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 10th, 2019

INRIA, Rennes

Section 1

In the previous episodes

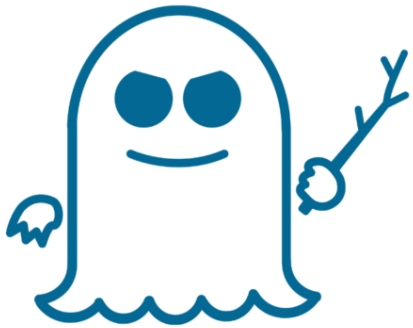
The SoC is king

Image from t3.com

The SoC is king

Image from http://www.netmarine.net

Micro-architectural attacks



```
if(condition) {
    func1();
}
else {
    func2();
}
```

The Instruction Set Architecture (ISA) abstraction

The ISA abstraction leaks.

The Instruction Set Architecture (ISA) abstraction

The ISA abstraction leaks.

You can prove your software correct,
the attacker doesn't care...

Physical attacks

Scenario: the attacker can grab the targeted device and bring it in her supervillain's lab.
She can measure the environment of the target and interact with it.

Figure: Faustine in the LHS.

Fault attacks

Modify the chip environment to induce failure. We use electromagnetic fault injection.

- Signal: Sinus 275MHz, 1 period, -14dBm (before ≈50dB amplification)
- XYZ stage, $1\mu m$ resolution.
- Langer RF U 5-2 magnetic probe.

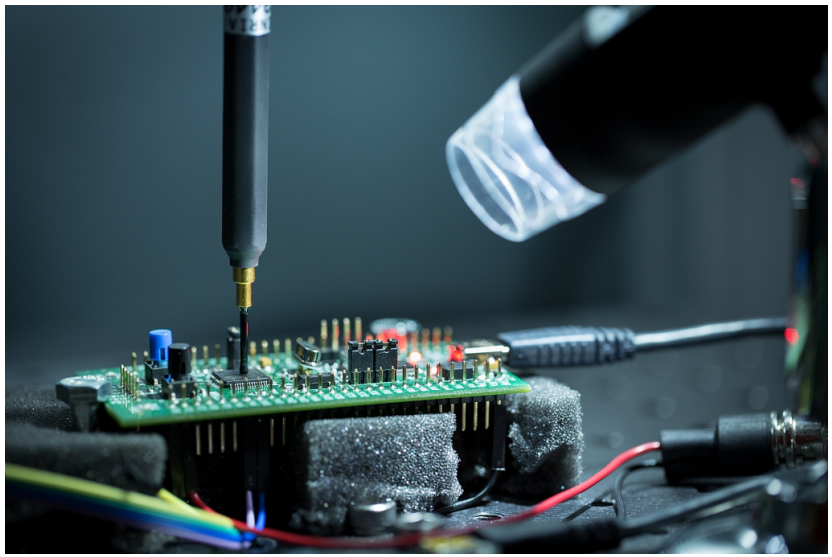
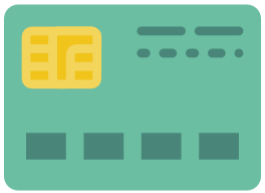


Figure: Still Faustine in the LHS.

Fault models on microcontrollers

Fault models

A formal description of the achievable faults is called a fault model. E.g. bit flip, instruction skip, ...
It is always the interpretation of a physical behavior at a specific abstraction level.



On microcontrollers

- Data faults: random, bit flip, stuck at.
- Control flow faults: (virtual) instruction skip.
- Microarchitectural faults: preventing instruction fetch, bus disturbance.

ISA fault models

Definition

A fault model that can be described by instruction modifications.
E.g.: b pin_verif_failed $\rightarrow$ nop

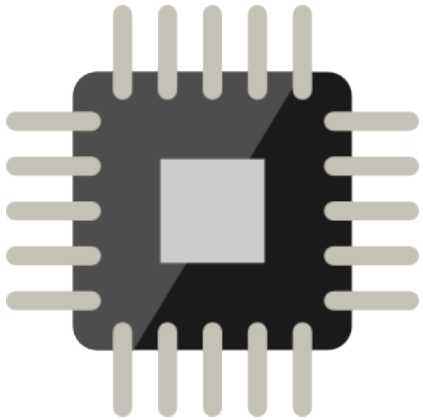
Why ?

The hardware can be modeled with software: allows software countermeasures to hardware attacks. E.g. duplicate instructions to counter single-faults in the ISA model.

Microcontrollers vs SoCs

Before

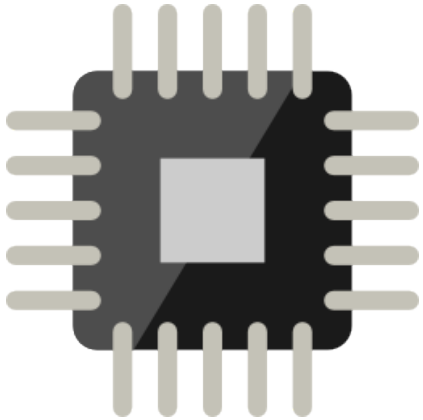
- Slow: < 50 MHz.
- Simple: in-order, single-issue, single core.
- No MMU.
- Limited cache hierarchy (L1 if any).



Microcontrollers vs SoCs

Now: Cortex-A53

- **Fast: 1.2 GHz (x24).**
- Complex: in-order, dual-issue, multiple cores.
- MMU present.
- Cache hierarchy: L1I, L1D, L2 (unified).

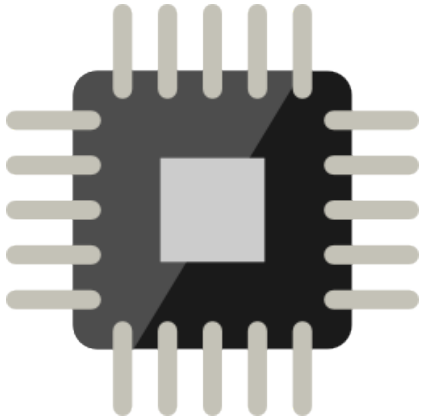


Microcontrollers vs SoCs

Now: Cortex-A53

- **Fast: 1.2 GHz (x24).**
- Complex: in-order, dual-issue, multiple cores.
- MMU present.
- Cache hierarchy: L1I, L1D, L2 (unified).

