Let's shock our IoT's heart: ARMv7-M under (fault) attacks

Sebanjila K. Bukasa
LHS-PEC INRIA
sebanjila.bukasa@inria.fr

Ronan Lashermes
LHS-PEC INRIA
ronan.lashermes@inria.fr

Jean-Louis Lanet
LHS-PEC INRIA
jean-louis.lanet@inria.fr

Axel Leqay
TAMIS INRIA
axel.legay@inria.fr

ABSTRACT

A fault attack is a well-known technique where the behaviour of a chip is voluntarily disturbed by hardware means in order to undermine the security of the information handled by the target. In this paper, we explore how Electromagnetic fault injection (EMFI) can be used to create vulnerabilities in sound software, targeting a Cortex-M3 microcontroller. Several use-cases are shown experimentally: control flow hijacking, buffer overflow (even with the presence of a canary), covert backdoor insertion and Return Oriented Programming can be achieved even if programs are not vulnerable in a software point of view. These results suggest that the protection of any software against vulnerabilities must take hardware into account as well.

KEYWORDS

Physical attacks, Fault injection, Electromagnetic Fault Injection (EMFI), Microcontroller, Backdoor, Buffer Overflow, Vulnerability Insertion, Return Oriented Programming

ACM Reference Format:
Sebanjila K. Bukasa, Ronan Lashermes, Jean-Louis Lanet, and Axel Leqay. 2018. Let's shock our IoT's heart: ARMv7-M under (fault) attacks. In *ARES 2018: International Conference on Availability, Reliability and Security, August 27–30, 2018, Hamburg, Germany.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3230833.3230842

1 INTRODUCTION

Internet-of-Things (IoT) devices are embedded devices with network connectivity. This definition embrace a wide range of devices, from a scale uploading daily your weight to a server, to an automated vacuum cleaner controlled by a smartphone, passing by a drone able to follow you when you are running and taking pictures or a smart watch containing your credit card informations. IoT devices can be wearable or transportable, leading them to interact with an unfriendly environment. The attackers have physical access to them.

In this paper, we evaluate if such devices can withstand physical attacks. Physical attacks are mainly divided between two categories: Observation Attacks (OA) where the attacker only observes (measures) the environment of the target, and Fault Attacks (FA).

FA are now a well-known class of physical attacks where a device undergoes physical parameters' modification in order to obtain an incorrect behaviour. Most classical hardware fault injection means are power glitches, clock glitches, laser pulses and electromagnetic pulses, the last technique being used in this paper. To inject a hardware fault, in most cases, it is necessary that the attacker has a physical access to the device.

Counterexamples where hardware faults are generated with pure software are presented in section 2.

FA have been shown extremely efficient against cryptography, *e.g.* the Bellcore attack [4] requires any fault, at the correct time, on an RSA-CRT signature to recover the secret. Therefore nowadays no cryptographic algorithm can be seriously published without an assessment of its sensitivity to faults attacks. But cryptography is only a small fraction of the possible applications in embedded devices. And the security of any system does not rely solely on cryptography.

The evaluator attacker model is used: in particular synchronization is achieved using a trigger signal directly issued by the targeted chip. We acknowledge that going from Proofs-of-Concept to real attacks on real devices is not straightforward (mostly because of the difficulties to achieve synchronization) but it is not a stretch to imagine that well funded attackers, or more advanced techniques would see such attacks becoming a reality.

In this paper we explore possibilities to create software vulnerabilities with hardware fault injection (with EM pulses), not on cryptography but targeting regular software running on IoT devices. These use-cases are demonstrated experimentally on an ARMv7-M (Cortex-M3) microcontroller which is present at the heart of a wide-range of embedded systems.

These examples intend to prove that a fault attack is able to create a vulnerability in a code where there is none in the usual software security meaning. Protecting against vulnerabilities must encompass protecting against both software and hardware attacks.

In order to prove our results and ease their reproducibility, we are publishing our source codes, our data etc. on a git repository available at **https://gitlab.inria.fr/rlasherm/ARMv7M-under-attacks**. Licenses are MIT for software and CC-BY-4.0 for non-software.

