EM Fault Model Characterization on SoCs: From Different Architectures to the Same Fault Model

Thomas Trouchkine*, Guillaume Bouffard*† and Jessy Clédière‡
*National Cybersecurity Agency of France (ANSSI), Paris, France
Email: thomas.trouchkine@ssi.gouv.fr – guillaume.bouffard@ssi.gouv.fr
†DIENS, École Normale Supérieure, CNRS, PSL University, Paris, France
‡ CEA, LETI, MINATEC Campus, Grenoble, France
Email: 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. Only few works try to characterize them at the Instruction Set Architecture (ISA) level.**

In this article, we apply a state-of-the-art faults model characterization approach on modern CPU to evaluate the fault model on two different CPUs from different architectures with the same injection mediums. We target the CPU of the Raspberry Pi 3 (ARM) and an Intel Core i3 (x86) and perturbing them with ElectroMagnetic Fault Injection (EMFI). From the ISA point of view, we disclose a similar fault model on each component. Additionally, we evaluate a widely used complex software, OpenSSL, against this fault model.

***Index Terms*—Complex CPU; Fault injection attacks; Code analysis**

I. Introduction

Nowadays, modern System on Chips (SoCs) [10] are used for sensitive operations such as payment, identification, communication, *etc* as they power mobile devices, for instance, smartphones.

This situation raises some questions about their security, in particular their physical security. Indeed, as they provide an important computational power, they usually embed strong software security mechanisms like the Trusted Execution Environments (TEEs).

The drawback of this computational power is a complex hardware layout which is, for now, lacking security analysis, in particular, regarding hardware attacks. However, some recent works have demonstrated that these attacks are efficient to lower the security of modern SoCs [5], [11], [12], [13]. Therefore, we believe it is important to understand the underlying effect induced by the proposed perturbations to be able to measure their impact and design adapted countermeasures. The used injection mediums were the laser [13], the Rowhammer attack [12] and the ElectroMagnetic Fault Injection (EMFI) [11].

As the Rowhammer fault model is well known (bit flips in the memory) and the laser induces important constraints on the chip access (involving chemistry for removing the package for instance), we decided to focus on the EMFI as it provides an interesting source of perturbation with a minimum effort on chip preparation which is interesting regarding modern SoCs.

Therefore, we realized the characterization of the effects of EMFIs on two modern processors, a BCM2837 and an Intel Core i3-6100T. These targets are representative of the chips commonly powering smartphones and computers which are the most used devices. By using the same characterization method on both targets, we are able to determine the fault model on each one and compare the results. For instance, despite the fact that the considered targets have different architectures (ARM and x86), we observe a similar fault model on both of them.

A. Related works

Fault characterization on modern SoCs is very nascent and all the works about this topic focus on the SoC Central Processing Unit (CPU) as it is the heart of the component. The first characterization work [8] proposed some test codes for determining the effect of an EMFI on a Cortex-A based CPU. The method presented in this work was formalized and extended in [10] and applied on a BCM2837. This paper introduces the characterization method we use in our work. Additionally, to these works, which mainly focus on a characterization at the Instruction Set Architecture (ISA) level, a characterization of the micro-architectural behaviour of a BCM2837 against EMFI has been done in [11] with an exploitation on the AES algorithm using a Persistent Fault Analysis (PFA). In this work, the authors target only one architecture. In this article, we target two different architectures.

B. Contribution

In this article, we use the characterization method introduced by Trouchkine *et al.*'s [10] to determine how different architecture components are perturbed with the same fault injection medium. Their methodology is based on simple and small fragments of code to characterize the fault effects at the ISA level. These codes aim at moving out the software complexity. They also aim at studying the fault model closer to the ISA.

Using their methodology, we propose a study on two modern CPUs against EMFI. Our targets are the BCM2837 (ARM) powering the Raspberry Pi 3 boards and an Intel Core i3-6100T CPU (x86) powering a modified motherboard.

Based on the same approach, we compare the fault model obtained on different component architectures through EMFI

medium. We disclose the same fault model. To confirm our hypothesis, we scale the observed fault model on some complex security software to analyse it. We focus on an application usually embedded on evaluated CPUs: the OpenSSL crypto-library.

This article is organized as follows. Section II introduces our EMFI setup and the evaluated components. In section III, we characterize the fault model on CPUs embedded in the Raspberry Pi 3's SoC and the Intel Core i3 CPU. Based on the obtained fault model, we target the OpenSSL AES implementation in section IV. Section V discusses how to secure complex CPU. Section VI concludes and opens on future works.

II. Experimental process methodology

We use EMFIs to perturb our targets. We characterize the perturbation effects at the ISA level with the characterization approach introduced in [10]. This method imposes some conditions on both the initial values and the tested code.