What faults can be experimentally achieved ?



Our target

Raspberry Pi 3 B

BCM2837: 4 × Cortex-A53

Content

- The impact of the operating system.
- Fault on L1I.
- Fault on MMU.
- Fault on L2 (unified).

Section 2

Related works

Teams working on fault models on SoC

- ANSSI (& co), EMFI: Thomas Throuchkine, Guillaume Bouffard, Jessy Cleydière.
- eshard, laser FI: ???
- INRIA, EMFI: Sébanjila Bukasa, Ronan Lashermes, Jean-Louis Lanet.
- INVIA/Thales (& co), EMFI: Julien Proy, Alexandre Berzati, Karine Heydemann, Albert Cohen.

All teams will publish in 2019 (expectedly).

Section 3

The impact of the operating system

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();
```

Crashes cartography

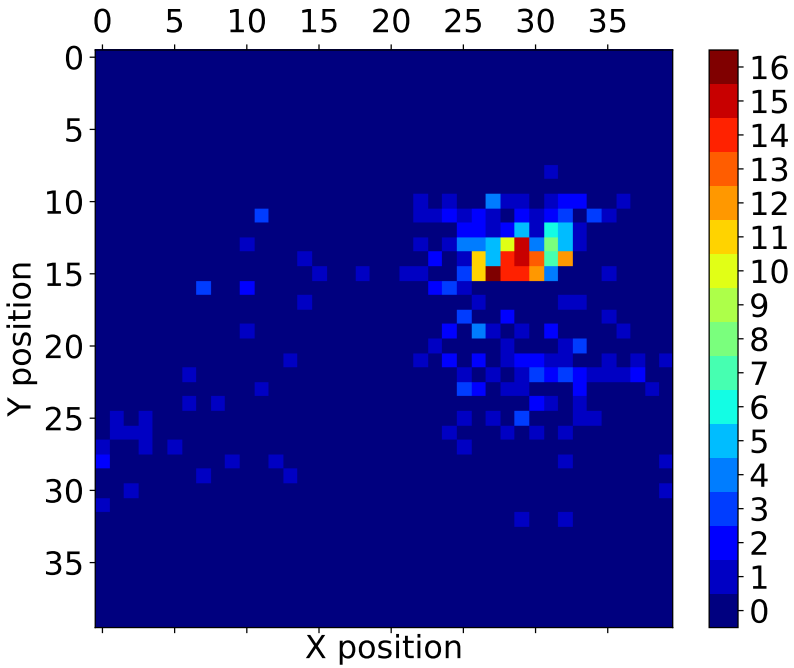


Figure: Bare-metal

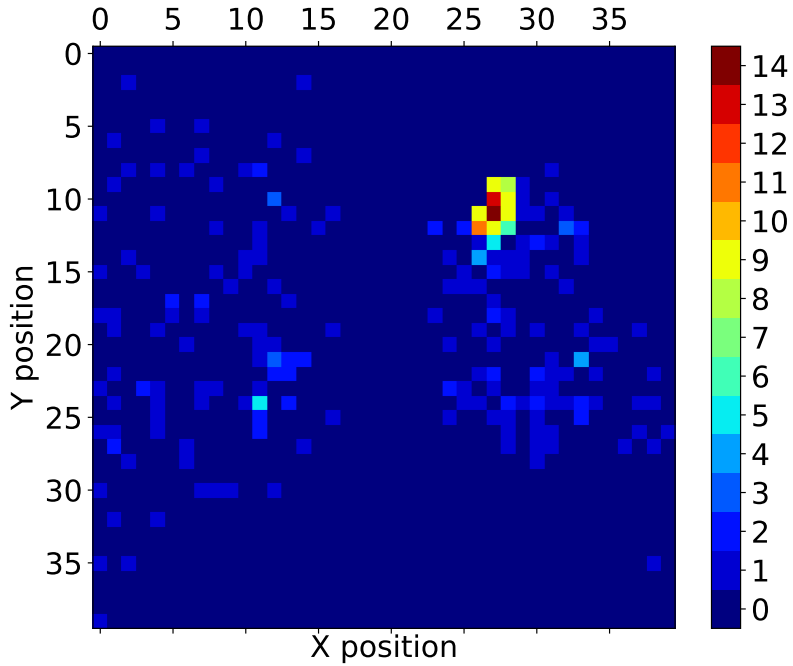


Figure: Linux (Raspbian)

Cartography valid with a different board and a different experimental setup.

Fault models

- Linux: faults on registers (ISA model valid).
- Bare-metal: no faults (we tried hard).

Fault models

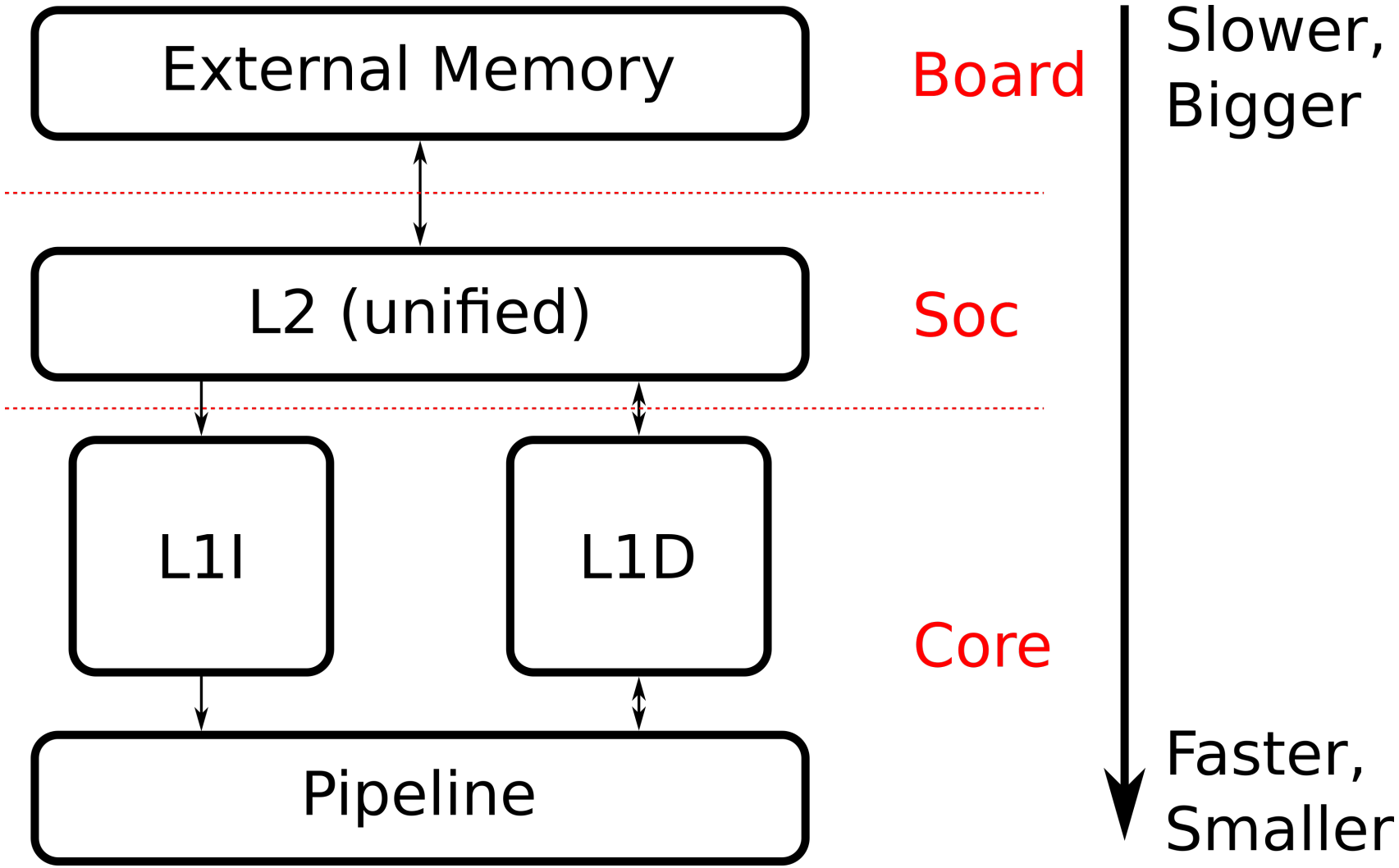
- Linux: faults on registers (ISA model valid).
