Do Not Trust Modern System-on-Chips

Electromagnetic fault injection against a System-on-Chip

Thomas Trouchkine [3] Sébanjila Kevin Bukasa [1] Mathieu Escouteloup [1] Ronan Lashermes [2] Guillaume Bouffard [3]

[1]INRIA/CIDRE [2]INRIA/SED&LHS [3]ANSSI

May 23th, 2019

JAIF, Grenoble

Section 1

(Really) short introduction

Introduction

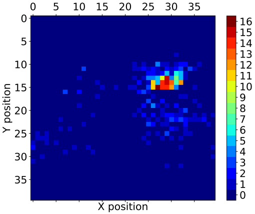
Objectives

- EM fault attack on modern ARM SoC.
- What fault models ?
- Methods for characterization
  - ISA and micro-architectural layers
  - Top-down approach

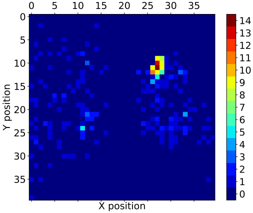
Section 2

Fault on instructions
-
Characterization methodology

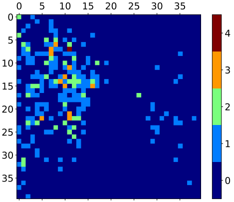
Determination of hotspots



Crashes Bare-metal



Crashes on Linux



Faults on Linux

Characterization generic methodology

1. Determination of possible error E induced by the perturbation

$$v_f = E(v)$$

2. Fault hypothesis from error E

$$v'_{fh} = E(v')$$

3. Experimental confirmation

$$v'_{fr} = E(v')$$

4. Conclusion

$$v'_{fh} = v'_{fr} \ ?$$

Code under test

Pipeline characterization

- only data processing instructions
- no instructions changing state

Code example:

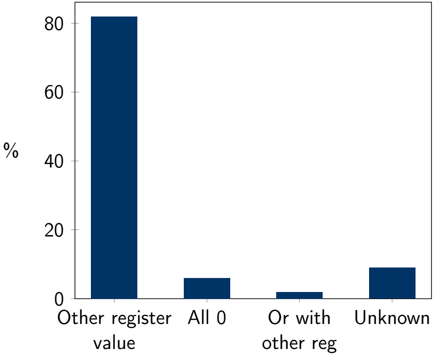
```
mov r3,r3
.
. /* 100 times */
.
mov r3,r3
```

```
nop
mov rX, rX
and rX, rX
orr rX, rX
```

Opcode analysis

```
mov r0, r0
 r0 <= r0
```

Pattern of the faulted value

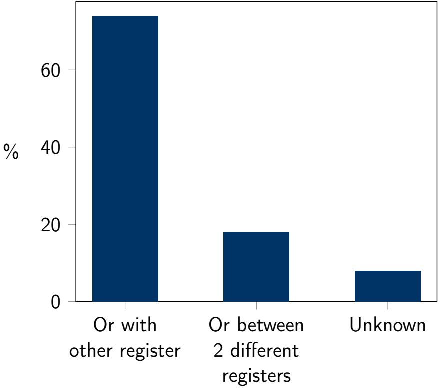


- check on `r0` to `r9`
- the operand doesn't change (80%)
- `rX <= rY`

Opcode analysis

```
or r0, r0
r0 <= r0 or r0
```

Pattern of the faulted value

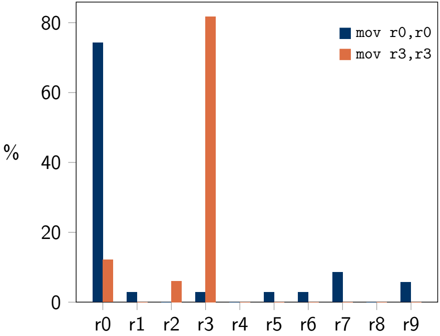


- `rX <= rY or rX` (70%)
- `rX <= rY or rZ` (20%)

Destination analysis

```
mov r0, r0
mov r3, r3
```

Number of faults per register



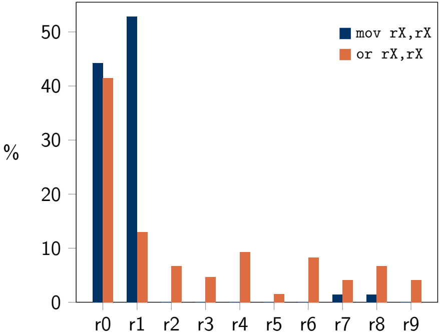
- destination register doesn't change (75%)
- `r0 <= rX`

