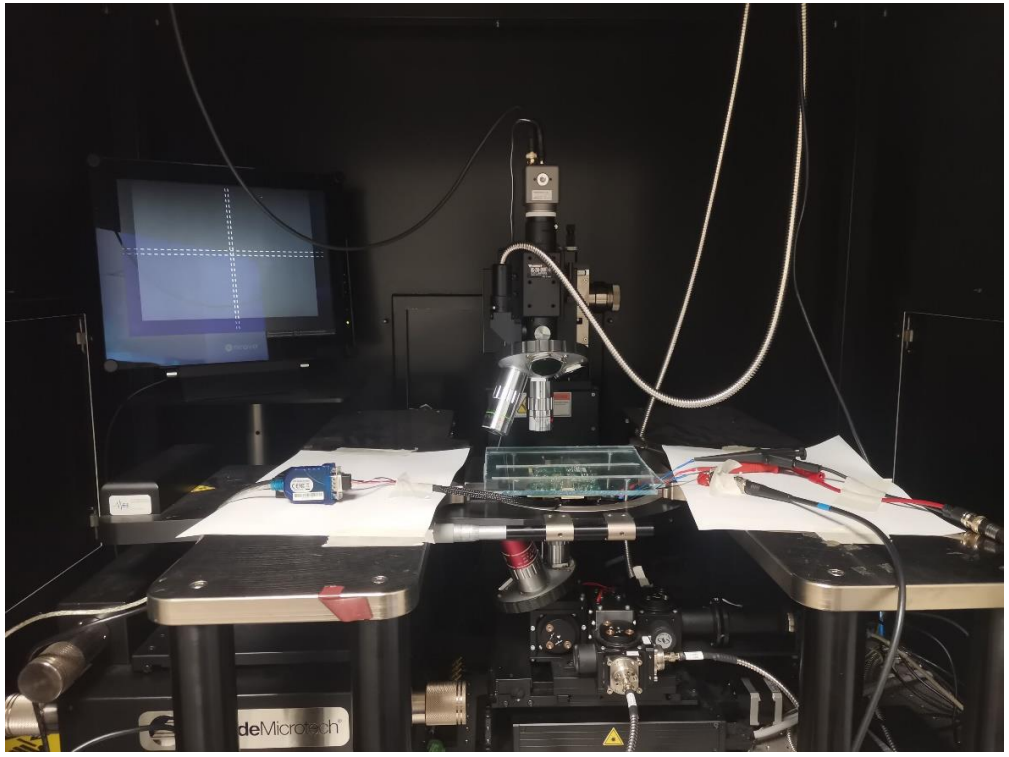
Laser Fault Injection Exploration on System-on-Chip

Soline Casavecchia
Jessy Clédière[1], Jean-Max Dutertre[2]
Simon Pontié[1], Driss Aboulkassimi[1]
(1) CEA-Leti, (2) Mines Saint-Étienne



Pulsys Laser

1. General Context

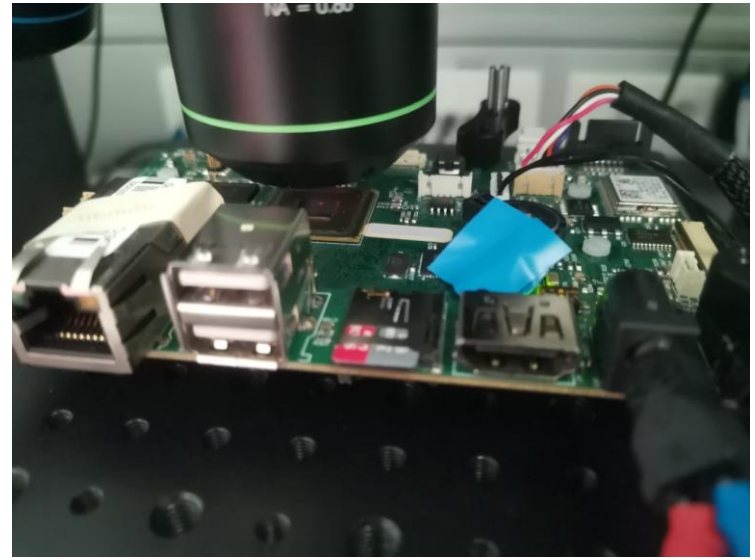
a. State of the Art

b. Objectives & Challenges

c. Problem

State of the Art

- LFI: modifying target intended behaviour through laser pulse illumination [1]
- PEM/PEA: Photon Emission Microscopy/Analysis, using transistors' state change light emissions to detect chip activity [2,3]
- SoC: complex integrated circuit, "all-in-one" functionalities of a system on a single chip
- Existing results of LFI on SoC:
  - LFI on CPU & cache (mainly results on cache) [4]
  - LFI (static) on special status registers [5]
  - Few details on repeatability and full characteristics of faults [4]
  - Exploit: bypass of secure boot on smartphone [5]

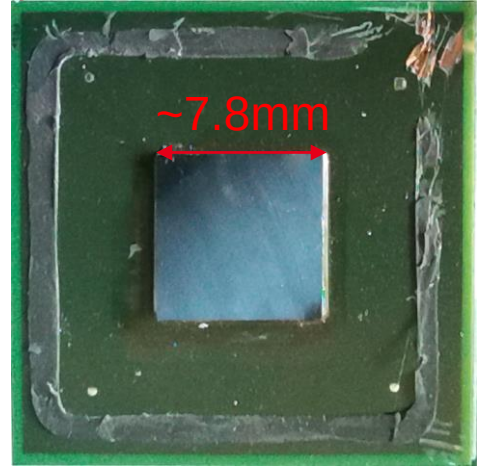


Target under lens, ALPhANOV laser

[1] S.P. Skorobogatov et al., "Optical fault induction attacks", 2002.
[2] R.S. Lima, "Reverse-Engineering and Data Extraction from SRAM using Photon Emission Analysis", 2024.
[3] H. Perrin, "Betrayed by Light: How Photon Emission Microscopy Empowers Register Bit-Level Laser Attacks on Microcontrollers", 2024.
[4] T. Trouchkine et al., "Soc physical security evaluation", Ph.D. dissertation, Université Grenoble Alpes, 2021.
[5] A. Vasselle et al., "Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version", 2020

Objectives & Challenges

Chip exposed backside



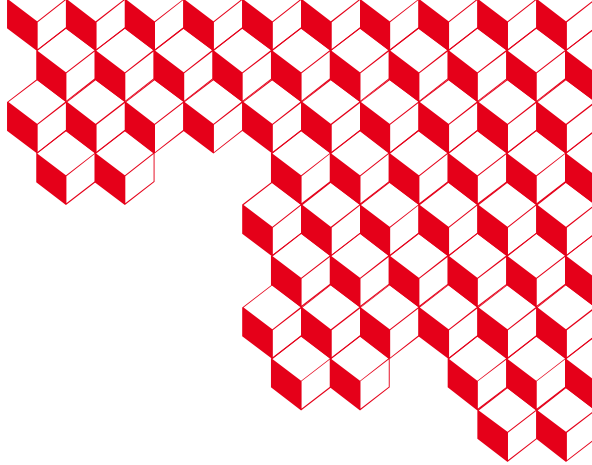
Objectives

- Improve LFI methodologies for multi-core SoCs to conduct campaigns more efficiently time-wise
- Characterise SoC CPU and cache vulnerabilities
- Consider potential exploitations of said vulnerabilities (e.g., cryptographic key extraction)

Challenges

- Large Si die area to cover
- Multi-core activity
- Multiway set-associative cache → difficulty ascertaining data position/location
- Understanding cache structure → descrambling/reverse needed to understand cache organisation [6,7]
- Synchronisation of attacks with runtime processing

[6] A. Stuffer, “Interactive and non-destructive verification of sram-descrambling with laser”, *Microelectronics Reliability*, 2004.
[7] S. Chef et al., “Descrambling if embedded SRAM using a laser probe”, 2018



“How can we better apply – and modify – current LFI methodologies to better assess and analyse SoC vulnerabilities? ”

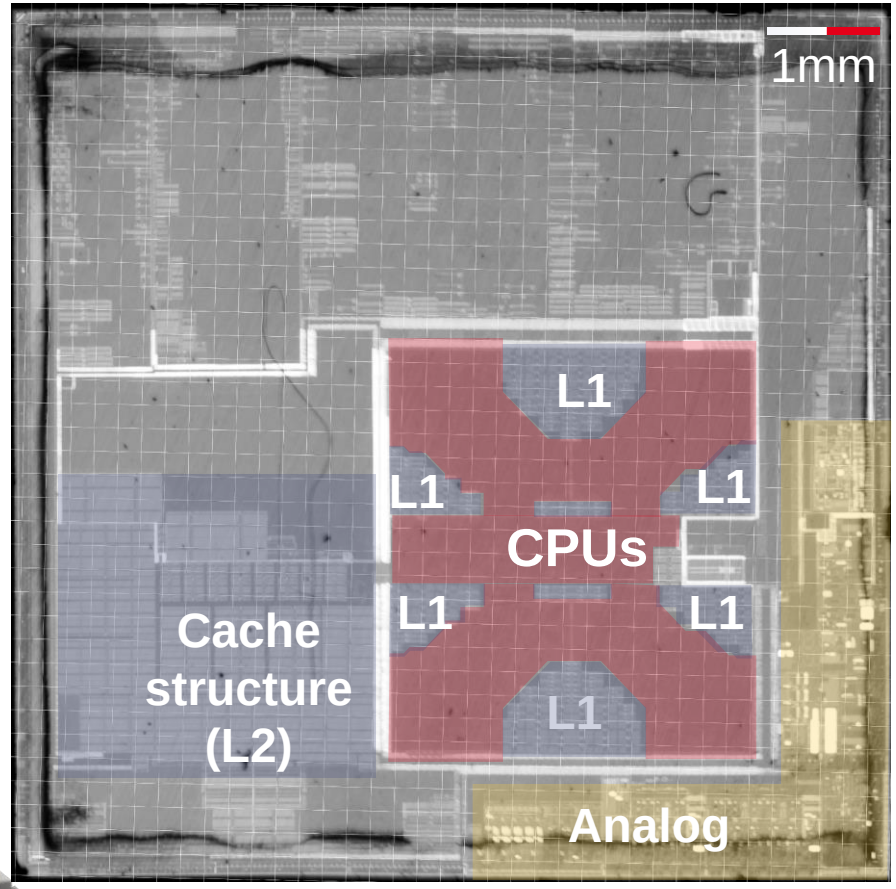
2. Technical Context

a. Target Information

b. Methodology

Target Information

- 32-bit architecture, technological node 40nm
- Linux Yocto OS – ARMv7 Cortex-A9 – quad-core
- CPU frequency – up to 996MHz
- Si thickness (backside LFI): 255 – 280µm
- total die area: ~ 61mm^2 → ~ 15·10^6 positions to cover with a 2µm-grid ⇔ laser spot diameter ~5 µm
- 32KB L1I & L1D caches per CPU (SRAM)
- 1MB L2 cache (SRAM)
- Greybox
- Thumb2



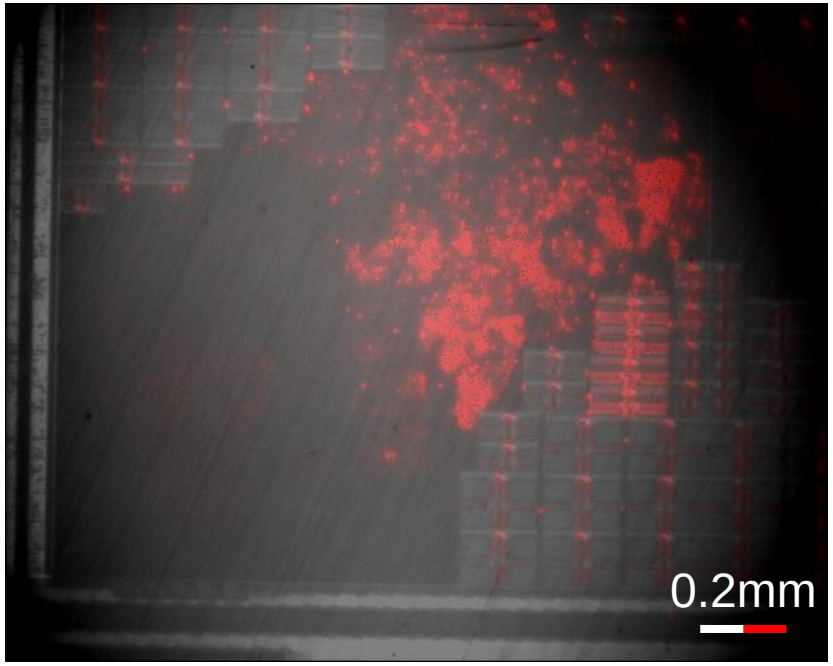
Chip NIR imaging

Target chip on its board

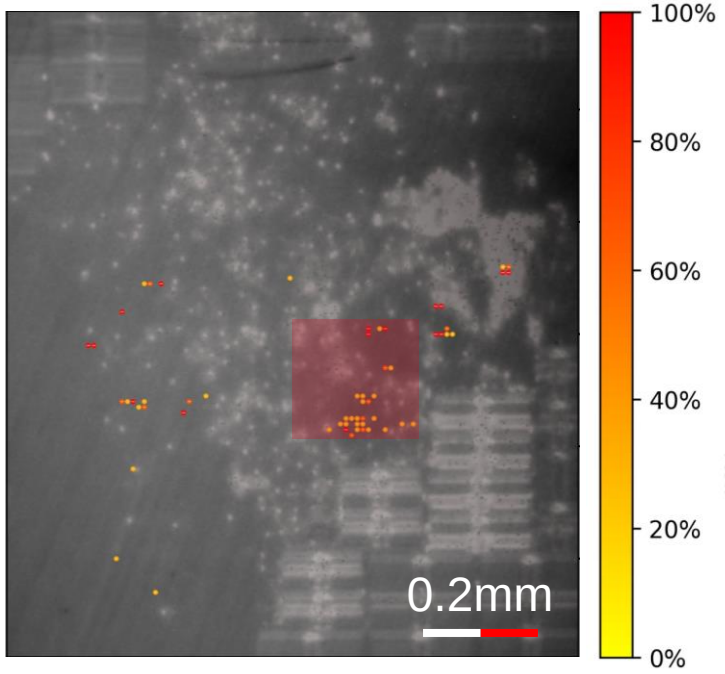
Methodology

1	2	3	4	5	6
SoC Visual Analysis:	**Photon Emission Microscopy (PEM):**	**Coarse grain LFI (x5 LENS):**	**Fine grain LFI (x20 LENS):**	**Fault modelling & characterisation**	**Repeatability testing**
Identify structures on NIR imagery (CPUs, etc.)	Transistors emit light when switching state → reflects CPU & cache activity	Explore active area with coarse grain to narrow down parameters (timing, pulse power, etc.)	After first results analysis, target specific area with high fault density	Sort & categorise faults based on possible fault model (e.g. mono-bit flip)	Select specific fault categories of interest and test repeatability in small area