- Bare-metal: no faults (we tried hard).

Until...

Section 4

Fault on L1I

Reminder on memory hierarchy



First fault observed

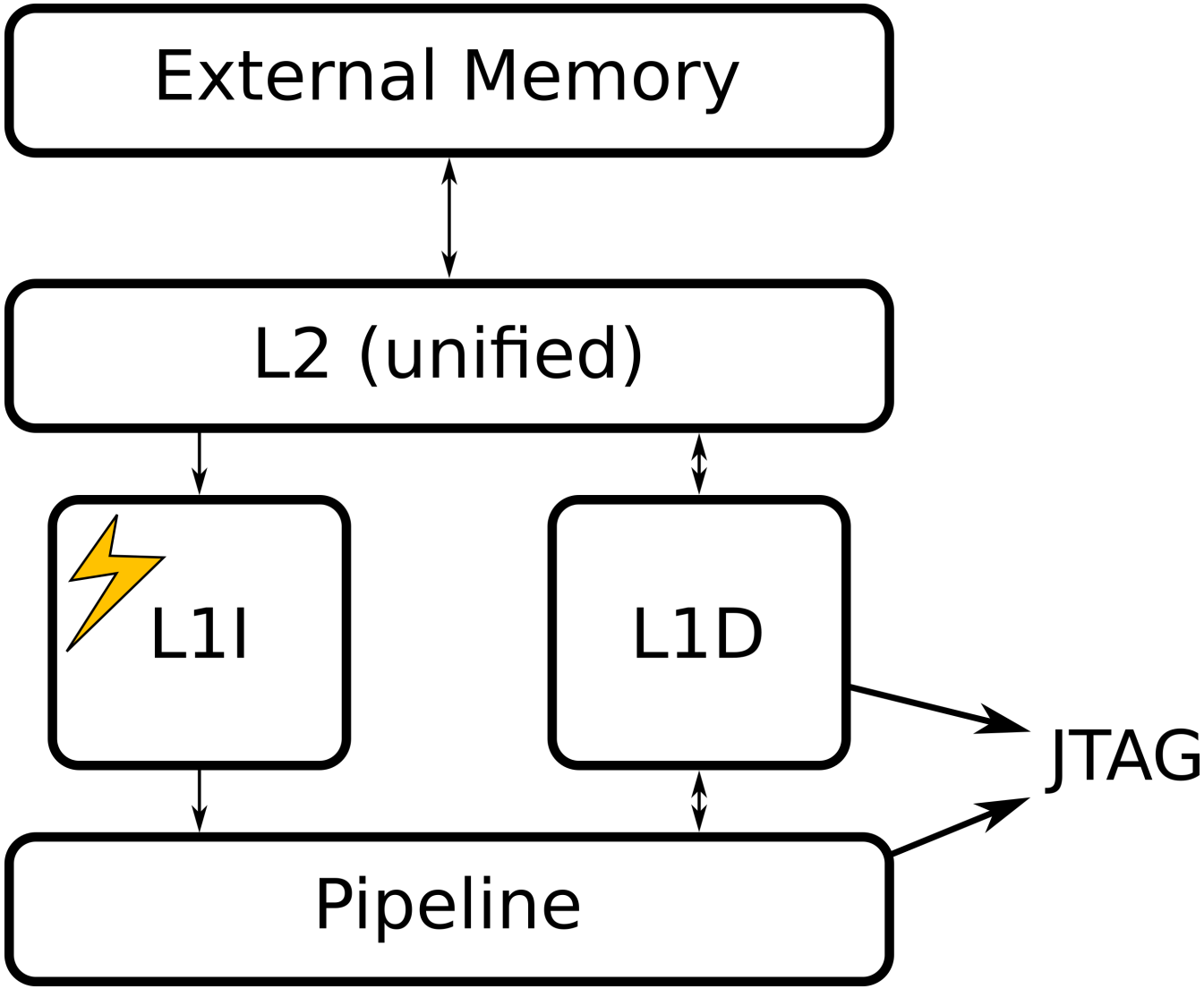
On the first execution of a campaign, the expected result (2500) is not the one received.

Hypotheses

- ISA model: instruction skip.
- Micro-architectural model: L1I modification.

We use JTAG to read the internal state (with L1D viewpoint).

First fault observed



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.

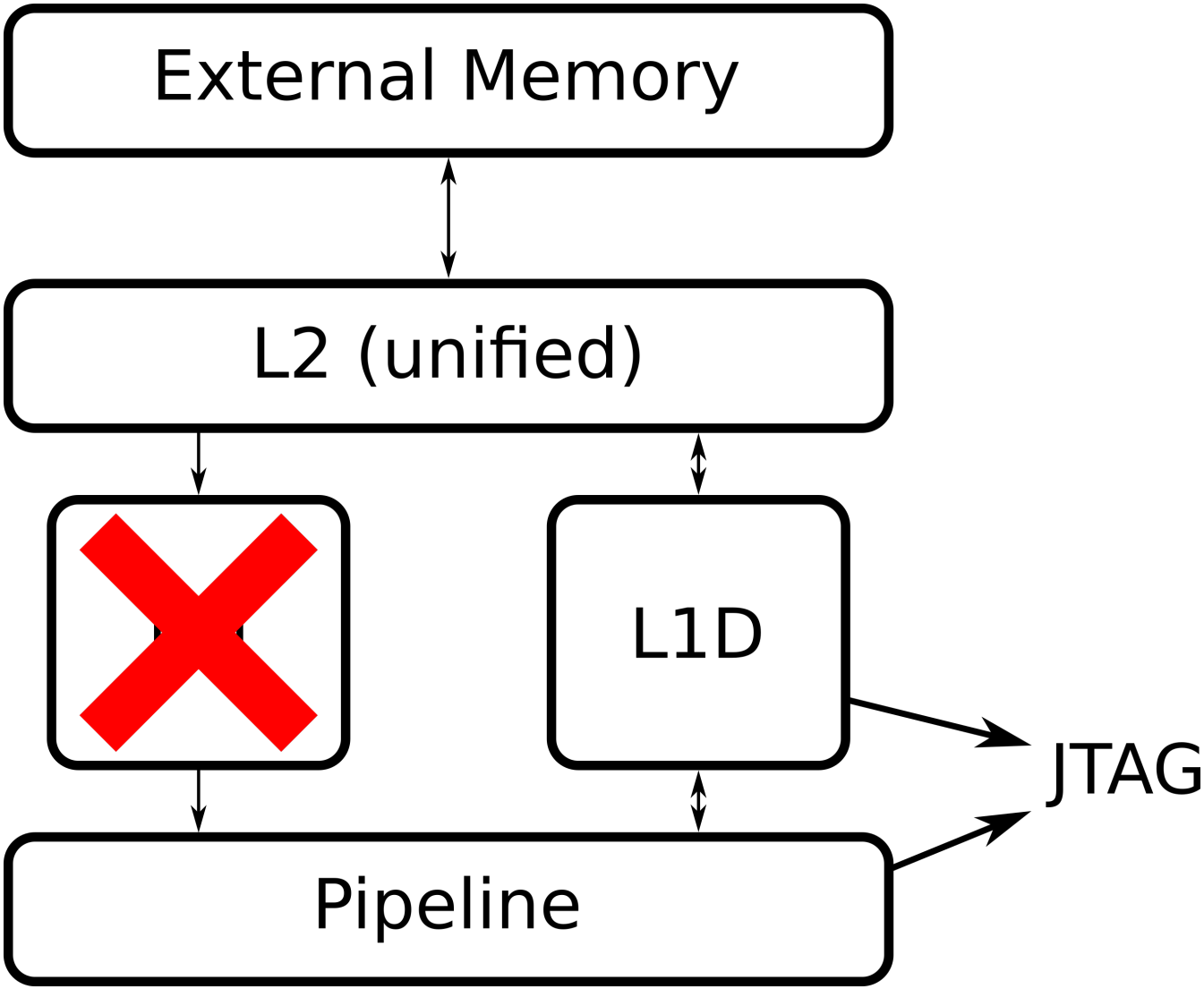
Listing 2: Loop target assembly