On the one hand, the initial values are chosen to match the following condition: if we observe a faulted value, and this value is the simple operation between two initial register values, we want to be able to determine the involved initial values and the logical or arithmetical operation without any ambiguity. On the other hand, the tested code must be a data processing instruction which does not change the register values. These conditions are met for all our test code and initial value sets.

A. Devices Under Test

We target different CPUs embedded in modern SoCs. The Devices Under Test (DUT) are:

- the ARM BCM2837 embedded in the Raspberry Pi 3 board within 64-bit 4 cores running at 1.2 GHz and 512 kB cache memory. The CPU lithography is 28 nm;
- and the x86 Intel Core i3-6100T CPU: a 64-bit 2 cores CPU running at 3.2 GHz and embedding 3 MB cache memory. Instead of the previous ARM processor, its lithography is 14 nm. This CPU is put on a modified motherboard to have a trigger and a path to the CPU.

On each target, a Linux Debian 10 distribution runs test codes. Those codes are faulted during their execution using an EMFI.

B. BCM2837 setup

The BCM2837 was characterized by the following parameters:

a) Test codes: Two codes are used for the characterization. One composed of the repetition of the `and r8,r8` instruction and the other one composed of `orr r5,r5` instructions.

b) Registers initial values: The initial values were initialized differently regarding the tested program. For the `and r8, r8` test program, we generated initial register values to match the constraints defined by the fault model characterization approach [10]. For the `orr r5,r5` test program, we randomly generated initial register values as this is a suitable method to fulfil the constraints on the initial values. The drawback is that randomly generated values are less identifiable for a human. The values we used are presented in table I.

Register	`and r8,r8`	`orr r5,r5`
r0	0xfffe0001	0xc3d0c220
r1	0xfffd0002	0x72b8ccd6
r2	0xfffb0004	0xf25f29b9
r3	0xfff70008	0x22c7271d
r4	0xffef0010	0xd3f8f3b1
r5	0xffdf0020	0x3ba81d04
r6	0xffbf0040	0x7c22b133
r7	0xff7f0080	0xcc302f01
r8	0xfeff0100	0xafa42878
r9	0xfdff0200	0xdd4c70ca

Table I: BCM2837 registers initial values.

C. Intel Core i3-6100T setup

The Intel Core i3-6100T was characterized by the following parameters.

a) Test codes: The two used test codes are the repetition of the `mov rbx,rbx` instruction and the repetition of the `or rbx,rbx` instruction.

b) Registers initial values: In both experiments, we used initial register values presented in table II.

Register	Initial value
rax	0x8000000000000001
rbx	0x4000000000000002
rcx	0x2000000000000004
rdx	0x1000000000000008
rsi	0x0800000000000010
rdi	0x0400000000000020
r8	0x0200000000000040
r9	0x0100000000000080
r10	0x0080000000000100
r11	0x0040000000000200
r12	0x0020000000000400
r13	0x0010000000000800
r14	0x0008000000001000
r15	0x0004000000002000

Table II: Intel Core i3-6100T registers initial values.

D. EMFI setup

EMFI is a very common approach where the target preparation is not required. Our bench is composed of a high-voltage pulse generator (800 V/16 A) with a rising time of 6 ns and 20 ns pulse width, a homemade probe which is an 8-round copper wire around a 2 mm diameter ferrite, an Arduino based reset system (also called defibrillator) and a motorized two-axis table.

The injection parameters determination is a mandatory step which consists in finding the spatial location and the input voltage which maximizes the probability of obtaining a fault.

Determining the best spatial location for obtaining faults, *i.e.* the probe position over the chip, we divided the chip in a 40×40 grid and tested every position. In the end, we have six tests per position which leads to 9600 tests; corresponding to almost three days of experiments. We consider a fault when the executed program is terminating without errors; the board is still running but one or more of the observed registers have an unexpected value.

Another parameter we can tune is the input voltage in the probe, *i.e.* the amplitude of the pulse. We tested every voltage from $400\,\mathrm{V}$ to $800\,\mathrm{V}$ with a step of $10\,\mathrm{V}$ but no specific impact of the input voltage on the fault probability or the fault/crash ratio was observed.

On the one hand, for ARM BCM2837 experiments, we decide to keep sweeping the voltage input between $400\,\mathrm{V}$ and $600\,\mathrm{V}$.

On the other hand, as the x86 Intel Core i3-6100T is less sensitive than the BCM2837, we had to power up the input voltage. By sweeping between $600\,\mathrm{V}$ to $800\,\mathrm{V}$, the best fault/reboot ratios are obtained around $600\,\mathrm{V}$. Therefore, we kept that input voltage value during the experiments.