This paper is organized as follows. A review of other works on similar topics is presented in section 2 to show how our own work is articulated with them. Then the fault injection process is presented in section 3: the experimental setup, how data is recovered and the assumed fault model are discussed. The core of our work is presented in section 4. The results of fault attacks achieving different effects are presented and discussed. Finally the conclusion is drawn in section 5.

2 PREVIOUS WORKS

Fault injection is not a novel technique. It was first used to simulate the effect of radiation (of cosmic or nuclear origin) [10] in the 60s. In order to ensure that a chip could withstand the harsh conditions of space, they were tested under a laser to prove their resilience to radiation. In parallel, techniques were invented to have a better resilience: error detecting/correcting codes, adding redundancy...

The use of fault attacks against cryptography originated in 1997. This year, two papers [4, 5] showed how an attacker could harness a fault to recover a cryptographic key. From this date, all cryptographic algorithms must be evaluated with respect to their sensitivity to fault attacks.

Consequently, fault attacks are mostly explored along two research directions. Cryptographers are mainly interested in theoretical fault attacks: "Is an implementation secure with respect to a particular fault model?" [6]. Whereas the experimental side evaluates the fault models and validate some theoretical attacks in practice [9].

Fault attacks have been used in the past targeting chips similar to our target, namely Cortex-M3 (or closely related Cortex-M4) microcontrollers and mainly targeting cryptographic algorithms.

Underfeeding, clock glitches and Electromagnetic fault injection (EMFI) rely on the same fault mechanism, timing violation. In [3], the authors characterize this mechanism on a FPGA with underfeeding.

Barenghi *et al.* demonstrated a fault attack on an AES running on Linux with a constant underfeeding in [2]. By lowering the feeding voltage, faults started to appear. They showed that some instructions are more sensitive than others. In their case, LOAD instructions were more easily faulted.

EMFI has been demonstrated effective against cryptography in [7]. All instructions could be skipped. The setup proposed in this paper has become the de-facto method to perform EMFI, as described in section 3.

What faults can be achieved and what is the fault model is an active area of research. In [13, 14], Moro *et al.* demonstrate instruction skips (1 skip for 32-bit instructions and 2 consecutive skips for 16-bit instructions) achieved with EMFI. The fault mechanism description proposed in this paper is the most accurate to date (details in section 3.3). The authors evaluate assembly-level countermeasures and show that they are efficient to protect a specific routine but are less efficient to protect the codebase as a whole (here FreeRTOS).

In [15], the authors demonstrate another mechanism to achieve faulty behaviours with EMFI. By targeting precisely the instruction cache, they were able to obtain a high reproducibility of the faults. Since several instructions are handled by the cache at the same time, they showed that up to 4 consecutive instructions can be skipped, undermining the single instruction skip fault model.

Confirming the power of fault attacks, Kelly *et al.* in [11], evaluate the fault model on an Atmel ATtiny841 microcontroller with laser fault injection. They stress out that if some fault is achieved once, there always exists a set of parameters that is able to repeat it, even if not in 100% of the cases. Faults are not random. Consequently, double or more faults during an execution is neither hard nor more expensive.

3 FAULT INJECTION PROCESS

3.1 Experimental setup

Experiments have been performed in our laboratory. The targeted board is an STM32VLDISCOVERY board with an STM32F100RB chip, embedding an ARM Cortex-M3 core running at 24MHz (41.7ns period).

Fault injection is performed with a signal forming chain consisting in a Keysight 33509B pulse generator, a Keysight 81160A signal generator and a Milmega 80RF1000-175 power amplifier. The created powerful signal is connected to a Langer RF probe RF B 0.3-3 located on the targeted chip.

In order to launch a fault injection, a synchronization signal (a trigger) is sent by the targeted chip General-Purpose Input/Output (GPIO) (controlled from the code) directly to the 33509B pulse generator. This experimental trick, possible when the attacker has control of the code (*i.e.* never but for vulnerability assessment) is not mandatory. Other synchronization possibilities include sniffing communications with the target or measuring its EM emissions to find a relevant pattern.

