédière[3]

[1]Agence Nationale de la Sécurité des Systémes d'Information, Paris, France, thomas.trouchkine@ssi.gouv.fr
[2]DIENS, École Normale Supérieure, CNRS, PSL University, Paris, France, guillaume.bouffard@ens.fr
[3]CEA, LETI, Grenoble, France, jessy.clediere@cea.fr

Abstract. Recently, several Fault Attacks (FAs) targeting 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 a modern CPU. For that, we explain an approach to characterize the fault model on modern CPU from the assembly instruction level to the micro-architectural level. This fault model helps at determining which micro-architecture elements are disrupted and how. The fault model determination aims at finding original attack paths and design efficient countermeasures. To confront our approach to real modern CPUs, we apply our approach on a Raspberry Pi 3 CPU on which the determined fault model is reused to corrupt an AES implementation.

I. 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 [3].
Since 2015, several Fault Attacks (FAs) on modern CPUs [10], [11], have been presented, some are new and some others already applied on Micro-Controller Units (MCUs) CPUs [5]. 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 [1], [6] but only few on modern CPUs [8]. For determining the fault model on such targets and for making it reproducible, we introduce in [12] a characterization method. In this article, we describe how our method can be used to break an AES implementation with an ElectroMagnetic Fault Injection (EMFI).

This article is organized as follows. Section II introduces how a CPU works and the section III describes our approach to characterize fault from instruction assembly to Micro-Architectural Blocks (MABs). In the section IV, we applied the introduced approach on the Raspberry Pi 3 CPU and exploit our results on an AES implementation. Finally, section V concludes and opens on future works.

II. A quick overview of modern CPU modeling

Any CPU can be modeled with three functional elements:

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

On modern devices, the memory is usually external to the CPU. However, there is always an internal one, called *cache*, where a part of the external memory is copied. The three functional elements are based on MABs as introduced in Figure 1.

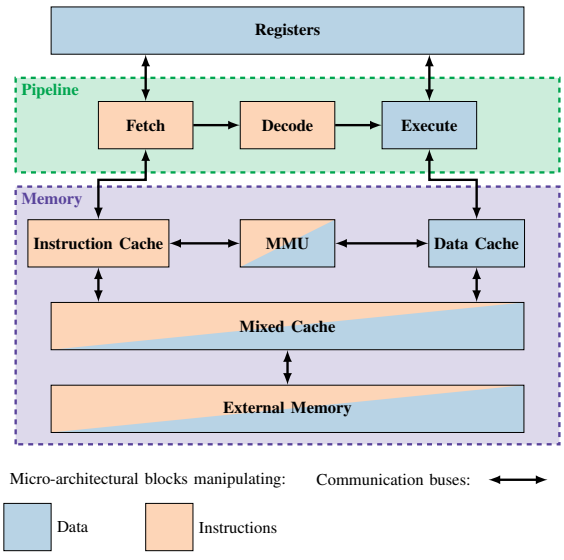

Fig. 1: CPU model

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 levels and a Memory Management Unit (MMU). Usually, CPUs have a mixed architecture where the data and the instruction paths are separated only at the lowest

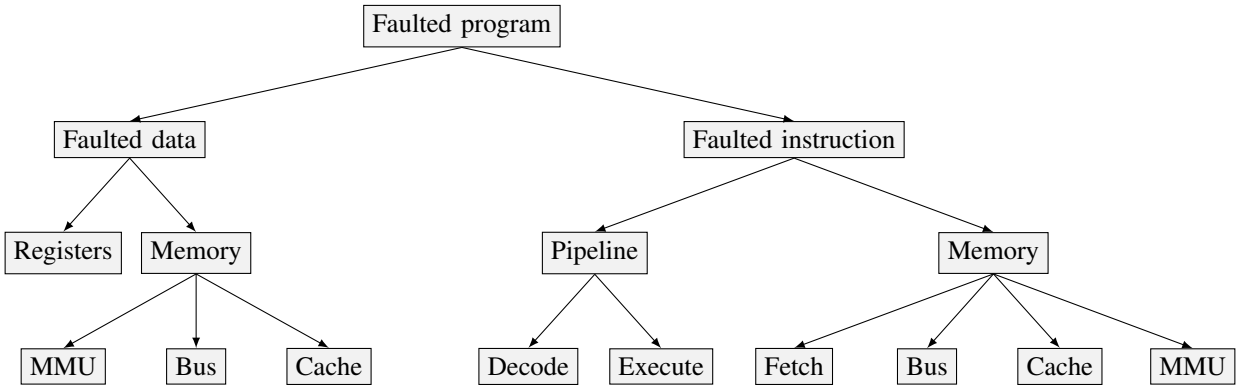

Fig. 2: Fault effect characterization overview [12].

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 can be explained by the perturbation of MABs presented in Figure 1. 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.

III. 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 [4], [9]. 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.

Introduced in [12] and summarized in figure 2, the method general idea is to apply a top-down approach. We start by determining whether the fault affects the data or the instructions. Once we know which element is affected, we determine which of its MABs is faulted. To discriminate which element is faulted, we repeatedly execute the same instruction on a known state CPU.