E. Fault model characterization

To characterize the fault model, we perform a complete analysis of the observed faulted values. Based on the values stored in registers before and after the fault, we are able to determine a fault model describing the fault. We have the following determined fault models.

- *Bit reset* corresponds to a faulted value set to 0;
- *Other observed registers value* means the faulted value is a copy of another register value;
- *And with other observed registers* means the faulted value comes from the logical `and` between the initial values of the faulted register and another one;
- *Or with other observed registers* means the faulted value comes from the logical `or` between the initial values the faulted register and another one;
- *Other observed registers value after execution* is the same as in *Other observed registers value* but where a register has been overwritten[1] during the execution of the program then copied in the faulted register
- *Or with two other observed registers* means the faulted value comes from the logical `or` between the initial values of two registers that are not the faulted register

III. Fault model characterization applied on DUT

In this section, we apply the fault model characterization approach on ARM BCM2837 and x86 Intel i3-6100T. The experiment results[2] and the tool[3] used to analyse them are available on GitHub.

[1]Overwritten means that the program, or the kernel, writes a value into that register independently of the presence of a perturbation. This is observed on the Intel Core i3. The consequence is that some registers initial values are never kept unchanged during the program execution for unknown reason.

[2]https://github.com/ANSSI-FR/Faults_experiments

[3]https://github.com/ANSSI-FR/Faults_analyzer

A. ARM BCM2837

1) Faults localization: Figure 1 shows the different locations where a fault was obtained. The X-axis and Y-axis represent the position of the probe in mm. Each dot represents a probe position where at least one fault was obtained.

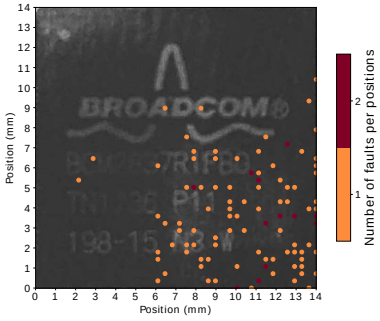

Figure 1: Probe positions over the BCM2837 leading to faults.

From this map, we have to choose a position we consider interesting regarding the obtained faults. We decide to focus at a position where the number of faults is significant. Therefore, to characterize the fault model, we put the probe at $X = 12.5\,\mathrm{mm}$ and $Y = 7.1\,\mathrm{mm}$ which is a position which lot of faults.

2) Fault characterization: One observation from these experiments is that the fault is dependent on the executed instructions. The experiments with an `orr` are faulted with a probability around $3\,\%$ while the experiments with an `and` are faulted with a probability around $1\,\%$.

Another observation is about the register manipulated by the instruction: it is the most faulted one. In the experiments with the `orr r5,r5` instruction, the `r5` is faulted in around $87.5\,\%$ of the cases. In the case of the `and r8,r8` instruction, the `r8` register is faulted in around $65.5\,\%$ of the cases.

Moreover, in any experiments, the `r0` register is always significantly faulted with a probability between $10\,\%$ and $25\,\%$ of the cases.

Other registers might be faulted but with a probability always lower than $2\,\%$.

3) Fault model: Figure 2 describes the identified fault model.

Some faulted values correspond to a undetermined fault model. This happens for around $20\,\%$ of the observed faulted values. They generally correspond to system values, data-dependent values or come from complicated fault models.

By analysing the fault effects, two kinds of corruption are disclosed, either on the registers or on the instructions.

a) Register corruption: One observed fault is a register corruption. This corruption is a complete reset of the faulted register. This happens only during the `and r8,r8` experiment with a relatively low ($3.27\,\%$) probability.

b) Instruction corruption: The other observed fault is an instruction corruption. Regarding figure 2, the main fault model for the `orr r5,r5` instruction is the *Or with other observed registers*. This fault model corresponds to modifying

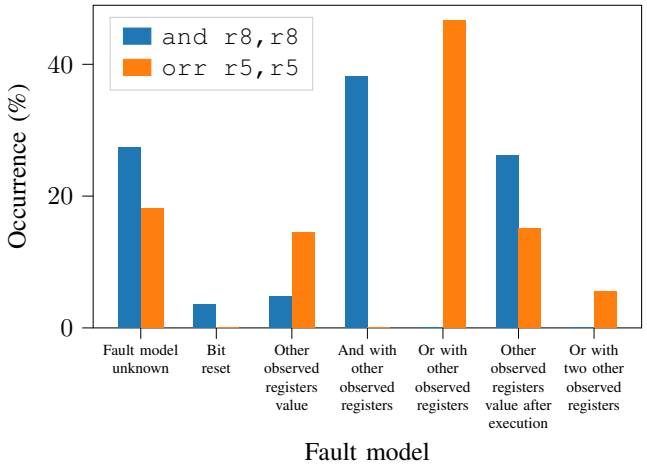

Figure 2: Fault models distribution and their probability of occurrence on the BCM2837 perturbed with EMFI.

the operands of the instruction. The faulted instruction is `orr r5,r1` in 92.54 % of the cases, `orr r5,r0` in 6.14 % of the cases and `orr r5,r7` in 1.32 % of the cases. The *Or with two other observed registers* fault model is similar to this one as, in this case, the two operands are modified. The observed faulted instructions are `orr r5, r0, r1` and `orr r5, r4, r1`.

