Electromagnetic Fault Injection on System-on-Chip and Exploitation

Clément Fanjas

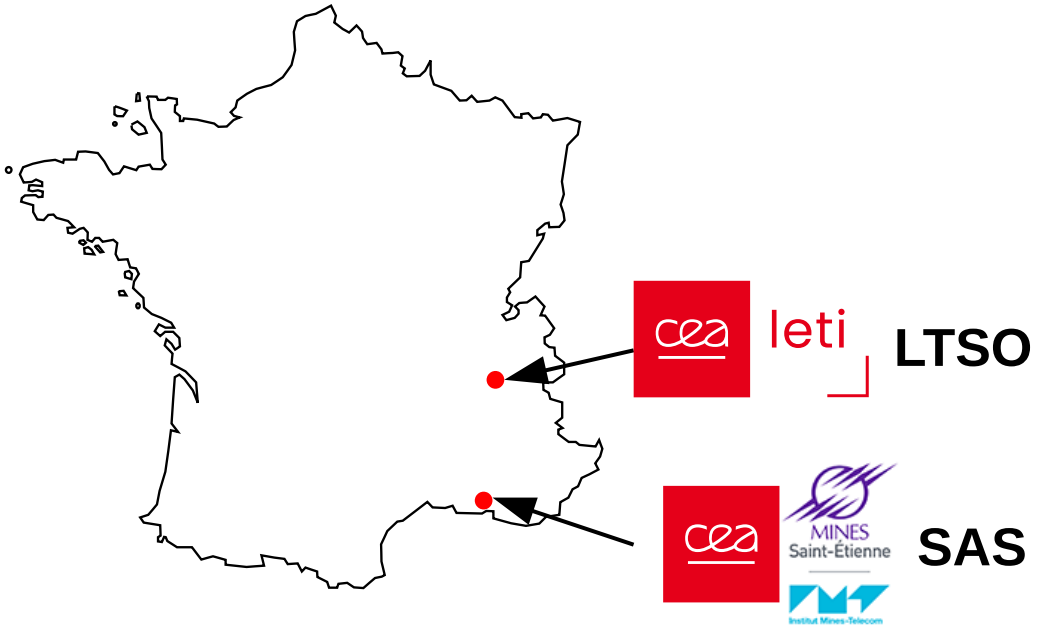
clement.fanjas@cea.fr

About me

Clément Fanjas : PHD student
Thesis supervisors : Jessy Clédière, Simon Pontié, Driss Aboulkassimi

Subject :
Hardware vulnerability exploitation for FORENSIC use-cases on mobile devices

Affiliation:
- Systèmes et Architectures Sécurisées (SAS)
  ⇒ Common research team CEA/EMSE (Gardanne)
- Laboratoire de Test de Sécurité et Outils (LTSO)
  ⇒ CEA LETI (Grenoble)



Context : Forensic Analysis

Law Enforcement Agencies (LEA) :

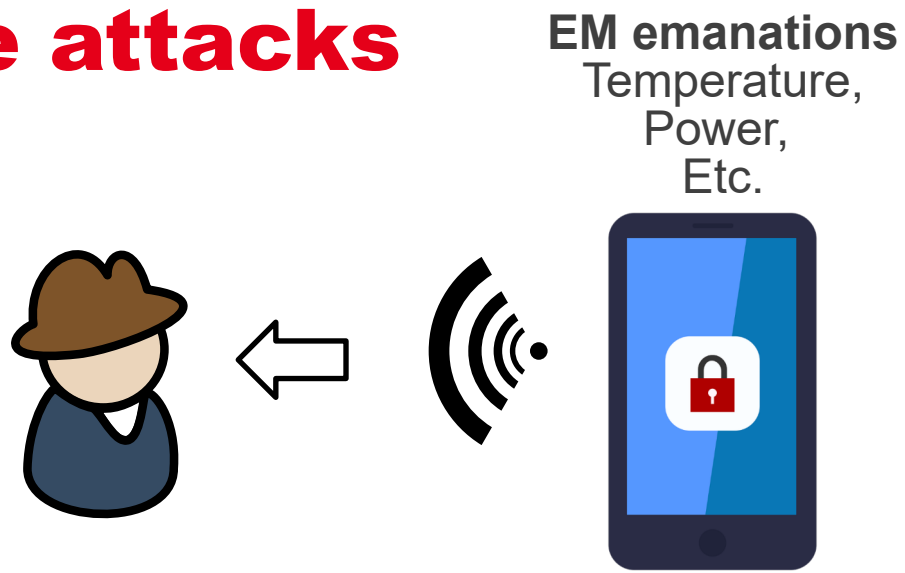
⇒ **Need to extract data from mobile devices such as Smartphones**

Hardware attacks are interesting to explore in this context

Context : Hardware attacks

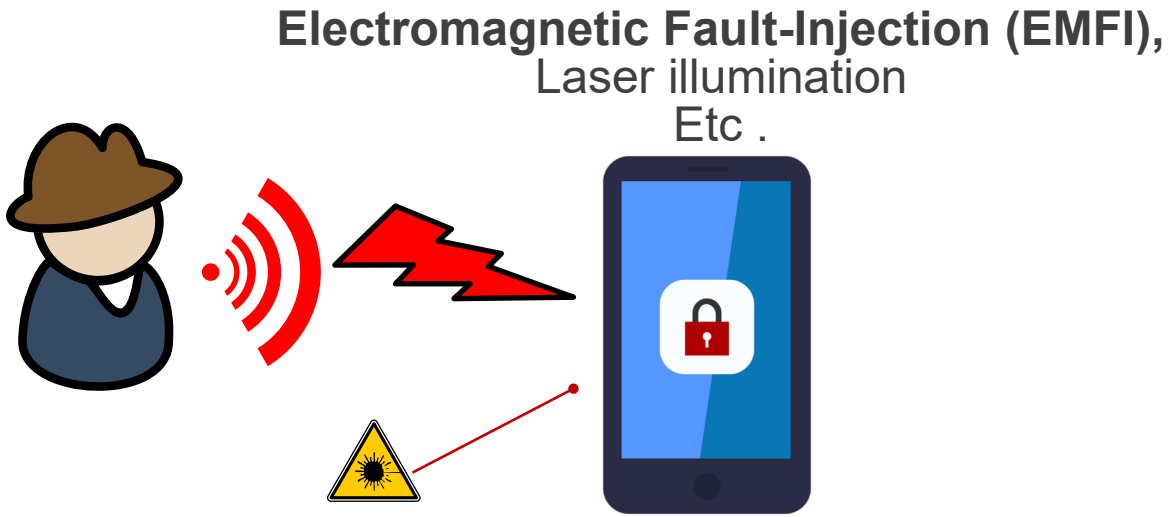
Side-Channel Analysis:

- Extract informations that could leak through a Side-Channel



Fault-Injection:

- Disrupt a target

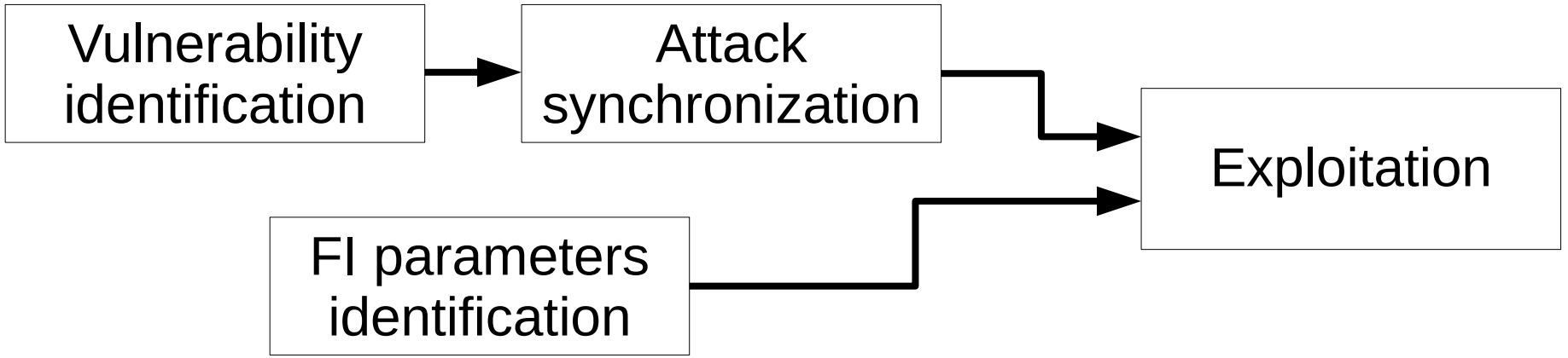


Context : Hardware attacks

Issues posed of hardware attacks on System-on-Chip

- **Attack synchronization**
- **Vulnerability identification**
- **Target Packaging (Package-on-Package)**
- Fault Injection parameters identification

Fault Injection attack



Context : Electromagnetic Fault Injection (EMFI)

How do we inject fault using EMFI:

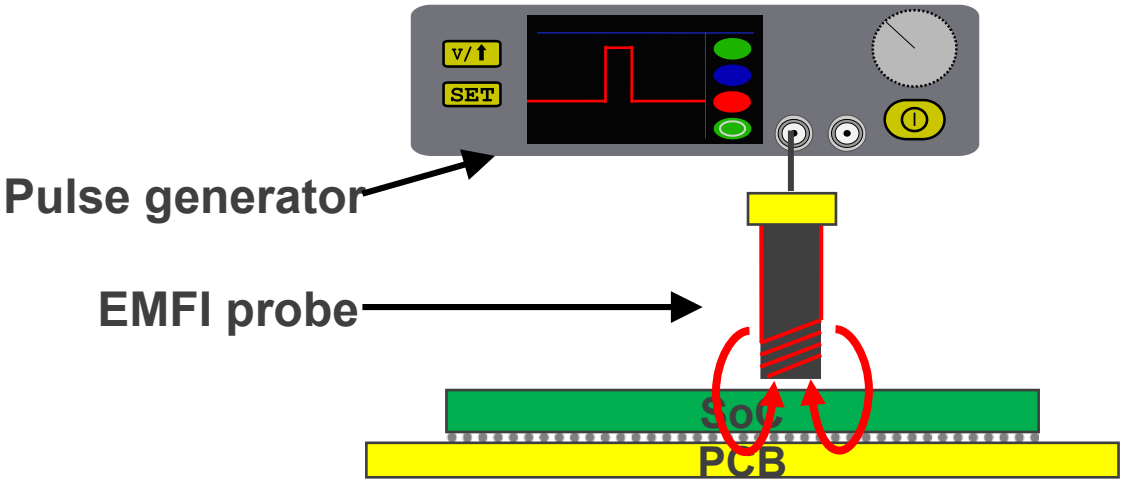
- Sending a voltage pulse into an active probe located over the targeted chip.

- Depending on the probe position over the chip, an EM coupling is created between the target and the probe.

- This coupling induces a transient voltage inside the chip which can corrupt the normal operation.

We need to find the good parameters:

- **Probe position**
- **Pulse parameters**: amplitude and width



Outline

I. Attack synchronization
   1. Synchronization issue
   2. Proposed solution
   3. Proof of concept
      ⇒ Synchronizing EMFI to bypass the Android Linux Kernel authentication

II. Vulnerability identification
   1. High level analysis
   2. Side-Channel Analysis
   3. Proof of concept
      ⇒ EMFI to bypass the SSBL authentication by the FSBL

III. FI on Package-on-Package SoC
   1. Software setup: Code under attack and Fault Model
   2. Fault Injections methods: EMFI / EM-induced glitches / Voltage glitches
   3. Faults obtained with each method
   4. Faults comparison

IV. Conclusion

Outline

I. Attack synchronization

1. **Synchronization issue**
2. **Proposed solution**
3. **Proof of concept**

   ⇒ **Synchronizing EMFI to bypass the Android Linux Kernel authentication**

II. Vulnerability identification

1. High level analysis
2. Side-Channel Analysis
3. Proof of concept

   ⇒ EMFI to bypass the SSBL authentication by the FSBL

III. FI on Package-on-Package SoC

1. Software setup: Code under attack and Fault Model
2. Fault Injections methods: EMFI / EM-induced glitches / Voltage glitches
3. Faults obtained with each method
4. Faults comparison

IV. Conclusion

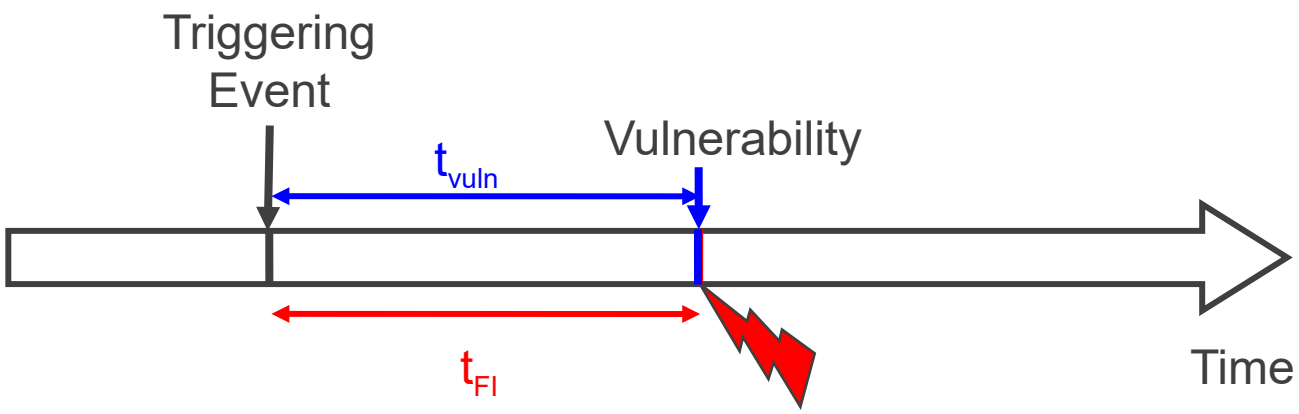
Attack synchronization

A triggering event is used as temporal reference to synchronize the injection.

t_{vuln} = delay between the triggering event and the vulnerability.

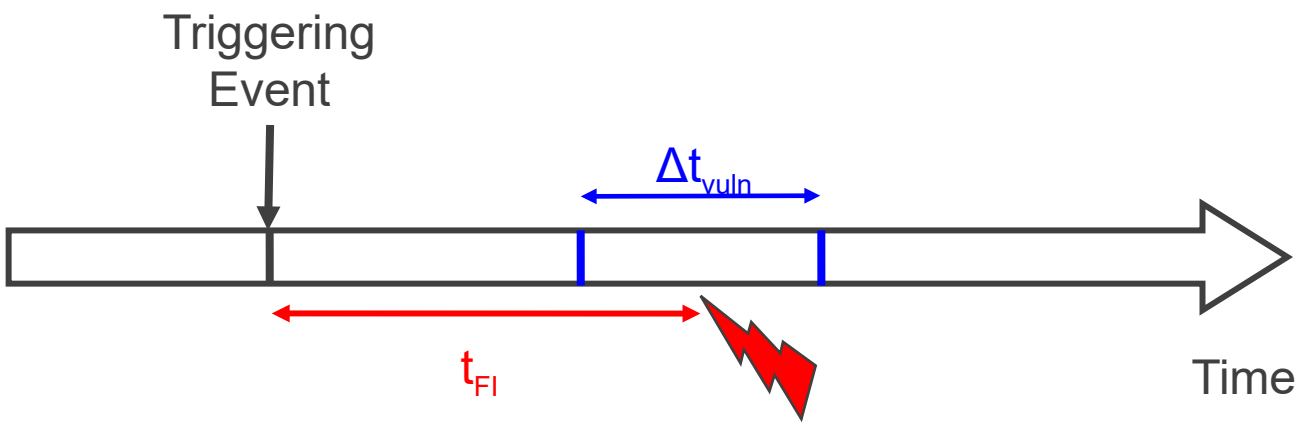
t_{FI} = delay between the triggering event and the attack.