```
...
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

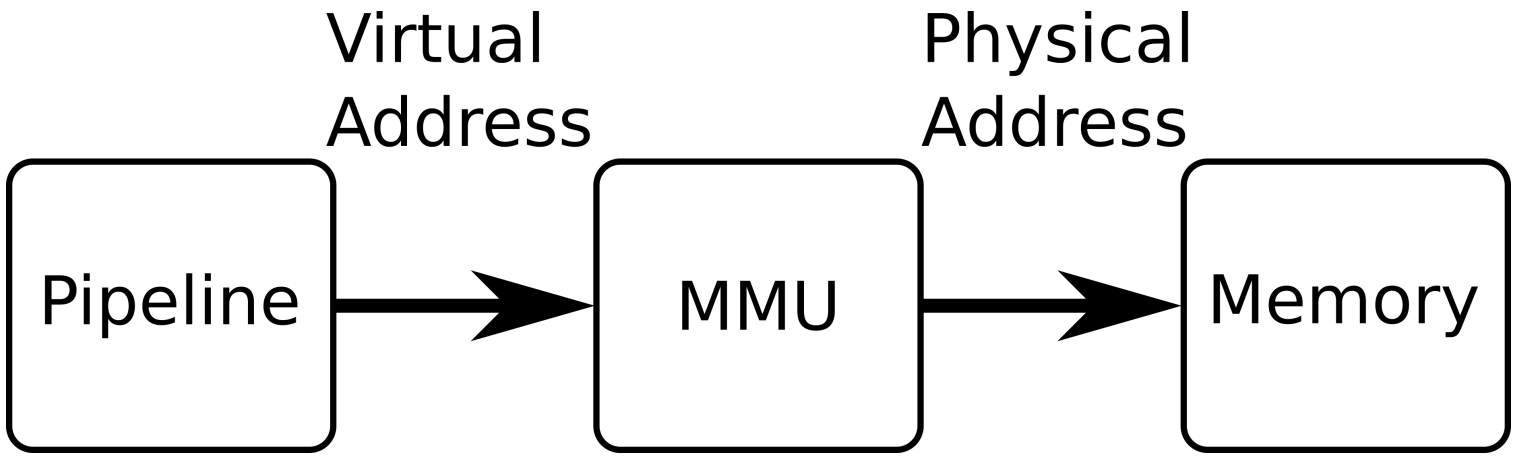
Now, we can reproduce the previous fault, if we inject during the cache reload (lasts $2\mu s$).