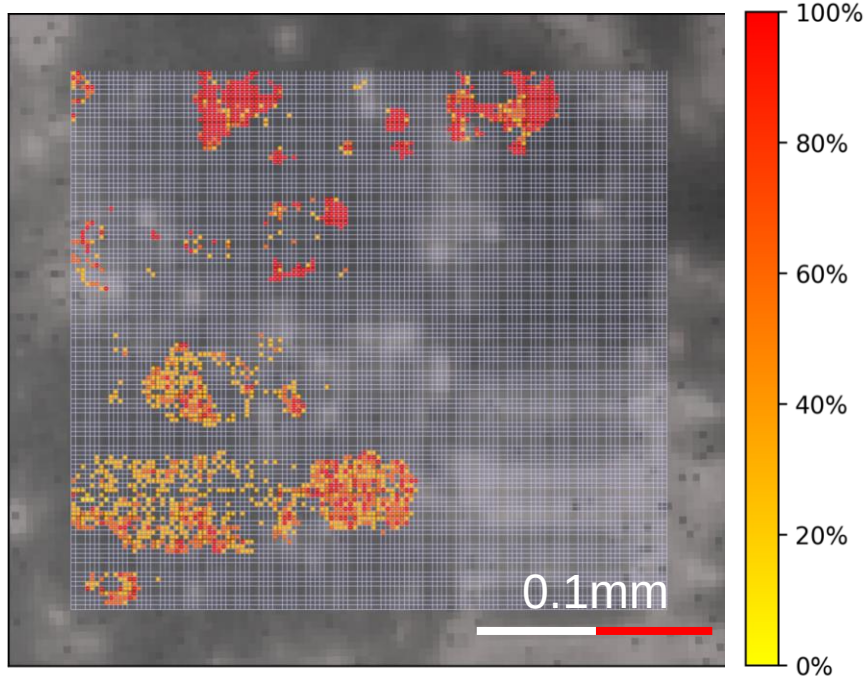
Methodology: LFI on CPU (steps 2, 3, 4)



PEM: CPU active area (ADD with 0)



Faults with x5 lens across CPU active area: identifying dense fault area (red area) (power = 0.5W, pulse width = 50ns)



Faults with x20 lens in fault-dense area (power = 0.3W, pulse width = 50ns)

3. Contributions

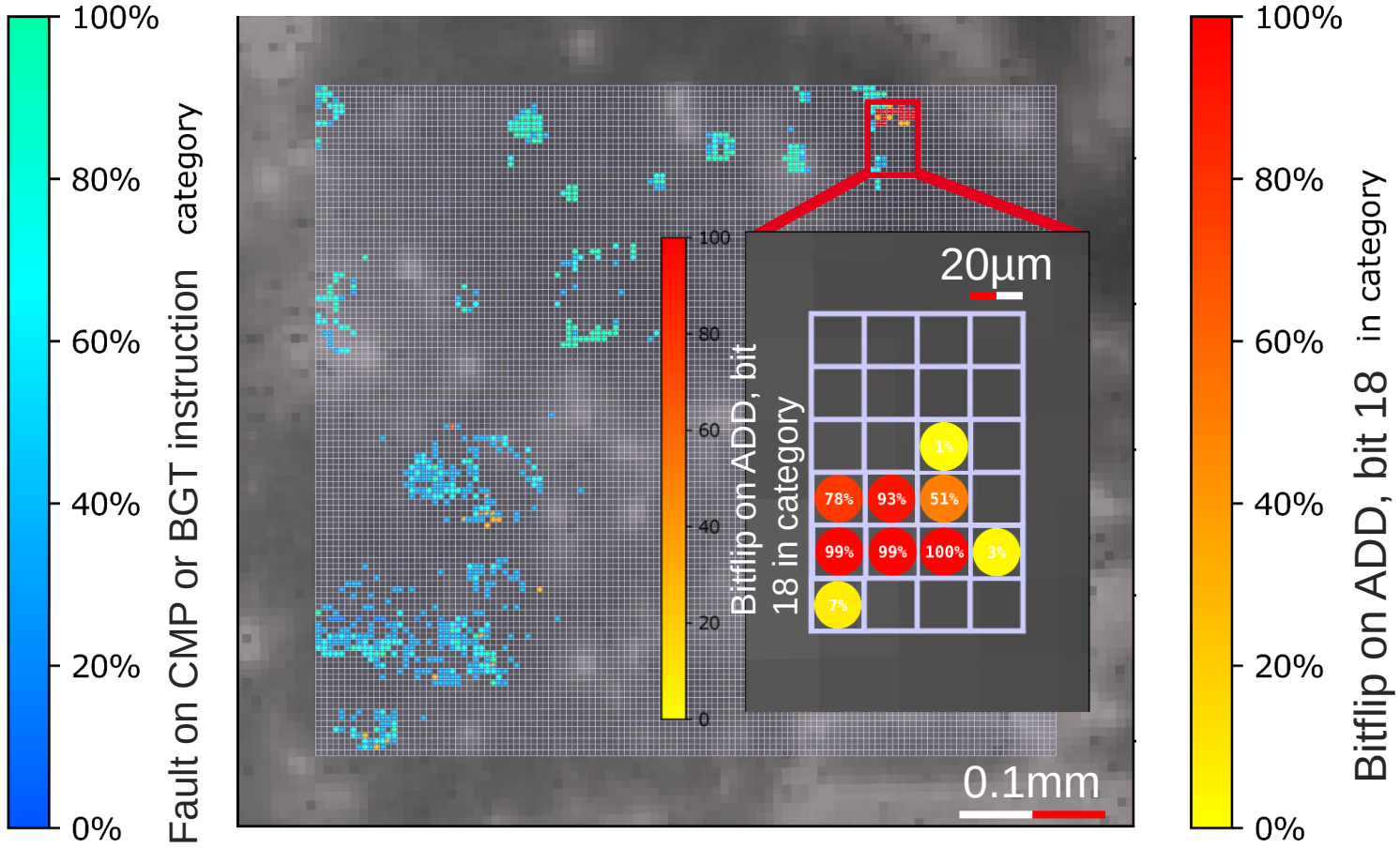
a. Results of fault injection on CPU

b. Cache characterisation / Reverse Engineering cache

c. Results of fault injection on cache

Analysis & Results LFI CPU

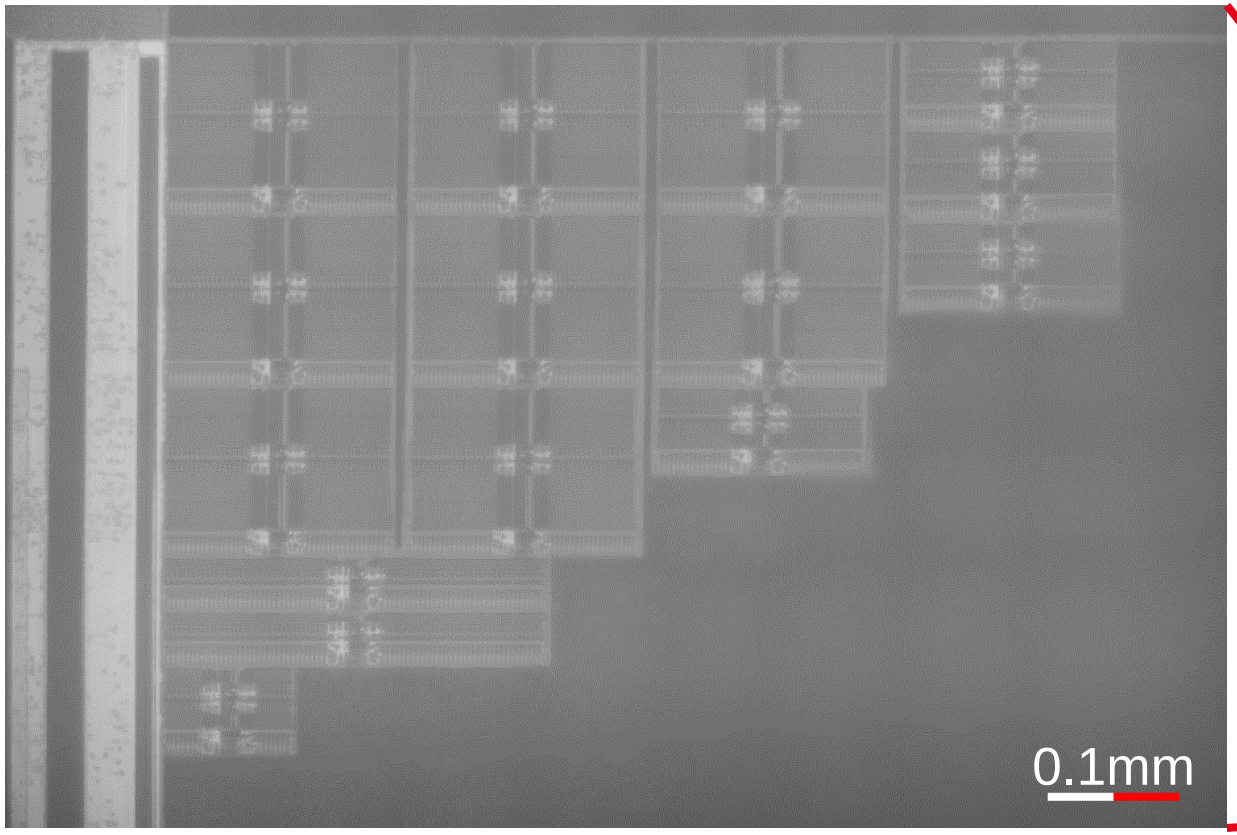
- Observed: difference in register state after attack with golden reference
- Model [8]: monobit flip on 1 instruction (hypothesis)
  - faulting instruction (uncompressed)
- (a) Bitflip bit 18 of ADD encoding: source register R5 → **R1**
- (b) Premature loop exit (counter partially incremented)
- Repeatability (a): 100%
- Repeatability (b): 85%



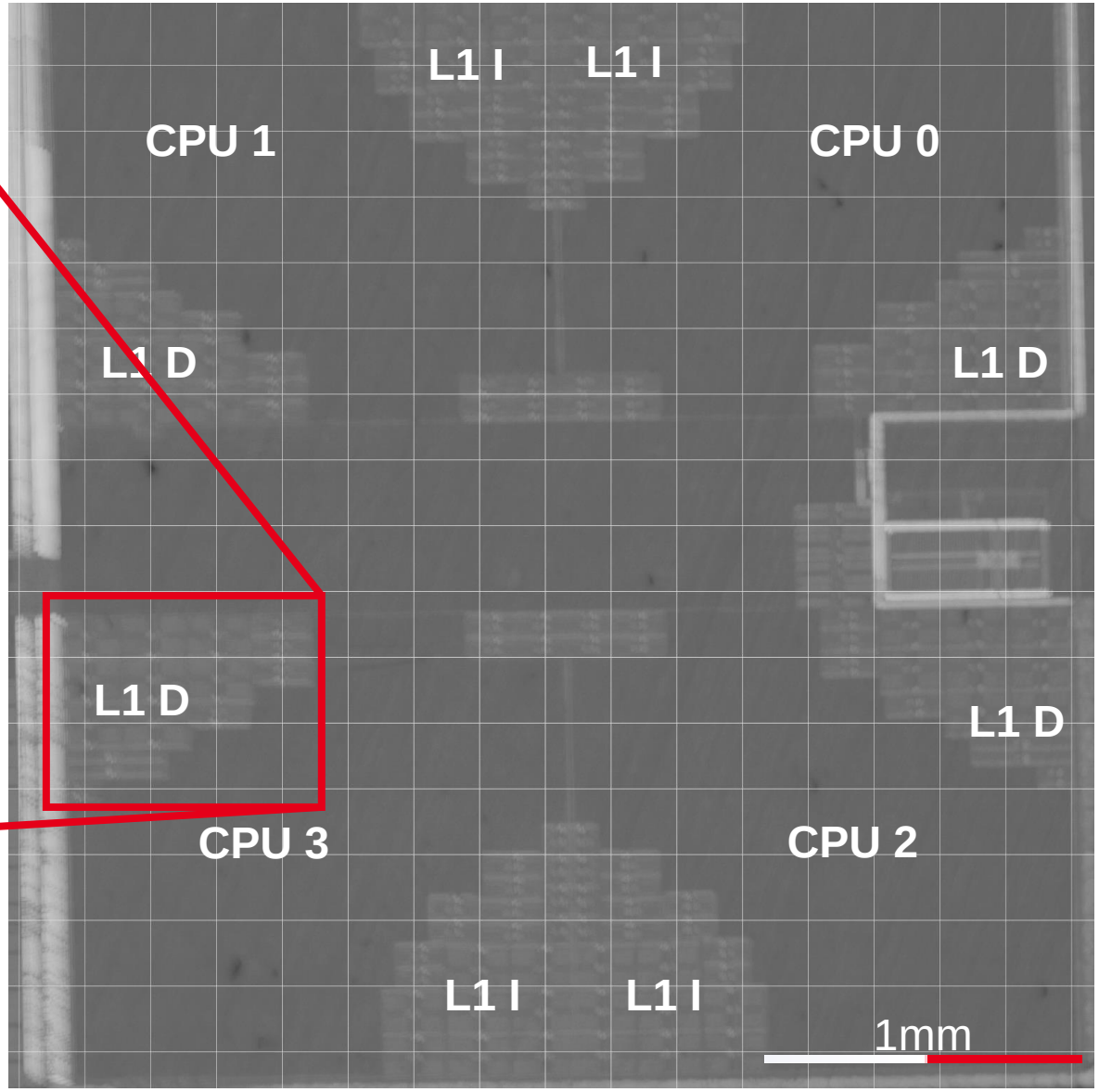
CPU fault injection for ADD and branch; repeatability test for fault on ADD instruction (zoom) (power = 0.3W, pulse width = 50ns)

[8] J.-M. Dutertre et al., “Laser Fault Injection at the CMOS 28 nm Technology Node: an Analysis of the Fault Model”, 2018.