The location of the probe on the chip was chosen after a scan that determined the most sensitive area on the chip. The same location was kept for all experiments.

Synchronization is the main experimental difficulty with our target. An irreducible latency of 600ns is added by our fault injection platform between the input trigger and the EM pulse.

The Sparkbench [12] software was used for the fault injection (open source, MIT license, see in reference). It is in charge of controlling the apparatus, orchestrating the commands to the target, retrieving the results and processing them. It includes a way to reset the target upon a crash and more generally to deal with errors that occur during a fault injection campaign.

3.2 Instrumentation

The targeted chip is compatible with OpenOCD [1], a tool that allows a debug access to the chip through a JTAG interface. In particular, it is possible to set breakpoints (the chip halts if a particular memory address is about to be executed), and to arbitrarily read in memory.

This tool was used to infer the behaviour of the chip upon a fault. Unfortunately, only partial information is available to us, there is no way to trace all previously executed instructions to the best of our knowledge.

When temporally scanning the chip (the injection timing sweeps over a predefined range) crashes occur frequently. OpenOCD allowed us to verify that these crashes were always explainable by software modifications e.g. reading in memory outside of the RAM boundary (invalid address) or jumping to an address not corresponding to a valid program.

Sparkbench was configured to issue a report upon a crash. This report include the timing of the crash, the Stack Pointer (SP), and the Program Counter (PC) causing the software fault (in the ARM hard fault meaning). It was used to get a feedback on the memory location of the fault, but approximately, since the software fault often occurs after the hardware one. E.g. if the stack is corrupted by a fault injection (hardware fault), the program may still run for a few instructions before reaching a point where the corrupted stack provokes an incorrect value to be loaded into the PC (software fault).

3.3 Unobservability of the fault model

The effect of a fault on the chip is of critical importance both for the attacker and for the defender, it is the so-called fault model. A detailed fault model as been proposed for our targeted chip in [13]. Moro *et al.* explain that timing based fault injection (EMFI included) is able to modify the instruction fetched. If the new instruction has no side effect, a virtual nop (No-operation, an instruction that does nothing) is obtained that cannot easily be differentiated with a true nop. If the new instruction has side effects, the result can be unpredictable.

It is important to notice that to establish this fault model, the observability difficulties are balanced by the controllability. I.e. since it is not possible to observe in details what is happening at the micro-architecture level, the authors carefully put the chip in the desired state (e.g. using nops to have an empty pipeline) to retroactively deduce the fault mechanism.

To our knowledge there is no better way to establish a detailed fault model on this chip. In particular it is not possible to know what instructions have really been executed upon a fault (execution traces are branch traces only). Yet it is also unsatisfying: what guarantee is there that another undocumented behaviour is exhibited if the chip is in another state (behaviour already proved in [8])?

The evaluator is left with a choice: either force the chip state to explain some of the fault mechanisms or do not alter the chip state, allowing all faulty behaviours, but leaving her unable to explain the fault mechanism.

If most previous works chose the first path [9, 13, 15], here the second one has been taken. As seen in section 4.2, complex behaviours did arise. Yet as unsatisfying as it is, it becomes impossible to explain precisely some fault mechanisms since we cannot trade controllability for observability.

In this paper, the details of the fault mechanism leading to the observed behaviour is not explained since it is impossible without altering the targeted software. Which, in the end, is an explication of the fault mechanism for another target program. The reader is referred to [13], for a plausible fault model for this exact same chip.

4 FAULT ATTACKS

In this section, we will demonstrate how to create vulnerabilities in a sound codebase with fault injection. Several use-cases have been experimentally implemented to illustrate the kind of vulnerabilities it is possible to achieve with fault attacks on real-world devices. Each scenario is based on well-known attacks scheme, and fault injection is a mean to activate a vulnerability.