Considering the `and r8, r8` instruction, the most probable fault model is the *And with other observed registers* which also corresponds to modify the operands of the instruction. In this case, the faulted instruction always moves to `and r8,r0`.

For both experiments, the *Other observed registers value* corresponds to a modification of both the operands and the instruction opcode, mutating it into a `mov` instruction.

Another corruption was observed and is more complicated. In some cases, the faulted value is computed from the executed instruction such as in equation 1.

$$v_f = (i \; rot_l \; 8) \wedge \texttt{0xfff} \tag{1}$$

where v_f is the faulted value, i the executed instruction machine code, and rot_l is a bit rotation to the left operation.

This fault model is complicated and may be explained by the "rolling" feature of the ARM instruction set architecture. Indeed, some bits of the instruction can be used to specify a shift to apply to the manipulated value. However, as this is a complicated fault model, we did not investigate further. It appears in between 20 % and 35 % of the cases for each experiment. In our analysis, this fault model is classified as *Other observed registers value after execution*.

Regarding these experiments, we conclude that our fault, often, corrupts the operands of the executed instruction.

c) Number of faulted instructions: When working with modern CPUs with high frequencies ($> 1\,\text{GHz}$), one important fault parameter is the spreading. We determined the fault affects instructions but we cannot determine how many instructions are actually faulted.

Regarding the BCM2837 CPU frequency (1.2 GHz) and the injector rising edge frequency (500 MHz), we can assume that a fault perturbs around 3 instructions. To confirm this hypothesis, we fault a test program composed of the repetition of the `mov rX,rX` with $X \in [\![0, 9]\!]$ and the first initial values in the registers. We observed that in $84.34\,\%$ of the cases, the fault corrupts the instruction which becomes `mov rX,rY` with $(X, Y) \in [\![0, 9]\!]^2$.

By faulting such a program, on average the fault affects 1.45 instructions. Since this result is different from the expected one, one explanation is that the CPU does not always run at maximum speed. Regarding this result, can focus our software analysis on the corruption of one or two instructions.

In conclusion, regarding the characterization of the BCM2837 CPU, one may consider different fault models for software analysis. The register corruption, which depends on the executed instructions but can be considered as a random corruption and the instruction corruption affecting the operands or the opcode. The fault can also be considered as affecting only one or two instructions.

B. x86 Intel Core i3-6100T (EMFI)

This section presents the EMFI perturbation effects characterization on an x86 CPU. The target is an Intel Core i3-6100T SoC.

1) Faults localization: The same method as for the ARM CPUs is also used but with a 40×20 grid resolution.

The Intel Core i3-6100T appeared to be slightly more difficult to perturb than the BCM2837, although it packaging has been opened. The reason behind this observation remains unknown. However, we were able to identify sensitive locations on the die. They correspond to positions where the fault forced the system in a state where we had to reboot it. By focusing on these areas, we were able to obtain exploitable faults. Figure 3 presents the obtained results.

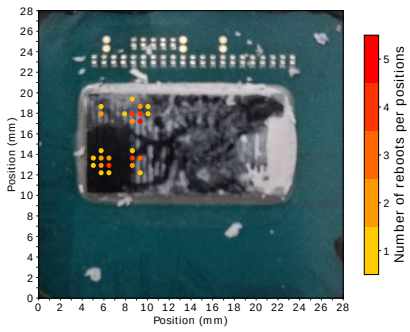

Figure 3: Probe positions over the Intel Core i3 where the system is sensitive to fault.

Figure 3 shows that two, almost symmetrically similar, areas are sensitive to faults. Our first hypothesis is that these areas

match with the cores as the Intel Core i3-6100T is a dual-core CPU device. For any other experiment on this target, we put the probe at the position $X = 8.5\,\text{mm}$ and $Y = 13.8\,\text{mm}$.

2) Fault model characterization:

a) Fault probability: On Intel Core i3, the fault probability slightly varies between $0.046\,\%$ and $1.2\,\%$. This variation is mainly due to the probe position over the die.

b) Targeted registers: An interesting result is that the faulted register is always the manipulated one. In other words, as our test program are either composed of `or rbx, rbx` or `mov rbx, rbx` instructions, the observed faulted register is always `rbx`.