Cache characterisation/reverse



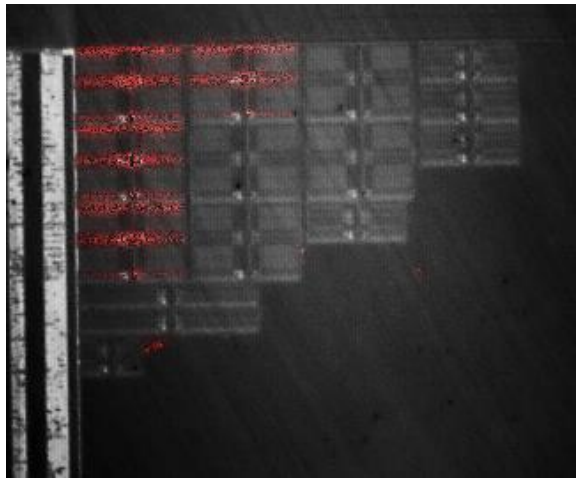
Zoom on L1 D cache

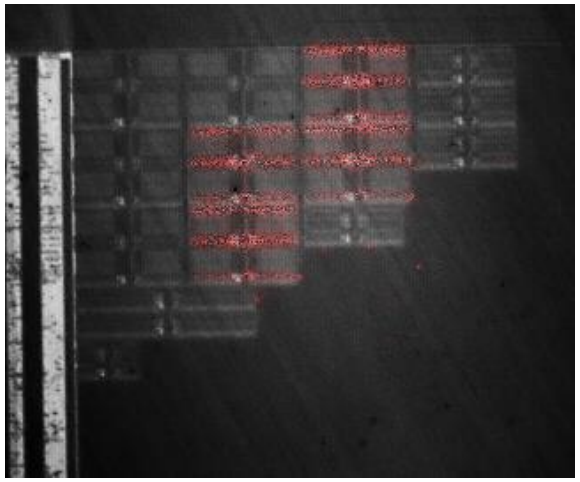


The four CPUs with their L1 caches

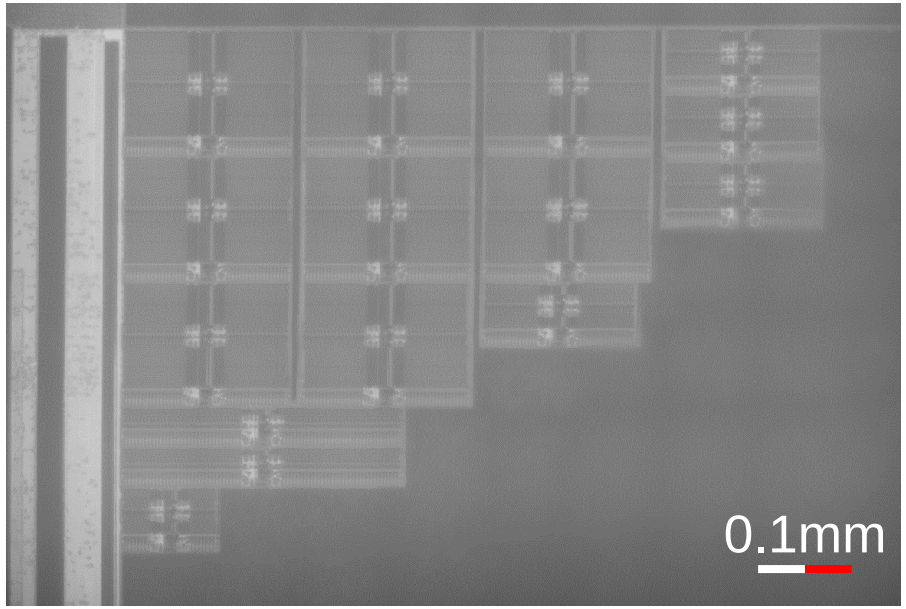
Cache characterisation/reverse

- Characteristics L1 D cache:
  - 32 KB
  - 256 sets / 4-way cache
  - 1 word = 32 bits
  - 1 line = 8 words
  - 32-bit physical address:
    - 8 bits line address,
    - 5 bits byte offset (13 bits),
    - 19 bits tag





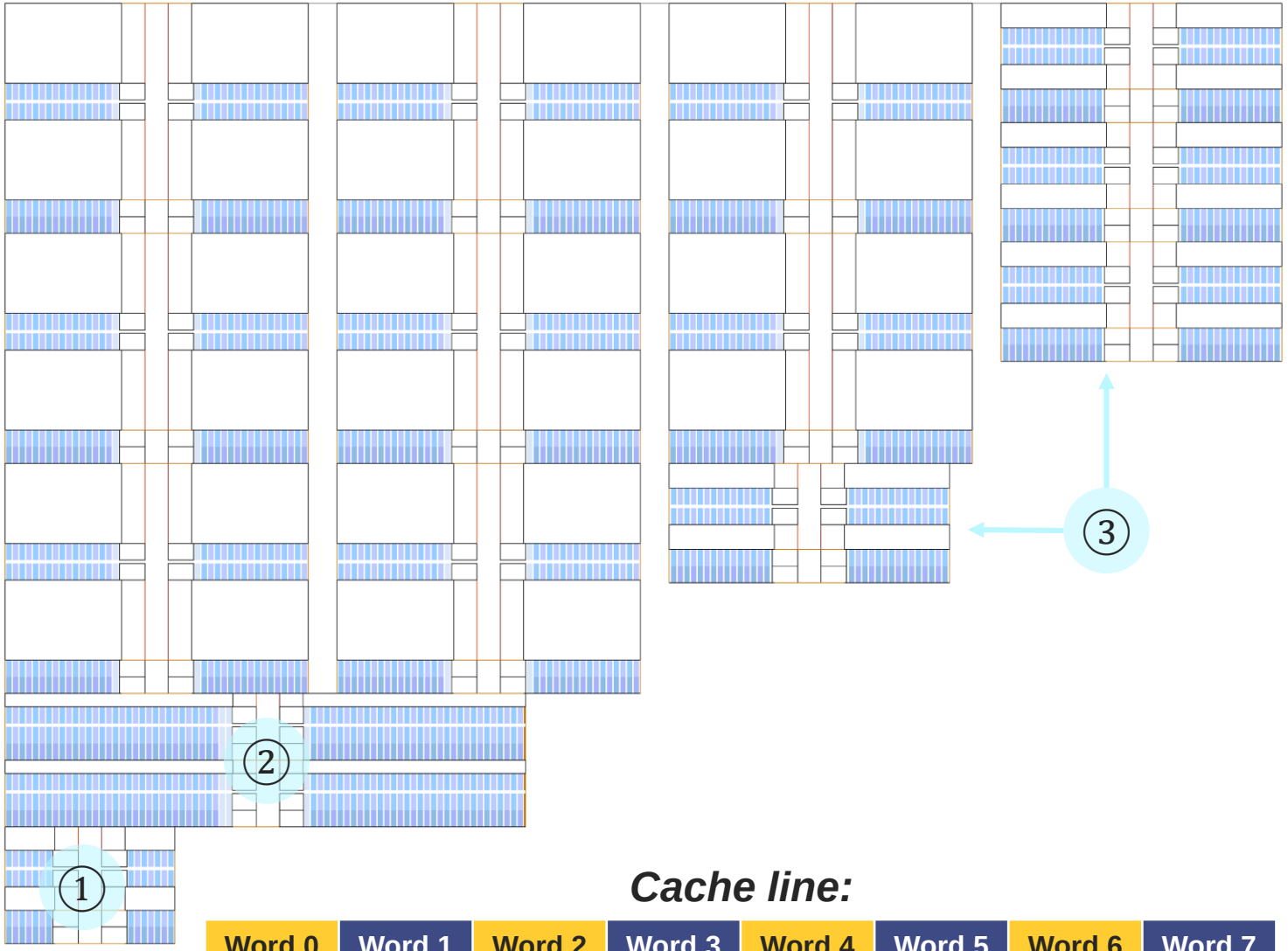
PEM addressing even offset word (left) vs. odd offset word (left)



NIR imagery of CPU3 L1D cache

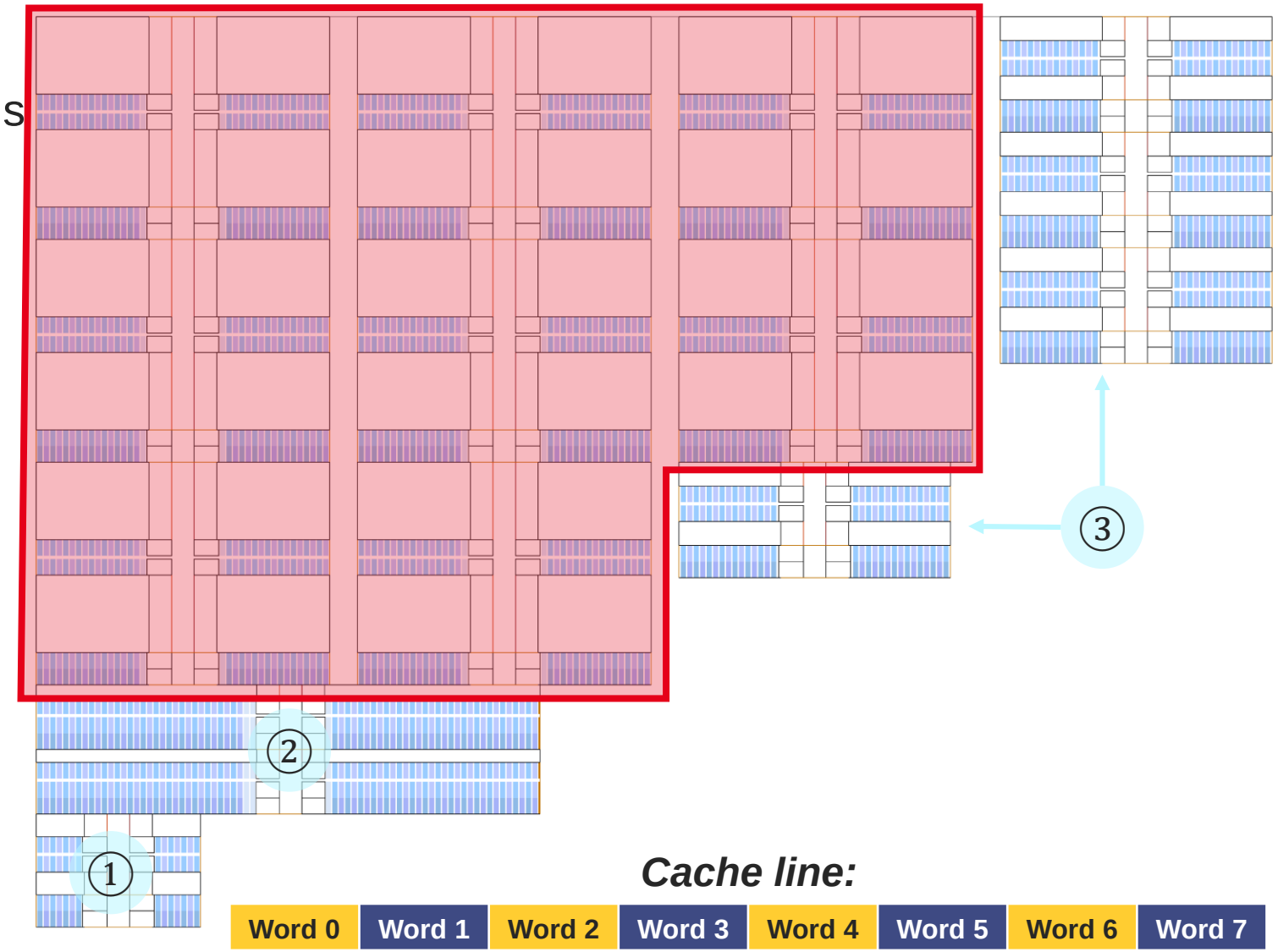
Cache characterisation/reverse

- PEM + IR:
  - Fill the cache: 256 lines × 8 words
  - 8 blocks containing data
    - 4 blocks for even offset words
    - 4 blocks for odd offset words
  - 4 even offset words/line
    × 4 cache ways
    = 16 even offset words
    in 4 blocks
    - ➢ **1 block ⇔ 1 word offset & 4 cache ways?**
    - ➢ **1 block ⇔ 1 cache way & 4 word offsets?**
  - 1 data block:
    - 1 word loaded ⇔ 4 bands
    - 2 buffer zones



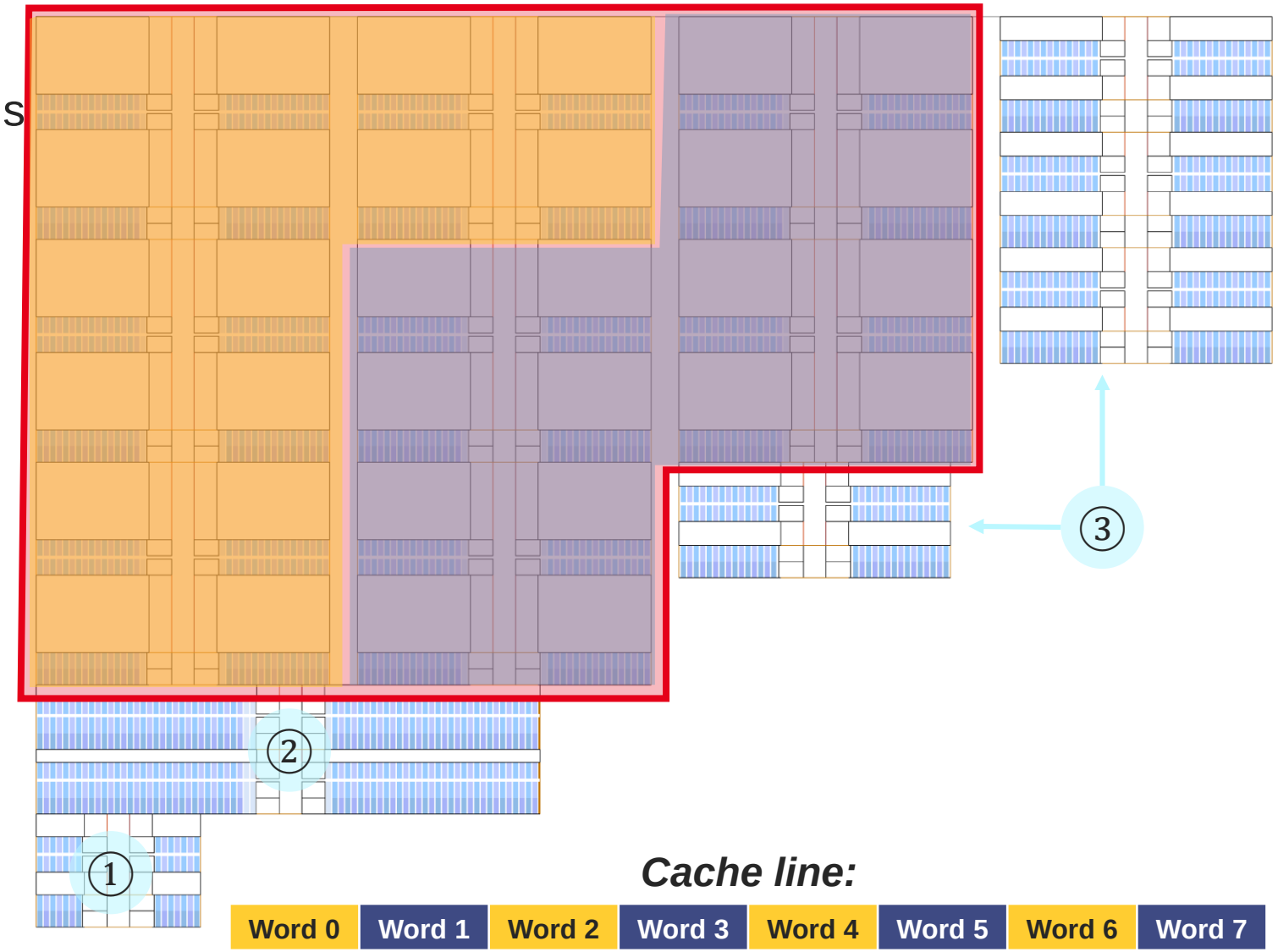
Cache characterisation/reverse

- PEM + IR:
  - Fill the cache: 256 lines × 8 words
  - 8 blocks containing data
    - 4 blocks for even offset words
    - 4 blocks for odd offset words
  - 4 even offset words/line
    × 4 cache ways
    = 16 even offset words
    in 4 blocks
    - ➢ **1 block ⇔ 1 word offset & 4 cache ways?**
    - ➢ **1 block ⇔ 1 cache way & 4 word offsets?**
  - 1 data block:
    - 1 word loaded ⇔ 4 bands
    - 2 buffer zones