4.1 Controlflow hijacking

4.1.1 Description. The control flow hijacking consists in forcing a program to follow a branch that it should normally not. A real world usage of this attack is a PIN code validation, if the given PIN is false, access should not be granted, ie. first condition is incorrect.

Listing 1: Targeted C code

```
if(correct == 1) {
  status = 0xFFFFFFFF; }
else {
  status = 0x55555555; }
```

Listing 2: Resulting assembly (thumb2)

```
cmp   r3, #1                  ; r3 contains *correct*
ite   eq                      ; if then else
moveq.w r4, #4294967295       ; 0xffffffff
movne.w r4, #1431655765       ; 0x55555555
```

From the C code on listing 1, the normal behaviour is for *status* to be equal to `0xFFFFFFFF` after its execution. Our goal is to follow the "else" branch instead (we detect *status* equal to `0x55555555` and *correct* still equal to 1).

Upon a correct fault, we reach an incoherent program state (that should not be reachable).

4.1.2 Attack results. To perform our attack, as in the other cases, we insert a triggering method just before our area of interest. The search for the correct parameters is done semi-automatically: first interesting parameters (that crash the target) are found for the number of pulses, the pulses period, width, leading and trailing edges duration, and the pulse amplitude. Then we scan the chip with a XY stage to detect the most sensitive location. In front of the number of parameter dimensions to explore, a part of the work depends on the experience of the operator to find the correct parameters.

In the end a set of parameters is found that trigger the expected behaviour. The signal sent to the EM probe is a train pulse with 15 pulses (period 3.1ns, width 1.6ns) with a −9dBm amplitude before the amplifier.

Faulty outputs (`status=0x55555555`) are observed for 10% of the executions during ≈ 200ns (the timing of the fault injection varies), or ≈ 5 instructions. Possible explanations are: several instructions may be faulted and giving the same faulty behaviour, a fault may be effective at different stage of the pipeline for a same instruction.

As discussed in section 3.3, it is not possible to precisely explain what is happening in the chip after a fault injection without altering the target program (and in this case we explain the fault behaviour for another program).

4.1.3 Consequences. This very simple pattern shows that with a fault attack, we may modify the control flow of our program *dynamically*, and even reach an incoherent state (that should not be reachable). Actually this pattern is already taken into account in the design of Verify PIN algorithms where fault attacks have been, rightly, considered a threat.

What this pattern reveals is that any branch in the code can be hijacked, easily leading to the compromise of the system. All branching must be protected or demonstrated innocuous.

4.2 Buffer overflow

4.2.1 Description. A buffer overflow is achieved when data is written outside the boundaries of the destination buffer during memory handling, leading to values written to adjacent memory locations. Here an attacker which can only access some variables can copy informations from internal unknown variables to readable ones.

Considerable effort have been devoted to mitigate this class of attacks in the past. In this use-case, it is demonstrated that a buffer overflow vulnerability can be created in a sound program with a fault attack.

Two different buffer overflows have been attempted. In the first attack (hereafter called *BO1*), the targeted value *key* is located close to the buffer written to (*text*). In a second attack (*BO2*), the two values are separated by a *canary*: a special buffer that issues an error if modified. The layout can be seen on listing 3.

Listing 3: Memory layout targeted by the buffer overflow

```
char text[n]                ; // n=128 for BO1, n=8 for BO2
char canary[8]              ; // BO2 only
unsigned char key[16]       ;
```

The targeted function is a *strncpy* that copy data from a big buffer called *big_text* filled with a distinctive pattern (`01 02 03 04 ...`) to *text* with the correct size parameter *n*. Our objective is to modify the *key* by disturbing the correct *strncpy* behaviour.