Section 5

Fault on the MMU

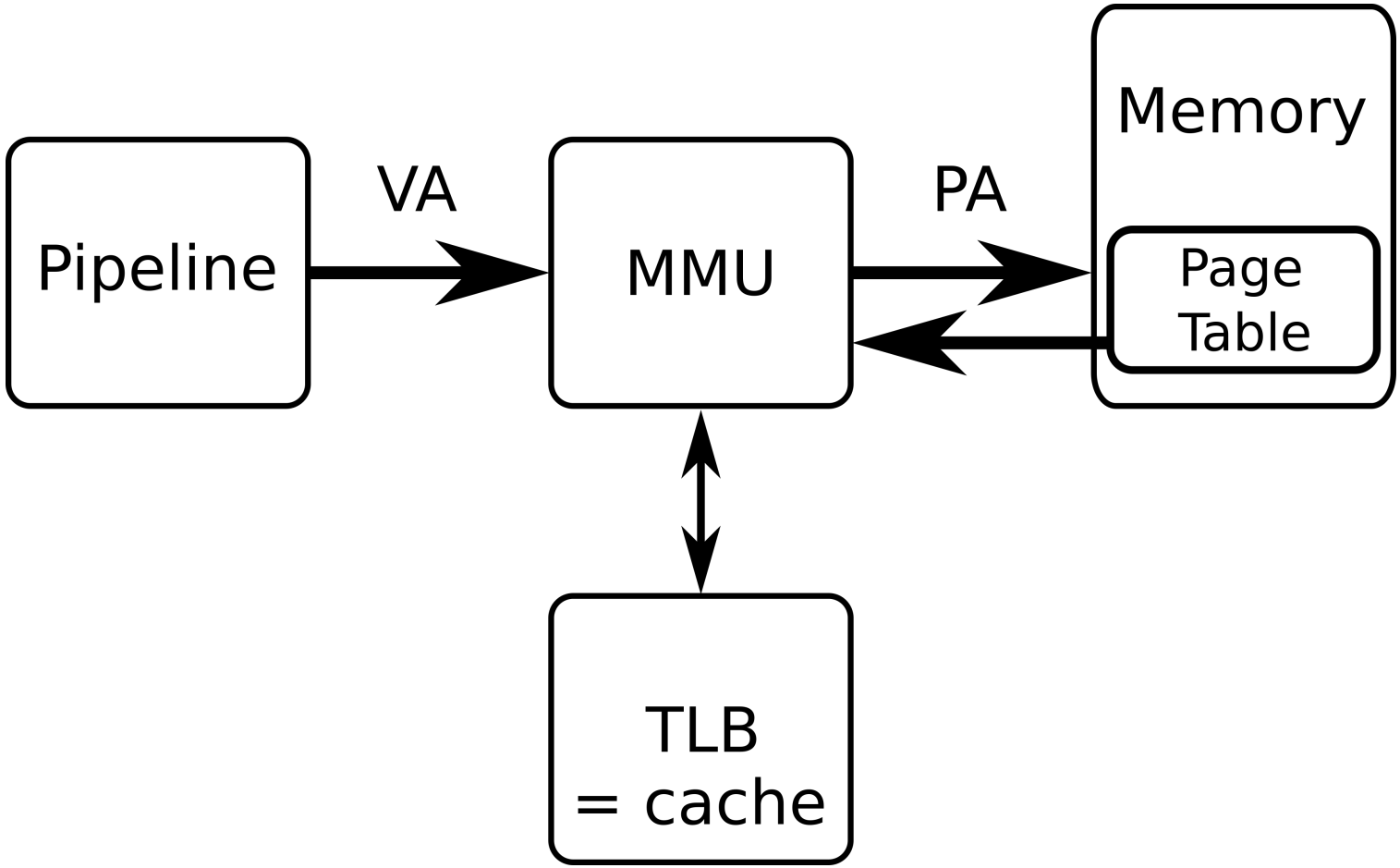
Reminder on the MMU

Principle



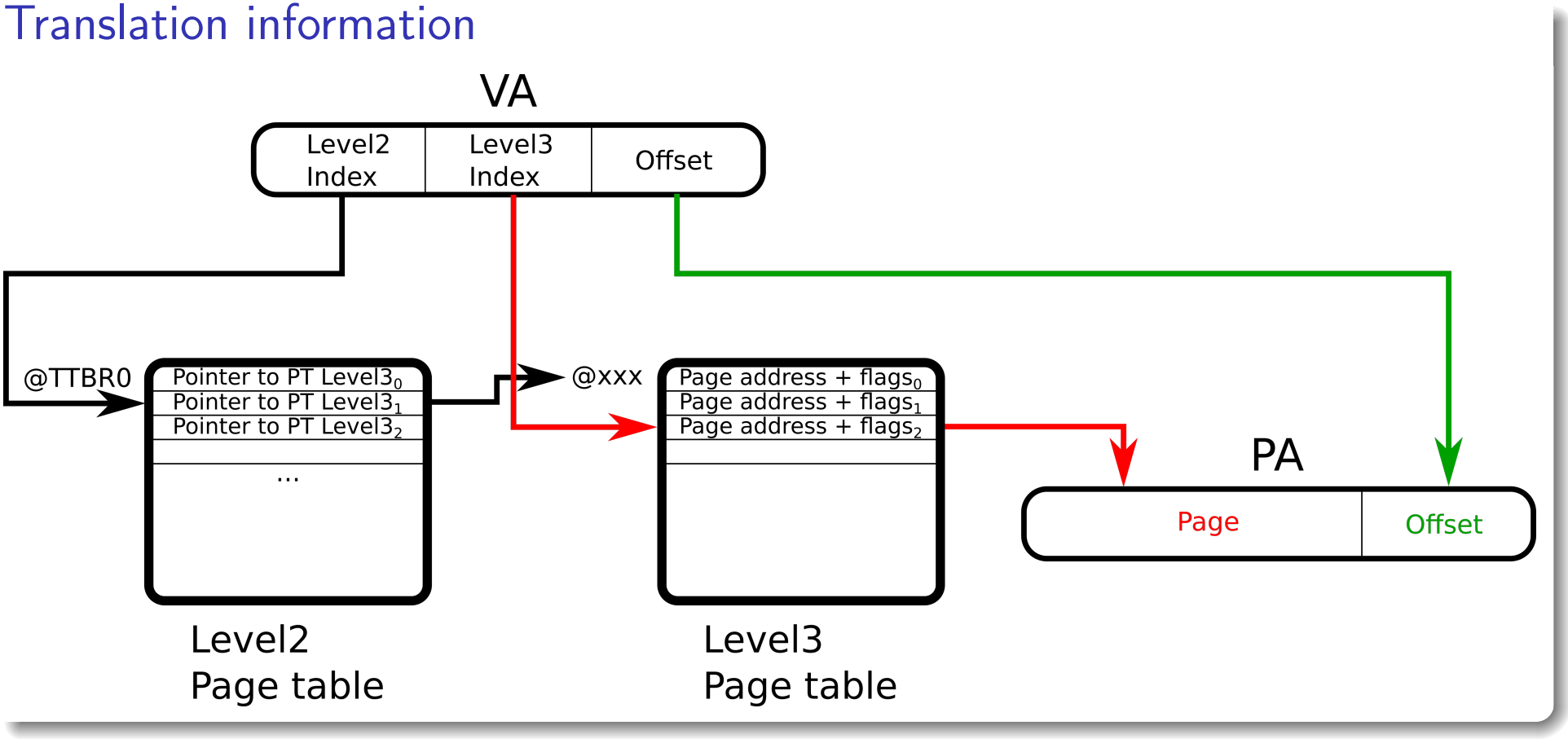
Reminder on the MMU

Principle



ARMv8 implementation

Translation information



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
```

A sneak peek at page tables

Level2

```
0x00380000: 00390003 00000000 003a0003 00000000 003b0003 00000000
```

Level3

```
0x00390000: 00000703 00000000 00010703 00000000 00020703 00000000 00030703 00000000
0x00390020: 00040703 00000000 00050703 00000000 00060703 00000000 00070703 00000000
0x00390040: 00080703 00000000 00090703 00000000 000a0703 00000000 000b0703 00000000
0x00390060: 000c0703 00000000 000d0703 00000000 000e0703 00000000 000f0703 00000000
0x00390080: 00100703 00000000 00110703 00000000 00120703 00000000 00130703 00000000
0x003900a0: 00140703 00000000 00150703 00000000 00160703 00000000 00170703 00000000
0x003900c0: 00180703 00000000 00190703 00000000 001a0703 00000000 001b0703 00000000
0x003900e0: 001c0703 00000000 001d0703 00000000 001e0703 00000000 001f0703 00000000
```

8192 blocks of size 64KiB $\times 3 \approx 1.5 GB$.

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 !

How to recover this mapping ?

Translation instructions

```
at s1e3r, x0
mrs x0, PAR_EL1
```

Put virtual address in `x0`, get physical address and flags in `x0`.

How to recover this mapping ?

Translation instructions

```
at s1e3r, x0
mrs x0, PAR_EL1
```

Put virtual address in `x0`, get physical address and flags in `x0`.

How to execute them ?

1. We do not know if they are present in the code and where.
2. We do not want to dump the whole memory $\rightarrow$ cache interaction.

Solution: we write the instructions in memory with the JTAG.

Page tables, after a fault

Level2

```
0x00380000: 20020703 00000000 20030703 00000000 20040703 00000000
```

Level3

```
0x00390000: 40020607 00000000 40030607 00000000 40040607 00000000 40050607 00000000
0x00390020: 40060607 00000000 40070607 00000000 40000000 8a210002 40000000 8a210002
0x00390040: 400a0607 00000000 400b0607 00000000 400c0607 00000000 400d0607 00000000
0x00390060: 400e0607 00000000 400f0607 00000000 40000000 8a210002 40000000 8a210002
0x00390080: 40120607 00000000 40130607 00000000 40140607 00000000 40150607 00000000
0x003900a0: 40160607 00000000 40170607 00000000 d2b82001 8a210000 40000000 8a210006
0x003900c0: 401a0607 00000000 401b0607 00000000 401c0607 00000000 401d0607 00000000
0x003900e0: 401e0607 00000000 401f0607 00000000 40000000 8a210002 40000000 8a210002
```

Failure cause

Mostly unknown

- Flushing TLB does not change anything.
- The page tables 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 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 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.

F1 and F2

The observed behavior is reproducible, but memory dumps show 2 variants: F1 and F2.

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 (F1). Underlined instructions are part of the infinite loop.