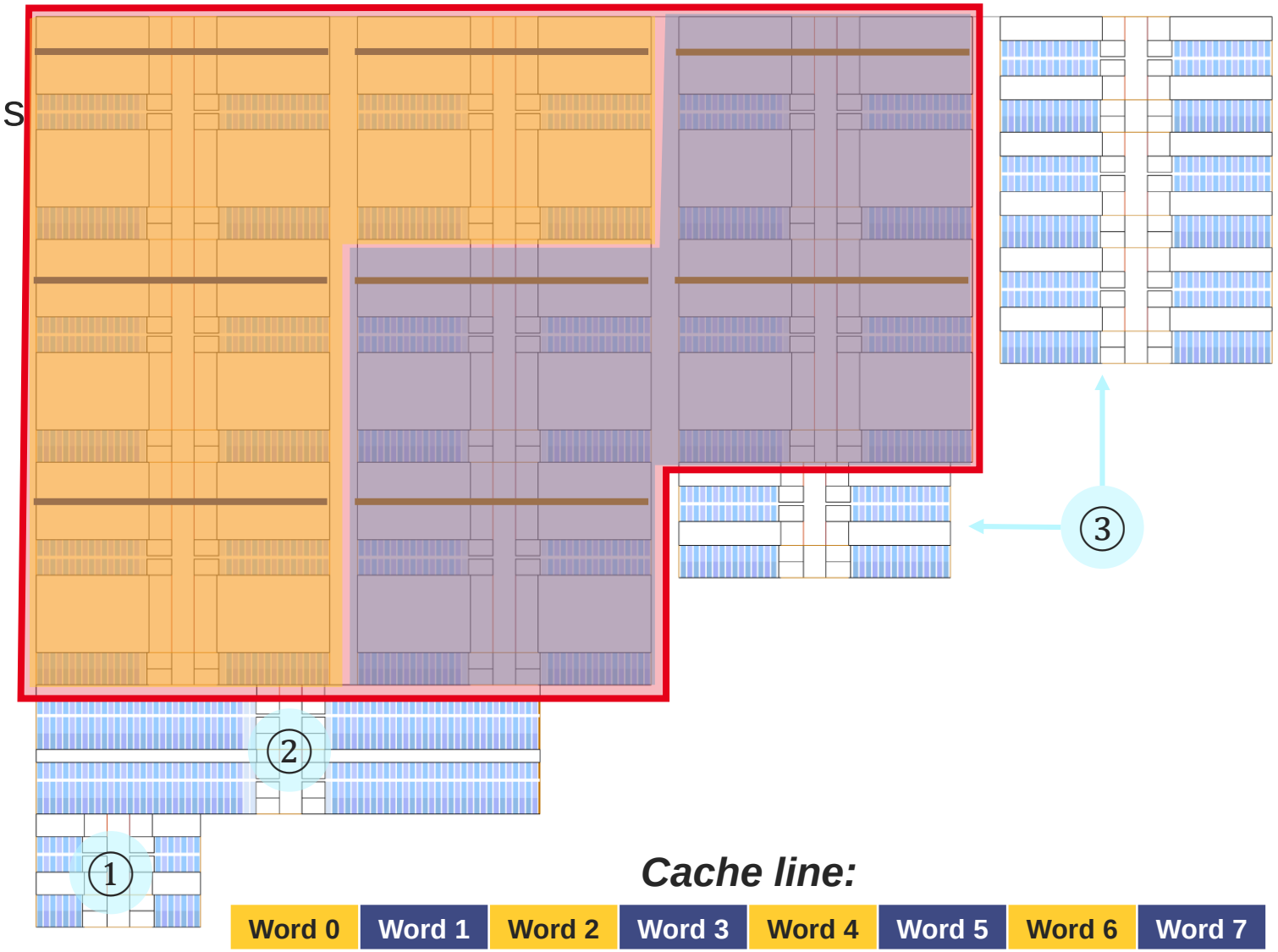
Cache characterisation/reverse

- PEM + IR:
  - Fill the cache: 256 lines × 8 words
  - 8 blocks containing data
    - 4 blocks for even offset words
    - 4 blocks for odd offset words
  - 4 even offset words/line
    × 4 cache ways
    = 16 even offset words
    in 4 blocks
    - ➢ **1 block ⇔ 1 word offset & 4 cache ways?**
    - ➢ **1 block ⇔ 1 cache way & 4 word offsets?**
  - 1 data block:
    - 1 word loaded ⇔ 4 bands
    - 2 buffer zones



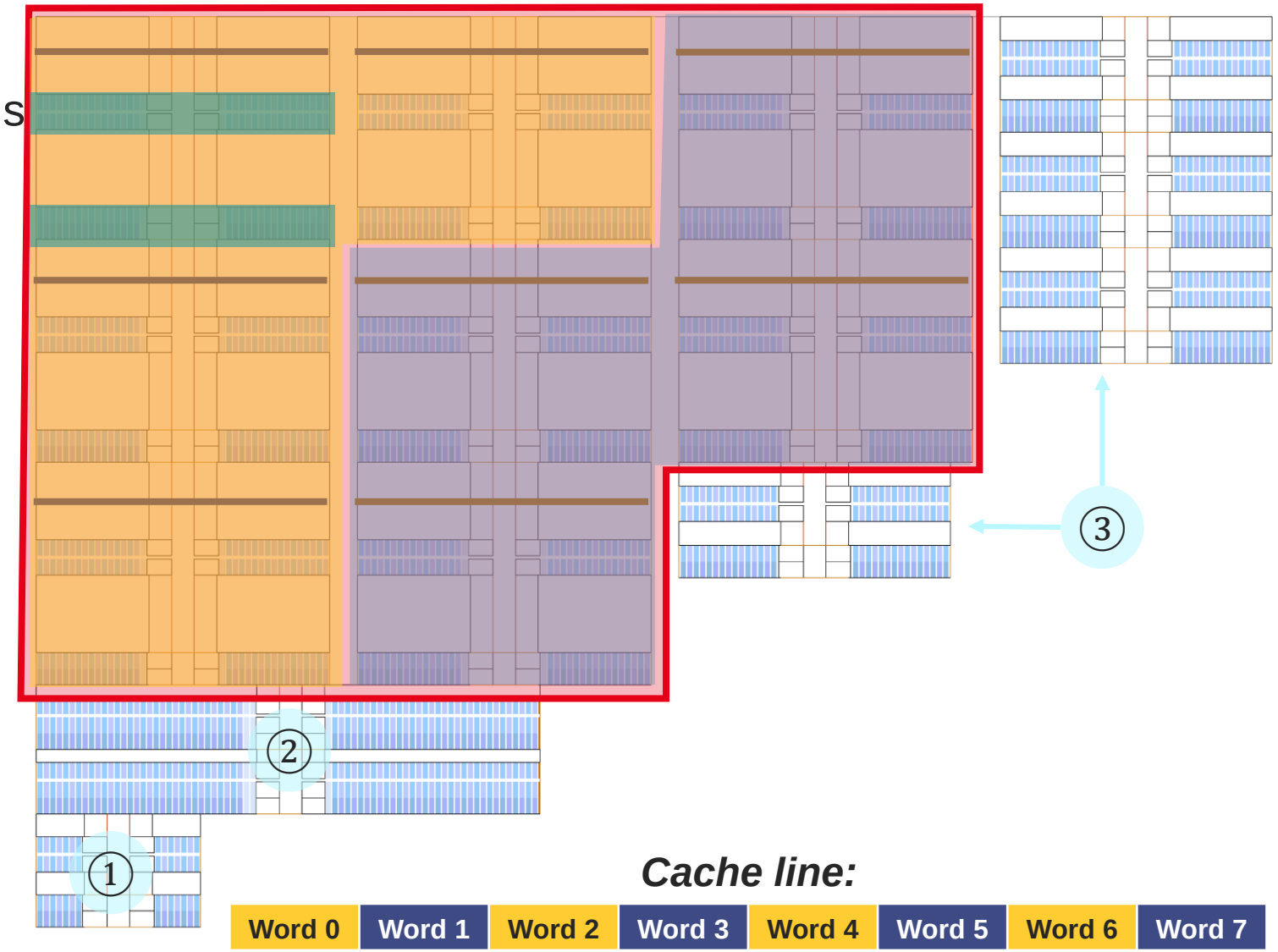
Cache characterisation/reverse

- PEM + IR:
  - Fill the cache: 256 lines × 8 words
  - 8 blocks containing data
    - 4 blocks for even offset words
    - 4 blocks for odd offset words
  - 4 even offset words/line
    × 4 cache ways
    = 16 even offset words
    in 4 blocks
    - ➢ **1 block ⇔ 1 word offset & 4 cache ways?**
    - ➢ **1 block ⇔ 1 cache way & 4 word offsets?**
  - 1 data block:
    - 1 word loaded ⇔ 4 bands
    - 2 buffer zones

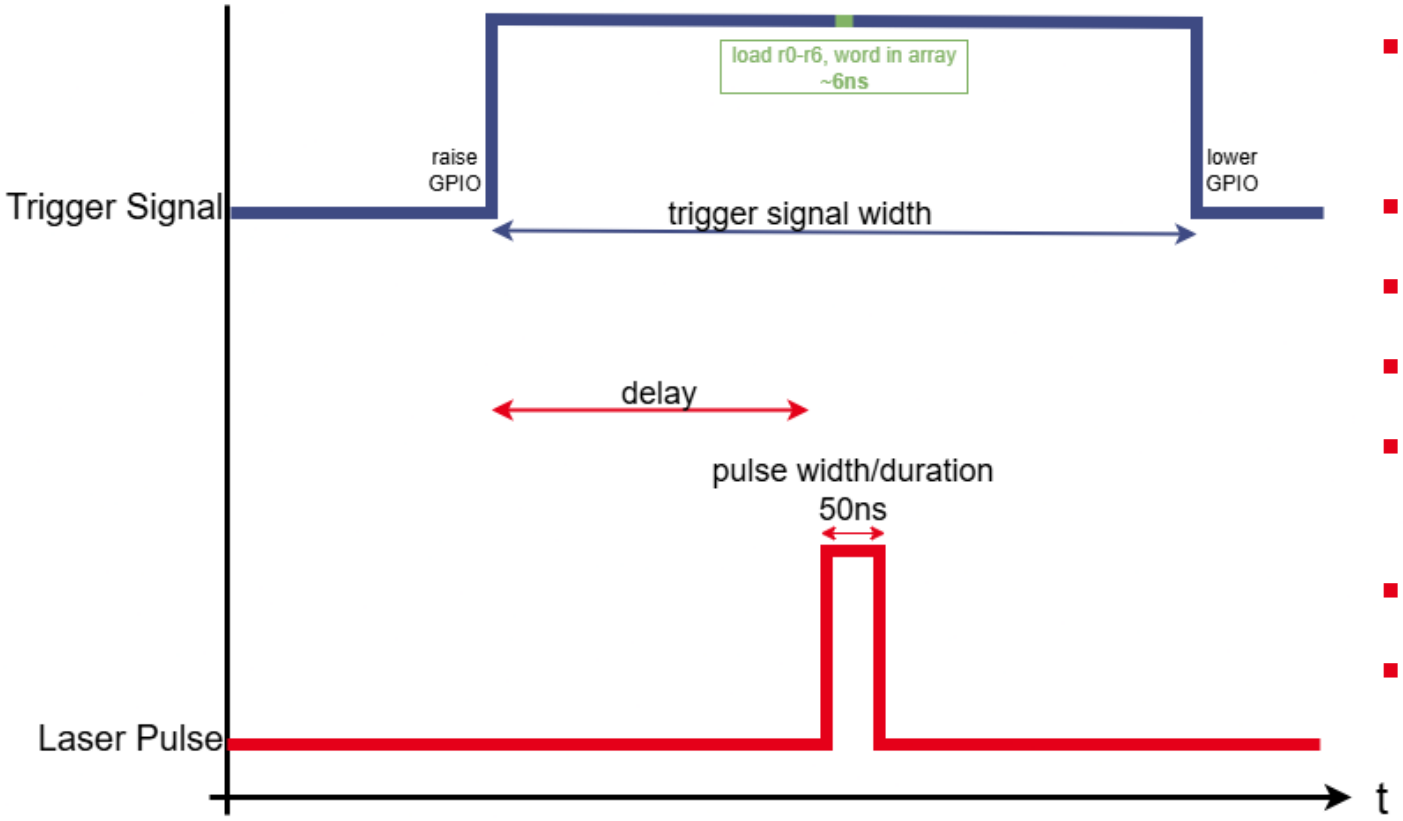


Cache characterisation/reverse

- PEM + IR:
  - Fill the cache: 256 lines × 8 words
  - 8 blocks containing data
    - 4 blocks for even offset words
    - 4 blocks for odd offset words
  - 4 even offset words/line
    × 4 cache ways
    = 16 even offset words
    in 4 blocks
    - ➢ **1 block ⇔ 1 word offset & 4 cache ways?**
    - ➢ **1 block ⇔ 1 cache way & 4 word offsets?**
  - 1 data block:
    - 1 word loaded ⇔ 4 bands
    - 2 buffer zones



LFI cache setup

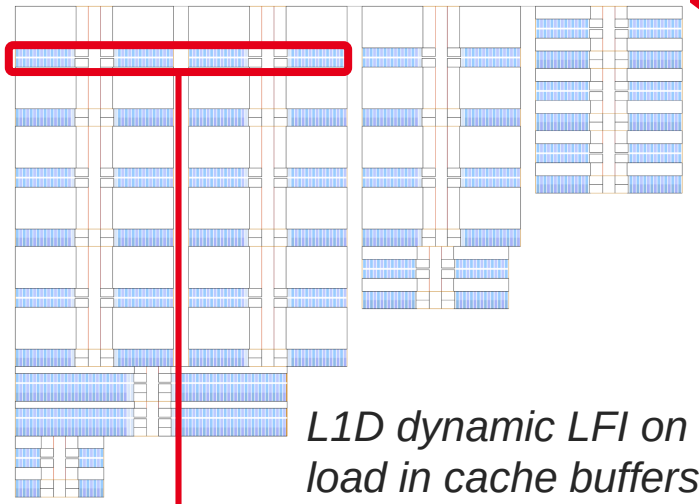


- Cache fully filled with array 256 lines x 8 words
- Loading into 6 registers 1 value from array
  - ~1ns/load
- Pipeline flush before and after to buffer
- Pulse width: 50ns
- Delay: 500 – 530ns
- Synchronisation using (EM) SCA & using performance counters
- Faulting data when loaded to registers
- Fixed line and word offset; cache way attributed randomly; power = 0.6W – 1.2W, pulse width = 50ns)

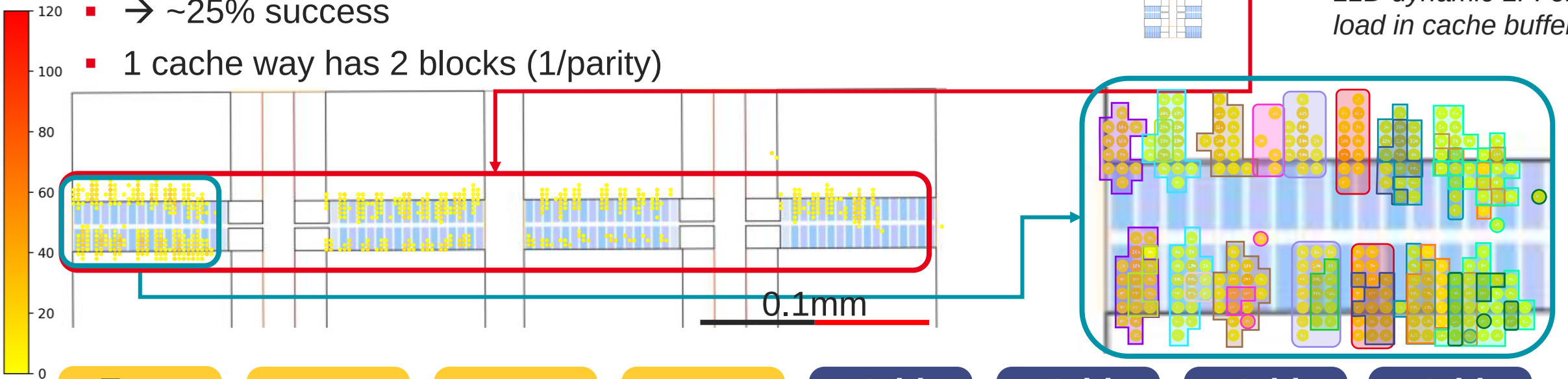
LFI cache analysis

- **1 block ⇔ 1 word offset & 4 cache ways?**
- **1 block ⇔ 1 cache way & 4 word offsets?**

- If 1 block = 1 word: only 1 of 2 blocks is fault sensitive
- If 1 block = 1 cache way: both blocks are fault sensitive → observed!
  - → ~25% success
  - 1 cache way has 2 blocks (1/parity)