The attack is successful when $t_{vuln} = t_{FI}$



Attack synchronization

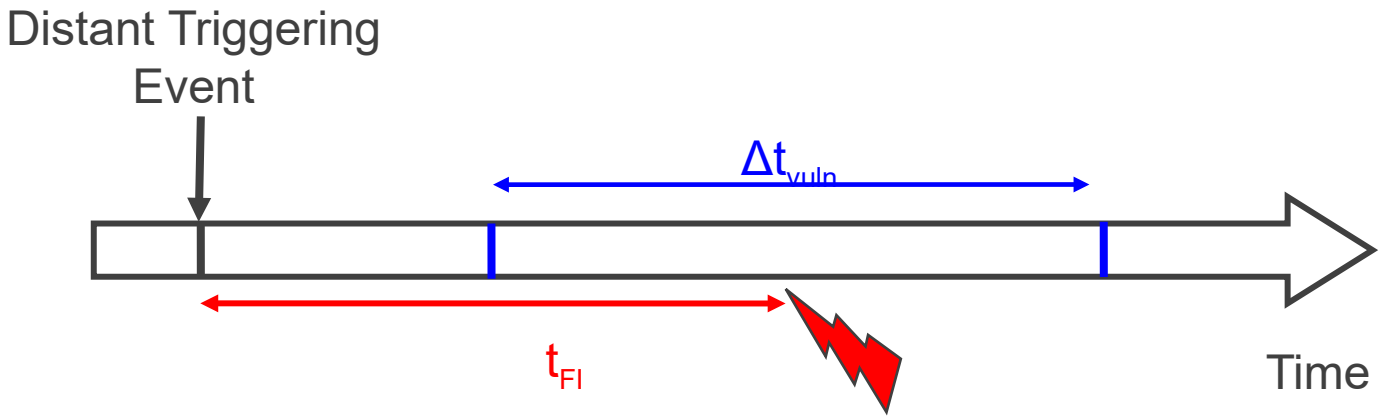
We set t_{FI}, but t_{vuln} is confined in a temporal window Δt_{vuln} also called jitter.

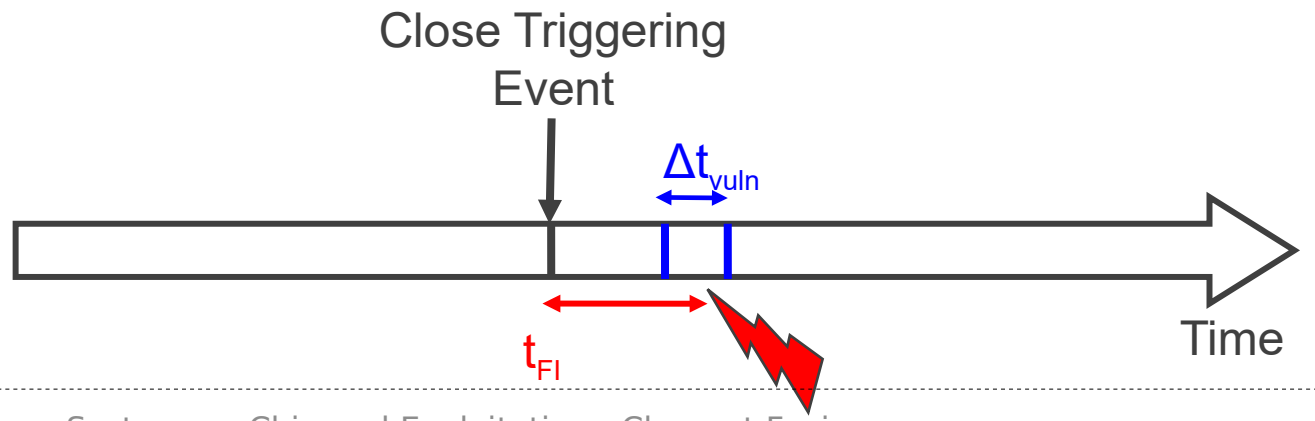


Attack synchronization

To maximize the success rate we need to reduce Δt_{vuln}.

The best way to do it is to get the triggering event as close as possible to the vulnerability.



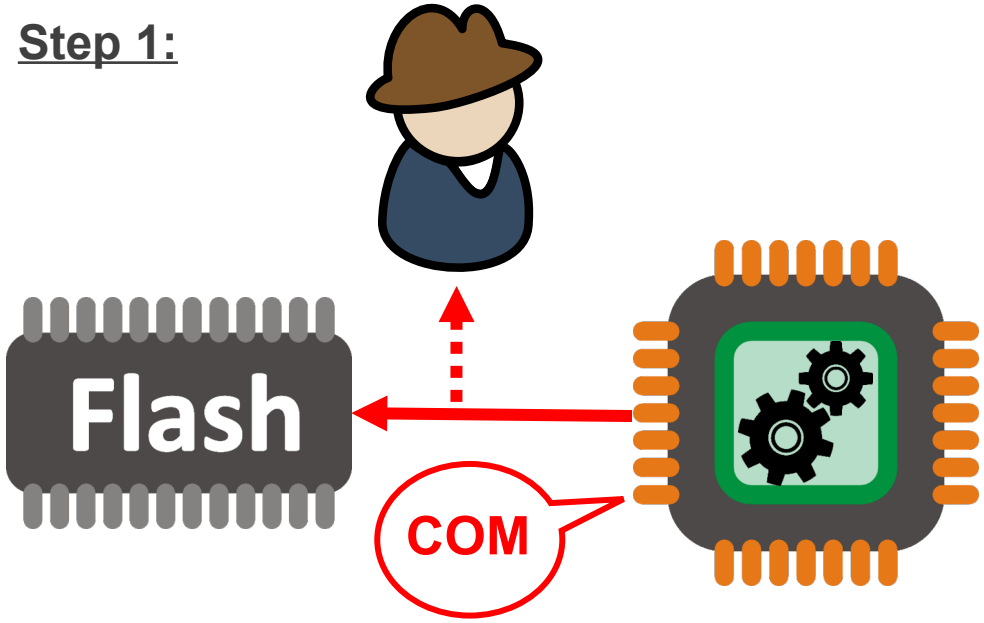


Attack synchronization

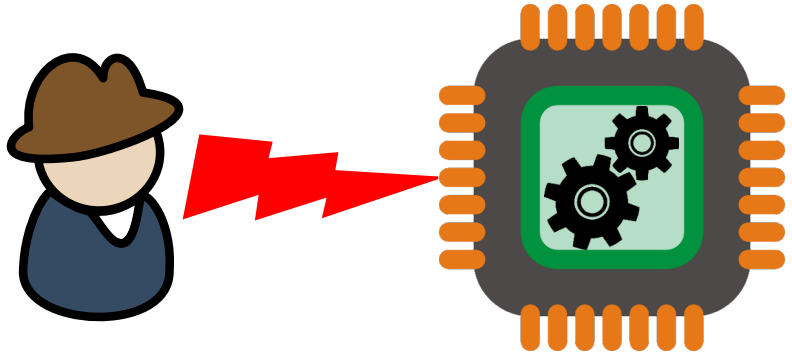
Existing triggering solutions in Black-Box context:

Solution 1: Trigger on uncontrolled I/O

Step 1:



Step 2:



This kind of triggering event is not always accurate but there is no need to control the target.

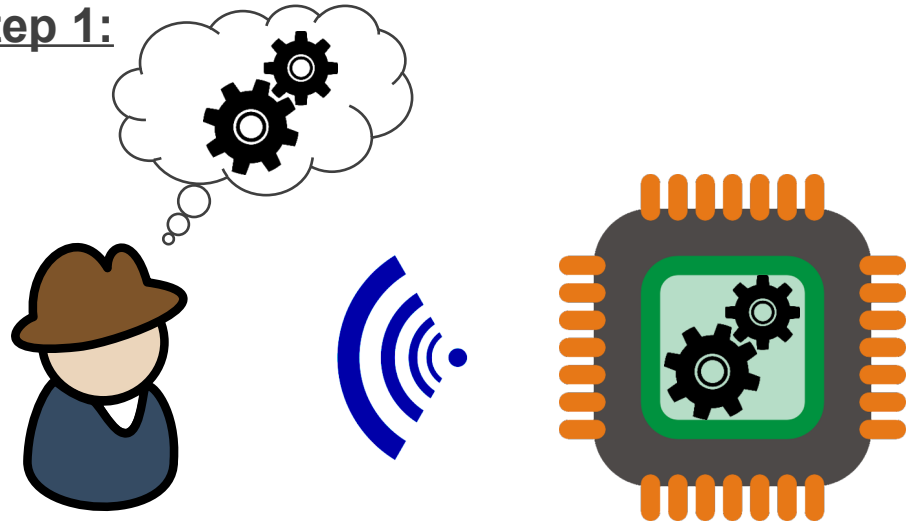
Spruyt et al. (2021) : Fault injection as an oscilloscope: Fault correlation analysis.
Bittner et al. (2021) : The forgotten threat of voltage glitching: A case study on nvidia tegra x2 socs

Attack synchronization

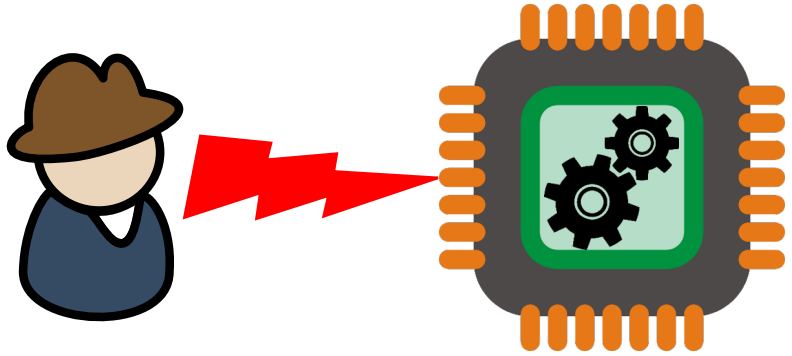
Existing triggering solutions in Black-Box context:

Solution 2: Triggering on a Side-Channel event

Step 1:



Step 2:

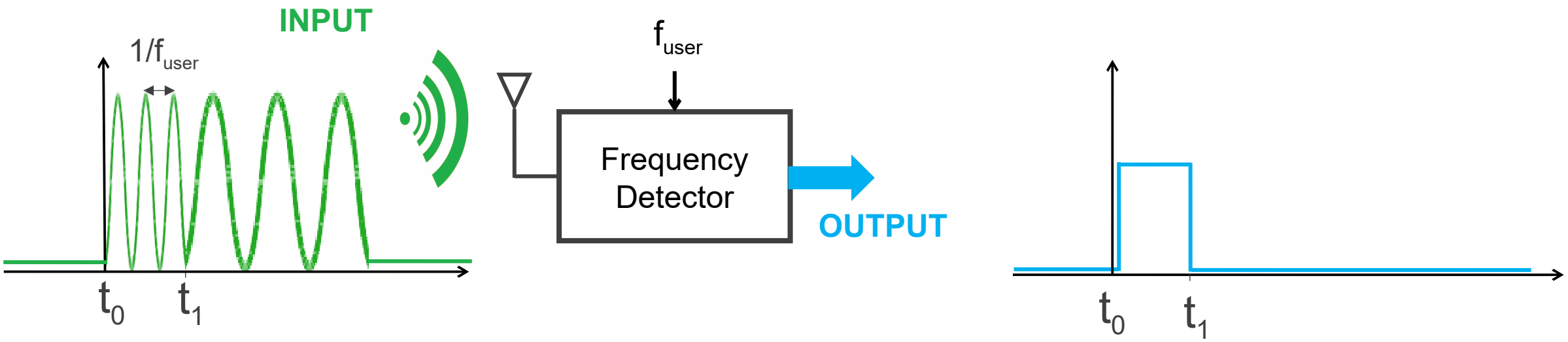


The attacker has a great degree of freedom in the event choice.
For Fault Injection, a real time analysis is needed:
Analysis of high frequency events may be difficult

Camurati et al. (2018) : Screaming channels
Beckers et al. (2016) : Design and implementation of a waveform-matching based triggering system

Frequency detector

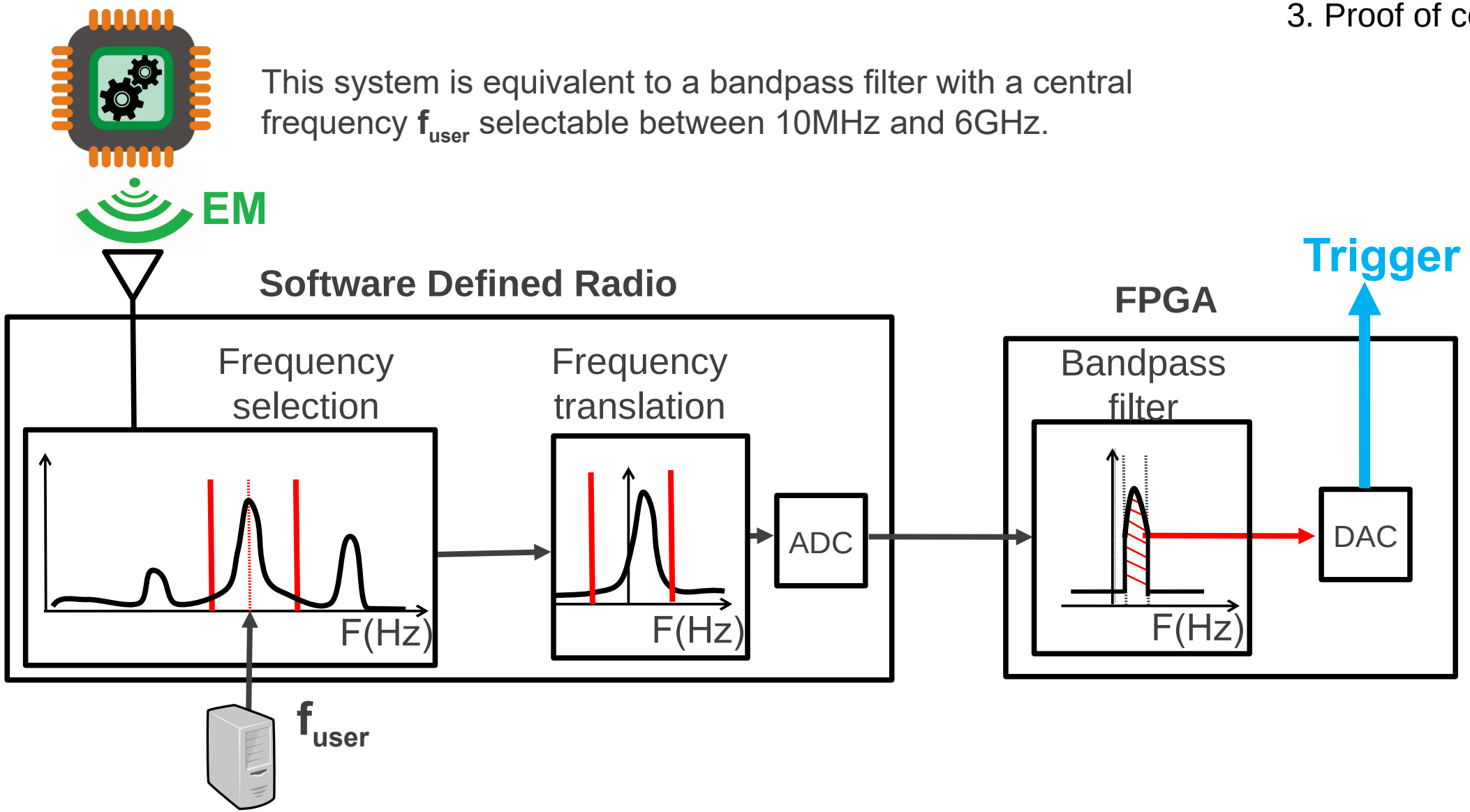
Proposed solution : Synchronization by Frequency Detection in the EM Side-Channel



We need a device that can generate a trigger signal when a chosen frequency is active.

Frequency detector

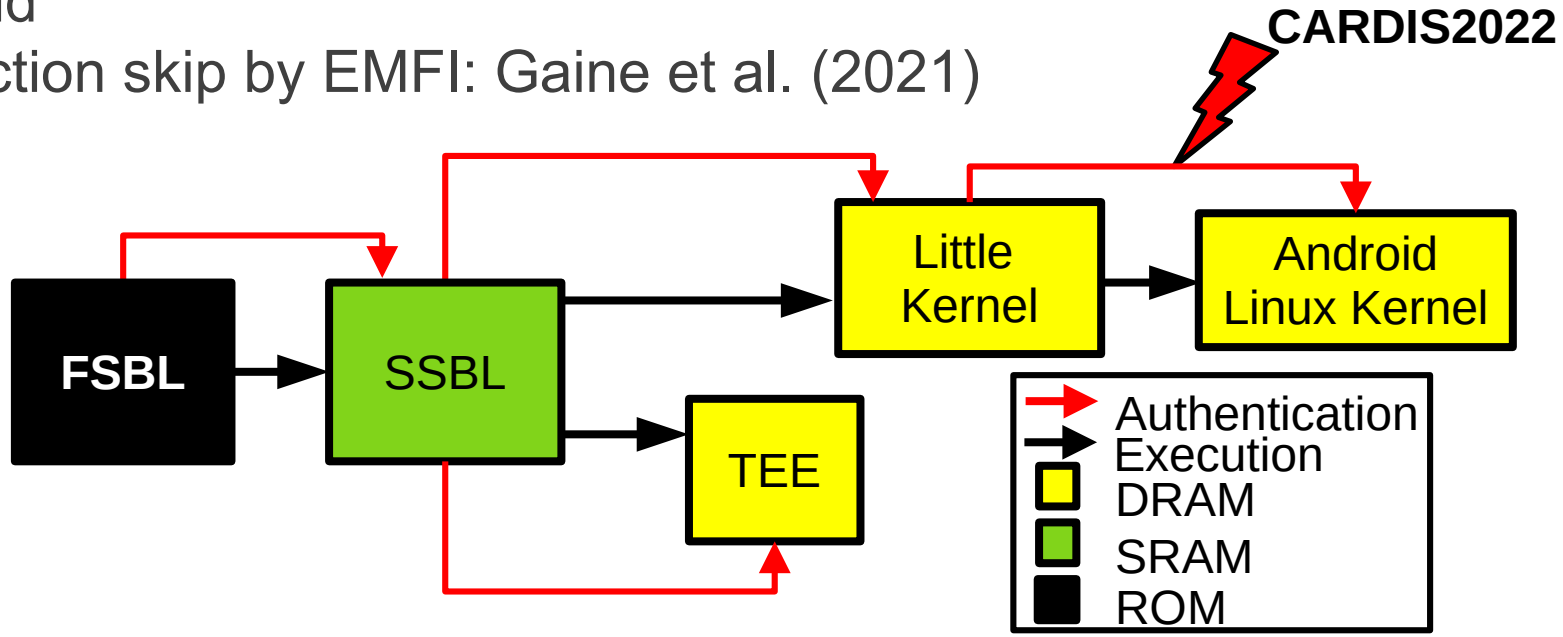
This system is equivalent to a bandpass filter with a central frequency $\mathbf{f_{user}}$ selectable between 10MHz and 6GHz.