This is similar to the observation made on the Raspberry Pi 3 CPUs where the most faulted register was also the one manipulated by the instruction. This suggests that the fault on each target is similar.

c) Fault models: Regarding the fault models, they are less diversified on the Intel Core i3 CPU than on the Raspberry's. Figure 4 is presenting the observed fault models on the Intel Core i3.

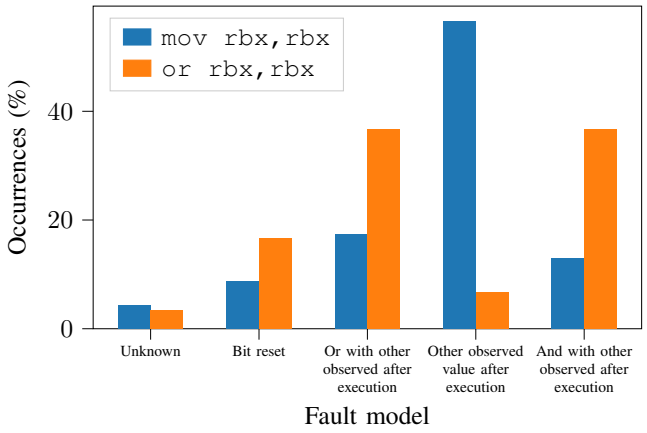

Figure 4: Fault models distribution and their probability of occurrence on the Intel Core i3-6100T perturbed with EMFI.

d) Register corruption: During this characterization, register corruption is observed. However, the corruption is really simple and consists in forcing the faulted register to specific values. Only two different values were observed. The *bit reset* value which appears in around $13\,\%$ of the cases and the register value set to 2 appears in around $9\,\%$ of the cases.

e) Instruction corruption: Regarding instruction corruption, we observe that the fault is strongly linked with the executed instruction. In some cases, with the `or rbx, rbx` test code, the opcode is corrupted and forced to `and`. However in most of the cases, only the operand is corrupted which is very similar to our observations on the Raspberry Pi 3 CPU and therefore suggests that all targets suffer similar faults.

Another interesting observation is that the faulted second operand is only among two registers. In $78\,\%$ of the second operand corruption, the instruction comes from `or rbx, rbx` to `or rbx, rax`. In $22\,\%$ of the cases, it becomes `or rbx, rcx`. Regarding the `mov rbx, rbx` test code, the corrupted instruction is always `mov rbx, rax`.

This fault model is very similar to the observed fault model on the Raspberry Pi 3 CPU. This strongly suggests that despite the fact the architectures and the technologies of these devices are different, the faults perturb them in a similar way. We can therefore extrapolate by considering that the faults perturb a component these devices have in common.

C. Conclusion on EMFI characterization

In this section, we characterized faults on commonly used modern SoCs, the BCM2837 and the Intel Core i3-6100T. We used the same injection mediums on both targets: the EMFI.

By applying the same characterization approach for each device, we observed similar perturbations on every target despite different embedded technologies and different architectures.

This conclusion is very interesting because we consider that the fault model we observed is an inter-platform fault model. Therefore, software analysis done with this fault model is relevant for different devices. Moreover, any countermeasure protecting a device against this fault model will be efficient regardless the device it is implemented on.

IV. Exploitability of the obtained fault model

Since the fault model on components is characterized, this section aims at evaluating the security of a widely used Linux program. The evaluation approach of such programs is very different from the one used on secure components software. Indeed, as they execute on a modern CPU, their execution is not sequential but parallel due to the presence of CPU optimizations; as out-of-order execution and branch prediction. These make the synchronization of the perturbation at a particular moment of the program execution tricky.

Moreover, complex programs rely on mechanisms provided by the kernel like the shared libraries. These libraries are either included in the executable (static linking), loaded at the program startup (dynamic linking) or loaded during the program execution (module loading). Therefore, as a complex program is composed of an executable and several libraries it relies on, its attack surface is important but it is also more complex to analyse. In particular, due to a lot of runtime mechanisms, a static analysis of such program may be superficial compared with what actually happens at runtime, in particular considering faults.

Because the security of such programs against fault attacks is a novel topic, there are no fault security analysis method, no fault analysis tools and only few works that target such systems.

This section presents the first steps of such security evaluation. We target OpenSSL application. Therefore, we focus on the OpenSSL library which links at the program startup the required dependencies. We use a Differential Fault Analysis (DFA) where we successfully target the ninth round of an AES-128.