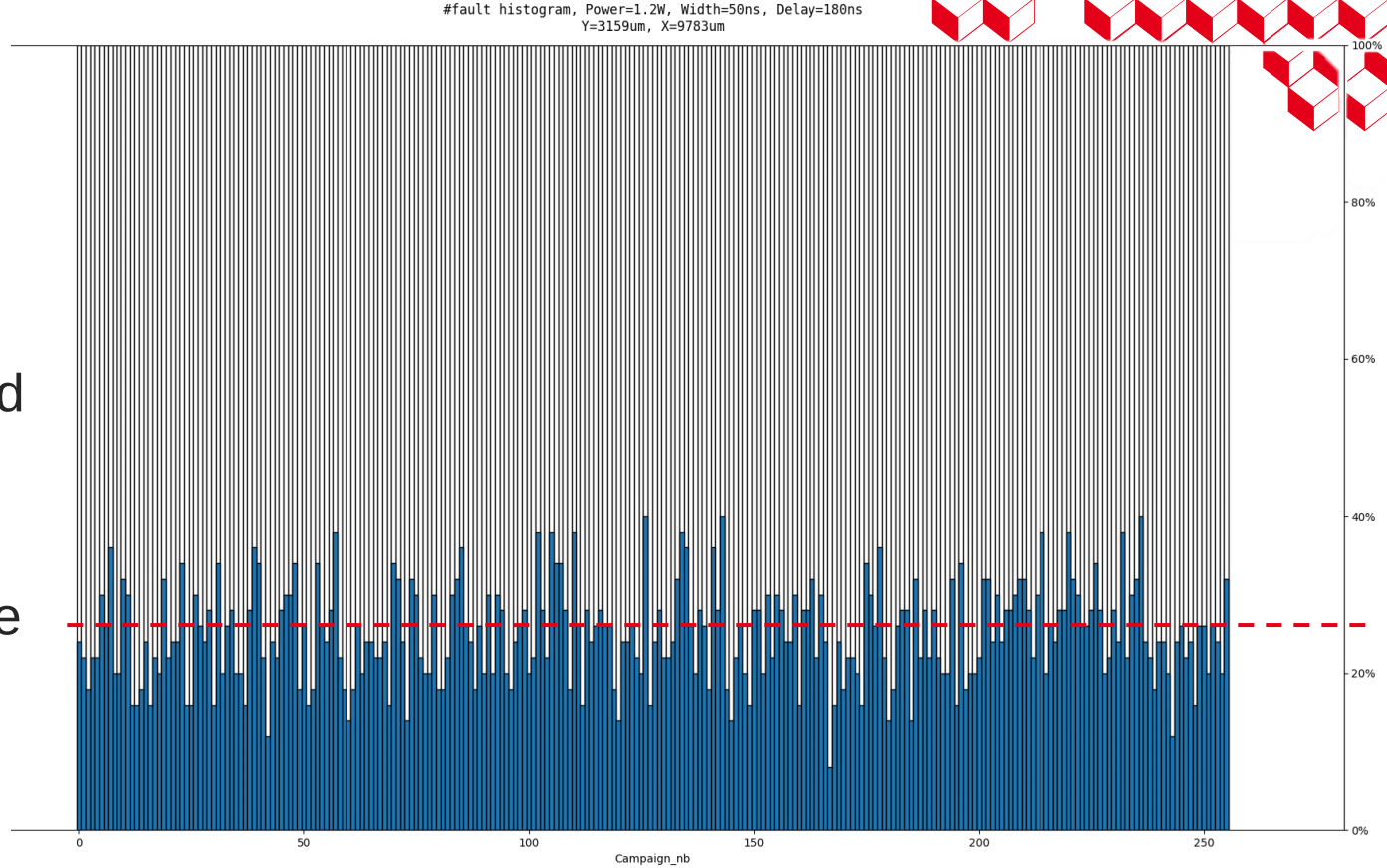
L1D dynamic LFI on load in cache buffers



Even block	Even block	Even block	Even block	Odd block	Odd block	Odd block	Odd block
C.Way #0	C.Way #1	C.Way #2	C.Way #3	C.Way #0	C.Way #1	C.Way #2	C.Way #3

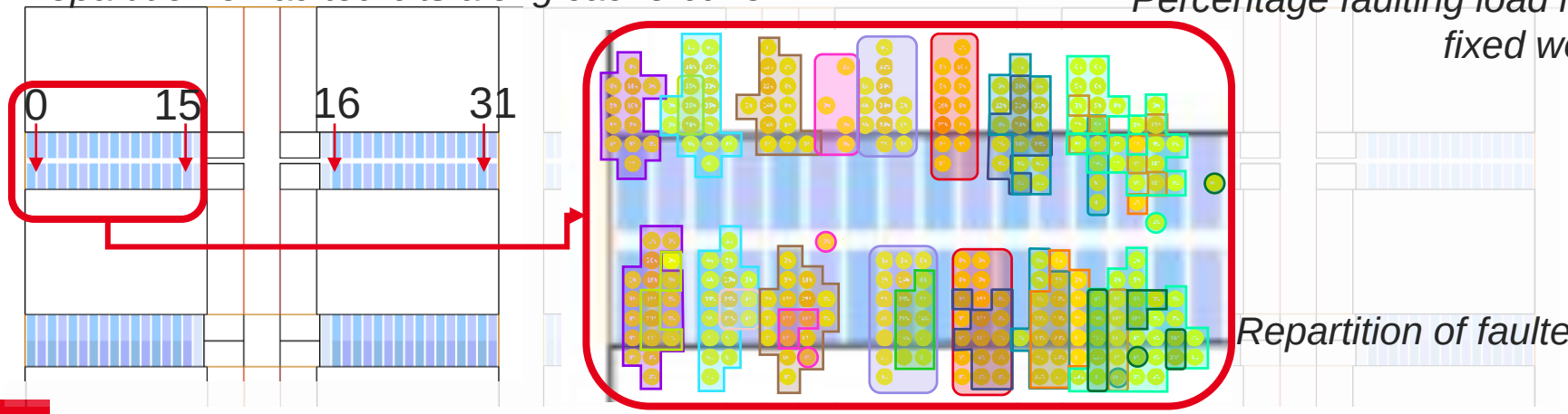
LFI cache results

- Faulting data when loaded to registers across 1 buffer
  - Fixed word offset; cache way attributed randomly; iterate line 0-255
    - Averaging at 25% fault rate
  - Fixed word offset & line; random cache way
    - Mainly mono-bit reset 0→31 across buffer



Percentage faulting load instructions for all 256 lines, fixed word offset

Repartition of faulted bits along cache buffer



Repartition of faulted bits along cache buffer (zoom)

4. Conclusion

a. Conclusion

b. Prospects

Conclusion

Methodology:

- ✓ PEM quickly reduces area to explore & finds points of interest

CPU LFI:

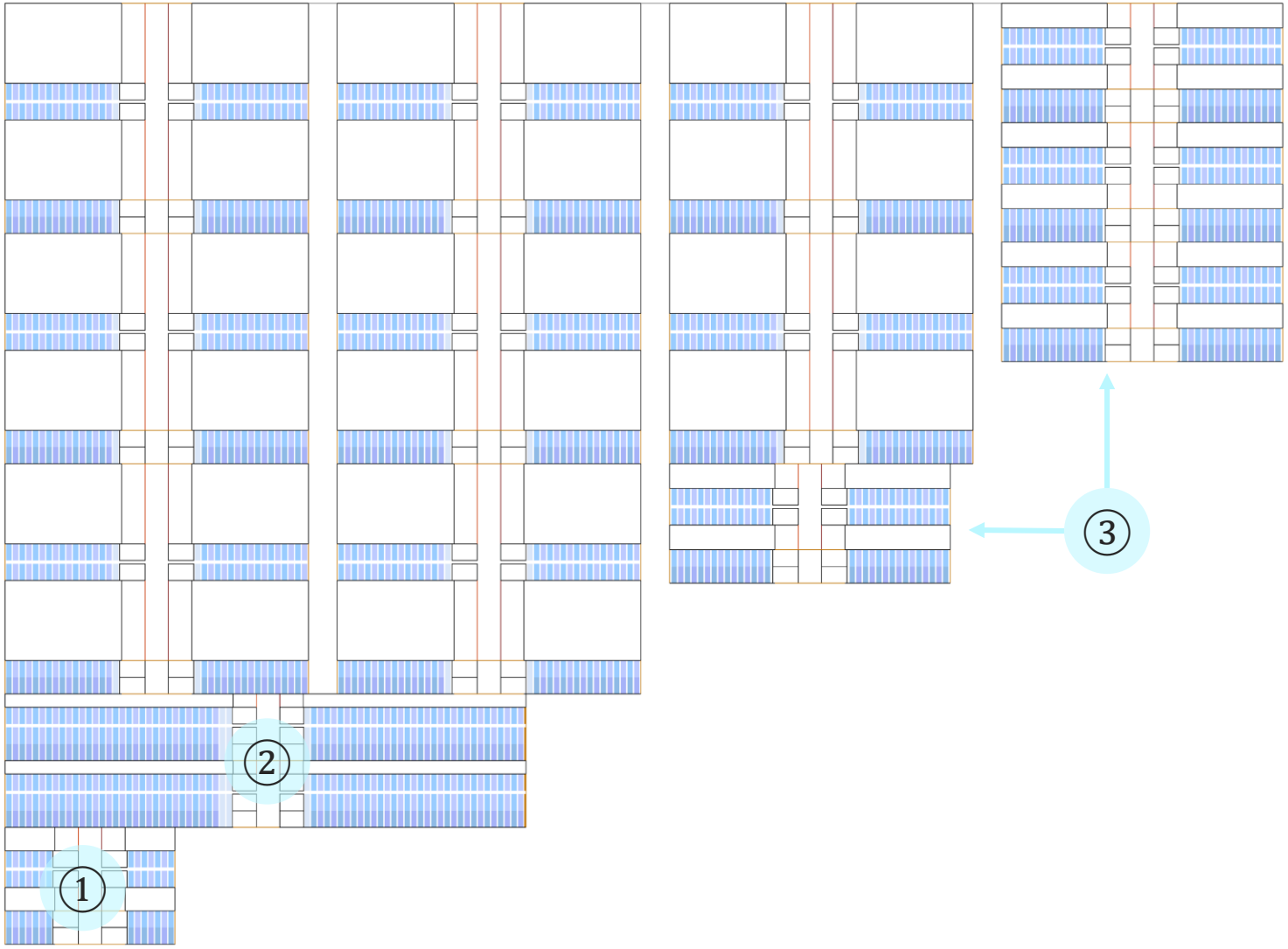
- ✓ Multiple faults on CPU (dynamic) following monobit flip on 32-bit instruction model (e.g., ADD bit 18)
- ✓ High repeatability (e.g., 100% for ADD bit 18)

Cache LFI:

- ✓ Dynamic monobit faults on loading data from cache to registers
- ✓ Repeatability: average 25% due to 1-in-4 chance of correct position

Prospects

- Further explore CPU faults repeatability (different faults, different areas in CPU)
- Attack different zones of cache
- Find & attack area where all data converges (100% repeatability)
- Characterise and fault an Aarch64 SoC (28nm FD-SOI)
- Potential in using LFI to aid or imitate microarchitectural attacks



Bibliography

[1] S.P. Skorobogatov et al., “Optical fault induction attacks,” *Cryptographic Hardware and Embedded Systems*, *CHES*, 2002.

[2] R.S. Lima, “Reverse-Engineering and Data Extraction from SRAM using Photon Emission Analysis,” *IEEE PAINE*, 2024.

[3] H. Perrin et al., “Betrayed by Light: How Photon Emission Microscopy Empowers Register Bit-Level Laser Attacks on Microcontrollers,” *IEEE HOST*, 2025.

[4] T. Trouchkine et al., “Soc physical security evaluation,” Ph.D. dissertation, Université Grenoble Alpes, 2021.

[5] A. Vasselle et al., “Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version,” in *IEEE Transactions on Computers*, 2020.

[6] A. Stuffer, “Interactive and non-destructive verification of sram-descrambling with laser”, in *Microelectronics Reliability*, 2004.

[7] S. Chef et al., “Descrambling if embedded SRAM using a laser probe”, *IEEE IPFA*, 2018.

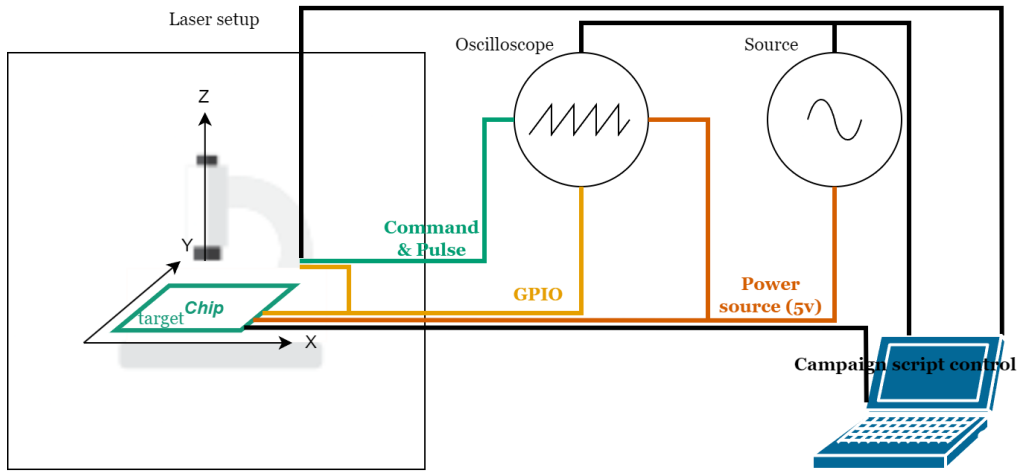
[8] J.-M. Dutertre et al., “Laser Fault Injection at the CMOS 28 nm Technology Node: an Analysis of the Fault Model”, *FDTC*, 2018.