Vulnerability Identification

Target: Smartphone System-on-Chip on dev-board:

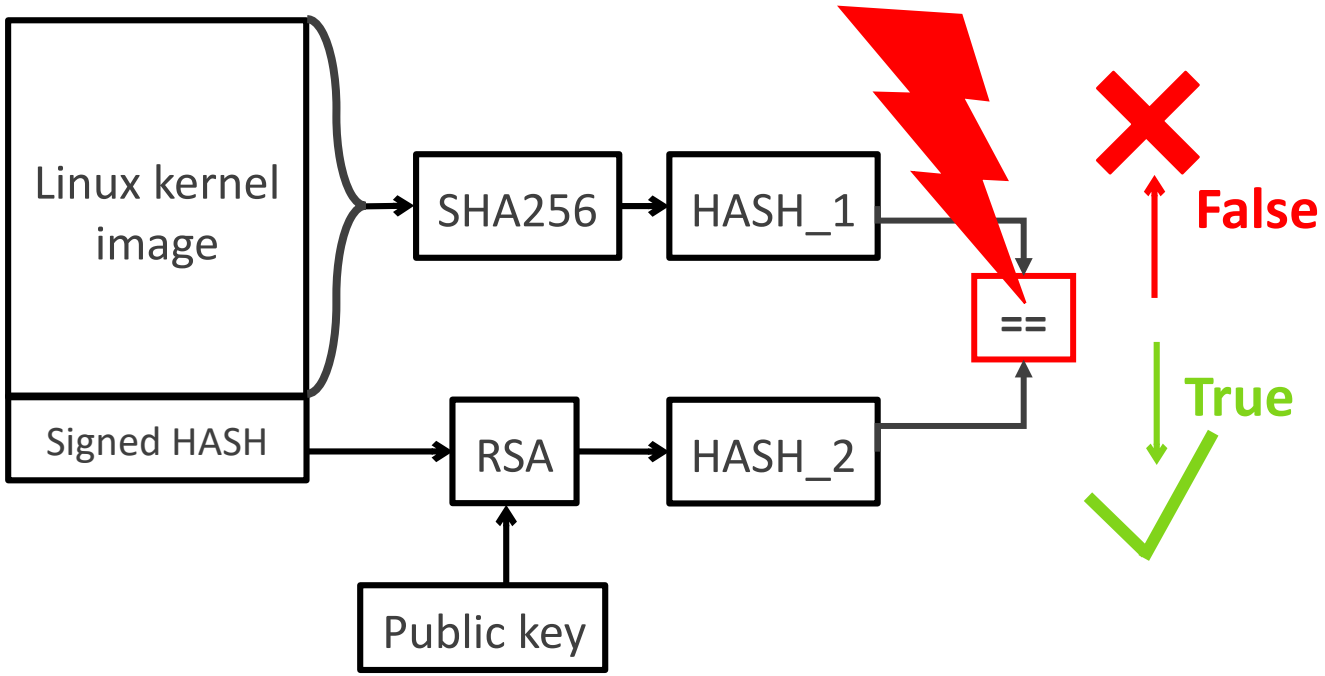
- 4 cores ARM Cortex A53
- Up to 1.2GHz ⇒ 800MHz during the boot phase
- Android
- Instruction skip by EMFI: Gaine et al. (2021)



Objective: Synchronize an EMFI to bypass the Linux Kernel authentication

Attack synchronization

Linux kernel authentication:



HASH Comparison in Little Kernel (C code)

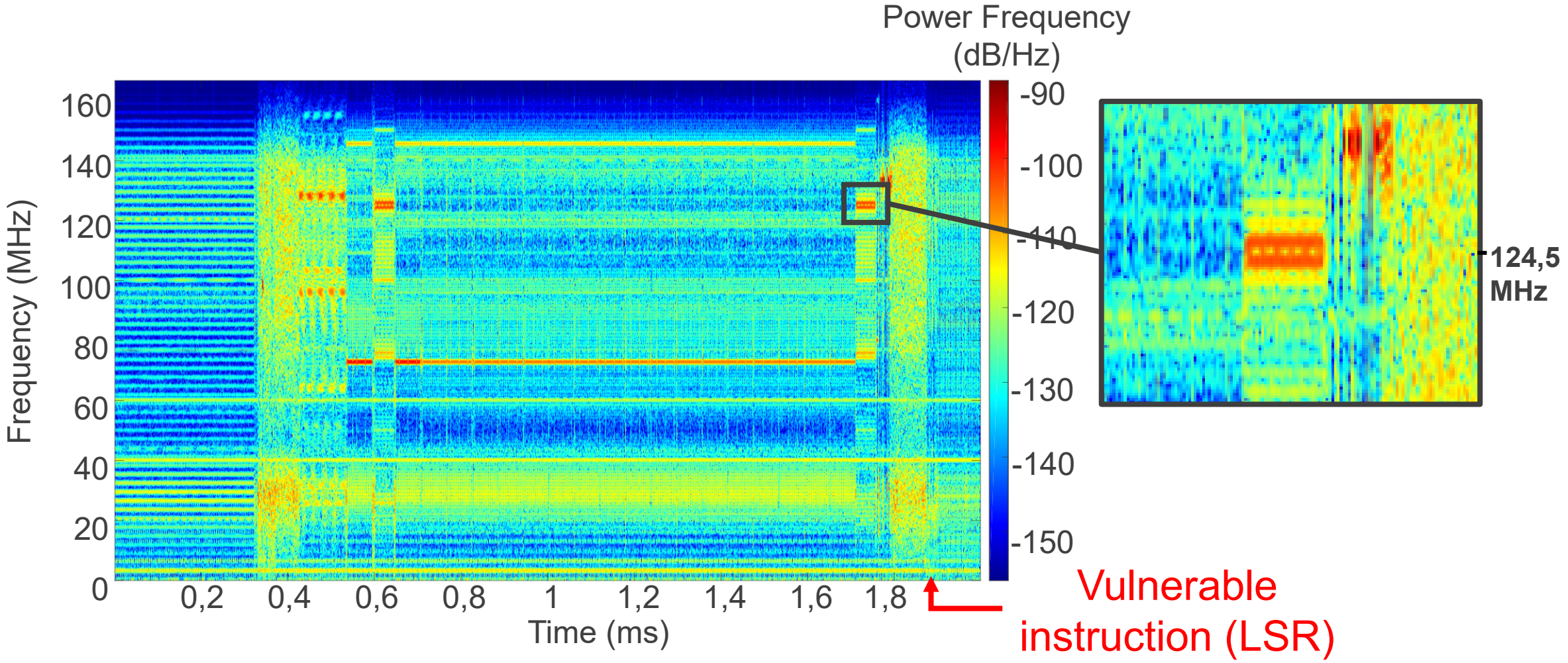
```
ret = memcmp(HASH_1, HASH_2);
if(ret == 0)
    auth = 1;
```

HASH Comparison in Little Kernel (ASM code)

```
bl <memcmp>
clz r6, r0
lsr r6, r6, #5
```

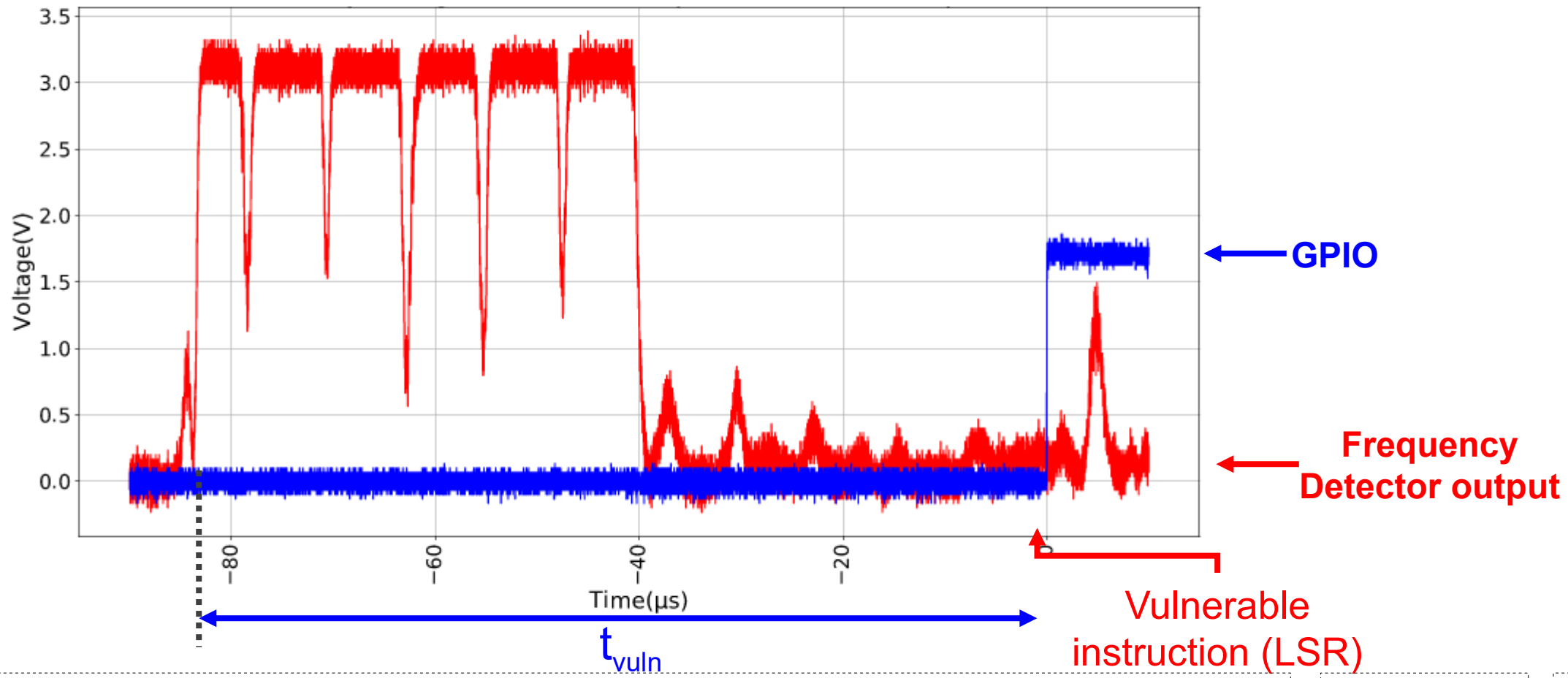
Attack synchronization

Offline spectrogram of the target EM emanations before the vulnerable instruction.



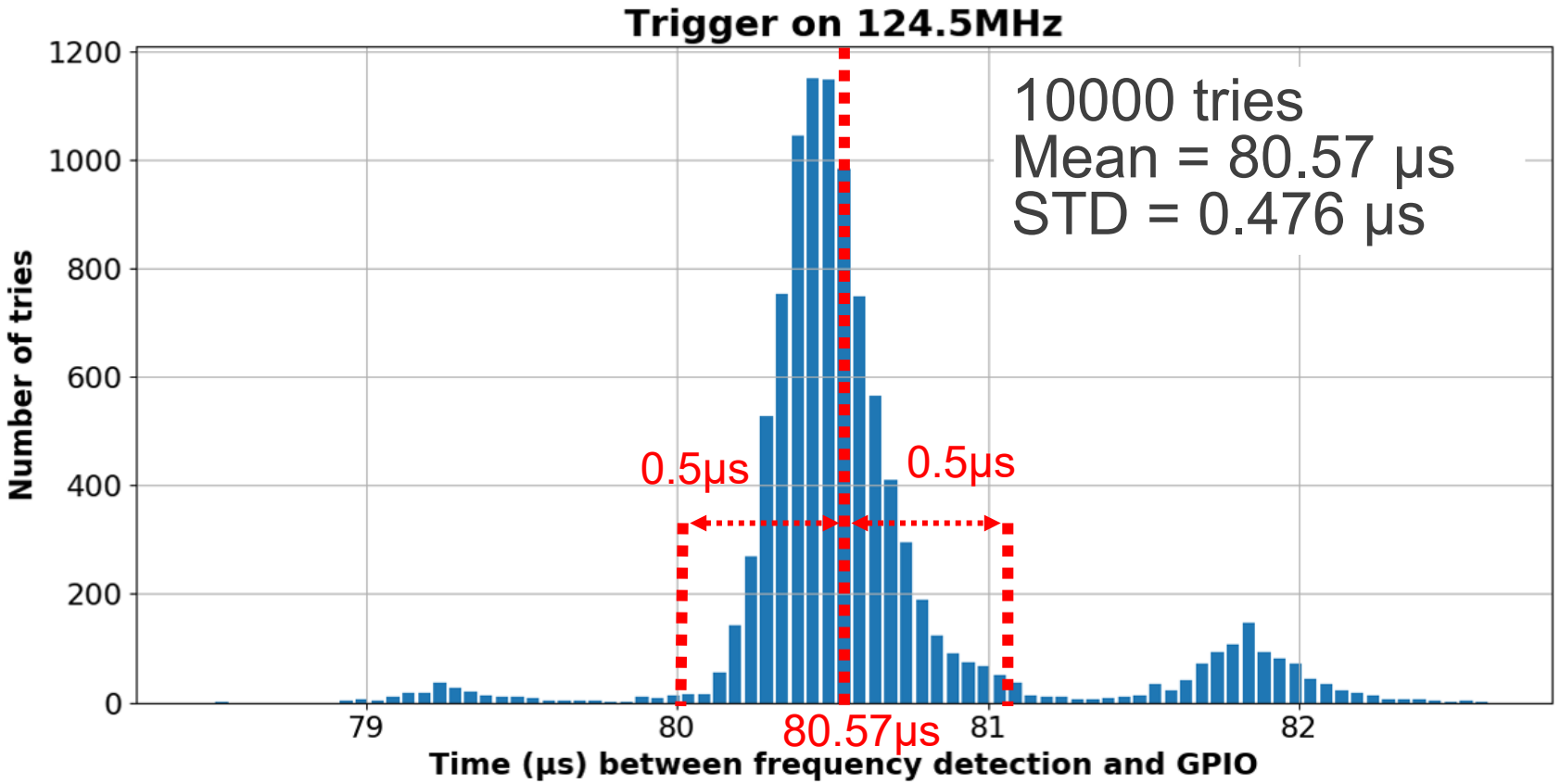
Attack synchronization

We set the frequency detector to trigger on the 124.5MHz frequency identified with an **offline spectrogram**.
We use a modified Little Kernel to rise a GPIO just **after** the vulnerability.