A. Differential Fault Analysis on AES

This section aims at demonstrating that the performed characterizations are relevant by recovering an AES key using a DFA [6], [7]. In particular, being able to target a byte in a specific round of the AES will confirm that the spreading of our fault is reasonable. Also, this will give us information about our synchronization capacity. This experiment targets the AES implementation provided by OpenSSL library packaged in Debian 10 distribution. This experimentation was both done on the BCM2837 and Intel i3-6100T CPUs. On each of them, we obtain similar result.

1) Background: 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 proposed method was introduced in [7] and extended in [6].

The principle of the attack is to perform a fault on a byte of the AES state before the last `MixColumns` operation, in the 9$^{\text{th}}$ round as presented in figure 5.

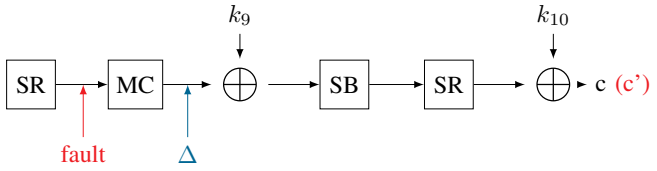

Figure 5: DFA Principle

Once the faulty ciphertext `c'` is obtained, we compute the value Δ such as presented in equation 2.

$$\Delta = SB^{-1}(SR^{-1}(c \oplus k_{10})) \oplus SB^{-1}(SR^{-1}(c' \oplus k_{10})) \quad (2)$$

Faulting one byte of the AES state before the last `MixColumns` implies that the faulted output of the `MixColumns` (Δ) has only 256 possible values. In equation 2, as the possible values for Δ are known, 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 four 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 produce a set of possible values for the k_{10} bytes. The correct k_{10} value is the one that verifies equation 2 for every faulted ciphertexts. In [6], 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 allows to recover the 16 bytes of the key with only 8 faulty ciphertexts.

2) OpenSSL AES implementation: OpenSSL is an open source general-purpose cryptography library. It is used in various programs needing cryptographic mechanisms such as web navigators, mail boxes, videoconferences, user authentication, *etc*.

For the setup, our test program calls the `AES_encrypt()` function from the OpenSSL library to encrypt data. To execute the AES encryption, our program has dependencies to the `libssl.so` and `libcrypto.so` shared libraries. They are statically loaded at the startup by the linker. There is no shared library loaded at runtime. Therefore, perturbing the program during its execution will not affect the dynamic linker.

a) Source code location: The OpenSSL source code is available on the official OpenSSL website[4]. We focus on the `AES_encrypt()` function available in the `crypto/aes/aes_core.c` file. Some implementations specific to the architecture are provided, for instance, on the Raspberry Pi 3, the executed function is the `_armv4_AES_encrypt()` function in the `crypto/aes/asm/aes-armv4.pl` file. However, our analysis on the assembly code will not come from this file but directly from the disassembly of the program. The reason is that the ARMv4 specific implementation is embedded in a Perl script which performs several optimizations. Therefore, disassembling the binary gives the closest-to-reality assembly code.

3) Static code analysis: Before evaluating the AES implementation, a static analysis of the source code and binary is performed to determine whether the determined fault model reveals exploitable vulnerabilities.

a) AES round optimization: The `AES_encrypt()` function is optimized in time. This optimization is obtained by using pre-computed tables for every `SubBytes` and `MixColumns` operation. Listing 1 presents a part of these tables.

Listing 1: OpenSSL AES pre-computed tables (partial)

```
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U,
    0xf67b7b8dU, 0xfff2f20dU, 0xd66b6bbdU,
    0xde6f6fb1U, 0x91c5c554U, 0x60303050U,
    0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    // ...
}
```

The `Te0` table corresponds to the operation presented in equation 3 with S the `SubBytes` operation.

$$Te0[x] = S(x) \cdot [2, 1, 1, 3] \quad (3)$$

This operation performs the computation of the first column of the next AES state. Corresponding tables `Te1`, `Te2`, `Te3` are also pre-computed for the other columns. In the end, the outputs of these tables are recombined to obtain the current round AES state. Listing 2 presents how the states are recombined.

Listing 2: OpenSSL AES round computation

```
t0 = Te0[(s0 >> 24)]       ^ Te1[(s1 >> 16) & 0xff]
   ^ Te2[(s2 >>  8)& 0xff] ^ Te3[s3 & 0xff] ^ rk[4];
t1 = Te0[(s1 >> 24)]       ^ Te1[(s2 >> 16) & 0xff]
   ^ Te2[(s3 >>  8)& 0xff] ^ Te3[s0 & 0xff] ^ rk[5];
t2 = Te0[(s2 >> 24)]       ^ Te1[(s3 >> 16) & 0xff]
```