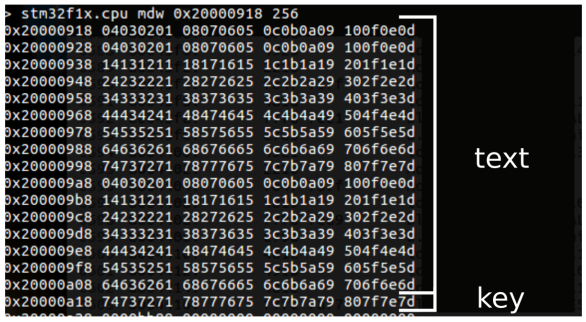
Listing 4: Assembly code for *strncpy* call

```
mov   r2, r5       ; pass n as argument (in r2)
mov   r0, r6       ; get destination (text) address (in r0)
ldr   r1, [pc, #24] ; get source (big_text) address (in r1)
bl 8004770 <strncpy>; call strncpy procedure
```

As can be seen on listing 4, if the instruction loading the n argument could be replaced by a nop, then the actual *n* value will depend on the value of the `r2` register at this call. If the new value is bigger than the previous one, a buffer overflow becomes possible. But if this new value is bigger than the source buffer, our memory will be filled with zeros according to *strncpy* documentation. But fault attacks allow us to modify any instruction, so we can also try to alter source or destination addresses.

4.2.2 Attack results. First, we confirm that preventing the proper loading of the size limiting argument (*n*) does not work. A nop was achieved, but the value stored in `r2` was an address (`0x2000xxxx`) and as specified in the documentation *n* bytes are always written to the destination buffer. First the source buffer, then a null padding. The fault, in fact wiped out our RAM, triggering a crash (probably by writing to unmapped addresses).

By methodically tuning the delay parameter in order to fault successively various instructions, a faulty ciphertext was finally found. OpenOCD was used, as explained in section 3.2, to observe the memory at the *text* and *key* locations. Figure 1 shows the result.



```
> stm32f1x.cpu mdw 0x20000918 256
0x20000918 04030201 08070605 0c0b0a09 100f0e0d
0x20000928 04030201 08070605 0c0b0a09 100f0e0d
0x20000938 14131211 18171615 1c1b1a19 201f1e1d
0x20000948 24232221 28272625 2c2b2a29 302f2e2d
0x20000958 34333231 38373635 3c3b3a39 403f3e3d
0x20000968 44434241 48474645 4c4b4a49 504f4e4d
0x20000978 54535251 58575655 5c5b5a59 605f5e5d
0x20000988 64636261 68676665 6c6b6a69 706f6e6d
0x20000998 74737271 78777675 7c7b7a79 807f7e7d
0x200009a8 04030201 08070605 0c0b0a09 100f0e0d
0x200009b8 14131211 18171615 1c1b1a19 201f1e1d
0x200009c8 24232221 28272625 2c2b2a29 302f2e2d
0x200009d8 34333231 38373635 3c3b3a39 403f3e3d
0x200009e8 44434241 48474645 4c4b4a49 504f4e4d
0x200009f8 54535251 58575655 5c5b5a59 605f5e5d
0x20000a08 64636261 68676665 6c6b6a69 706f6e6d
0x20000a18 74737271 78777675 7c7b7a79 807f7e7d
```

Figure 1: Memory observed with OpenOCD after a successful fault injection. The correct *text* address is 0x20000918, the first 4 words are correct, resulting from a previous unfaulted execution. The *key* address is 0x20000a18.

It shows that *n* was not modified. Instead, the destination address was altered and the *big_text* buffer was written at the wrong location (16 bytes shift). Finally, the *key* was overwritten at the end of the buffer. A buffer overflow was achieved.

A buffer overflow is usually achieved by modifying the number of bytes written. Here the destination address is modified. This fact motivated our second *BO2* campaign, where a canary is present in memory to detect buffer overflows. This canary is manually implemented in this case, but nowadays compilers are able to place them automatically, often at stack frame boundaries. In our application, if the canary is modified, the key is erased to 0 which can easily be detected by our platform.

The *BO2* campaign explored a big timing range to see what effect could be achieved. The most important point is that the *key* is not used in any way by the targeted function (*strncpy*), only *text* which is close in memory.