Soline CASAVECCHIA

soline.casavecchia@cea.fr

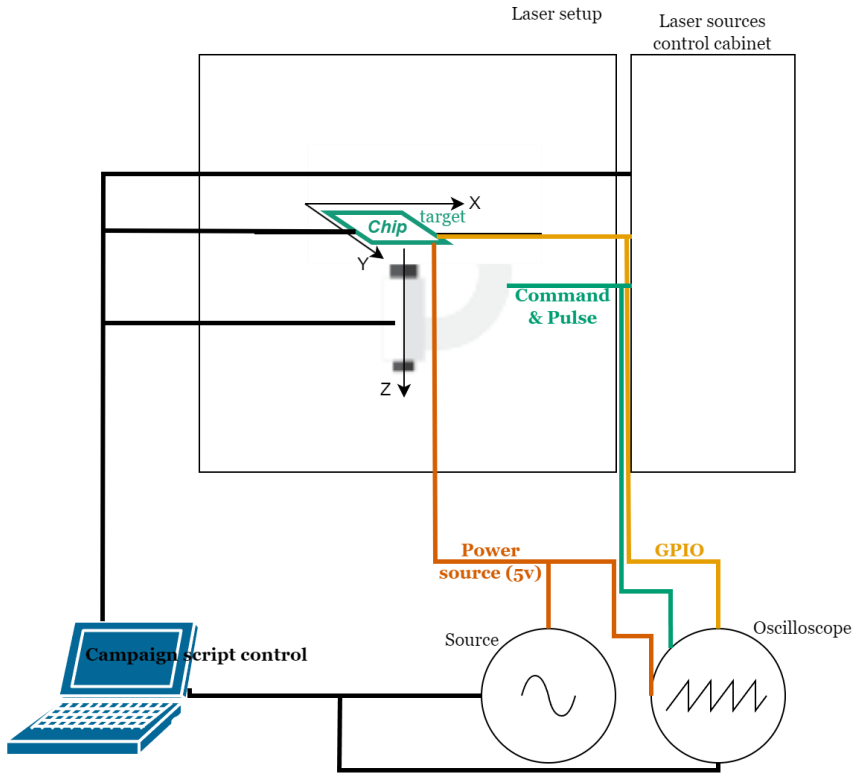
Definition: Fault injection

- Fault Injection Attack: target behaviour modification advantageous to the attacker
- Laser Fault Injection (LFI)
  - more precise than other FI methods (e.g., EM…)
  - Target Si die illuminated via laser pulse → integrated circuit faulted (photocurrents created)
    - Need optical access to die
    - IR wavelength to pass through backside Si
  - Spot diameter (multiple lenses, focalisation) : 20 µm ~ 1 µm



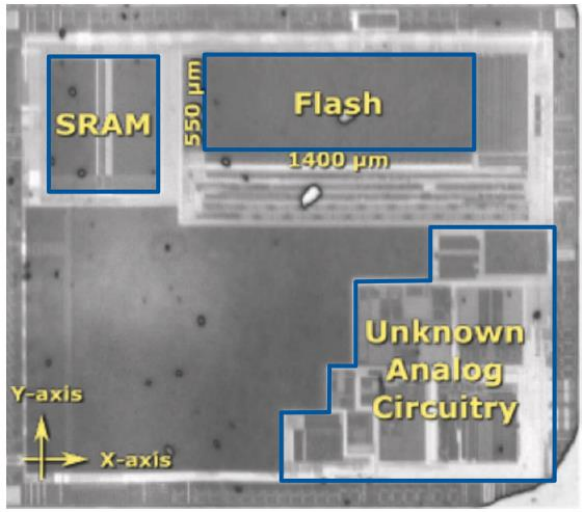
LFI setups

Laser Pulsys & board – zoom target & lenses

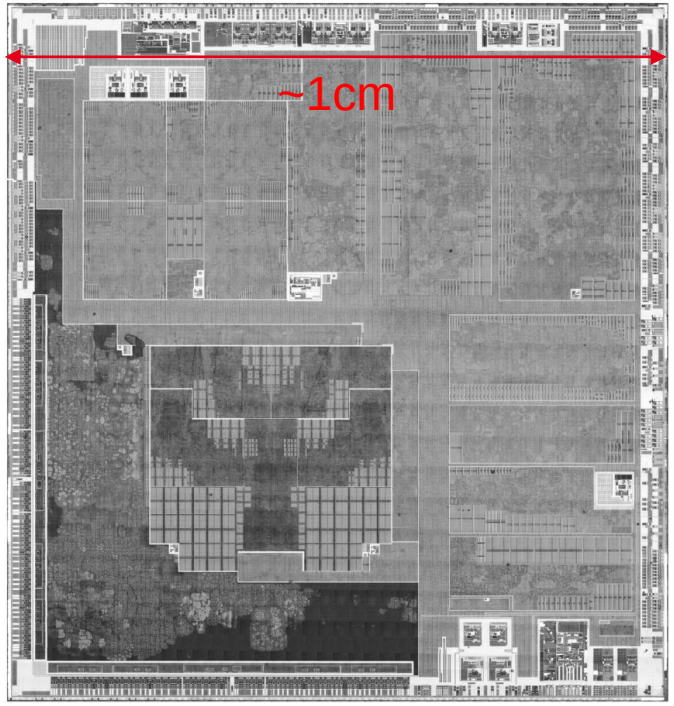


Definition: System-on-Chip

- Integrated Circuit (IC) with all functionalities of a system on a single chip (« all-in-one", e.g. smartphone)
  - Processor cores – e.g. ARM architecture
  - Memory – main memory (DRAM) + cache(s) (SRAM)
  - Interfaces – e.g. USB, Ethernet, etc.
  - Digital signal processors
  - Bus-based communication
  - Other



Backside infrared imagery of a microcontroller [1]



Backside infrared imagery of a quad-core cortex A9 SoC [2]

[9] R. Viera et al. “Tampering with the flash memory of microcontrollers: permanent fault injection via laser illumination during read operations”, 2024.
[5] A. Vasselle et al., “Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version”, 2020.

Results LFI CPU

Algorithm – Code under attack

Input : *n*, *c*

Output : registers list

initialise list

If c = ‘A’ **or** c=‘T’ **Then**

list ← snapshot registers

If ‘A’ **Then** raise GPIO **End If**

While *i* < *n*

i++

End While

```
loop:
    add R5, R5, #1
    cmp  R6, R5
    bgt    loop
```

lower GPIO

list ← snapshot registers

End If

Register	Value
R0	Pointer to GPIO peripheral
R1	GPIO low value
R2	GPIO high value
R3	Pointer to list of 32-bit words
R4	Received char through UART (A or T, variable c)
R5	Increment value i
R6	Max value variable (cst_r, n) = 1000 = 0x3E8
R7	Equal to Stack Pointer (SP) – unused
R8	Attack Mask (reset u. test; set u. attack)
R9	0x00000000 – unused
R10	– unused
R11	0x00000000 – unused
R12	0x00000000 – unused
SP	Stack Pointer
LR	Link Register, = R3
PC	Program Counter

Results LFI CPU

Algorithm – Code under attack

Input : *n*, *c*

Output : registers list

initialise list

If c = ‘A’ **or** c=‘T’ **Then**

list ← snapshot registers

If ‘A’ **Then** raise GPIO **End If**

While *i* < *n*

i++

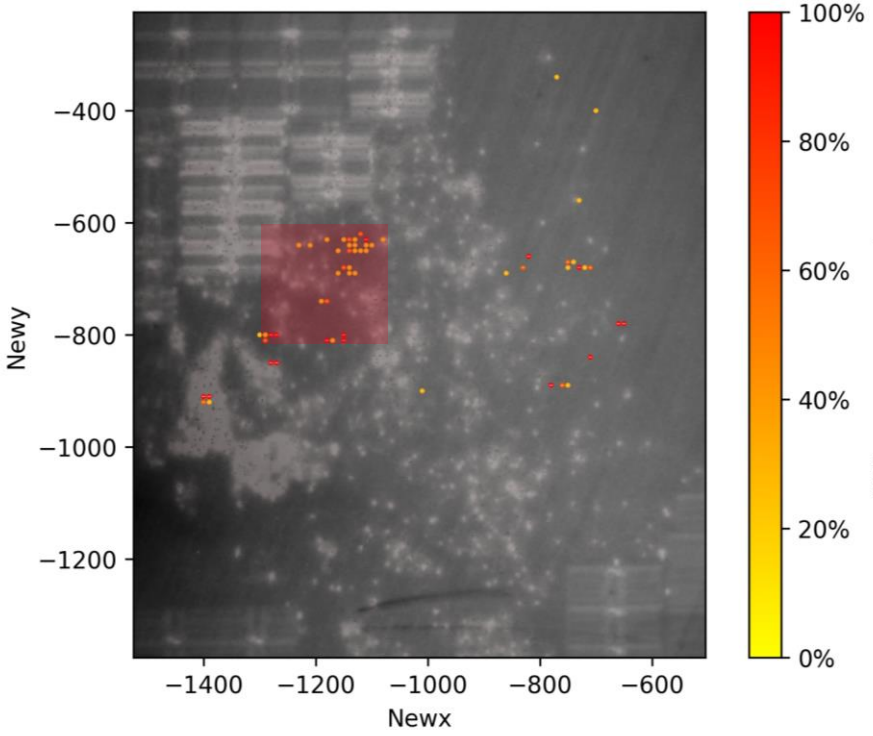
End While

```
loop:
    add R5, R5, #1
    cmp  R6, R5
    bgt   loop
```

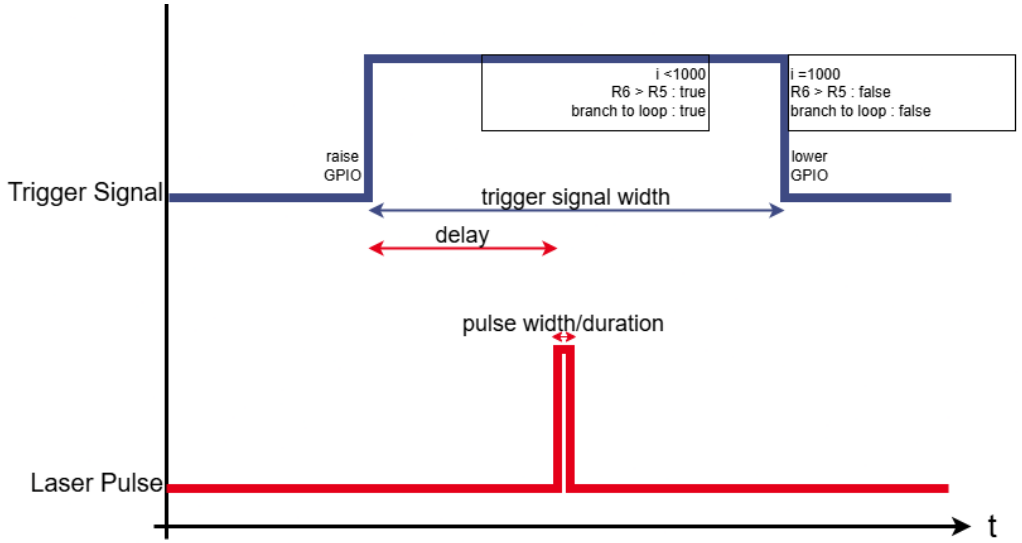
lower GPIO

list ← snapshot registers

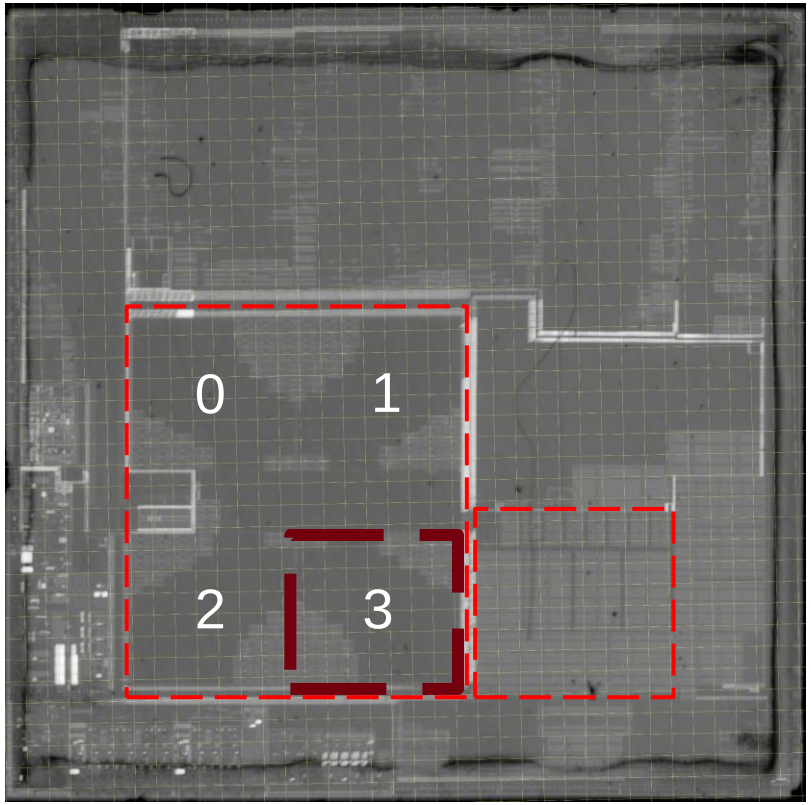
End If



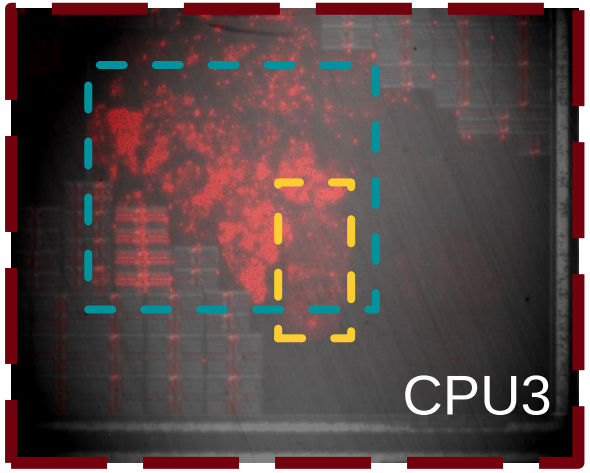
Faults with x05 lens across CPU active area: identifying dense fault area (red area) (power = 0.5W, pulse width = 50ns)



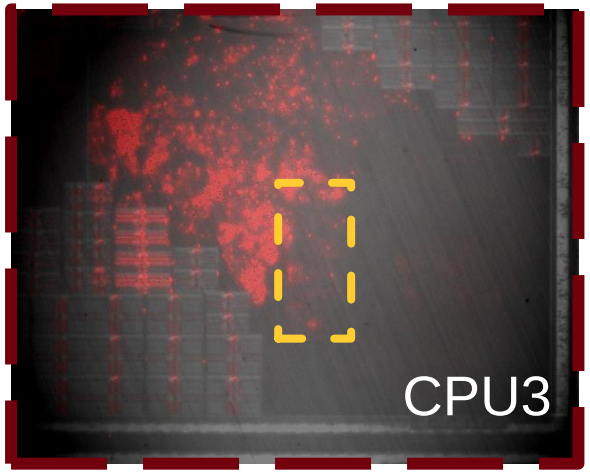
CPU PEM



Backside IR scan highlighting cores and memory structures for PEM



XOR with 0xFFFF FFFF

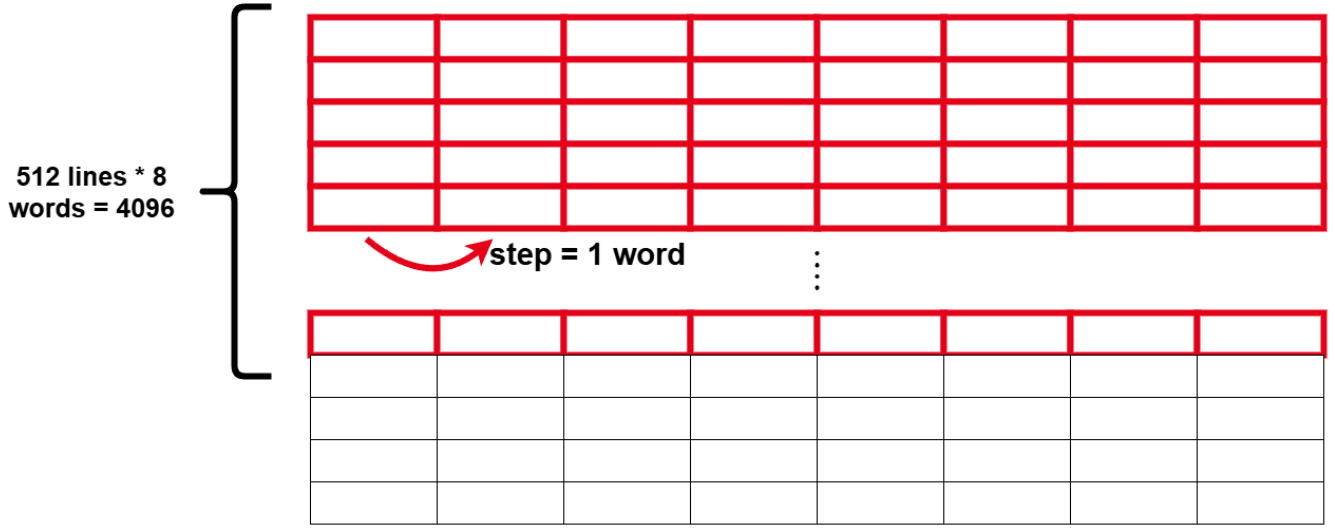


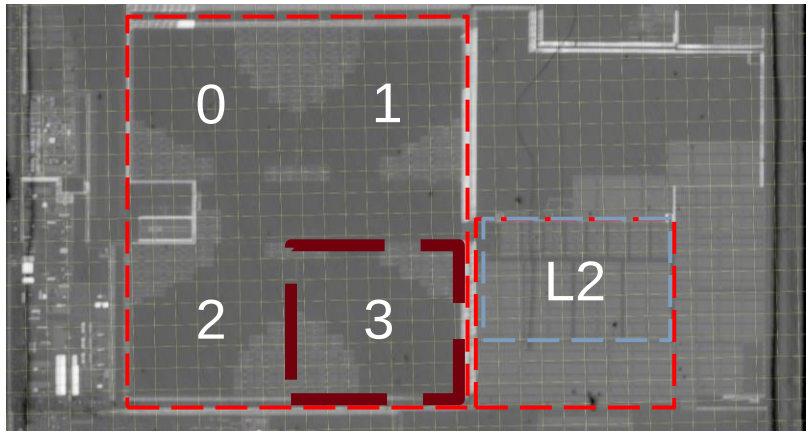
XOR with 0x0000 0000

- 4 CPUs – 1 separating band
- Frequency f_{max} = 996 MHz
- Scripts PEM :
  - Empty loop
  - ADD Rx, Rx, R0 (R0 = 0)
  - XOR Rx, Rx, #val
  - Load in register of array from cache
- Lit up area → target area for LFI