[4]https://www.openssl.org/source/gitrepo.html

```
    ^ Te2[(s0 >>  8)& 0xff] ^ Te3[s1 & 0xff] ^ rk[6];
t3 = Te0[(s3 >> 24)]        ^ Te1[(s0 >> 16) & 0xff]
    ^ Te2[(s1 >>  8)& 0xff] ^ Te3[s2 & 0xff] ^ rk[7];
```

The states recombination only consists in XORing the output of the tables between them. The table access is done by switching and masking the current state (s0, s1, s2, s3). The full computation consists in a loop repeating this operation alternating between the t_X and the s_X states as input/output of the tables.

b) OpenSSL AES vulnerability analysis: Regarding the source code of an AES round presented in listing 2, one can see that the only operations involved are the logical right shift, the logical AND, the logical XOR and a memory access to the tables. As a consequence, the computed code for a round will only use ldr, and, eor and lsr instructions. This is confirmed by the disassembled code of the AES_encrypt() function.

These instructions are very similar to the instructions used for characterizing the fault model on our targets. Therefore, we have a high probability of modifying the second operand of these instructions, as this is our main fault model on targeted DUT.

Also, for the DFA, we want to fault only one byte in the AES state before the last MixColumns. As many instructions manipulate 32 bits wide register, faulting the second operand will mainly fault 4 bytes. If the 4 bytes are in different columns, the obtained cipher is still usable for a DFA and even leak information on 4 bytes of the key. However, as presented earlier, the operations are done column by column and therefore, faulting a register will mainly modify the entire column making the faulted cipher not exploitable. This excludes any fault on the ldr, eor and lsr instructions.

However, there are the and instructions that remain and apply a byte wide mask on the register. Faulting the second operand of these instructions (*i.e.* the used mask) will result in applying a byte wide fault on the result. This corresponds to the fault required for the DFA.

The and instructions represent 23 % of the instructions composing an AES round. As presented on the BCM2837 in section III, we have a 1 % fault probability on and instructions. Therefore, by extrapolating, the probability to obtain a usable ciphertext for a DFA is around 0.23 % per injection. In other words, around 400 fault injections are needed before obtaining a usable ciphertext. As 8 ciphertexts are needed for a complete DFA, around 3 200 injections are needed to obtain all the ciphertexts needed for the DFA.

In practice, the instruction does not have the same execution time, in particular the memory access instructions (ldr) are, in general, slower than the data processing instructions (and, eor and lsr). However, estimating the execution time is a tricky job as, due to CPU optimizations [3], [4] (cache, fill buffers, *etc*), this time is quite variable. However, these optimizations aim at making a memory access as fast as a data processing instruction, making our hypothesis relevant.

4) The AES test program setup: For this experiment, the DUT executes the AES from the OpenSSL library as a test program. The program inputs are the key and the message to cipher and set to the following values:

```
k = 0x000102030405060708090a0b0c0d0e0f
m = 0x00112233445566778899aabbccddeeff
```

As the DFA does not require different plaintexts, these values are hardcoded in the test program. Our goal is to recover the value of k.

5) Exploitability: The fault campaign made on BCM2837 (ARM) consisted in 3 000 injections (around 3 200 are supposed to be needed for a complete DFA) and around an hour was needed to achieve them. Among these injections we obtained a fault percentage of 15.54% (466 faults). Among these 466 faults, only 16 have one diagonal faulted (4.348%) and considering these faulted ciphertexts, only 8 (50%) correspond to a one byte fault before the MixColumns operation. Also, faults appear with the same probability on every diagonal.

In the end, the probability to obtain a suitable faulted ciphertext for the DFA is 0.34% which corresponds to 1 ciphertext every 294 injections. This is a bit better than the rate extrapolated from the analysis in Section IV-A3b (0.23 %). This difference might come from the hypothesis we made that any fault on instructions other than and instruction will not give any interesting faulted ciphertext. Indeed, it is possible, even by faulting a 4 bytes register that the fault corresponds to a 1 byte fault. The reason is that 1 byte faults are a sub-part of 4 bytes faults.

Considering the global injection sequence needs 2 seconds, we obtain a usable ciphertext every 10 minutes. As 8 ciphertexts are needed for realizing the complete DFA and because every diagonal has the same probability to be faulted, 3 hours of injection are enough to obtain the needed ciphertexts.

6) DFA implementation: We 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® Core™ i7-8550U CPU clocked at 1.80 GHz with 16 GB of memory.

As our implementation works per diagonals, it is possible to run four instances of the program and therefore perform the cryptanalysis on the four diagonals in parallel.

Finally, once the faulted ciphertexts are obtained, only 1 hour is needed to recover the key. Adding the time needed to obtain these faulted ciphertexts, the complete cryptanalysis can be achieved in less than 4 hours where the whole secret key is obtained.