Attack synchronization

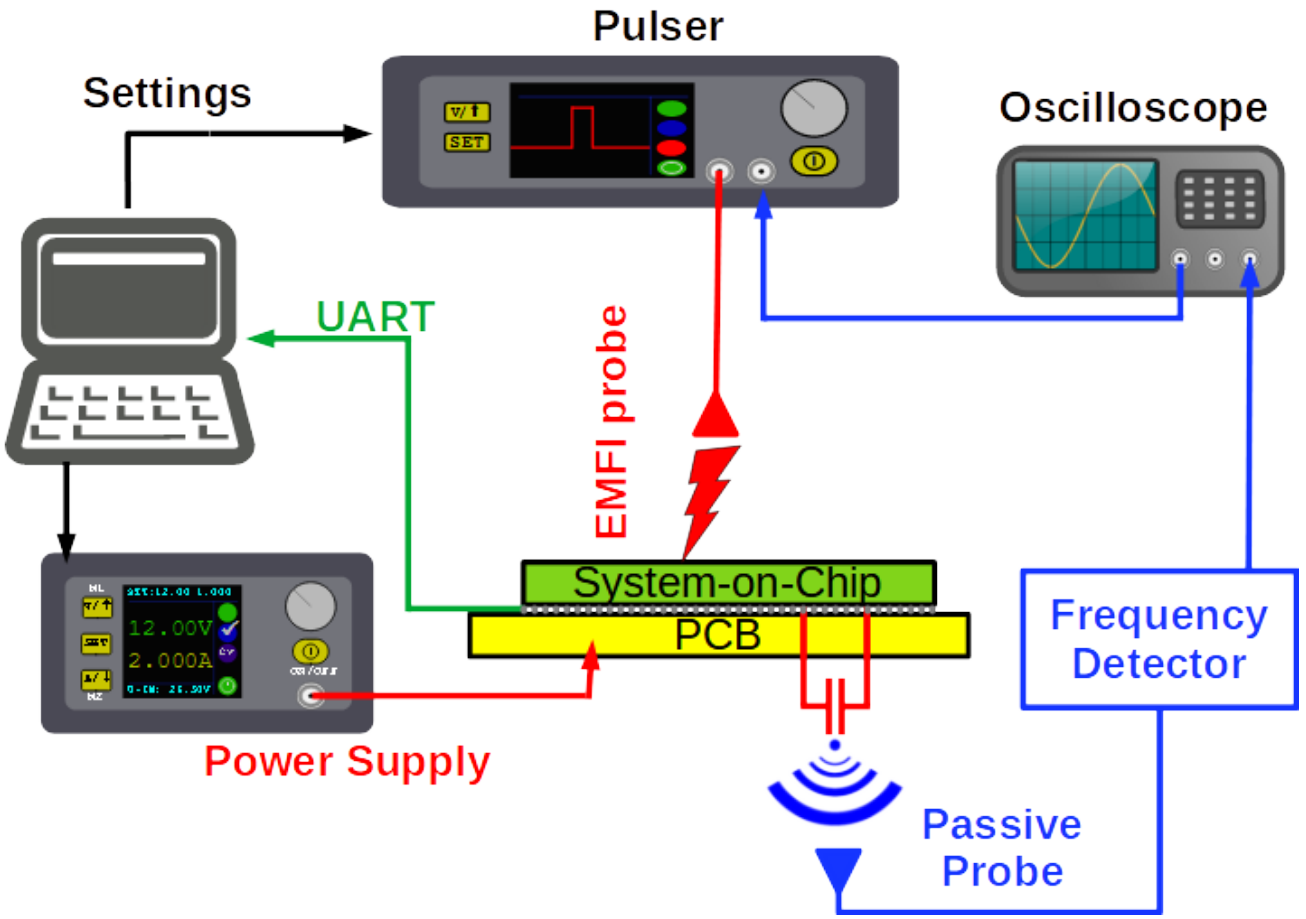
This measure is performed 10000 times to identify the mean delay and the jitter.



Attack synchronization

Experimental setup :

Target running an unmodified Little Kernel (no GPIO)



Attack synchronization

15000 attacks in 18h00:

Scenarios	Results
Crash	7005 (46,777%)
No effect	7912 (52,75%)
Authentication bypass	83 (0,53%)

≈ 1 bypass every 15 minutes.

Paper published at **CARDIS2022** (best student paper).

Presentation of this work at **JAIF2022**.

Outline

I. Attack synchronization
1. Synchronization issue
2. Proposed solution
3. Proof of concept
   ⇒ Synchronizing EMFI to bypass the Android Linux Kernel authentication

II. Vulnerability identification
1. High level analysis
2. Side-Channel Analysis
3. Proof of concept
   ⇒ **EMFI to bypass the SSBL authentication by the FSBL**

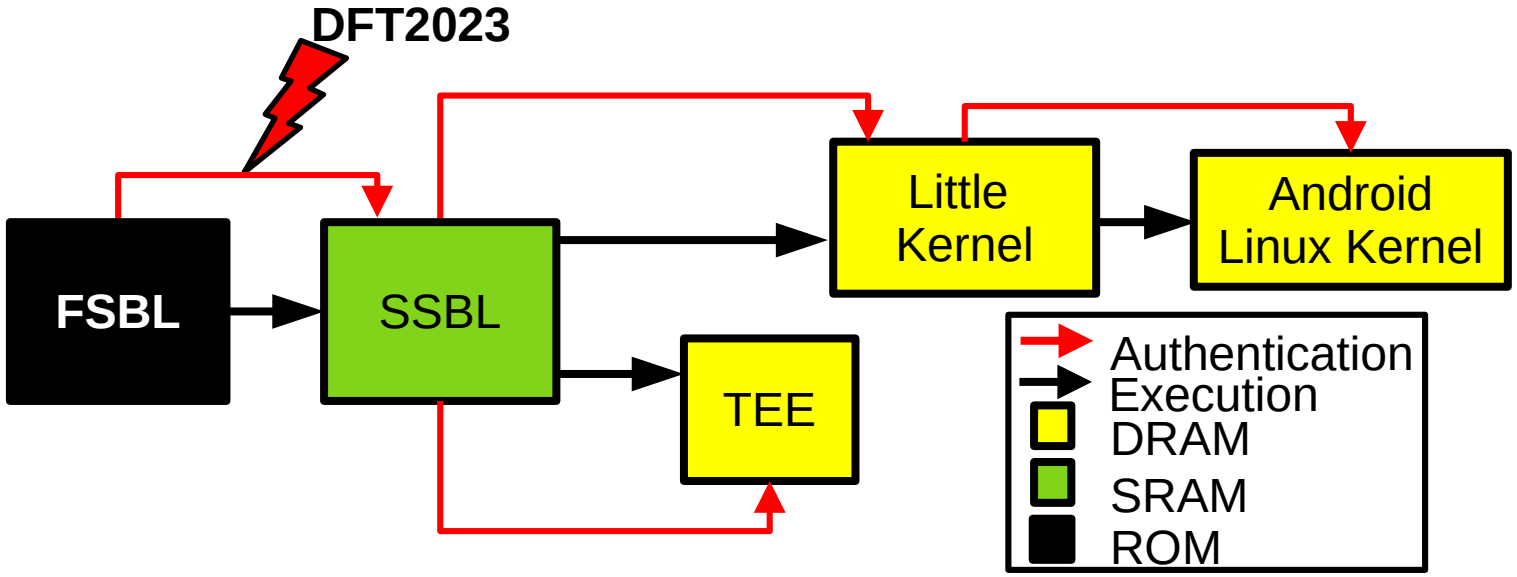
III. FI on Package-on-Package SoC
1. Software setup: Code under attack and Fault Model
2. Fault Injections setup: EMFI / EM-induced glitches / Voltage glitches
3. Faults comparison

IV. Conclusion

Vulnerability Identification

Target: Smartphone System-on-Chip on dev-board:

- 4 cores ARM Cortex A53
- Up to 1.2GHz ⇒ 800MHz during the boot phase
- Android



Objective: Bypass the SSBL authentication

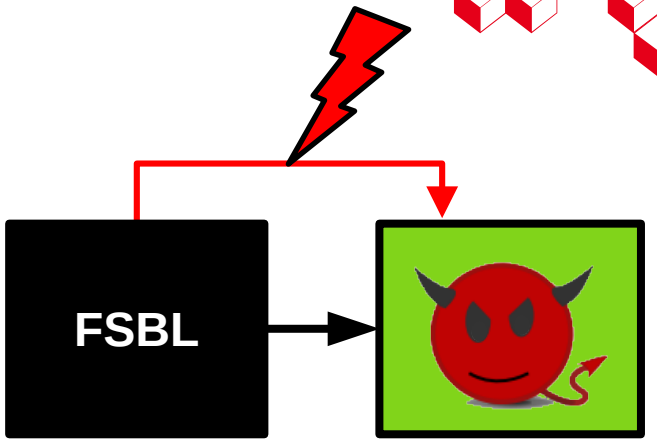
⇒ More privileges

⇒ FSBL harder to patch than Little Kernel

Vulnerability Identification

	Target SoC	Access to the code or binary	Methodology to find the vulnerability timing
Timmers et al. **FDTC2016**	AArch32 800MHz	YES	Anytime during a copy phase
Vasselle et al. **FDTC2017**	4-Cortex A9 – 1.4GHz	Binary dumped	Divergence analysis in Power Side-Channel + Static Faults
Bittner et al. **FDTC2021**	Tegra X2 Cortex R5	Binary dumped	Bruteforce
Fanjas et al. **CARDIS2022**	4-Cortex A53 – up to 1.2GHz (800MHz in practice)	YES	Use of I/O
Colin O'Flynn **eprint iacr2020 n°937**	µC MPC5xxx	NO	Divergence analysis in Power Side-Channel
Kühnapfel et al. **PAINE2022**	AMD-SP : Armv7	NO	Bruteforce
Marco et al. **BlackHat2022**	4-Cortex A53 – up to 1.2GHz	NO	Divergence analysis in power to set up a timing window
Nashimoto et al. **CHES2022**	X300 RISC-V 65MHz	YES	Identification of an instruction in EM waveform
Gaine et al. **WIFS2020**	4-Cortex A53 – 1.2GHz	YES	Use of I/O
Fanjas et al. DFT2023	4-Cortex A53 – up to 1.2GHz (800MHz in practice)	NO	Divergence analysis in EM Side-Channel

Vulnerability Identification



<u>Objective:</u> Bypass the SSBL authentication

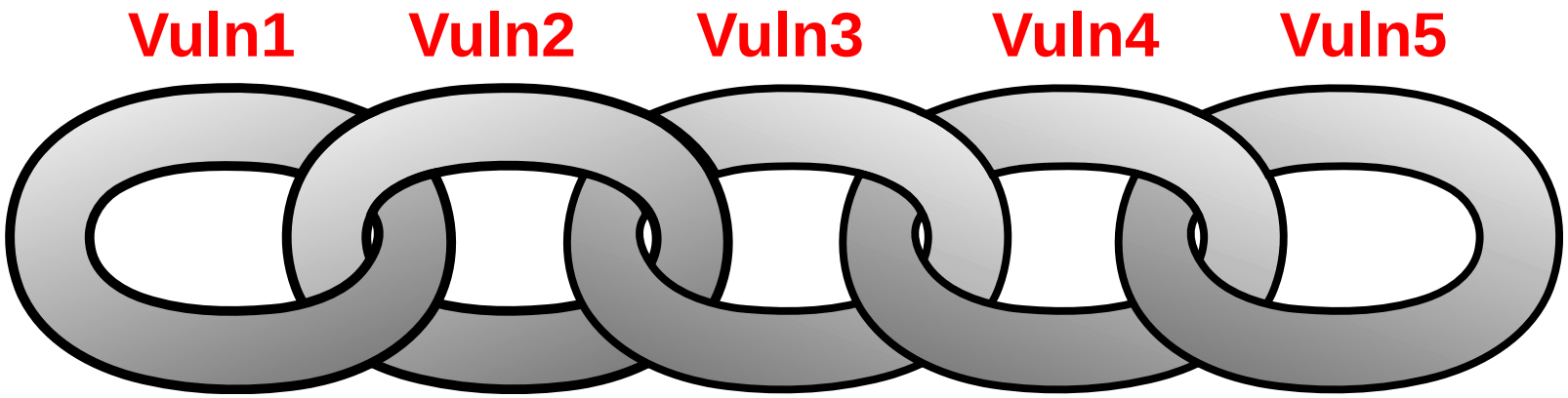
- <u>First Stage BootLoader in Read-Only Memory:</u>
  - **How to find a vulnerability to fault-injection without access to the code or binary?**
  - **When is this vulnerability executed?**
  - **How to find a detectable signal (trigger) suitable as temporal reference for the attack?**
  - How to find the best fault-injection parameters?

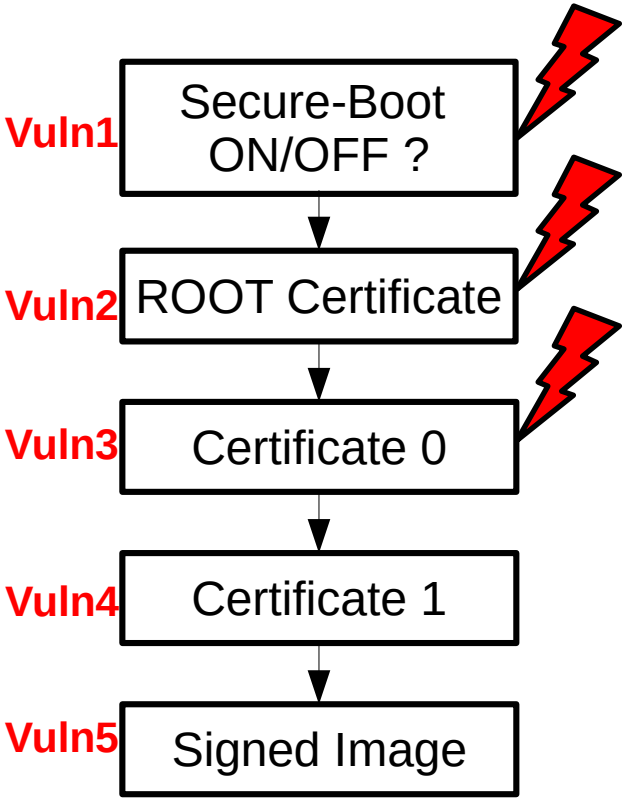
- <u>Proposed methodology:</u>
  - 1. Fault-Injection Vulnerability Analysis at high level
  - 2. Side-Channel Analysis to find the vulnerabilities timing
  - 3. Proof of concept using ElectroMagnetic Fault-Injection (EMFI)

Vulnerability Identification

SSBL image authentication:

1. **Secure-Boot status check** **(Vuln1)**
2. **Root Certificate hash verification** **(Vuln2)**
3. **Certificate 0 signature verification** **(Vuln3)**
4. Certificate 1 signature verification (Vuln4)
5. Image signature verification (Vuln5)

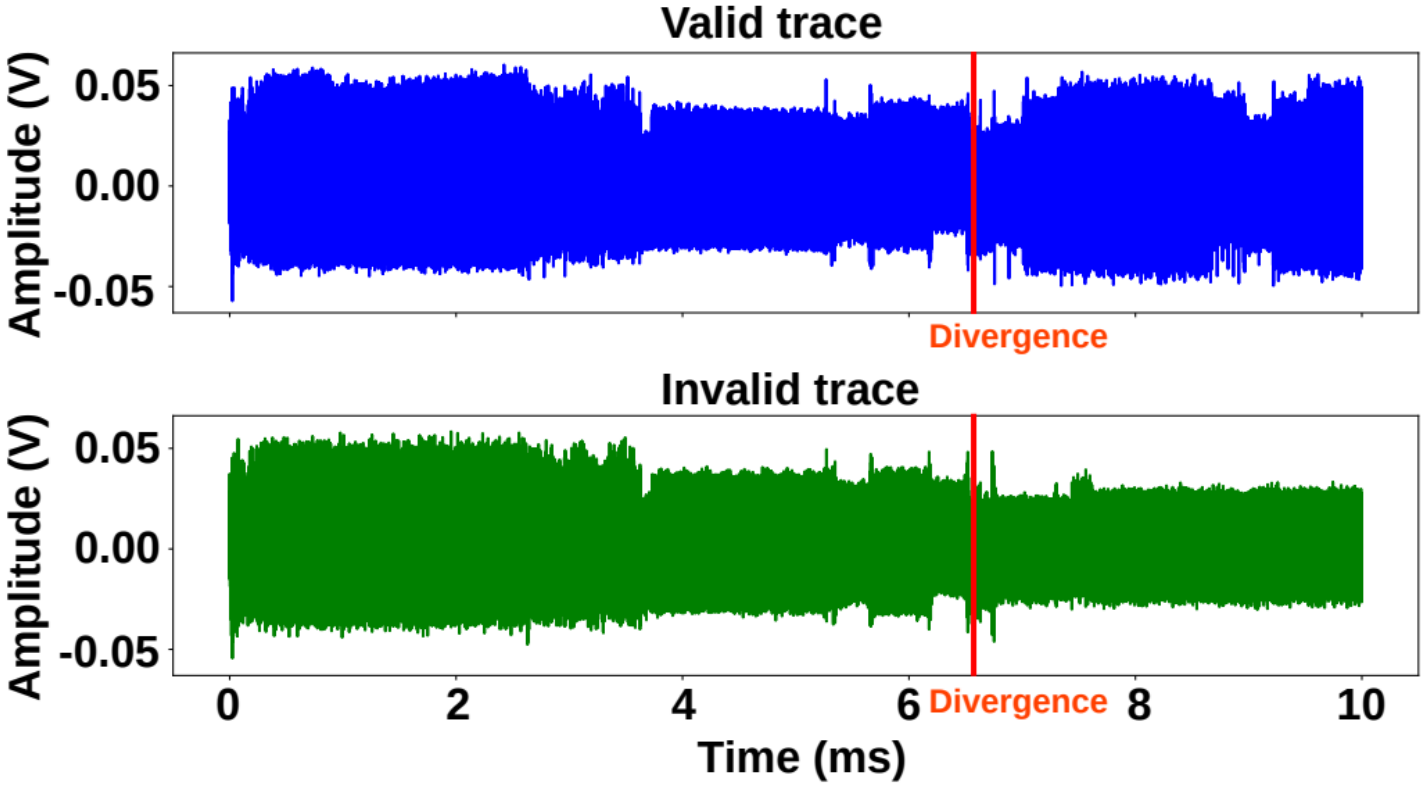


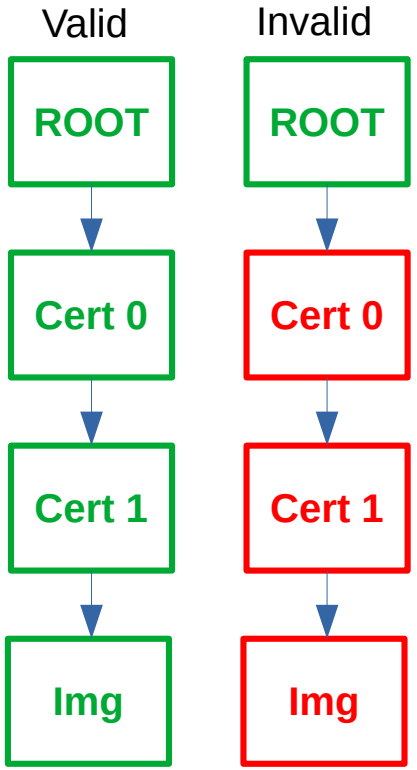


Vulnerability Identification

Vuln3: Certificate 0 signature verification

First approach : Comparison of EM traces with a valid and an invalid certificate 0.