```
0x000489f8: 940087c1 b94017a0 11000400 b90017a0
0x00048a08: b9401ba0 11000400 b9001ba0 b9401ba0
0x00048a18: 7100c41f 54fffeed b9401fa0 11000400
0x00048a28: b9001fa0 b9401fa0 7100c41f 54fffded
```

Figure: Faulty dump (F2).

First observations for F1

```
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 (F1).

- We can observe the fault with JTAG! Fault is present in L1D.
- We can confirm that what we get is what we see: by executing step by step and checking side-effects.
- What about F2 ?

F2

```
0x000489f8: 940087c1 b94017a0 11000400 b90017a0
0x00048a08: b9401ba0 11000400 b9001ba0 b9401ba0
0x00048a18: 7100c41f 54fffeed b9401fa0 11000400
0x00048a28: b9001fa0 b9401fa0 7100c41f 54fffded
```

Figure: Faulty dump (F2).

```
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 (F1).

First observation for F2

```
0x000489f8: 940087c1 b94017a0 11000400 b90017a0
0x00048a08: b9401ba0 11000400 b9001ba0 b9401ba0
0x00048a18: 7100c41f 54fffeed b9401fa0 11000400
0x00048a28: b9001fa0 b9401fa0 7100c41f 54fffded
```

Figure: Faulty dump (F2).

- Similar to F1 but for the first two instructions.
- If we execute step by step, we observe F1 behavior. In particular 940087c1 encodes an unconditional branch that is not taken.

Concluding that the fault is in L2

Counter-argument 1

A fault is present in the MMU: the mapping has been changed. But the mapping is correct for the program space (up to `0x7FFF`).

Argument 1

If we invalidate L1I cache in both cases, nothing change.

Argument 2

If we invalidate L1D to point of coherency at `0x489f8` after F2, the dump becomes the same as F1's. F2 = error in L2 + error in L1D with respect to L2.

L2 conclusion

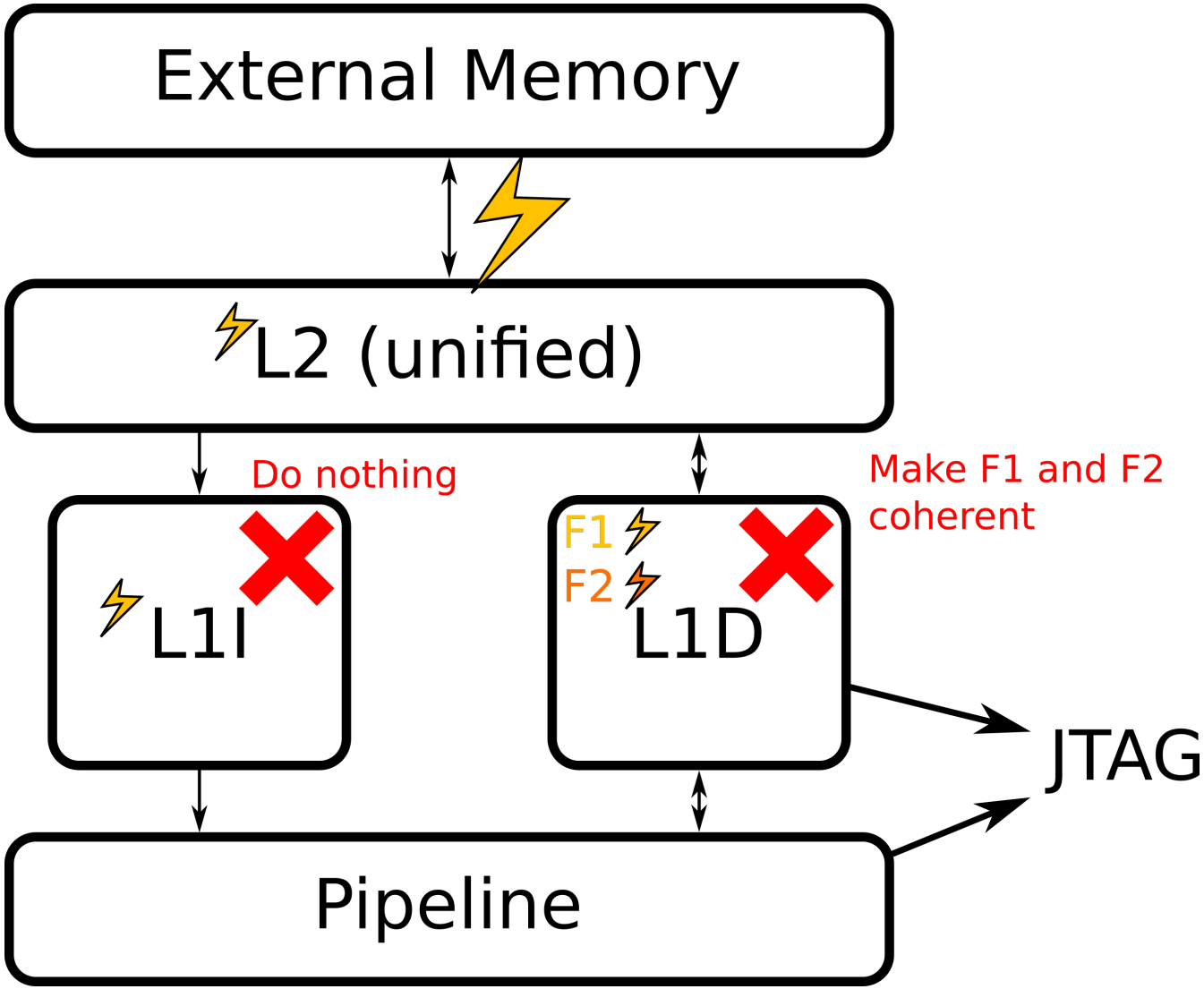
Cause

It appears that 128-bit blocks ($<$ cache line size $=$ 512-bit) are shifted locally in memory. 128-bit is the size of the L2 interface to external memory. **A low Hamming weight fault on the address during a memory transfer ?** Equivalently, a specific MMU error that is not observed with mapping reconstruction.

Lack of control