The results show that the canary has been modified in numerous cases, thus the attack failed in these cases. But faulty results were achieved nonetheless. Using OpenOCD, the *key* value was observed: the first word was replaced by either `0x08000000` or `0x00008000` depending on the fault injection timing. This is not the result of a misplaced buffer, but the exact cause of this error is still unknown. It is reminded that the targeted code does not handle memory at the *key* address, it cannot be explained by a fault at the decode stage on a load instruction.

4.2.3 Consequences. Unlike pure-software buffer overflows, it is possible with hardware fault injection to modify the size of the memory copy but also to alter the destination address. It has been demonstrated that it can jump over a canary, a classic countermeasure against buffer overflows. Yet the EMFI does not allow a good control on the fault effect.

4.3 Fault activated backdoor

4.3.1 Description. In this use-case, it is supposed that a malevolent insider (with access to source code) tries to add a backdoor into a cryptographic application. In order to not be detected by static analysis tools, the backdoor payload cannot be accessed normally by the code (no branch leads to the payload, it cannot be normally executed). As dead code, it can be detected as such but it can also be hidden as data. Yet a precisely timed fault injection is able to activate the payload.

The backdoor is split in two parts: the payload copies the key into the ciphertext buffer. The backdoor trigger (cf listing 5) is an (almost) innocent looking code that can launch the payload upon a fault injection.

Listing 5: Backdoor trigger

```c
void blink_wait()
{
  unsigned int wait_for = 3758874636;
  unsigned int counter;
  for(counter = 0; counter < wait_for; counter += 8000000);
}
```

The backdoor trigger compiles to the assembly code show on figure 6 (only the end of the function is shown).

Listing 6: Backdoor trigger (assembly)

```
   ...
 80005ca:  bd80         pop   {r7, pc}
 80005cc:  e00be00c     .word 0xe00be00c
```

This backdoor trigger lies in the fact that the ARMv7-M interleaves data and code if a data too big to be embedded in an instruction is used. Thus `3758874636` corresponds in fact, if executed, to two instructions that jump to the backdoor: `e00b e00c`. A fault instruction able to avoid the *blink_wait* return instruction at address `0x80005ca` implies that the data at `0x80005cc` is executed. A covert backdoor trigger is obtained. In our program the backdoor payload is in plain sight, but one could imagine sneakier ways to hide it.

4.3.2 Attack results. With an injection timing of $2.116\mu s$, the attack succeeded with a low success rate of 4%, in the other 96% nothing happened and the result was correct.

The value `00112233445566778899AABBCCDDEEFF` (key value) is observed as the ciphertext when the backdoor has been triggered. To verify that the backdoor trigger was responsible for the backdoor call, a hardware breakpoint was placed at address `0x80005cc` which is normally a data (and so never executed). As expected, on a successful fault injection the breakpoint did halt the execution and a "step" in debug mode did show a jump to the backdoor. The triggering path has been validated. Unfortunately, we have no way to be sure that a nop is indeed responsible for the data execution at `0x80005cc`.

But something else, strangely, did appear during the experiments. At $2.268\mu s$, a faulty value `279FB74A445566778899AABBCCDDEEFF` is observed (first word is correct ciphertext, the 3 other words correspond to the key). To investigate, a breakpoint was placed at the end of the backdoor payload. Indeed this breakpoint was reached. By reading the memory at this point, part of the program path leading to the backdoor payload was reconstructed. The backdoor trigger was not responsible but in a neighbouring part of the code, an address corresponding to the middle of the backdoor was found in the stack. A stack corruption leads to a jump to this location through a *pop {..., pc}*. How this address ended up in the stack is still a mystery (and unfortunately cannot be explained cf section 3.3).

4.3.3 Consequences. With this use-case it has been showed that a backdoor may be hidden in an inert part of the memory. As such, static analysis tools cannot detect this vulnerability with information flow analysis. This vulnerability is a reminder of the well known problem of executable data. Usual protection may be efficient in this case apart for the peculiarity of ARMv7-M to interleave some data with executable code.