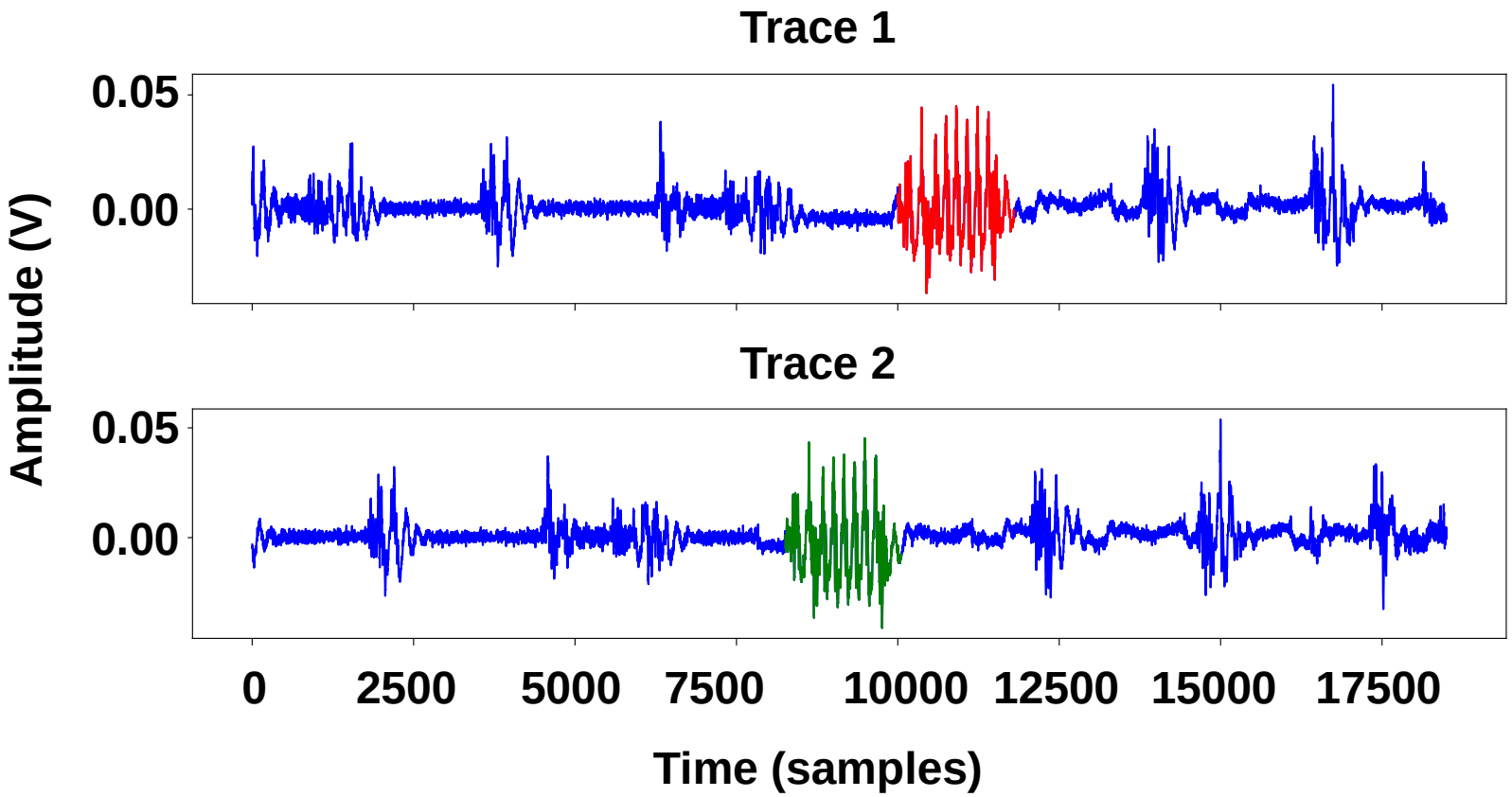




10ms ⇒ 8000000 clock cycles ⇒ **IMPOSSIBLE TO BRUTEFORCE!**

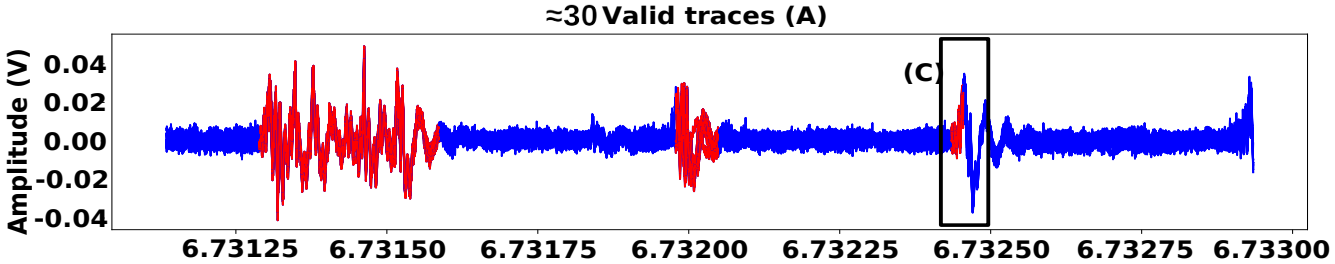
Vulnerability Identification

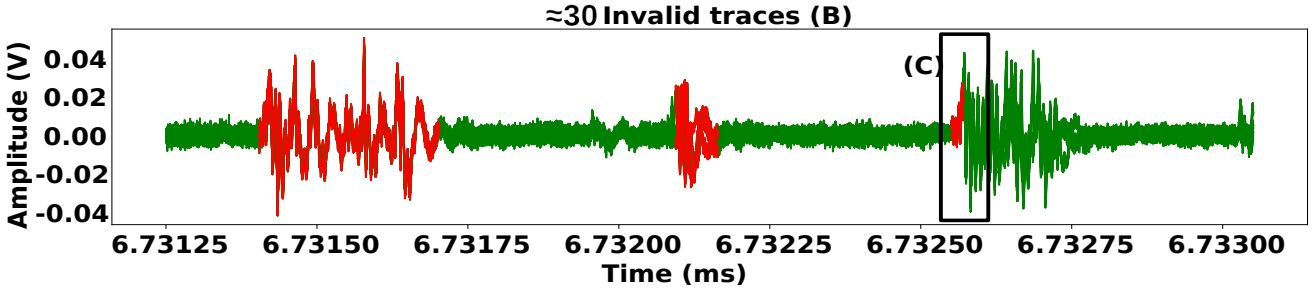
- **Patterns are representative of the target control-flow**
- **The timing of a pattern may change between two executions of the same program.**

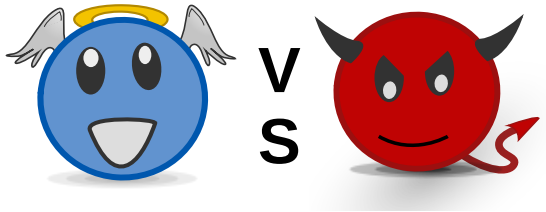


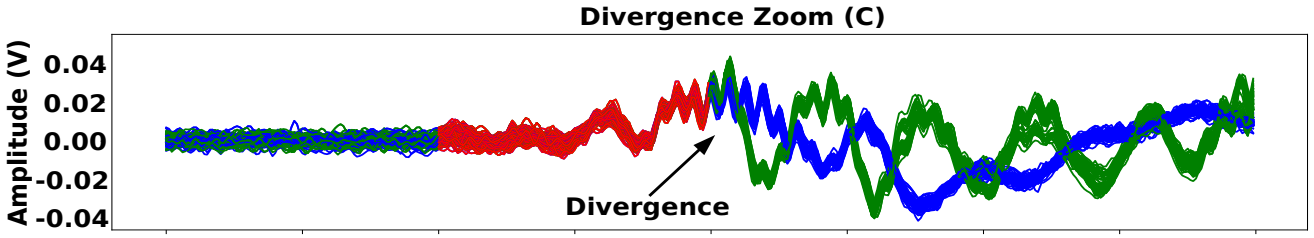
Vulnerability Identification

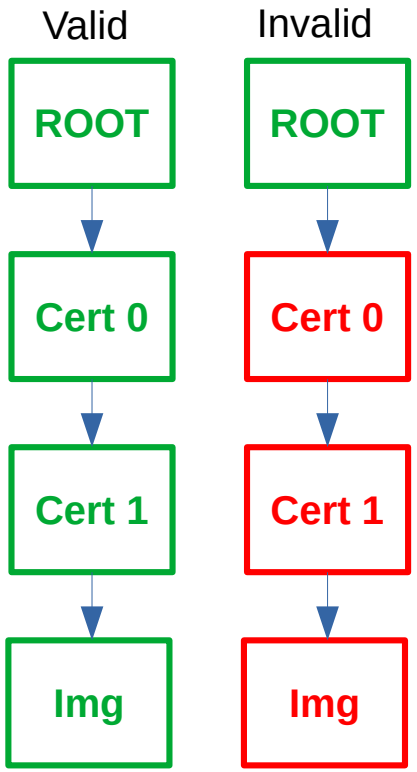
Solution : Divergence Analysis
Vuln3 : Certificate 0 : Comparison of EM traces with a valid and an invalid certificate 0
⇒ Identification of the mean timing of the diverging pattern









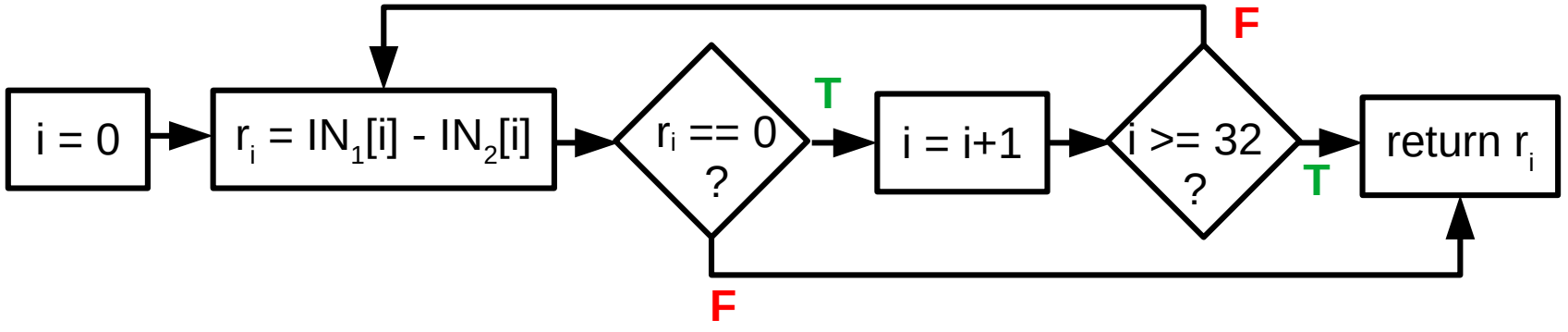


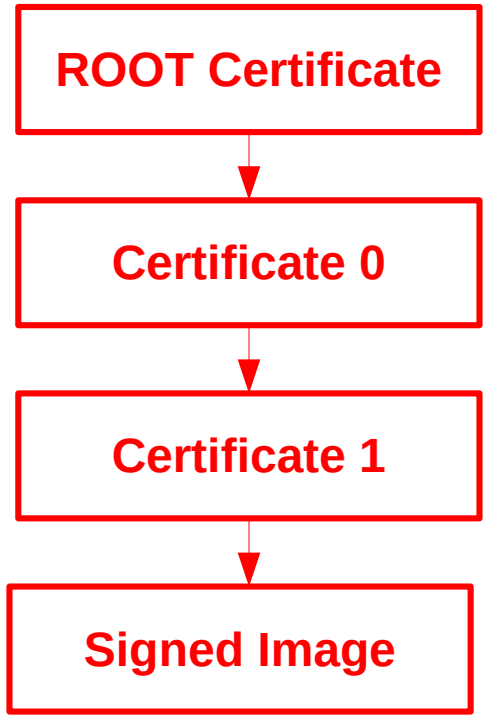
Vulnerability Identification

Other solutions: Trigger the divergence

Vuln2: Root certificate

- 2 HASH are compared using a ''MEMCMP'' function





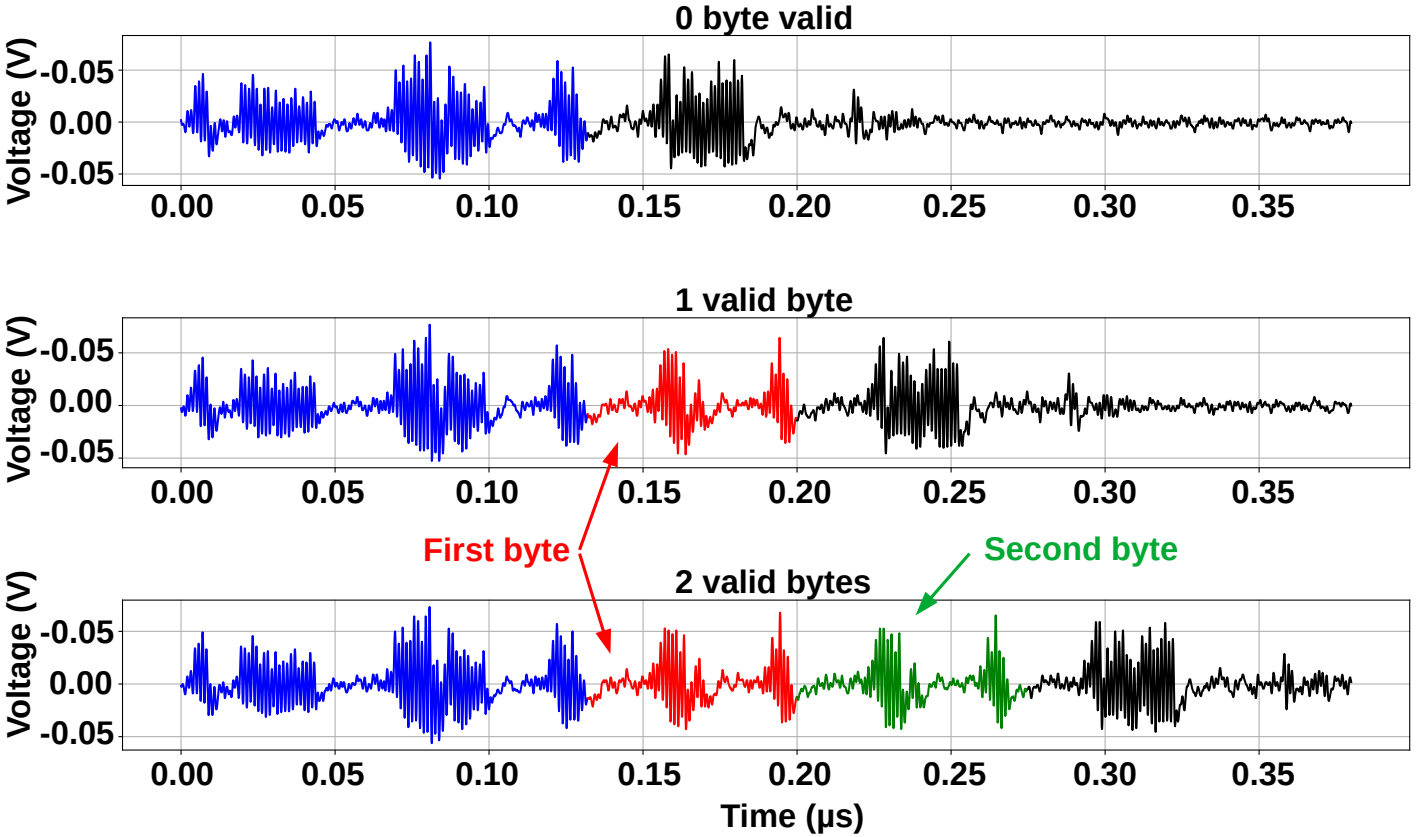
What happens if a HASH share the same first bytes as the HASH stored in fuse?