The attacker does not control the shifting. Exploitation depends heavily on the application, sometimes all it takes is a single modified instruction.

Graphical summary



Section 7

Conclusion

Synchronization

The problem

For a chip running at 1GHz, a clock cycle lasts 1ns. During this time, an electrical signal can propagate to a 15cm distance at most. Jitter is increased with more complex memory hierarchies.

Contrary to attacks on microcontrollers, it is not possible to efficiently target one clock cycle.

Synchronization

The problem

For a chip running at 1GHz, a clock cycle lasts 1ns. During this time, an electrical signal can propagate to a 15cm distance at most. Jitter is increased with more complex memory hierarchies.
Contrary to attacks on microcontrollers, it is not possible to efficiently target one clock cycle.

But we do not care

- Use jitter, and patience, to inject faults. You should eventually inject to the correct timing (if possible).
- Memory transfers are slow: faulting them is easier than targeting a clock cycle.
- RISC architectures: it takes many instructions to perform most tasks. Leaving lots of opportunities to get exploitable failure.

Countermeasures

Encrypt all the things!

The SoC is a network

- Ensure confidentiality, integrity, authenticity between subsystems.
- Prevent DoS (depending on the context).
- Use hardened primitives.

At what cost?

Conclusion / Attacks

What we did

- SoC computations can be disrupted by EMFI.
- We demonstrate faults on L1I, MMU and L2.
- The micro-architectural model is the most faithful model,
- since the ISA model changes with the OS used!

What we didn't do

- Explore faults with muliple cores enabled,
- impact of memory coherency.

Conclusion / Countermeasures

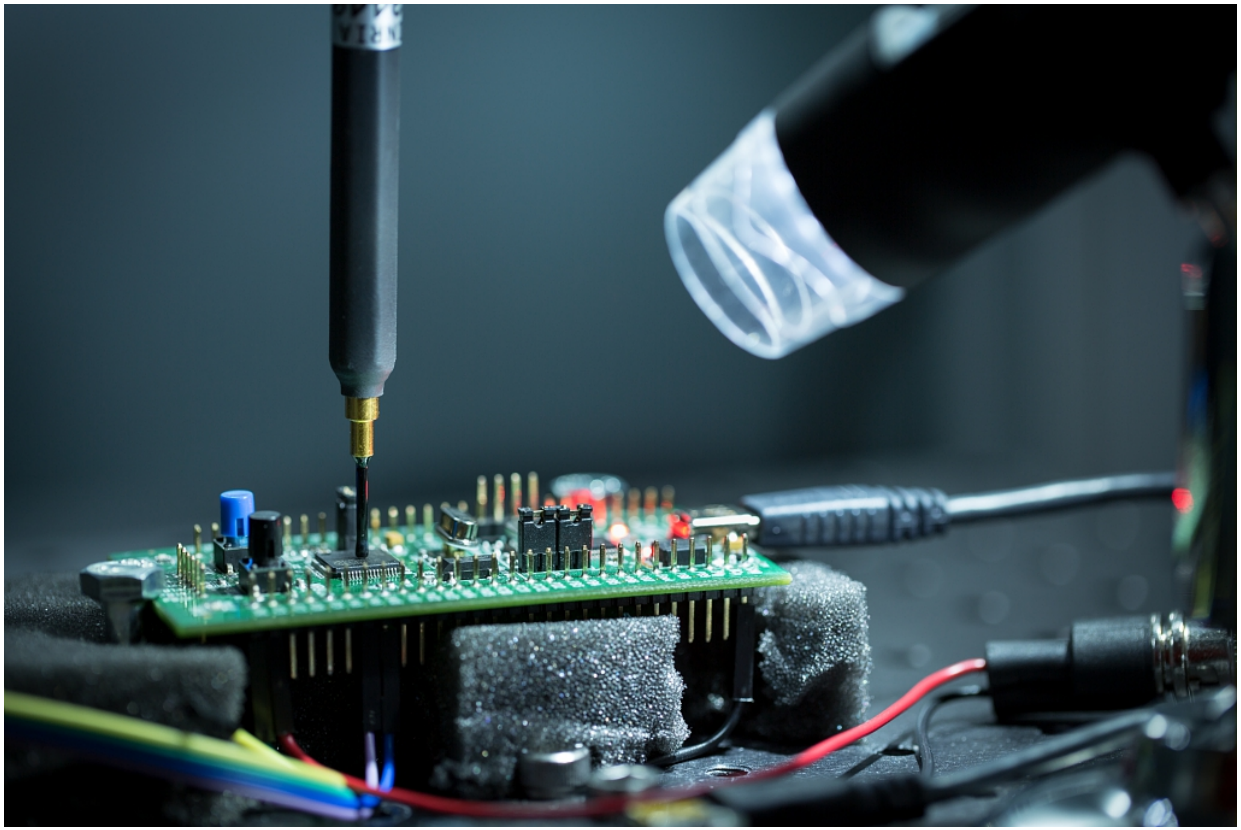
What should be done

- The hardware is not a magically secure black box.
- Secure hardware refoundation is necessary, **required** to run all critical code (today at least all kernel code).
- RISC-V is an opportunity to bring this renewal.

High performance, low power consumption, high security: pick one.

Thank you!

Any questions?



© Inria / Photo C. Morel