4.4 Return Oriented Programming

4.4.1 Description. In this use-case, a Return Oriented Programming (ROP) exploit is demonstrated. From her knowledge of the binary, an attacker tailors her own program by imposing her control flow graph on top of existing code. It becomes possible to take full control of a targeted device, including execution of privileged instructions. If the binary is big enough, code sections with a useful behaviour will always be present. ROP allows to use these sections and to chain them in a useful manner to achieve the desired behaviour and minimize side effects. These pieces of code are called "gadgets". Their common particularity is that they include a useful behaviour for an attacker, and the sequence of instructions finishes by a return. There are two different ways to "return" in ARMv7-M, namely `bx lr` or `pop {...,pc}`, the first one takes into account the Link Register (LR) whereas the second take values from the stack. The second class of return is the most practical in our case, because with full control on the stack it is easier to pop from stack than to assign a value to LR. This attack is interesting for an attacker because it allows to use existent code to create a different behaviour, thus keeping static integrity.

Listing 7: Example of gadget (assembly)

```
800039c:bd04    pop {r2, r5}
800039e:46ae    mov lr, r5
80003a0:bd00    pop {pc}
```

In listing 7 registers r2, r5 and lr are filled with values from the stack, then the last pop is a branch by filling the PC register with the next stack value.

If the attacker is able to branch to a controlled part of the stack, she can then use values that will call gadgets and then return to the stack. The challenge here is to gain control of the stack and fill it with the right values, to activate the ROP. As the attacker only manipulate addresses there are no modifications on the source code itself. But she must know all exact addresses and necessary values to fill correctly the stack.

The ROP goal is to copy the key value into the ciphertext buffer then to hand over to the normal control flow (similar to the backdoor scenario). So when reading the ciphertext after a successful fault injection, the key is read instead.

4.4.2 Attack results. Activating the ROP has some prerequisites. First of all, the stack must be filled with the ROP data (the branches to gadgets) and the SP correctly positioned.

In our setup, the stack has been filled at the current stack pointer with the adequate values. In a real attack, this part may be tricky.

Listing 8: ROP activation (assembly)

```
 80003dc  b.n 80003ec <some_function>
 80003de  nop
        ...
080003e2 <first_gadget>:
 80003e2  ldr r4, [pc, #4]; (80003e8)
 80003e4  mov lr, r4
 80003e6  pop {r0, r1, r2, pc}
 80003e8 .word 0x0800039d
```

As our test application is not big enough, some gadgets where missing. In particular, the first one has to be manually added: it saves the LR required to return to the normal program flow after the attack. Then it sets the PC from a value in the attacker controlled stack to the next gadget.

The ROP activation happens when the program flow reach a function preceding the first gadget in memory as shown in listing 8. Instruction at `0x80003dc` is skipped.

A NOP is present in listing 8 to illustrate a difficulty with fault injection. As clearly explained in [13], the fetch stage always load 32-bits of instruction (here corresponding to two instructions). Hence a fault may alter two instructions simultaneously depending on the alignment of the instructions, increasing the probability to get a crash. Indeed in our first experiments, the alignment was not correct and caused the fault to impact two instructions: the first instruction of the first gadget was not properly executed.

The ROP attack consists in filling the proper data in registers before calling *memcpy* to copy the key into the ciphertext buffer. Since our gadgets do not corresponds exactly to the desired program, padding values are placed into the stack so that the values of interest for the attack are properly located into the stack

The last gadget is a return to normal execution. Finally, the attack succeeded as the key was read in the ciphertext buffer, with no error on other functionalities.

4.4.3 Consequences. This use-case demonstrates that the previous attack schemes may be leveraged into a fully fledge attack, namely a ROP attack to execute an arbitrary payload designed by the attacker.

ROP is a real threat for devices as it allows to take full control of a target by completely hijacking the control flow.

This technique is powerful on several account. First, the ROP keeps the privilege the program has at the faulted instruction. It is a practical opening for privilege escalation. Then the static integrity is preserved before and after the attack. A forensic analysis cannot reconstruct easily what happened (if the stack is cleared after the attack).