0 valid byte hash	: **B043C4DD67**	**…**	**26BBD6A24C**
1 valid byte hash	: **E1EB1C648D**	**…**	**0ED518FBB7**
2 valid bytes hash	: **E11CEBC2DA**	**…**	**EC0AA76BA2**
Hash stored in fuse	: **E11C9B94AF**	**…**	**57256A5A08**

Vulnerability Identification

Vuln2 : ROOT Certificate

⇒ **Comparison of EM traces using partially bruteforced ROOT Certificate**



B043C4...

E1EB1C...

E11CEB...

Vulnerability Identification

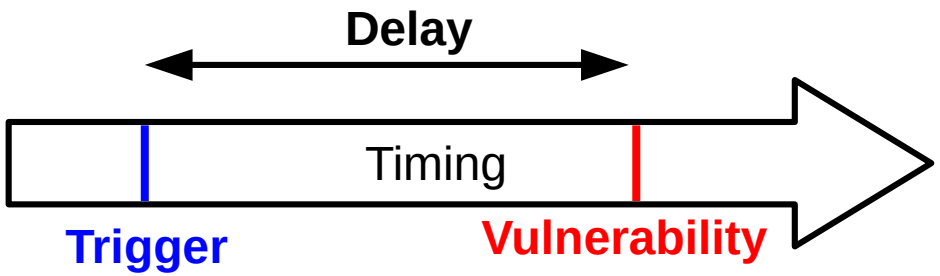
Step 2:

- Divergence found for each vulnerability
⇒ Trigger identified for each divergence
- Communication between SoC and EMMC
- Side-Channel based trigger: Frequency activation

⇒ Delay characterized between trigger and vulnerability

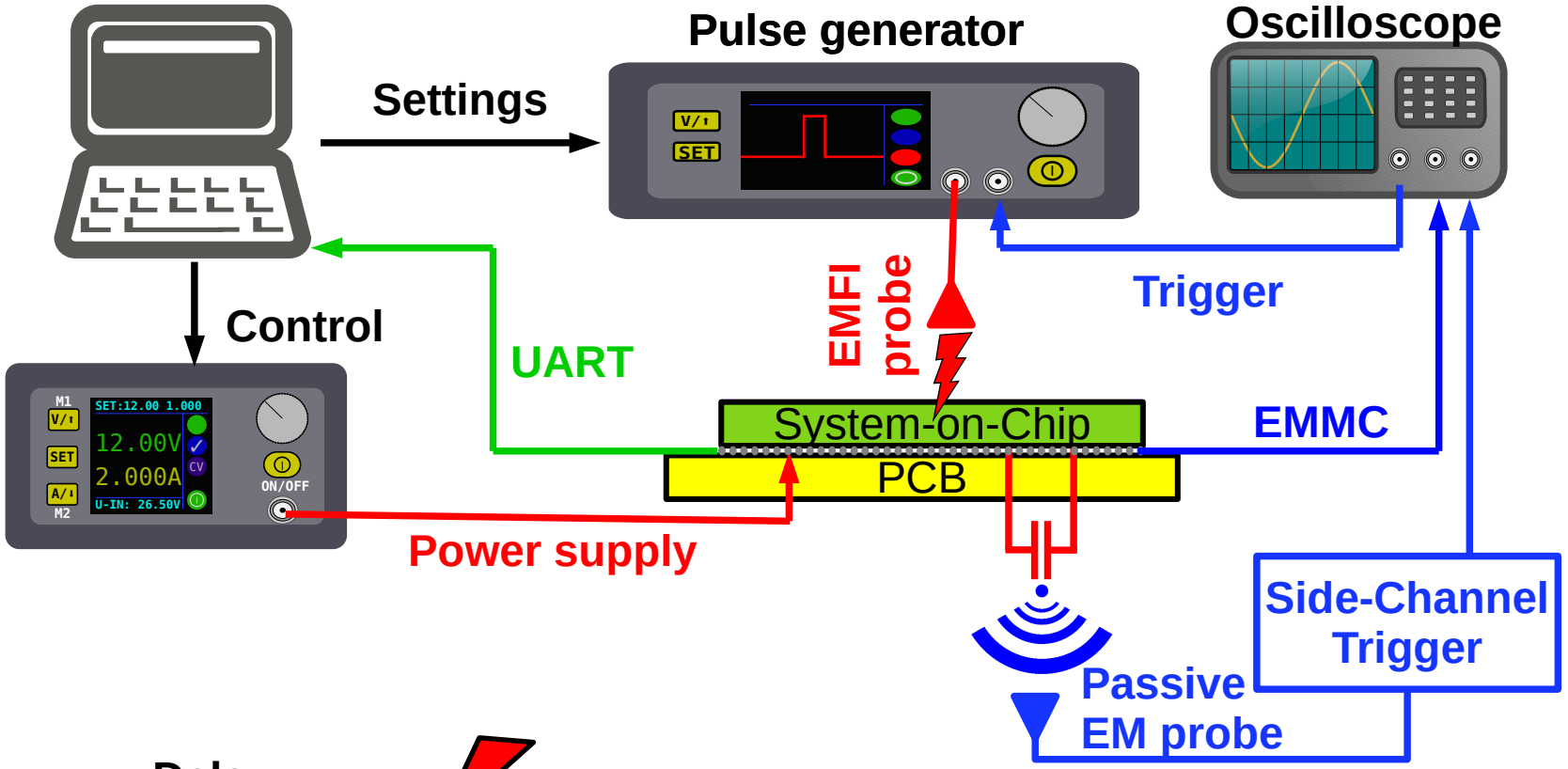
Vuln1:	**45.12µs**	**(jitter = 49ns)**	**⇒ EMMC communication**
Vuln3:	**6733.00µs**	**(jitter = 1870ns)**	**⇒ EMMC communication**
Vuln2:	**4.95µs**	**(jitter = 492ns)**	**⇒ Frequency Detector**

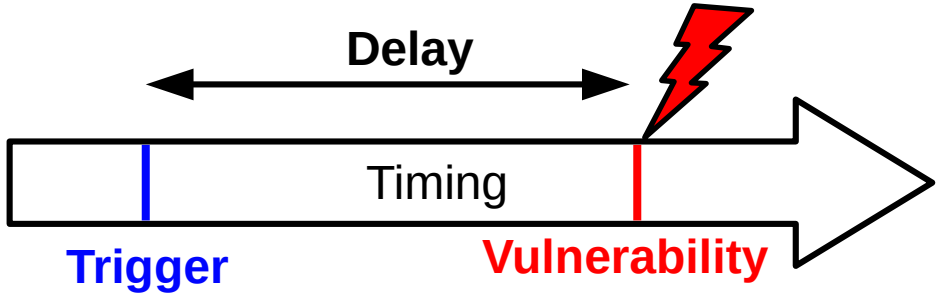
Vuln 1 = Secure-Boot Status check
Vuln 2 = Root certificate
Vuln 3 = Certificate 0



Vulnerability Identification

1. High level analysis
2. Side-Channel Analysis
3. **Proof of concept**





Vulnerability Identification

Step 3: Conclusion

Secure-Boot bypass ⇒ malicious SSBL executed

Vuln1: Attack fails when injecting faults using the measured delay
⇒ Success when bruteforcing the timing window between the trigger and the vulnerability
⇒ 3 bypass on 50000 experiments ⇒ 78h ⇒ 1 success in 26 hours

Vuln2: 10 bypass on 55000 experiments ⇒ 66h ⇒ 1 success in 6.6 hours

Vuln3: 2 bypass on 25000 experiments ⇒ 17h ⇒ 1 success in 8.5 hours

Work published and presented at DFT2023.

Outline

I. Attack synchronization
   1. Synchronization issue
   2. Proposed solution
   3. Proof of concept
      ⇒ Synchronizing EMFI to bypass the Android Linux Kernel authentication

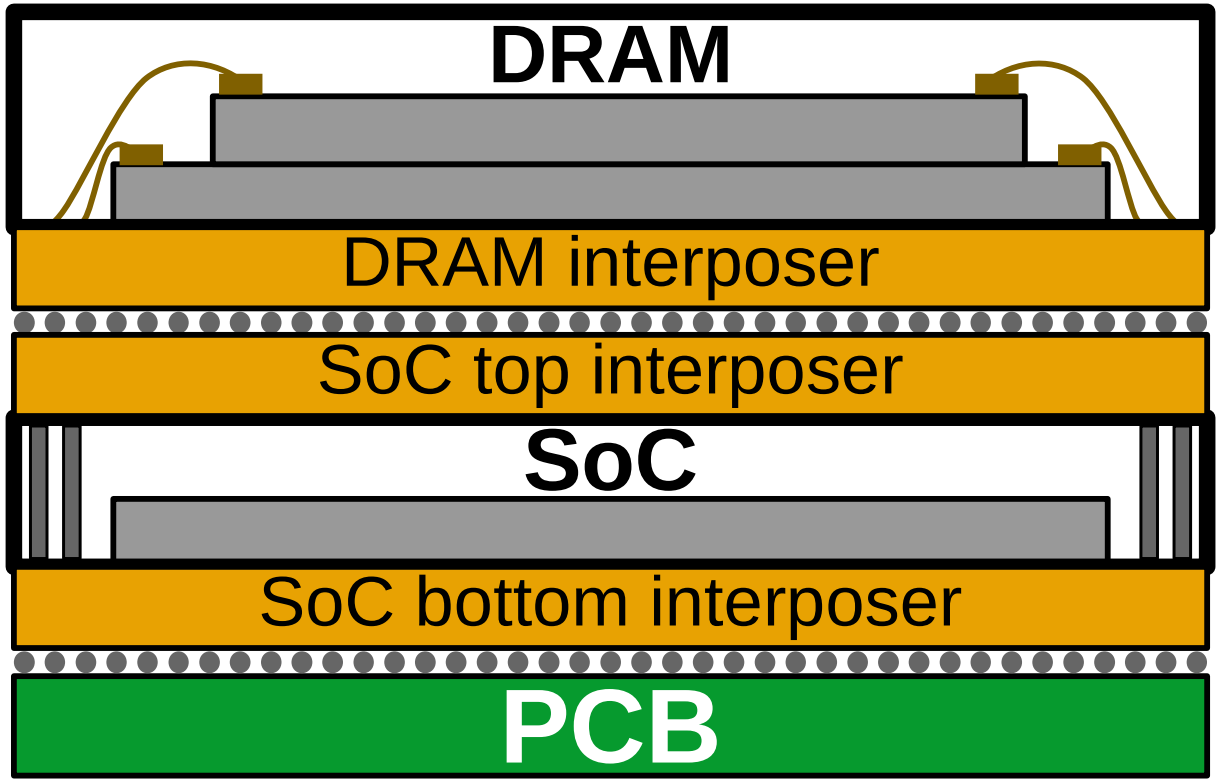
II. Vulnerability identification
   1. High level analysis
   2. Side-Channel Analysis
   3. Proof of concept
      ⇒ EMFI to bypass the SSBL authentication by the FSBL

III. FI on Package-on-Package SoC
   1. Software setup: Code under attack and Fault Model
   2. Fault Injections methods: EMFI / EM-induced glitches / Voltage glitches
   3. Faults obtained with each method
   4. Faults comparison

IV. Conclusion

Package-on-Package SoC

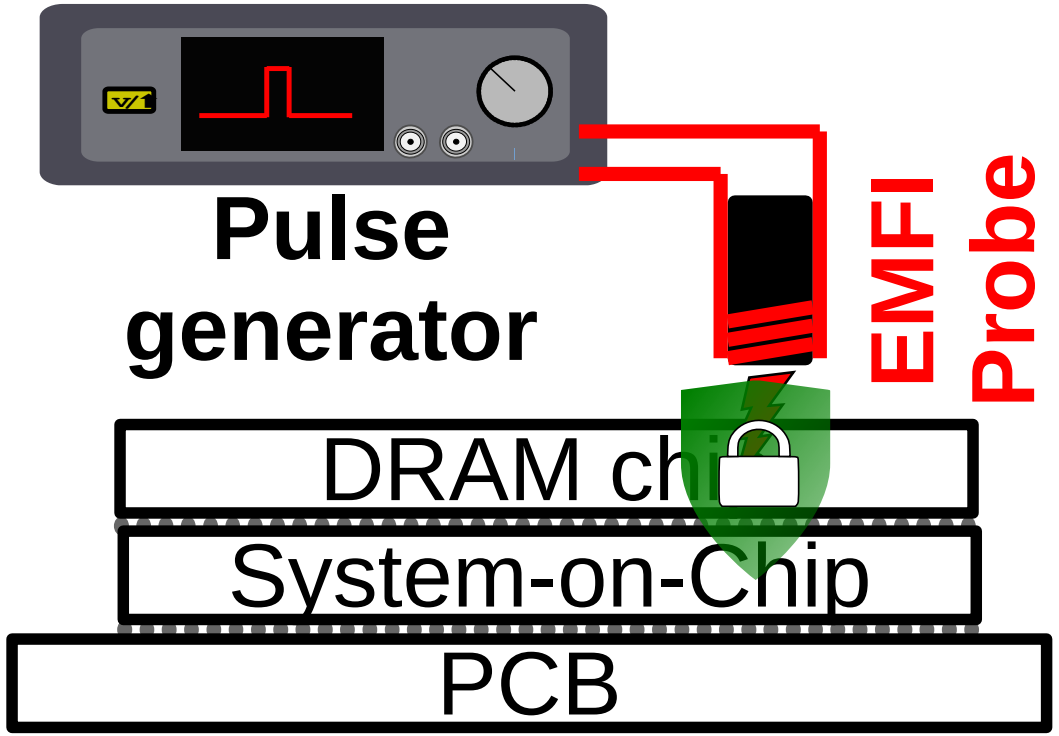
In modern SoC, the DRAM chip is stacked **above** the SoC.