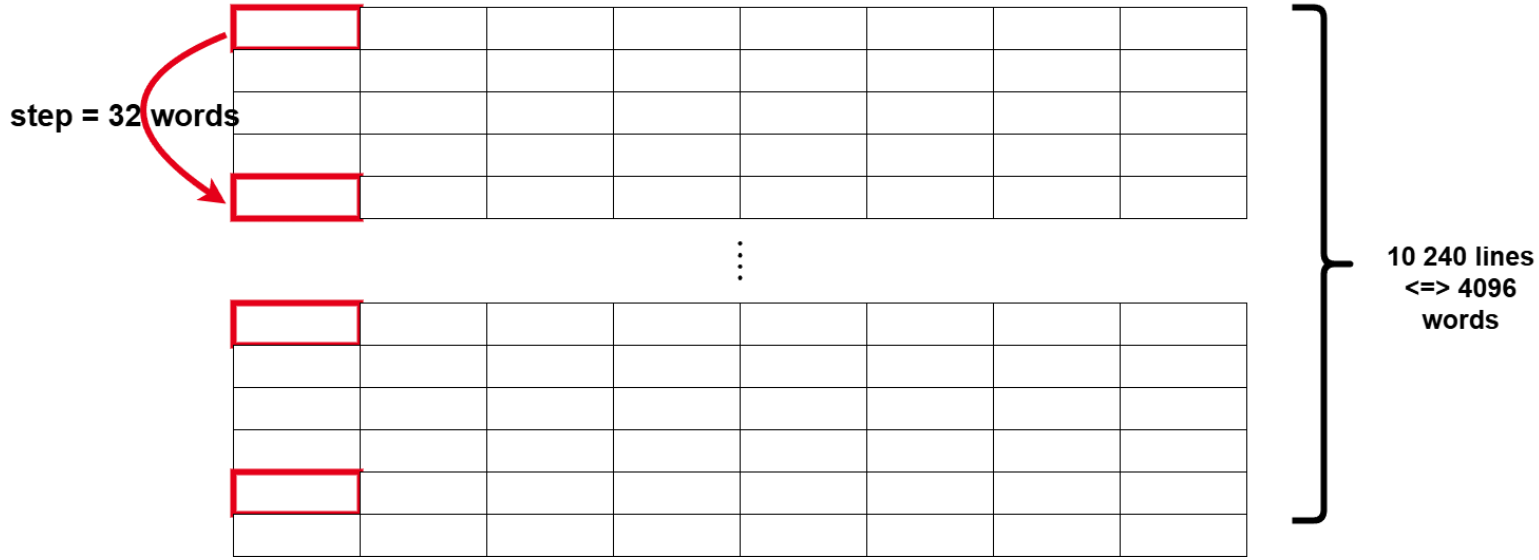
Caches

- 1 cache L1/CPU: L1-I, L1-D – 32 KB
- 1 cache global L2 – 1 MB
- Cache lines : 8 words/line ⇔ 32 bytes/line
- Cache HIT via step parameter (in words) and iteration (*n*)



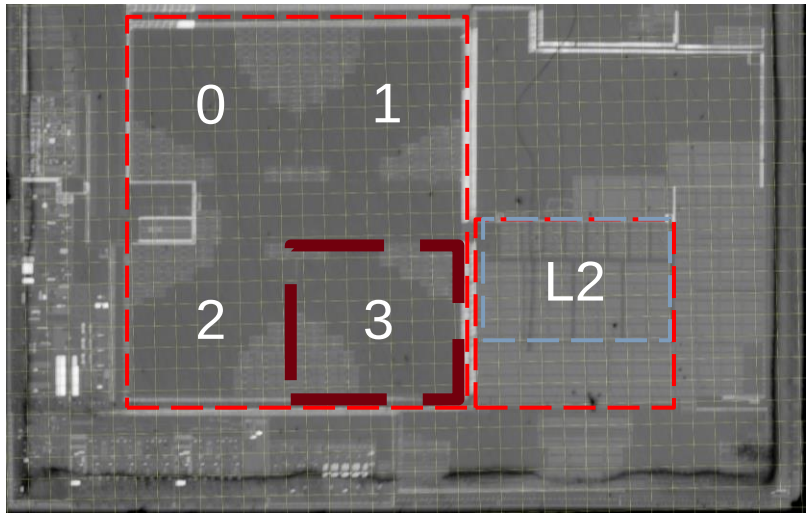


target cores & L2

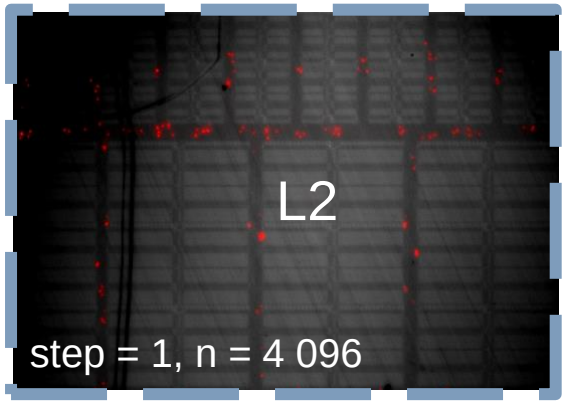


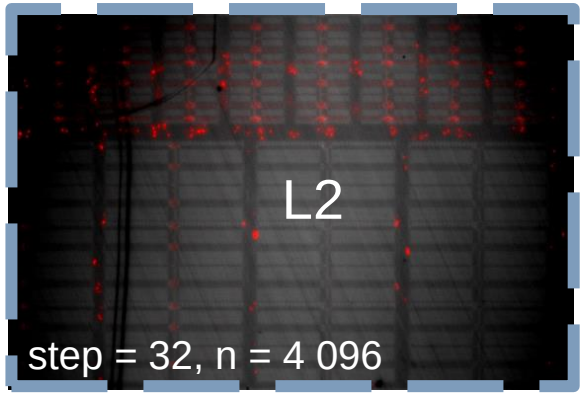
Caches

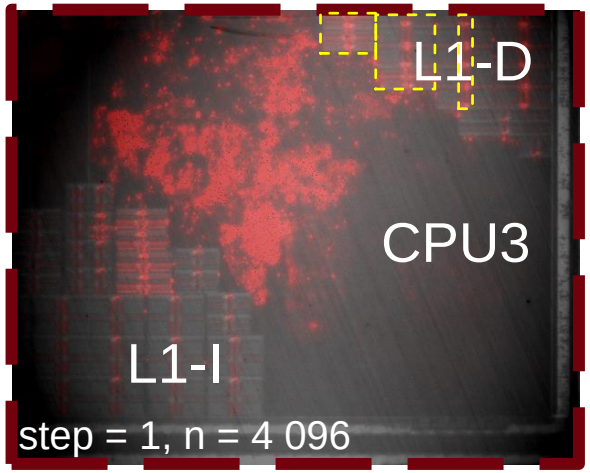
- 1 cache L1/CPU: L1-I, L1-D – 32 KB
- 1 cache global L2 – 1 MB
- Cache lines : 8 words/line ⇔ 32 bytes/line
- Cache HIT via step parameter (in words) and iteration (*n*)

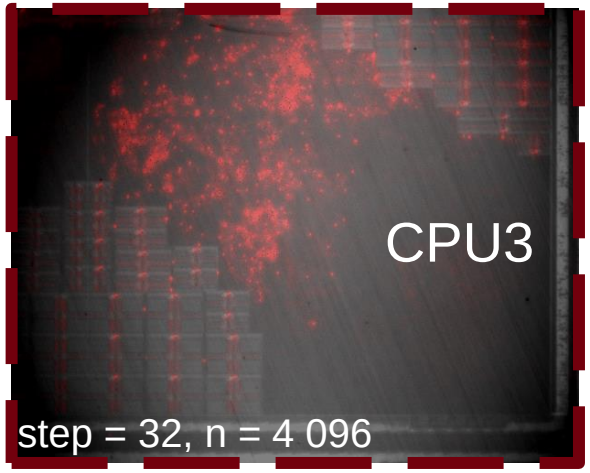


Target cores & L2





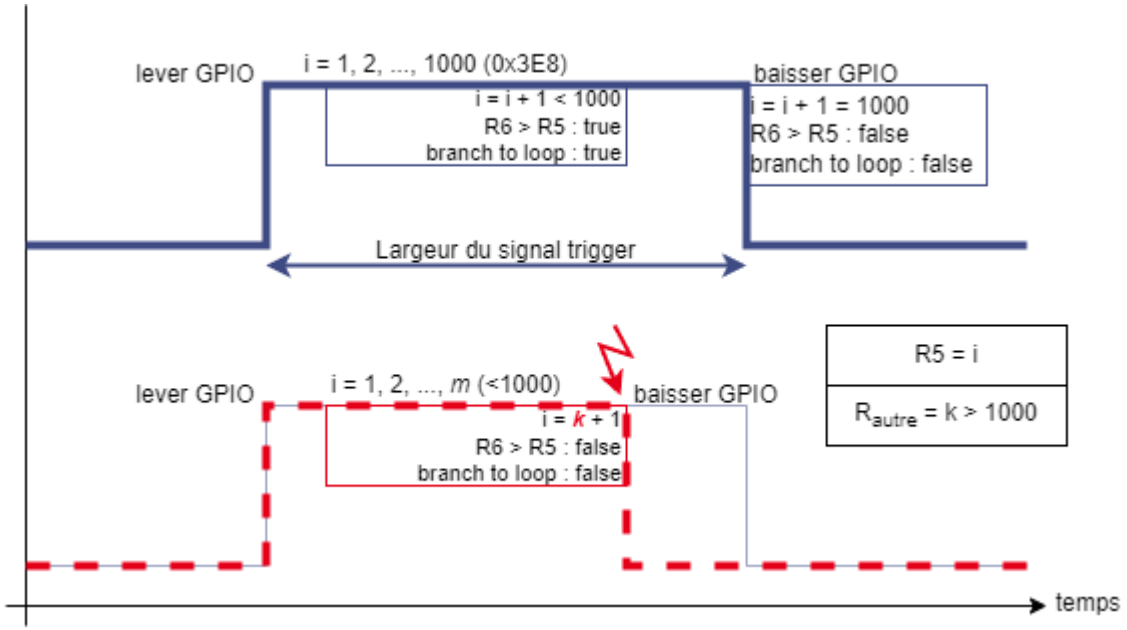




Load Rx ← array (cache)
Upper – *L2*
Lower – *CPU3, L1-I/D*

LFI CPU results analysis

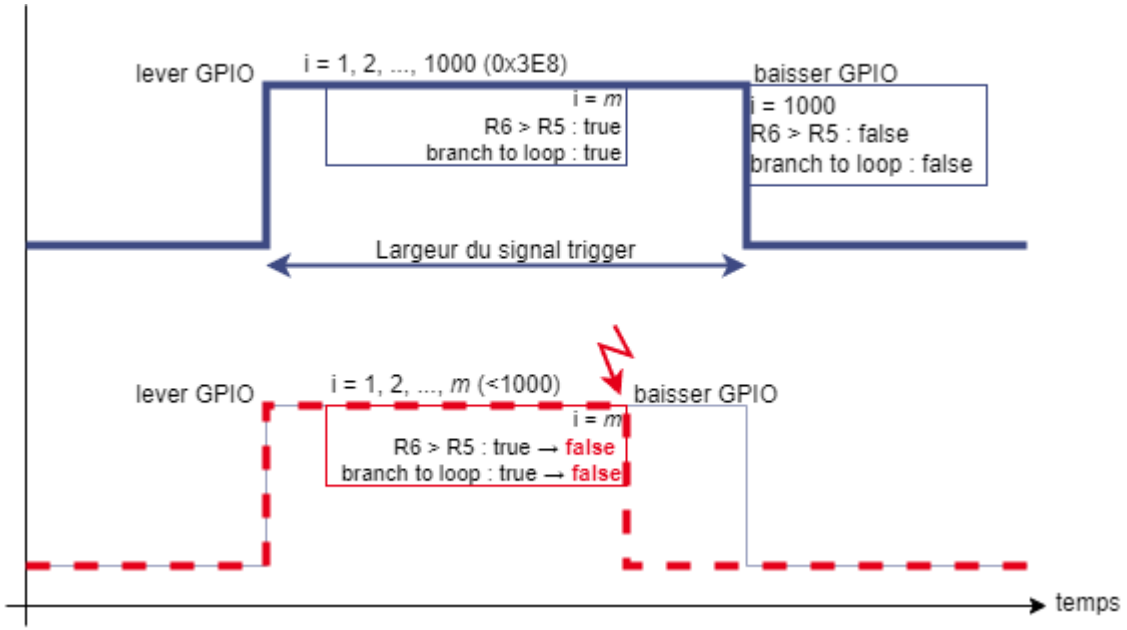
- Observed fault on R5 vs corresponding faulted instruction
  - R5 = R_{other} (> n) : ADD faulted, source register modified (bits [19:16]) : ADD R5, R5, #1 → ADD R5, **R_{other}**, #1; e.g. 0b0101 → 0b0**0**01 ⇔ R5 → R1