5 CONCLUSION

With this work, the EMFI has been shown to be an effective vulnerability injection mechanism. It asks for a re-evaluation of classic countermeasures against common software vulnerability, as our buffer overflow over a canary have shown.

The precise micro-architectural effect of the fault injection should be analysed in depth. For that, tooling must be developed to reconstruct the precise effect of the fault on real code. Indeed using specially crafted code for that purpose, even if informative, prevents from observing the whole range of effects seen on real code.

Tooling must also be designed to evaluate the fault vulnerability of a program. It can help the system designer to detect the most vulnerable parts of his system and act accordingly.

In this work, EMFI has been used on our own application. It is far simpler than a real application in a IoT device for example. The effectiveness of this technique against commercial products must be evaluated. EMFI is a threat, but is it as bad as we suspect? Not necessarily, since the complexity of the system may make the experiments more difficult. In the end, the main limitations for EMFI is of experimental nature, the attacker has a low control on the injected faults and the reproducibility may be low.

There are many possible countermeasures against EMFI (formally proven applications, secure elements, increased experimental difficulty, control flow integrity, etc).

In the future, at least one of the core would have to be hardened, probably with an efficient Control Flow Integrity (CFI) & System Integrity (SI) mechanism, to face EMFI threat.

REFERENCES

[1] 2017. OpenOCD. http://openocd.org Last accessed on July, 5th, 2017.
[2] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi. 2010. Low voltage fault attacks to AES. In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 7–12. https://doi.org/10.1109/HST.2010.5513121
[3] S. Bhasin, N. Selmane, S. Guilley, and J. L. Danger. 2009. Security evaluation of different AES implementations against practical setup time violation attacks in FPGAs. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*. 15–21. https://doi.org/10.1109/HST.2009.5225057
[4] Eli Biham and Adi Shamir. 1997. *Differential fault analysis of secret key cryptosystems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 513–525. https://doi.org/10.1007/BFb0052259
[5] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 1997. *On the Importance of Checking Cryptographic Protocols for Faults.* Springer Berlin Heidelberg, Berlin, Heidelberg, 37–51. https://doi.org/10.1007/3-540-69053-0_4
[6] H. Le Bouder, G. Thomas, Y. Linge, and A. Tria. 2014. On Fault Injections in Generalized Feistel Networks. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 83–93. https://doi.org/10.1109/FDTC.2014.18
[7] A. Dehbaoui, J. M. Dutertre, B. Robisson, and A. Tria. 2012. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 7–15. https://doi.org/10.1109/FDTC.2012.15
[8] Christopher Domas. 2017. Breaking the x86 ISA. In *BlackHat 2017*.
[9] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. 2016. *From Code Review to Fault Injection Attacks: Filling the Gap Using Fault Model Inference.* Springer International Publishing, Cham, 107–124. https://doi.org/10.1007/978-3-319-31271-2_7
[10] D. H. Habing. 1965. The Use of Lasers to Simulate Radiation-Induced Transients in Semiconductor Devices and Circuits. *IEEE Transactions on Nuclear Science* 12, 5 (Oct 1965), 91–100. https://doi.org/10.1109/TNS.1965.4323904
[11] M. S. Kelly, K. Mayes, and J. F. Walker. 2017. Characterising a CPU fault attack model via run-time data analysis. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 79–84. https://doi.org/10.1109/HST.2017.7951802
[12] Ronan Lashermes. 2017. Sparkbench. https://gitlab.com/Artefaritaj/Sparkbench
[13] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. 2013. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 77–88. https://doi.org/10.1109/FDTC.2013.9
[14] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz. 2014. Experimental evaluation of two software countermeasures against fault attacks. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 112–117. https://doi.org/10.1109/HST.2014.6855580
[15] L. Rivière, Z. Najm, P. Rauzy, J. L. Danger, J. Bringer, and L. Sauvage. 2015. High precision fault injections on the instruction cache of ARMv7-M architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 62–67. https://doi.org/10.1109/HST.2015.7140238