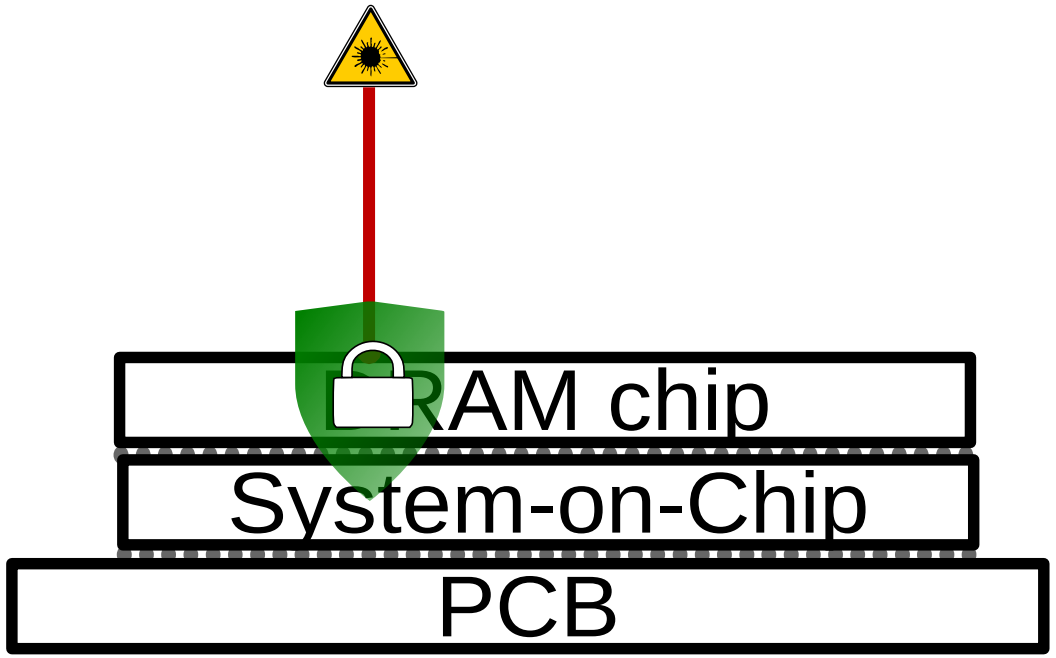


Package-on-Package SoC

It reduces the efficiency of local Fault Injection method such as EMFI or Optical Fault Injection :

Nourdin Aït El Mehdi [2019]: Analyzing the Resilience of Modern Smartphones Against Fault Injection Attacks
Aurélien Vasselle et al. [2017]: Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot

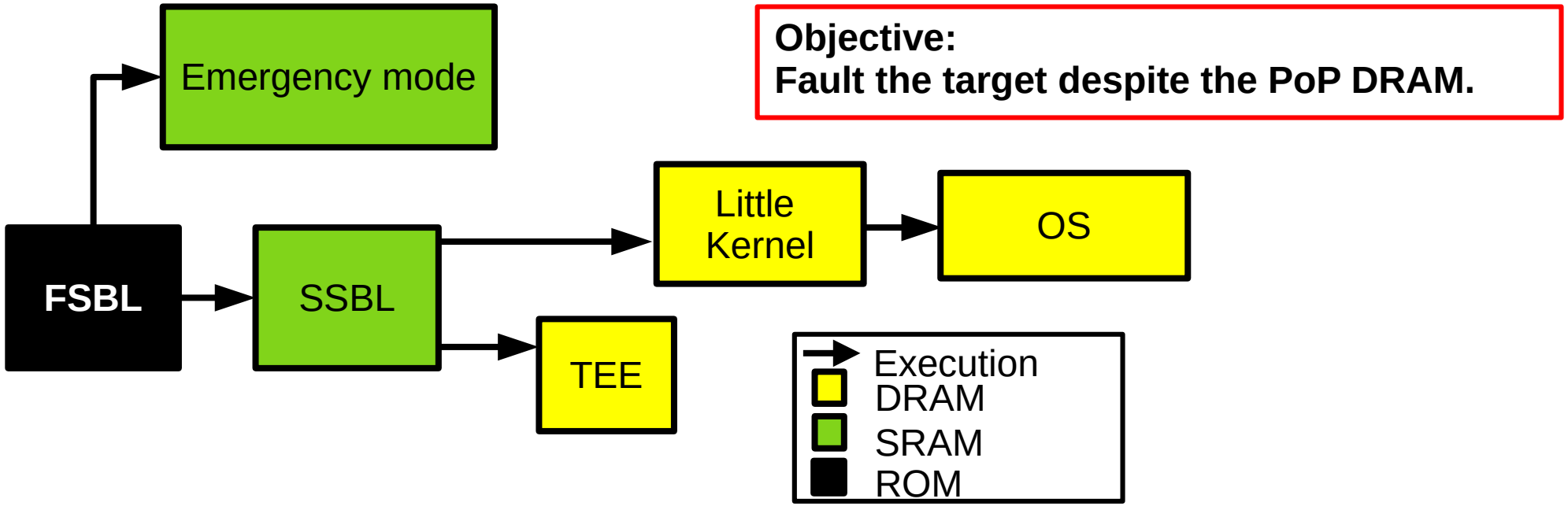




Package-on-Package SoC

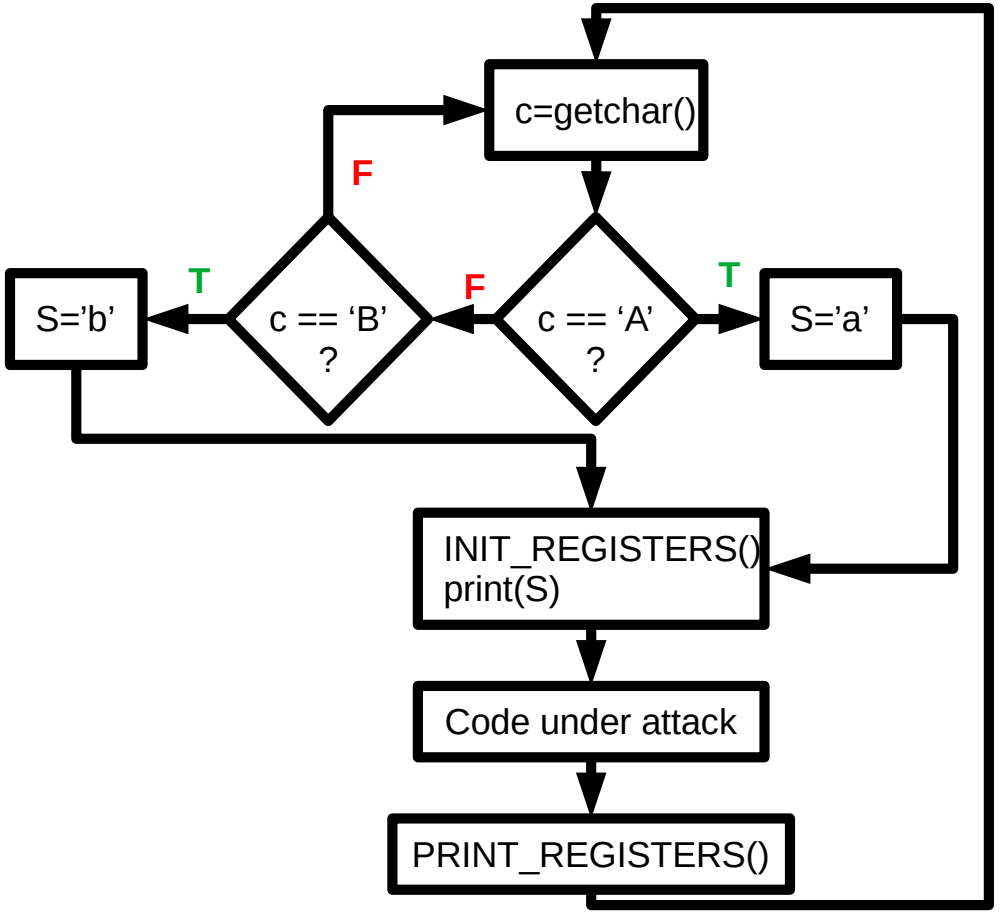
Target: Smartphone System-on-Chip on dev-board:

- 4 custom ARM cores
  - ⇒ 2 cores up to 2.15GHz
  - ⇒ 2 cores up to 1.59GHz
- Secure-Boot disabled
- Emergency mode running in SRAM at the SSBL level



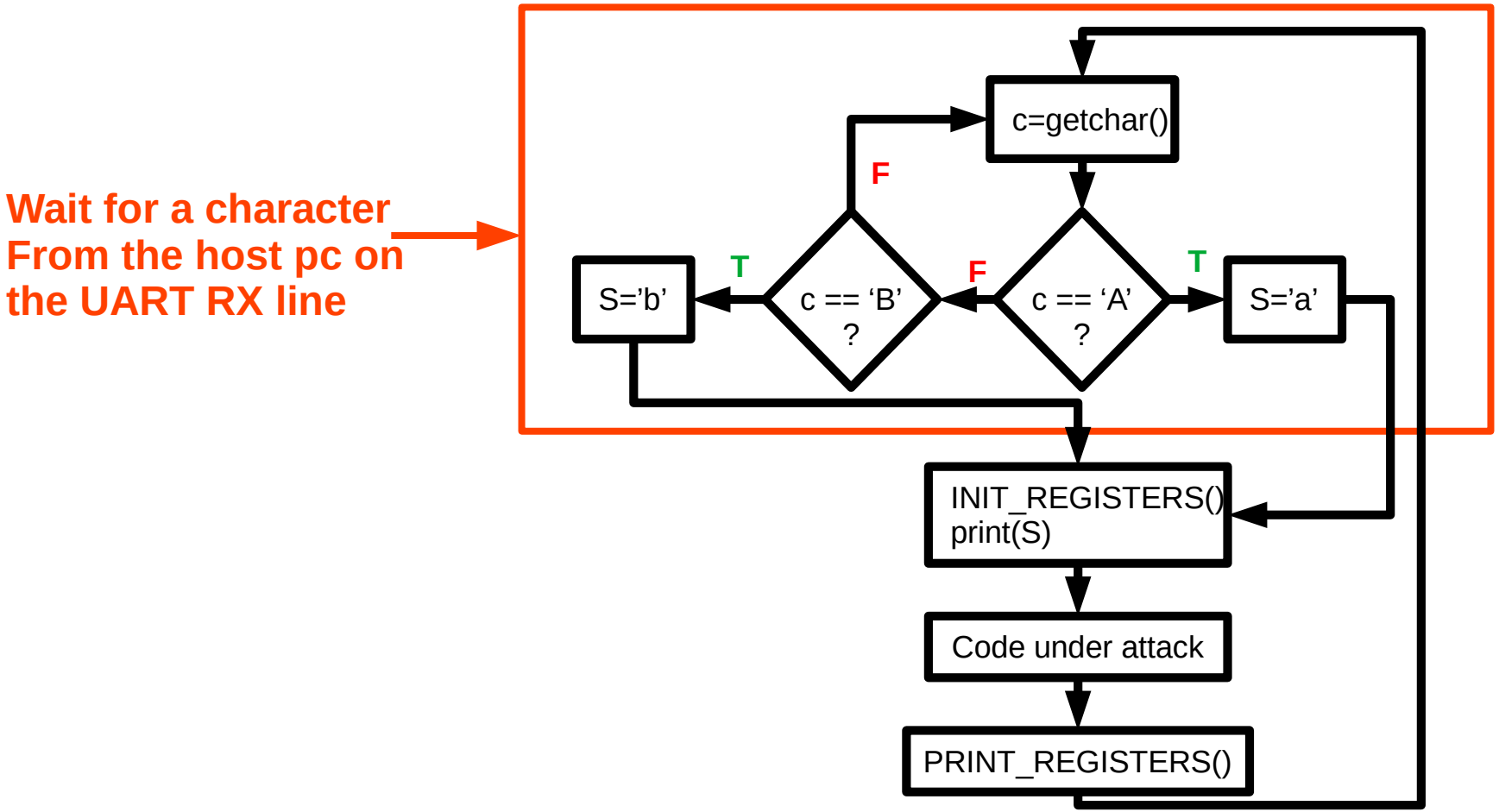
Package-on-Package SoC

Code under attack inserted into the emergency program.
The UART is used instead of GPIO to trigger the Fault Injection.



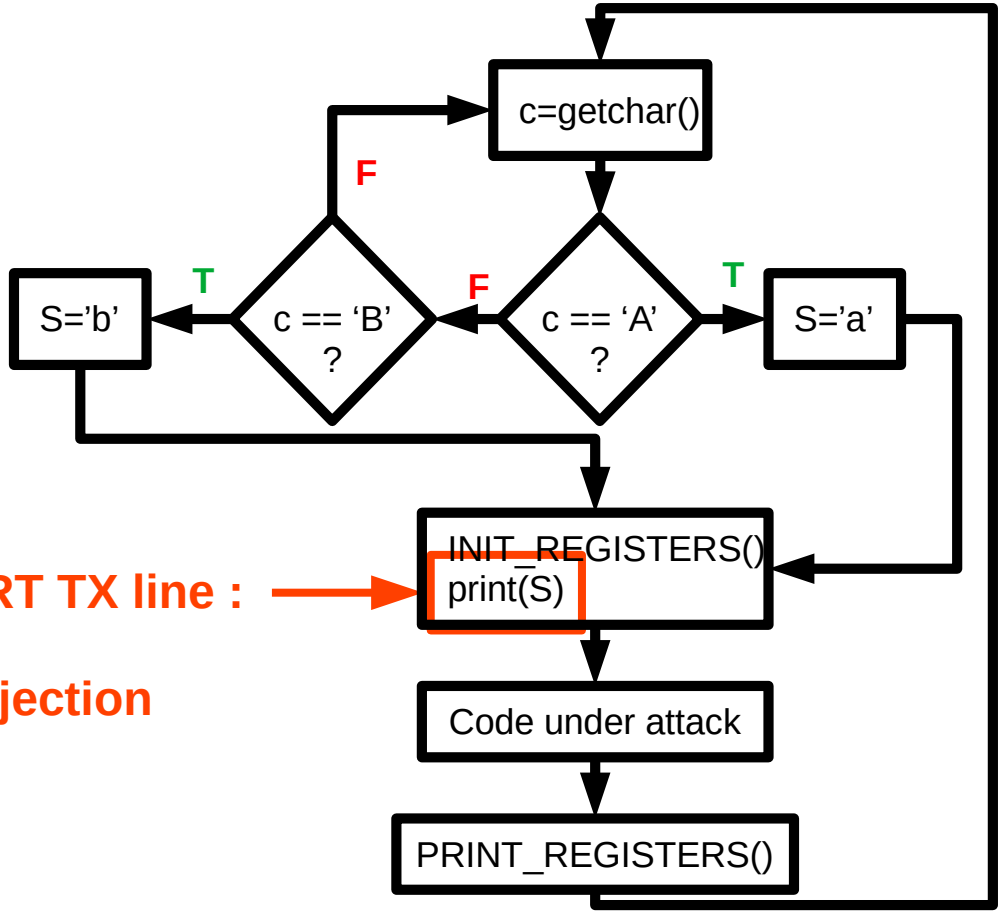
Package-on-Package SoC

Code under attack inserted into the emergency program.
The UART is used instead of GPIO to trigger the Fault Injection.



Package-on-Package SoC

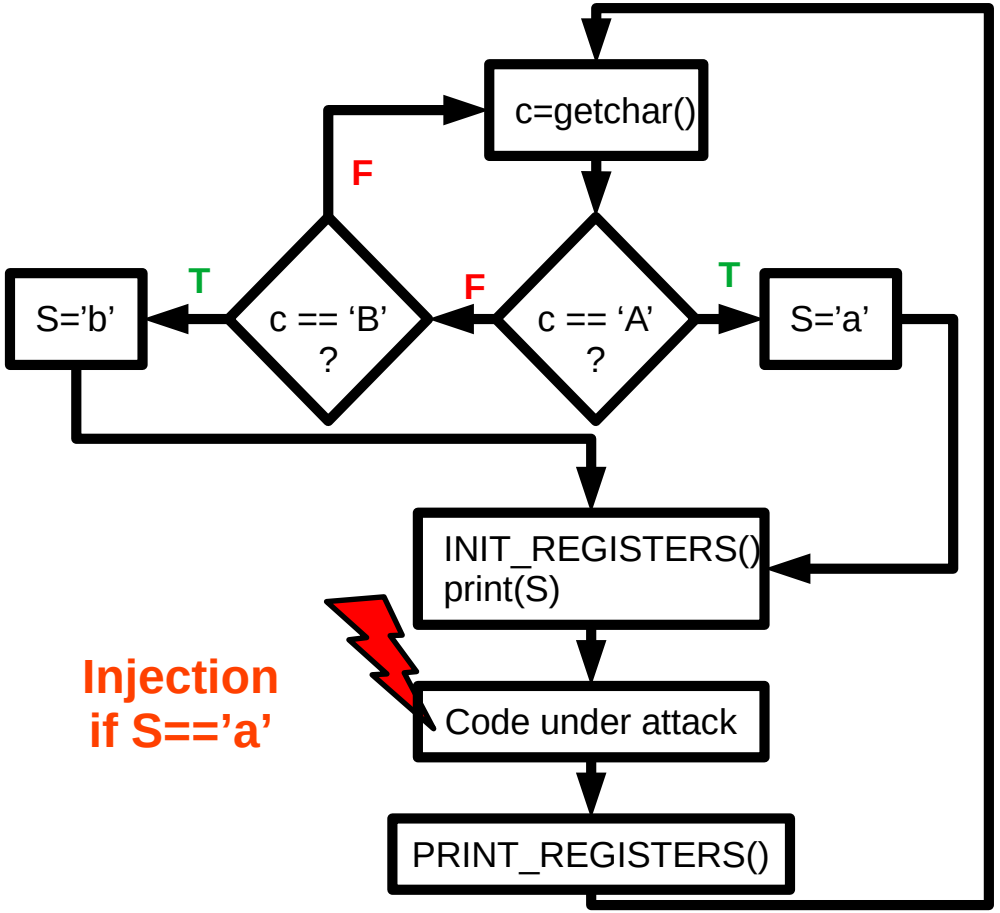
Code under attack inserted into the emergency program.
The UART is used instead of GPIO to trigger the Fault Injection.