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

In this section, we demonstrated that EMFI are a viable way to perturb a cryptographic algorithm executed on a modern CPU. The optimized implementation helped a lot during the analysis as only four instructions are used in the program. Also, the multi-core architecture of the CPU did not impact the

fault probability. It was even greater than the fault probability we theoretically calculated from our characterization and code analysis confirming that we are able to accurately target the 9^{th} AES round. This attack was made both on ARM BCM2837 and Intel Core i3-6100T CPUs.

V. Countermeasures

This article demonstrates that modern CPUs are very sensitive to Fault Attacks (FAs) and more especially their caches. However, cache memories are a critical element for CPUs performance. As a consequence, disabling this feature for security reasons is usually not worth the effort. Moreover, many other CPU modules may be faulted.

Also, modern SoCs surface is bigger than the Micro-Controller Units (MCUs) because more complex modules are embedded in them. Therefore, evaluating the security of complete SoC is a very time consuming process.

Regarding these constraints, the most natural countermeasure is to delegate the sensitive operations to a dedicated component, a Secure Element (SE). A SE is designed to be tamper-resistant against hardware and software attacks. This solution currently provides the best performance/security trade-off. Instead of modern SoC, SE is a small chip closer than a MCU with few hardware modules.

However, a major issue remains about the implementation and security evaluation of such component. Nowadays, two main solutions are explored by manufacturers.

A. Standalone secure modules

The first strategy consists in embedding the SE as a separated component on the device board, this solution is chosen in Google's Pixel 3 smartphones [1] for instance. This is currently the safest solution: several SEs have been evaluated under Common Criteria scheme. However, as the SE runs slower than the CPUs embedded in complex SoC. This approach slows down the system when a sensitive operation is executed.

B. Secure module integrated in a SoC

The second strategy is about embedding a SE in the complex SoC. This approach, named Smart Secure Platform (SSP), provides better speed performance than a standalone component. However, due to its youth, securing such implementation remains an open problem. Public and private certification bodies [2], helped by companies, are currently designing an evaluation scheme to check the security of a SSP. The component integration makes some invasive attacks (bus spying, component spoofing) more complicated but some warnings have to be taken to avoid a new kind of attack specific to highly integrated devices like ClkScrew [9] or integrated modules spying [14].

VI. Conclusion and future works

In this article, we compare the effects of EMFI medium on two components from different architectures (ARM and x86) and obtain the same fault model. Based on this fault model, we succeed in attacking AES implementation provided by OpenSSL library embedded in every target.

This works is a first step to compare fault medium injection on different architectures. Several next steps open to us.

On the one hand, it should be interesting to compare different medium injection on the same component. Will we get the same fault model?

On the other hand, we find the same fault model on different component architectures. Therefore, we need to investigate more deeply on which Micro-Architectural Block (MAB) is disturbed on the different targets. Working on an open component will help us to characterize the component logical behaviour and design efficient countermeasures.

References

[1] Building a titan: Better security through a tiny chip. https://security.googleblog.com/2018/10/building-titan-better-security-through.html. Accessed: 2021-04-19.

[2] Globalplatform takes first steps towards "integrated" secure element standardization. https://globalplatform.org/latest-news/globalplatform-takes-first-steps-towards-integrated-secure-element-standardization. Accessed: 2021-04-19.

[3] ARM® Cortex®-A75 Software Optimization Guide. 2018.

[4] Intel® 64 and ia-32 architecturesoptimization reference manual. 2020.

[5] Clément Gaine, Driss Aboulkassimi, Simon Pontié, Jean-Pierre Nikolovski, and Jean-Max Dutertre. Electromagnetic fault injection as a new forensic approach for socs. In *12th IEEE International Workshop on Information Forensics and Security, WIFS 2020, New York City, NY, USA, December 6-11, 2020*, pages 1–6. IEEE, 2020.

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

[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] 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.

[10] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clédière. Fault injection characterization on modern cpus. In Maryline Laurent and Thanassis Giannetsos, editors, *Information Security Theory and Practice - 13th IFIP WG 11.2 International Conference, WISTP 2019, Paris, France, December 11-12, 2019, Proceedings*, volume 12024 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2019.

[11] Thomas Trouchkine, Sébanjila Kevin Bukasa, Mathieu Escouteloup, Ronan Lashermes, and Guillaume Bouffard. Electromagnetic fault injection against a complex cpu, toward new micro-architectural fault models. *Journal of Cryptographic Engineering (JCEN)*, 3 2021.

[12] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1675–1689, 2016.

[13] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien Ermeneux. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot. In *FDTC 2017, Taipei, Taiwan*, pages 41–48. IEEE Computer Society, 2017.

[14] Mark Zhao and G. Edward Suh. Fpga-based remote power side-channel attacks. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 229–244. IEEE Computer Society, 2018.