Operands analysis

```
mov rX, rX
or rX, rX
```

$\texttt{X} \in [0, 9]$

Value in the faulted register



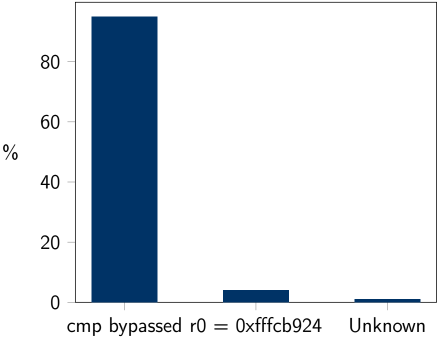
- all registers faulted with same probability
- `rX <= r{0,1}`
- second operand set to 0 or 1

Example of exploitation

Targeting cmp instruction

```
      init:  r3 <= 0xff
```

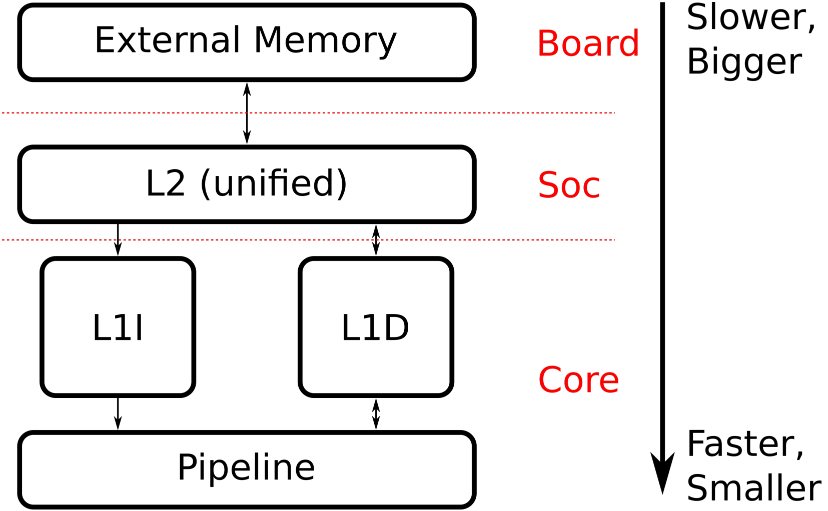
```
         cmp r3, #255
         bne fault
         b nofault
fault:   mov r9, #170
         b end
nofault: mov r9, #85
end:     nop
```



Section 3

Fault on L1I

Reminder on memory hierarchy



Targeted software (single-core)

Listing 1: Loop target application

```
trigger_up();
//wait to compensate bench latency
wait_us(2);
for(i = 0;i<50; i++) {
  for(j = 0;j<50;j++) {
    cnt++;
  }
}
trigger_down();
```

Forensic

Just after a fault, we set the Program Counter to the start of the loop. Then we execute step-by-step and check the side effects.

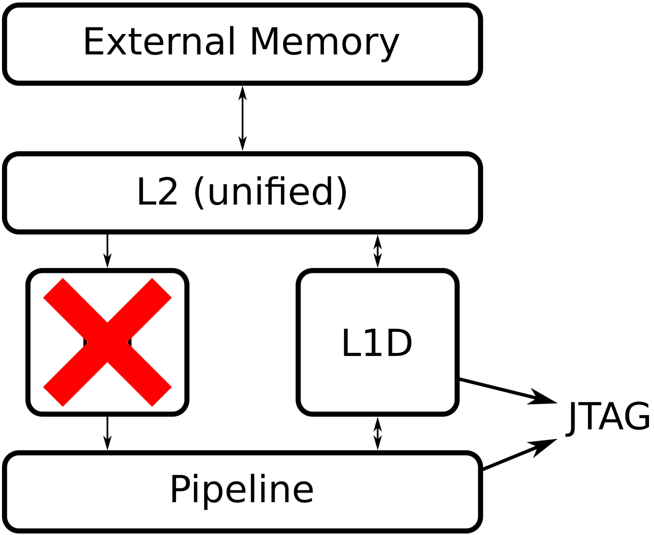
Listing 2: Loop target assembly

```
...
48a04:  b94017a0   ldr   w0, [x29,#20]
48a08:  11000400   add   w0, w0, #0x1
48a0c:  b90017a0   str   w0, [x29,#20]
48a10:  b9401ba0   ldr   w0, [x29,#24]
48a14:  11000400   add   w0, w0, #0x1
48a18:  b9001ba0   str   w0, [x29,#24]
48a1c:  b9401ba0   ldr   w0, [x29,#24]
48a20:  7100c41f   cmp   w0, #0x31
48a24:  54 ffff0d  b. le 48a04
...
```

```
pc: 0x48a04
> reg x0
x0 (/64): 0x1
> step
pc: 0x48a08
> reg x0
x0 (/64): 0x2
> step
pc: 0x48a0c
> reg x0
x0 (/64): 0x2
> mdw 0x48a08 1
0x00048a08: 11000400
```

Figure: JTAG session

Confirming micro-architectural model



Confirming micro-architectural model

How to confirm ?