The executed instructions must have two properties, they should not modify the state of the processor. This is helpful as, in this case, if we observe a modification in the processor state, it comes from the fault. Also the instructions should not fetch data from memory, reducing the analysis frame.

Disturbing the program execution will give a distribution of faulted values. The next step consists in determining whether these faulted values come from a fault on the manipulated data or on the instructions.

Once we know if the data or the instructions are faulted and based on the figure 1, it is possible, from the fault model on these elements to determine which MABs have been faulted. In the case of a register corruption, it is straightforward that the registers are faulted. In the case the wrong instruction is fetched from the memory, either the cache has loaded the wrong data or the MMU has failed the address translation. If an instruction corruption is observed, the fault affects either one of the pipeline MABs or the cache or the instruction bus.

IV. Experimental analysis

IV.1. Fault model determination on BCM2837

Using the method presented in section III, we determine the fault model on a BCM2837 SoC from a Raspberry Pi 3 model B board. The chosen injection medium is ElectroMagnetic Pulse (EMP). Our bench is composed of an high voltage (800V/16A) pulse generator, a home-made probe (copper wire around a ferrite) and an Arduino based defibrillator. The target runs a Raspbian Lite distribution.

The fault model determination process is split in two steps. The first step is the hot-spots determination, *i.e.* the parameters set (position over the target chip and power voltage) for which we obtain the best faults/crashes ratio. The second step consists in determining the fault effect on the executed program for a fixed set of parameters.

During the characterization, we observed that the prob-

ability to achieve an interesting fault is around 10%. Among these faults, the most probable effect (around 80% among all interesting faults) is a corruption of the second operand of the instructions.
This fault model was determined by realizing two experiments. The first experiment aims at faulting a code repeating the `mov r3, r3` instruction. We observe that in 80% of the cases, the value of `r3` is modified and correspond to the value of another register, `r2` for instance. This corresponds to fault the second operand of the `mov r3, r3` instruction into `mov r3, r2`. During the second experiment, we fault a code repeating the `orr r3, r3` instruction. We observed that in 80% of the cases, `r3` is faulted and the faulty value is the logical `or` between `r3` and `r2`. Corresponding to the faulty instruction `orr r3, r2`.
This experiments shown that our setup is perturbing the second operand of the executed instructions. Knowing the fault model we are achieving, we propose to use it on a real attack case, the cryptanalysis of the AES algorithm.

IV.2. Exploitation: Differential Fault Analysis (DFA) on OpenSSL AES

In this section, we aim at demonstrating that the realized characterization is relevant by recovering an AES key using a DFA [2], [7].

a - Background

DFA: The DFA is a cryptanalysis method which relies on the appearance of errors during a cryptographic calculus to extract information about the manipulated secret. The method we propose to apply was introduced in [7] and extended in [2].
The principle of the attack is to realize a fault on a byte of the AES state before the last `MixColumns` operation, in the 9th round as presented in figure 3.

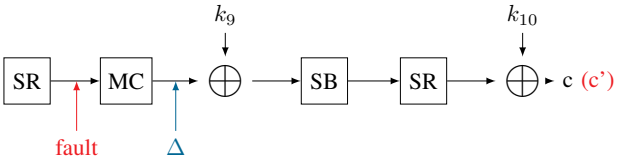

Fig. 3: DFA Principle

Once the faulty ciphertext `c'` is obtained, we compute the value Δ such as presented in equation (1).

$$\Delta = SB^{-1}(SR^{-1}(c \oplus k_{10})) \oplus SB^{-1}(SR^{-1}(c' \oplus k_{10})) \quad (1)$$

In this equation, as the possible values for Δ are known (there are only 256 possible values for Δ), the only unknown value is k_{10} and at this point Δ has only four bytes (among sixteen) that are not zeros. The next step consists in testing all the values for these bytes in k_{10} (2^{32} values in total) such as the corresponding Δ is among the possible values.
Given a faulty ciphertext c' this computation will give a set of possible values for the k_{10} bytes. The correct k_{10} value is the one that verify the equation (1) for every faulted ciphertexts. In [2], the authors demonstrate that the probability to recover the correct key with two faulty ciphertexts is around 98%, which match with our observations.
Repeating these steps 4 times able to recover the 16 bytes of the key with only 8 faulty ciphertexts.

b - The AES test program setup

For this experiment, the BCM2837 executes the AES from the OpenSSL library as a test program. The inputs of this program are the key and the message to cipher and have the following values:
`k = 0x000102030405060708090a0b0c0d0e0f`
`m = 0x00112233445566778899aabbccddeeff`
As the DFA we want to realize does not require to cipher different plaintexts, these values are hardcoded in the test program. Our goal is to recover the value of `k`.