Loop exit following ADD fault

LFI CPU results analysis

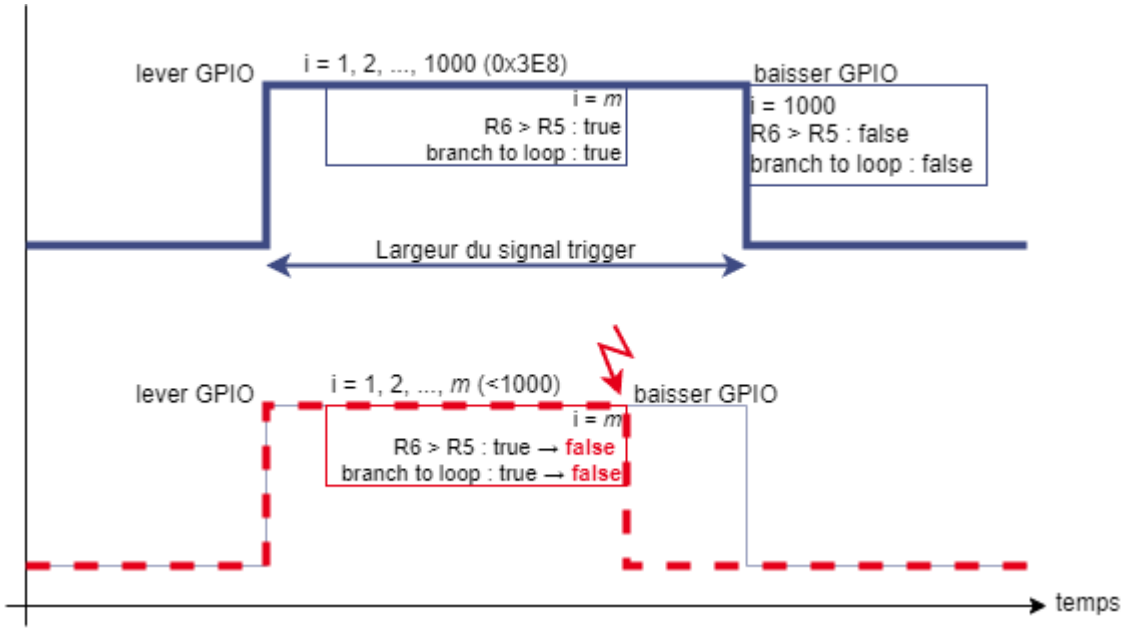
- Observed fault on R5 vs corresponding faulted instruction
  - R5 = R_{other} (> n) : ADD faulted, source register modified (bits [19:16]) : ADD R5, R5, #1 → ADD R5, **R_{other}**, #1; e.g. 0b0101 → 0b0**0**01 ⇔ R5 → R1
  - Partial increment: force loop exit, BGT faulted → trigger signal width reduced; e.g. BGT @loop → BGT **@other**



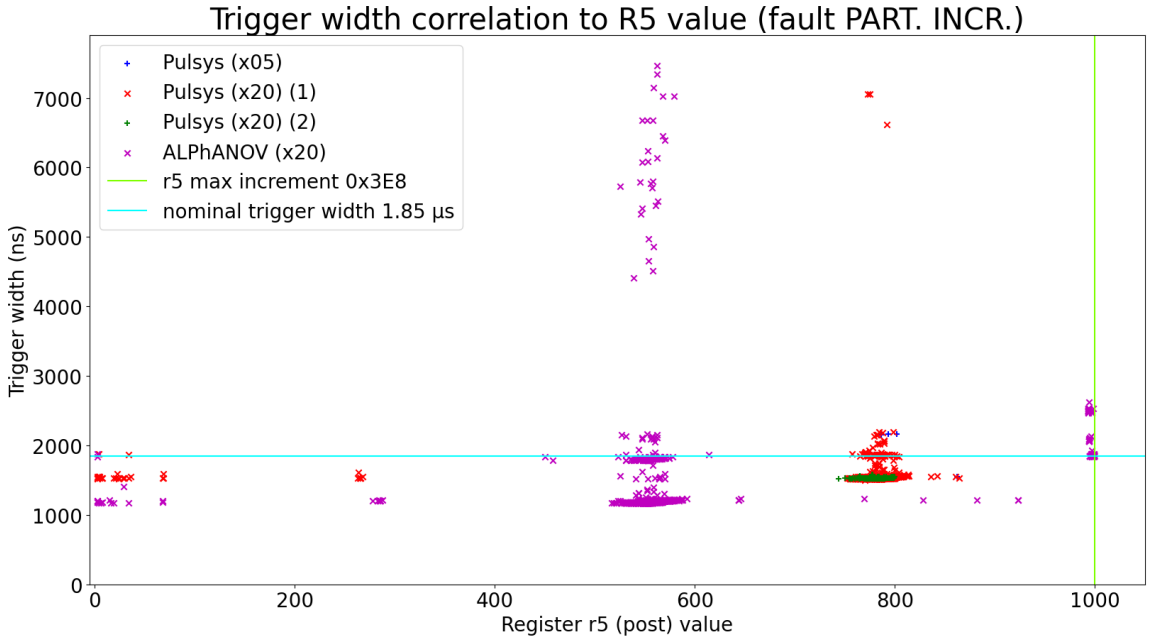
Loop exit following branch fault

LFI CPU results analysis

- Observed fault on R5 vs corresponding faulted instruction
  - R5 = R_{other} (> n) : ADD faulted, source register modified (bits [19:16]) : ADD R5, R5, #1 → ADD R5, **R_{other}**, #1; e.g. 0b0101 → 0b0**0**01 ⇔ R5 → R1
  - Partial increment: force loop exit, e.g. BGT faulted → trigger signal width reduced; e.g. BGT @loop → BGT **@other**
- No fault detected ≠ no fault occurred: e.g., immediate value reset in ADD → loop iteration increase

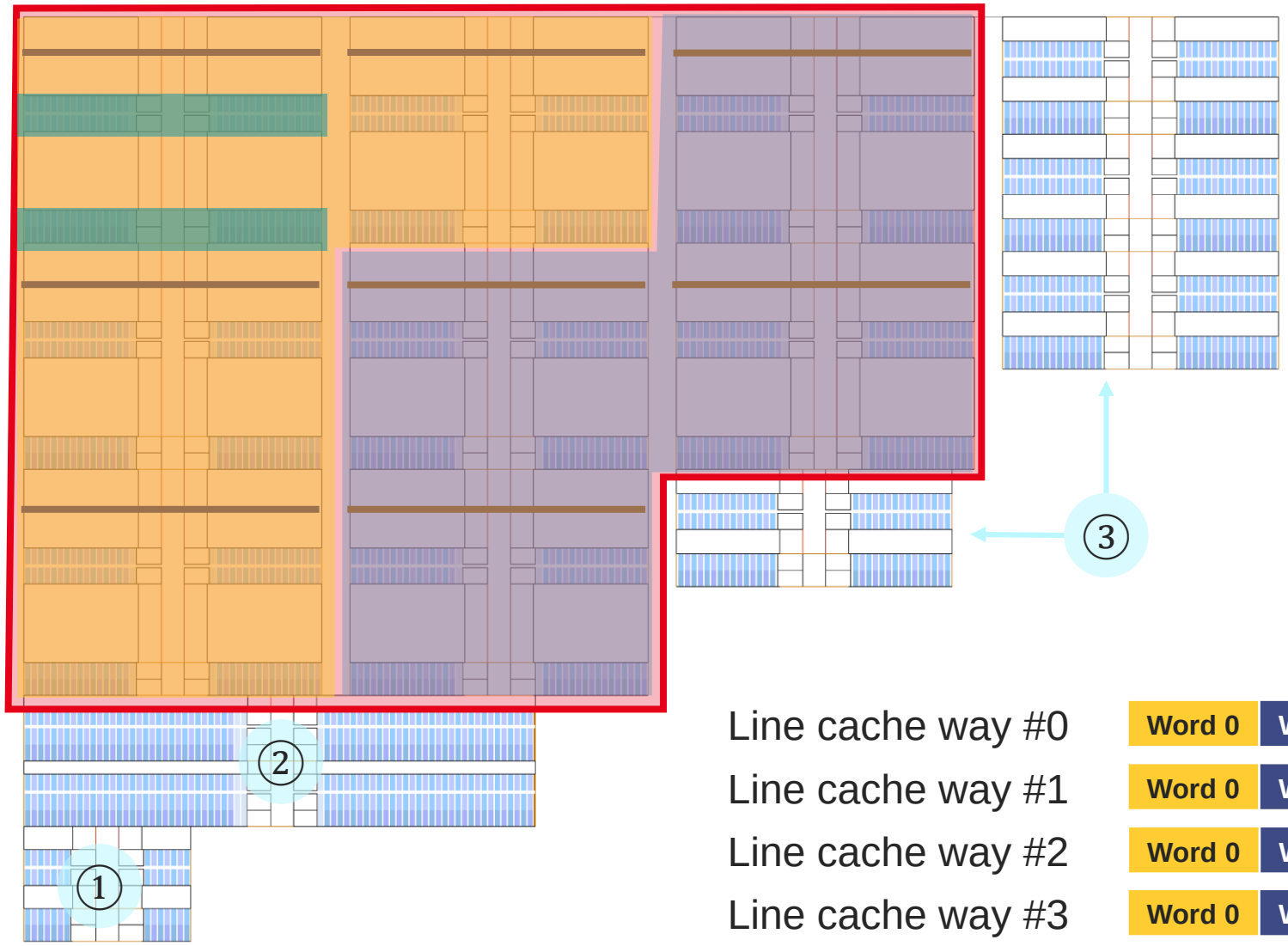


Loop exit following branch fault



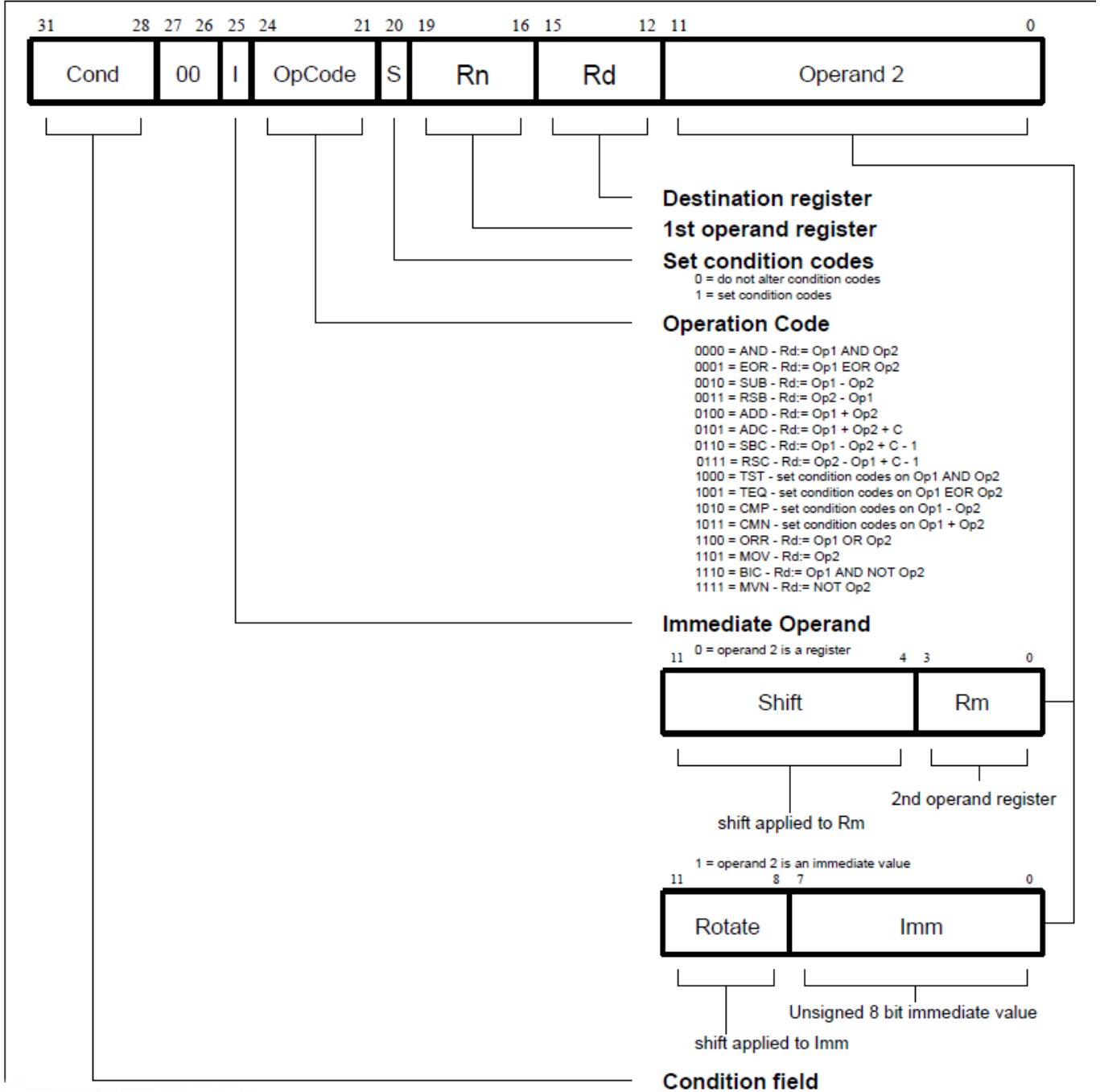
Trigger signal width for $R5_{observed}$ < $R5_{expected}$ (= 1000)

Cache characterisation/reverse



Cache lines for 1 index:

Line cache way #0	Word 0	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7
Line cache way #1	Word 0	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7
Line cache way #2	Word 0	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7
Line cache way #3	Word 0	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7


31 28 27 26 25 24 21 20 19 16 15 12 11 0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
Immediate Operand
0 = operand 2 is a register
11 4 3 0
Shift
Rm
2nd operand register
shift applied to Rm
1 = operand 2 is an immediate value
11 8 7 0
Rotate
Imm
Unsigned 8 bit immediate value
shift applied to Imm
Condition field

LFI CPU results analysis

- With LFI emulation following a mono-bit flip on 1 instruction model, possible effects are:

ADD	CMP	BGT
Increment change → $r5_e$ < max: reduced /increased cycles		Branch address change: early loop end, error (illegal destination address), increased cycles (branching back higher)
Increment change → $r5_e$ > max: early loop exit with r5>r6		
Destination register change: increased cycles/ potential crash (r7)	Operand 2 change/left shift: r6 < $op2_e$ → early loop end	
Source register change: increased/decreased cycle & early loop exit if $r5_e$>r6	Operand 1 change: $op1_e$ < r5 → early loop end	
Opcode modification: different instruction	Opcode modification: different instruction (possibly unauthorised)	Opcode modification: different instruction (possibly illegal)
R/I modification: instruction change (#1 → r1)	R/I modification: instruction change (r1 → #5)	
Condition change: error/ increased cycles by 1	Condition change: error/ increased/ decreased cycles	Condition change: branch under different condition (e.g. bgt → ble)

Bibliography

[1] S.P. Skorobogatov et al., “Optical fault induction attacks,” *Cryptographic Hardware and Embedded Systems*, *CHES*, 2002.

[2] R.S. Lima, “Reverse-Engineering and Data Extraction from SRAM using Photon Emission Analysis,” *IEEE PAINE*, 2024.

[3] H. Perrin et al., “Betrayed by Light: How Photon Emission Microscopy Empowers Register Bit-Level Laser Attacks on Microcontrollers,” *IEEE HOST*, 2025.

[4] T. Trouchkine et al., “Soc physical security evaluation,” Ph.D. dissertation, Université Grenoble Alpes, 2021.

[5] A. Vasselle et al., “Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version,” in *IEEE Transactions on Computers*, 2020.

[6] A. Stuffer, “Interactive and non-destructive verification of sram-descrambling with laser”, in *Microelectronics Reliability*, 2004.

[7] S. Chef et al., “Descrambling if embedded SRAM using a laser probe”, *IEEE IPFA*, 2018.

[8] J.-M. Dutertre et al., “Laser Fault Injection at the CMOS 28 nm Technology Node: an Analysis of the Fault Model”, *FDTC*, 2018.

[9] R. Viera et al. “Tampering with the flash memory of microcontrollers: permanent fault injection via laser illumination during read operations”, *Journal of Cryptographic Engineering*, 2024.