Invalidate L1I cache by executing corresponding instruction.

```
> reg pc 0x6a784
pc (/64): 0x000000000006A784
> step => IC IALLU
pc: 0x6a788
> step => ISB
pc: 0x6a78c
> reg pc 0x48a08
pc (/64): 0x0000000000048A08
> reg x0
x0 (/64): 0x0000000000000002
> step
pc: 0x48a0c
> reg x0
x0 (/64): 0x0000000000000003
```

Figure: JTAG session

Failure cause

Hypothesis

- Fault present only on first execution,
- and fault has an impact on L1I.

The fault occurs on a memory transfer when writing instructions to L1I.

Failure cause

Hypothesis

- Fault present only on first execution,
- and fault has an impact on L1I.

The fault occurs on a memory transfer when writing instructions to L1I.

Listing 3: Loop target assembly

```
trigger_up();
wait_us(2);
/* + */invalidate_icache();
for(i = 0;i<50; i++) {
  for(j = 0;j<50;j++) {
    cnt++;
  }
}
trigger_down();
```

Observations

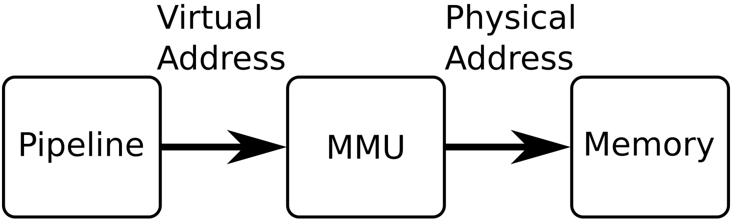
Now, we can reproduce the previous fault, if we inject during the cache reload (lasts $2\mu s$).

Section 4

Fault on the MMU

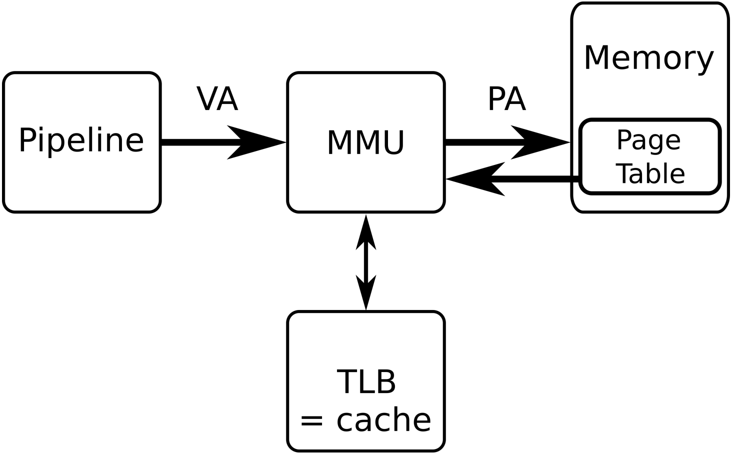
Reminder on the MMU

Principle



Reminder on the MMU

Principle



Correct memory mapping

Identity Mapping

```
VA      -> PA
0x0     -> 0x0      0x80000 -> 0x80000
0x10000 -> 0x10000  0x90000 -> 0x90000
0x20000 -> 0x20000  0xa0000 -> 0xa0000
0x30000 -> 0x30000  0xb0000 -> 0xb0000
0x40000 -> 0x40000  0xc0000 -> 0xc0000
0x50000 -> 0x50000  0xd0000 -> 0xd0000
0x60000 -> 0x60000  0xe0000 -> 0xe0000
0x70000 -> 0x70000  0xf0000 -> 0xf0000
```

Faulting the MMU

Setup

- Same code target (loop).
- Change injection timing (target the end of L1I loading).
- In this case, we investigate a crash (the application did not provide a result).

Voilà !