c - Specific to complex CPUs consideration

The DFA we propose to realize requires that we fault a byte in the AES state before the last `MixColumns` operation, i.e. during the 9th round. This constraint implies the need of a precise synchronization between the execution of the AES and the injection of the fault.
To evaluate our timing spreading, we realized an experiment that gave us the information that with our injection setup we usually fault 1 or 2 instructions. On average, we fault 1.012 instructions per perturbation. With this information, we are confident that we are able to fault only one byte of the AES state. Actually, the main constraint is to not fault more than one byte per column of the AES as the DFA aim at recovering the key column by column. Fault one byte in each column is the most efficient fault we can achieve because this will give us information on the four columns of the key from only one faulty ciphertext.
Another issue with our target is its multi-core and multi-thread architecture. This implies that we do not know on which core the AES is executing and that some operations may be executed in parallel. From our point of view, the best way to measure the impact of these features is to observe the probability of obtaining the awaited fault.

d - Results of the experiment

The fault campaign consisted in 3000 injections and around an hour was needed to achieve them. Among this injections we obtained a fault percentage of 15.54% (466 faults). Among these 466 faults, only 16 have only one diagonal faulted (4.348%) and considering these faulted ciphers, only 8 (50%) correspond to a one byte fault

before the `MixColumns` operation. Also, faults appear with the same probability on every diagonals.

In the end, the probability to obtain a suitable faulted cipher for the DFA is 0.34% which corresponds to 1 cipher every 294 injections. Considering an injection needs 2 seconds, we obtain a usable cipher every 10 minutes. As 8 ciphers are needed for realizing the complete DFA and because every diagonal has the same probability to be faulted, 3 hours of injection are completely enough to obtain the needed ciphers.

e - DFA implementation

We also implemented the DFA algorithm in C-language. From 2 faulted ciphertexts with the same faulted diagonal, our program is able to recover the 4 corresponding bytes of the key in an hour on average. The computer used for this computation is powered by an Intel(R) Core(TM) i7-8550U CPU clocked at 1.80GHz with 16GiB of memory. As our implementation works per diagonals, it is possible to run four instances of the program and therefore realize the cryptanalysis on the four diagonals in parallel.

Finally, once we obtained the faulted ciphers, only 1 hour is needed to recover the key. Adding the time needed to obtained these faulted ciphers, the complete cryptanalysis can be achieved in less than 4 hours.

This timing considers that the hot spots determination and the fault characterization are already done. These steps require a week of work but the results are reusable on every target powered by the characterized device.

V. Conclusion and future works

In this paper, we present our step by step analysis about the perturbation of a modern SoC, the BCM2837, powering a Raspberry Pi 3 board. As this SoC is complex, our analysis focus on its CPU for which we propose a model. This model is intended to be general and suitable for any kind of CPU. By faulting our target with EMFI and by applying the method introduced in [12] we are able to determine the fault model we achieve on this CPU.

After having determined how we perturb the target, we decide to use our fault injection setup to realize a DFA on the OpenSSL AES running on the BCM2837. Despite the number of cores of the target CPU, the Linux Operating System (OS) and the parallel program execution, we are able to obtain suitable faulted ciphertexts for realizing the cryptanalysis within a couple of hours.

This works shows that even on considered complex CPUs, it is possible to realize both a fault characterization from assembly instructions to MAB and a cryptanalysis in a reasonable amount of time. This questions the security of more “in production” devices such as smartphones. The final aim consisting in evaluating how resistant these devices are against EMFI attacks.

REFERENCES

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

[2] Christophe Giraud and Adrian Thillard. Piret and Quisquater's DFA on AES Revisited. *IACR Cryptology ePrint Archive*, 2010:440, 2010.

[3] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. pages 1–19, 2019.

[4] Thomas Korak and Michael Hoefler. On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms. In Assia Tria and Dooho Choi, editors, *FDTC 2014, Busan, South Korea*, pages 8–17. IEEE Computer Society, 2014.

[5] Fabien Majéric, Eric Bourbao, and Lilian Bossuet. Electromagnetic security tests for SoC. In *IEEE ICECS 2016, Monte Carlo, Monaco*, pages 265–268. IEEE, 2016.

[6] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In Wieland Fischer and Jörn-Marc Schmidt, editors, *FDTC 2013, Los Alamitos, CA, USA*, pages 77–88. IEEE Computer Society, 2013.

[7] Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES 2003, Cologne, Germany*, pages 77–88. Springer, 2003.

[8] Julien Proy, Karine Heydemann, Alexandre Berzati, Fabien Majéric, and Albert Cohen. A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures: A Secure Software perspective. In *ARES 2019, Canterbury, UK*, pages 7:1–7:10. ACM, 2019.

[9] Lionel Rivière, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of ARMv7-M architectures. In *HOST 2015, Washington, DC, USA*, pages 62–67. IEEE Computer Society, 2015.

[10] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 1057–1074. USENIX Association, 2017.

[11] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM Using Fault Injection. In *FDTC 2016, Santa Barbara, CA, USA*, pages 25–35. IEEE Computer Society, 2016.

[12] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clediere. Fault Injection Characterization on modern CPUs – From the ISA to the Micro-Architecture. In *WISTP 2019, Paris, France*, 2019.