Faulty mapping

```
VA -> PA
0x0 -> 0x0              0x100000 -> 0x0
0x10000 -> 0x10000    0x110000 -> 0x0
0x20000 -> 0x20000    0x120000 -> 0x0
0x30000 -> 0x30000    0x130000 -> 0x0
0x40000 -> 0x40000    0x140000 -> 0x100000
0x50000 -> 0x50000    0x150000 -> 0x110000
0x60000 -> 0x60000    0x160000 -> 0x120000
0x70000 -> 0x70000    0x170000 -> 0x130000
0x80000 -> 0x0        0x180000 -> 0x0
0x90000 -> 0x0        0x190000 -> 0x0
0xa0000 -> 0x0        0x1a0000 -> 0x0
0xb0000 -> 0x0        0x1b0000 -> 0x0
0xc0000 -> 0x80000    0x1c0000 -> 0x180000
0xd0000 -> 0x90000    0x1d0000 -> 0x190000
0xe0000 -> 0xa0000    0x1e0000 -> 0x1a0000
0xf0000 -> 0xb0000    0x1f0000 -> 0x1b0000
```

This is a working mapping !

Failure cause

Mostly unknown

- Flushing TLB does not change anything.
- The page tables are modified but do not match the mapping.
- Flags have changed in the new page tables.

Failure cause

Mostly unknown

- Flushing TLB does not change anything.
- The page tables are modified but do not match the mapping.
- Flags have changed in the new page tables.

Other observations

- Mapping is still correct for the program memory size.
- Fault is reproducible,
- but we do not achieve exactly the same mapping every time.
- The new mapping is often invalid (translation error).

MMU conclusion

Pointer authentication (PA)

PA, as in ARMv8.3, does not resist this fault model. Pointer security should guarantee the translation phase too.

MMU conclusion

Pointer authentication (PA)

PA, as in ARMv8.3, does not resist this fault model. Pointer security should guarantee the translation phase too.

OS

The MMU management is done very diffirently with an (full) OS present: pages are allocated on-the-fly.

MMU conclusion

Pointer authentication (PA)

PA, as in ARMv8.3, does not resist this fault model. Pointer security should guarantee the translation phase too.

OS

The MMU management is done very diffirently with an (full) OS present: pages are allocated on-the-fly.

No attacker control

The erroneous mapping is not controlled by the attacker, the danger is therefore limited. For now ?

Section 5

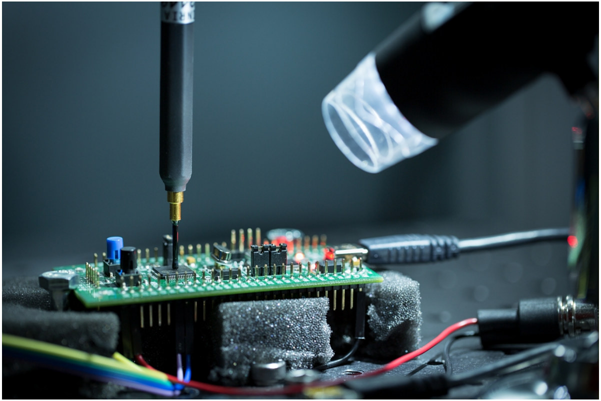
Conclusion

Conclusion / Attacks

- SoC computations can be disrupted by EMFI.
- We demonstrate faults on the pipeline, L1I, MMU and L2.
- We propose a methodology for fault model determination.

Thank you!

Any questions?



Section 6

Fault on L2

Yet another fault

Setup

- Same code target (loop).
- Change injection timing.
- We investigate a crash.

Why this fault ?

A step by step execution with JTAG rapidly shows that we are trapped into an infinite loop.

Comparing memory dumps

```
0x000489b8: d65f03c0 a9be7bfd 910003fd b9001fbf
0x000489c8: b9001bbf b90017bf 900001a0 912d2000
0x000489d8: d2802002 52800001 94000b28 97fefe67
0x000489e8: d2800040 97feffe2 94008765 940087ad
0x000489f8: b9001fbf 14000010 b9001bbf 14000008
0x00048a08: 940087c1 b94017a0 11000400 b90017a0
0x00048a18: b9401ba0 11000400 b9001ba0 b9401ba0
0x00048a28: 7100c41f 54fffeed b9401fa0 11000400
```

Figure: Correct dump.

```
0x000489d8: d2800040 97feffe2 00000002 00000008
0x000489e8: 00000002 00000008 910003fd b9001fbf
0x000489f8: b9001bbf b90017bf 11000400 b90017a0
0x00048a08: b9401ba0 11000400 b9001ba0 b9401ba0
0x00048a18: 7100c41f 54fffeed b9401fa0 11000400
0x00048a28: b9001fa0 b9401fa0 81040814 77777777
```

Figure: Faulty dump. Underlined instructions are part of the infinite loop.

Graphical summary