Character send on the UART TX line :
- 'a' ⇒ Trigger the injection
- 'b' ⇒ Do not trigger the injection

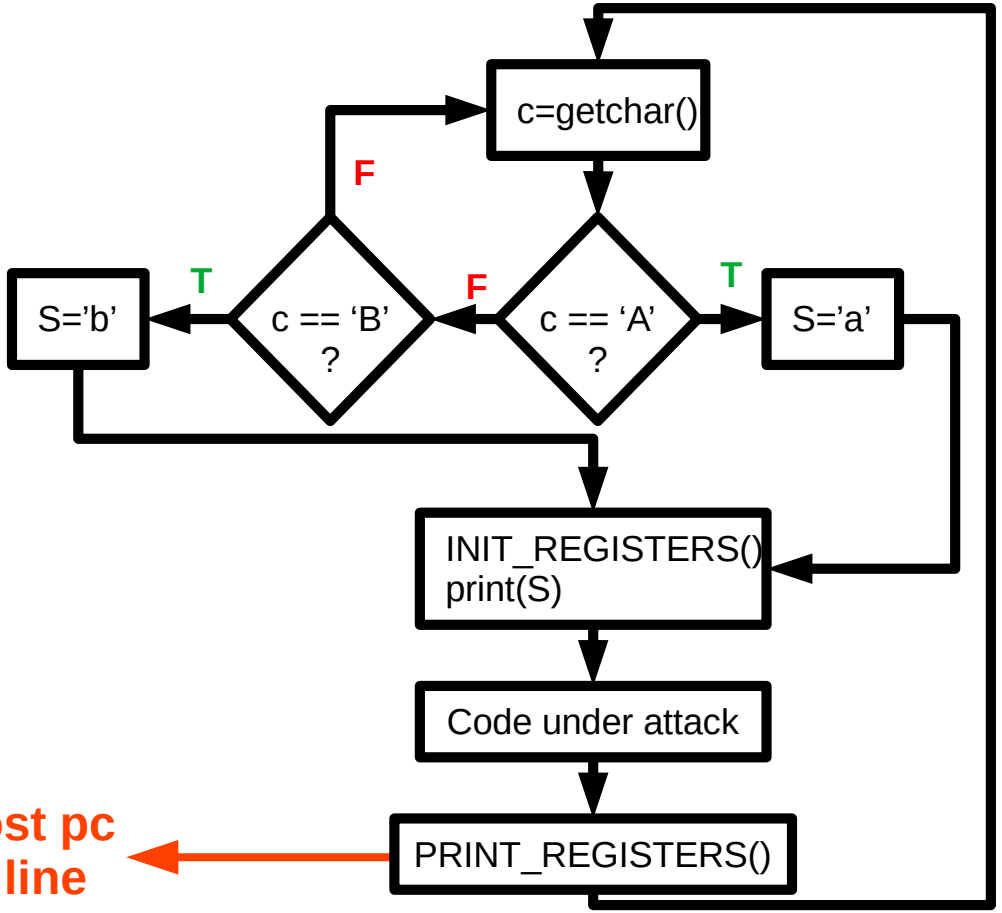
Package-on-Package SoC

Code under attack inserted into the emergency program.
The UART is used instead of GPIO to trigger the Fault Injection.



Package-on-Package SoC

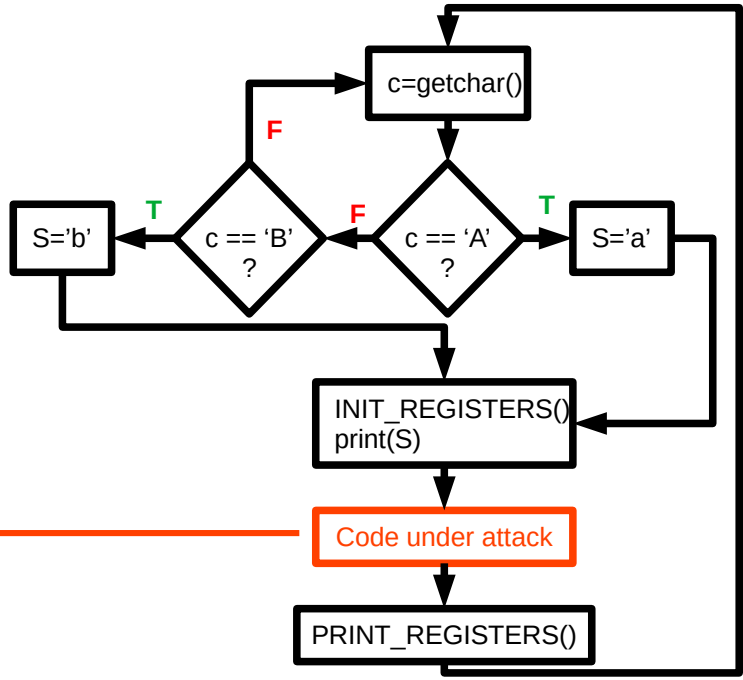
Code under attack inserted into the emergency program.
The UART is used instead of GPIO to trigger the Fault Injection.



Results sent to the host pc through the UART TX line

Package-on-Package SoC

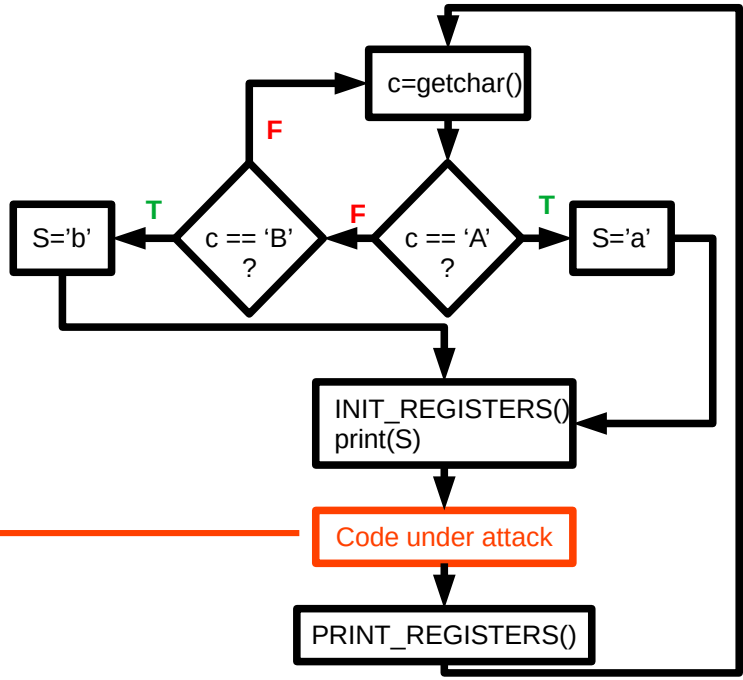
C pseudocode	ASM pseudocode
while w0>0 do	jmp0: sub w5, w6, #1
w5 ← w6 − 1	sub w4, w5, #1
w4 ← w5 − 1	sub w3, w4, #1
w3 ← w4 − 1	sub w2, w3, #1
w2 ← w3 − 1	sub w6, w2, #1
w6 ← w2 − 1	sub w0, w0, #1
w1 ← w1 − 1	sub w1, w1, #1
w0 ← w0 − 1	cmp w0, w1
if w0!=w1 then	bne fault
w10 ← 0x5555	cmp w0, #0
break	bhi jmp0
end if	b end
end while	fault: mov w10, #0x5555
	end: nop



Package-on-Package SoC

Sub

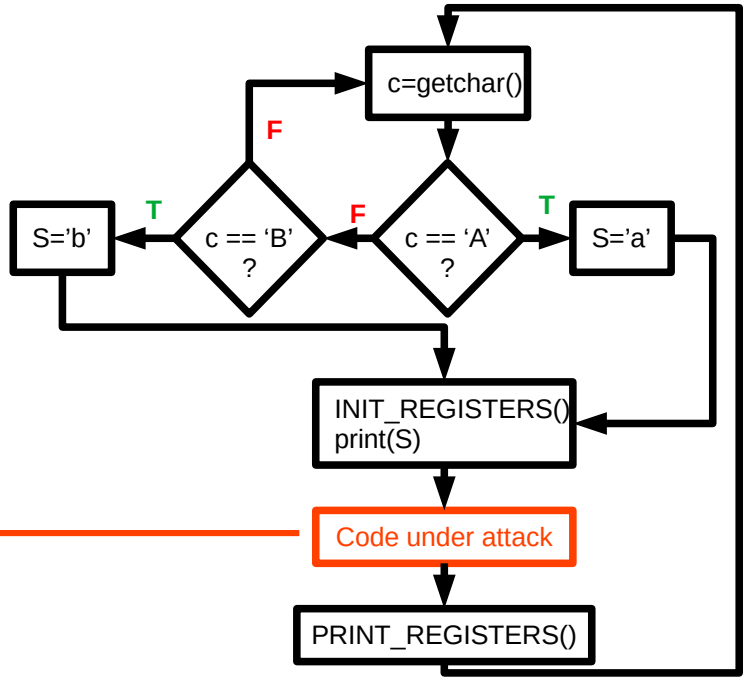
C pseudocode	ASM pseudocode
while w0>0 do	jmp0: sub w5, w6, #1
w5 ← w6 − 1	sub w4, w5, #1
w4 ← w5 − 1	sub w3, w4, #1
w3 ← w4 − 1	sub w2, w3, #1
w2 ← w3 − 1	sub w6, w2, #1
w6 ← w2 − 1	sub w0, w0, #1
w1 ← w1 − 1	sub w1, w1, #1
w0 ← w0 − 1	cmp w0, w1
if w0!=w1 then	bne fault
w10 ← 0x5555	cmp w0, #0
break	bhi jmp0
end if	b end
end while	fault: mov w10, #0x5555
	end: nop



Package-on-Package SoC

C pseudocode	ASM pseudocode
while w0>0 do	jmp0: sub w5, w6, #1
w5 ← w6 − 1	sub w4, w5, #1
w4 ← w5 − 1	sub w3, w4, #1
w3 ← w4 − 1	sub w2, w3, #1
w2 ← w3 − 1	sub w6, w2, #1
w6 ← w2 − 1	sub w0, w0, #1
w1 ← w1 − 1	sub w1, w1, #1
w0 ← w0 − 1	cmp w0, w1
if w0!=w1 then	bne fault
w10 ← 0x5555	cmp w0, #0
break	bhi jmp0
end if	b end
end while	fault: mov w10, #0x5555
	end: nop

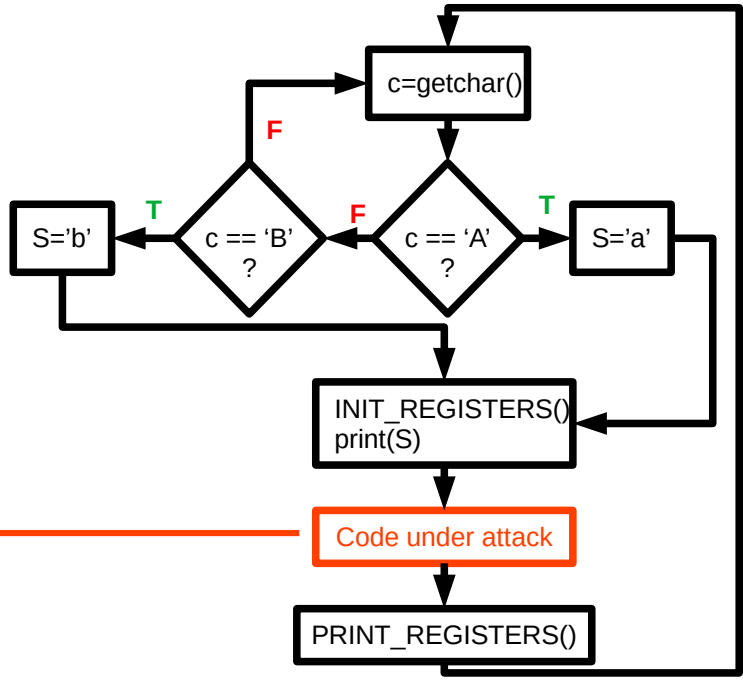
Decrement counters



Package-on-Package SoC

C pseudocode	ASM pseudocode
while w0>0 do	jmp0: sub w5, w6, #1
w5 ← w6 − 1	sub w4, w5, #1
w4 ← w5 − 1	sub w3, w4, #1
w3 ← w4 − 1	sub w2, w3, #1
w2 ← w3 − 1	sub w6, w2, #1
w6 ← w2 − 1	sub w0, w0, #1
w1 ← w1 − 1	sub w1, w1, #1
w0 ← w0 − 1	cmp w0, w1
if w0!=w1 then	bne fault
w10 ← 0x5555	cmp w0, #0
break	bhi jmp0
end if	b end
end while	fault: mov w10, #0x5555
	end: nop

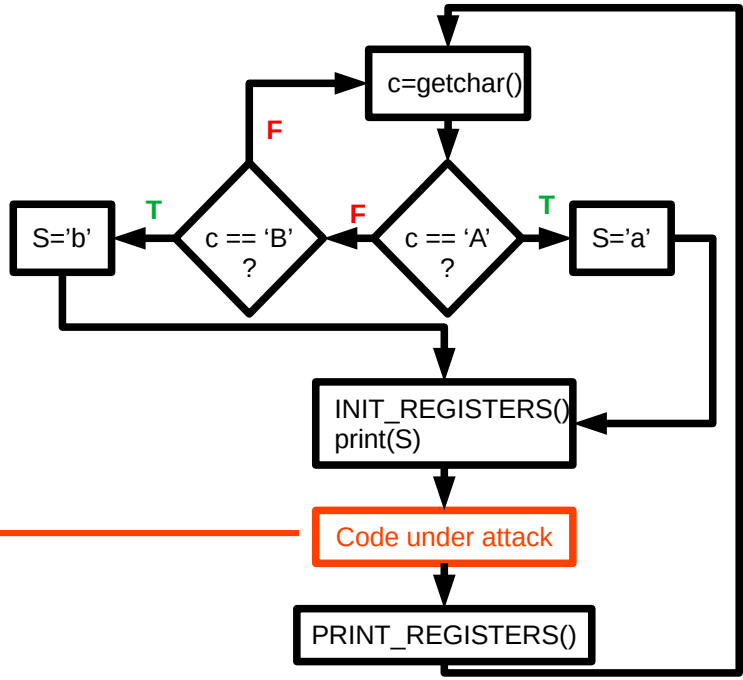
Check if counter are synchronized



Package-on-Package SoC

End if one counter is 0

C pseudocode	ASM pseudocode
while w0>0 do	jmp0: sub w5, w6, #1
w5 ← w6 − 1	sub w4, w5, #1
w4 ← w5 − 1	sub w3, w4, #1
w3 ← w4 − 1	sub w2, w3, #1
w2 ← w3 − 1	sub w6, w2, #1
w6 ← w2 − 1	sub w0, w0, #1
w1 ← w1 − 1	sub w1, w1, #1
w0 ← w0 − 1	cmp w0, w1
if w0!=w1 then	bne fault
w10 ← 0x5555	cmp w0, #0
break	bhi jmp0
end if	b end
end while	fault: mov w10, #0x5555
	end: nop



Package-on-Package SoC

Fault model identified:
(1) Instruction skip ⇒ F1, F2, F3, F4

ASM pseudocode	
jmp0: sub w5, w6, #1	
sub w4, w5, #1	
sub w3, w4, #1	F1
sub w2, w3, #1	
sub w6, w2, #1	
sub w0, w0, #1	F2
sub w1, w1, #1	
cmp w0, w1	F3
bne fault	
cmp w0, #0	F4
bhi jmp0	
b end	
fault: mov w10, #0x5555	
end: nop	

The different fault model can be identified by analysing the registers values that are sent to the host pc.

Package-on-Package SoC

Fault model identified:

(1) Instruction skip ⇒ F1, F2, F3, F4

(2) Other faults impacting the same instruction as (1) ⇒ F5, F6, F7, F8

ASM pseudocode	
jmp0: sub w5, w6, #1	
sub w4, w5, #1	
sub w3, w4, #1	F1, F5
sub w2, w3, #1	
sub w6, w2, #1	
sub w0, w0, #1	F2, F6
sub w1, w1, #1	
cmp w0, w1	F3, F7
bne fault	
cmp w0, #0	F4, F8
bhi jmp0	
b end	
fault: mov w10, #0x5555	
end: nop	

The different fault model can be identified by analysing the registers values that are sent to the host pc.

Package-on-Package SoC

Fault model identified:

(1) Instruction skip ⇒ F1, F2, F3, F4

(2) Other faults impacting the same instruction as (1) ⇒ F5, F6, F7, F8

ASM pseudocode

```
jmp0: sub w5, w6, #1
sub w4, w5, #1
sub w3, w4, #1
sub w2, w3, #1
sub w6, w2, #1
sub w0, w0, #1
sub w1, w1, #1
cmp w0, w1
bne fault
cmp w0, #0
bhi jmp0
b end
fault: mov w10, #0x5555
end: nop
```

F1, F5

F2, F6

F3, F7

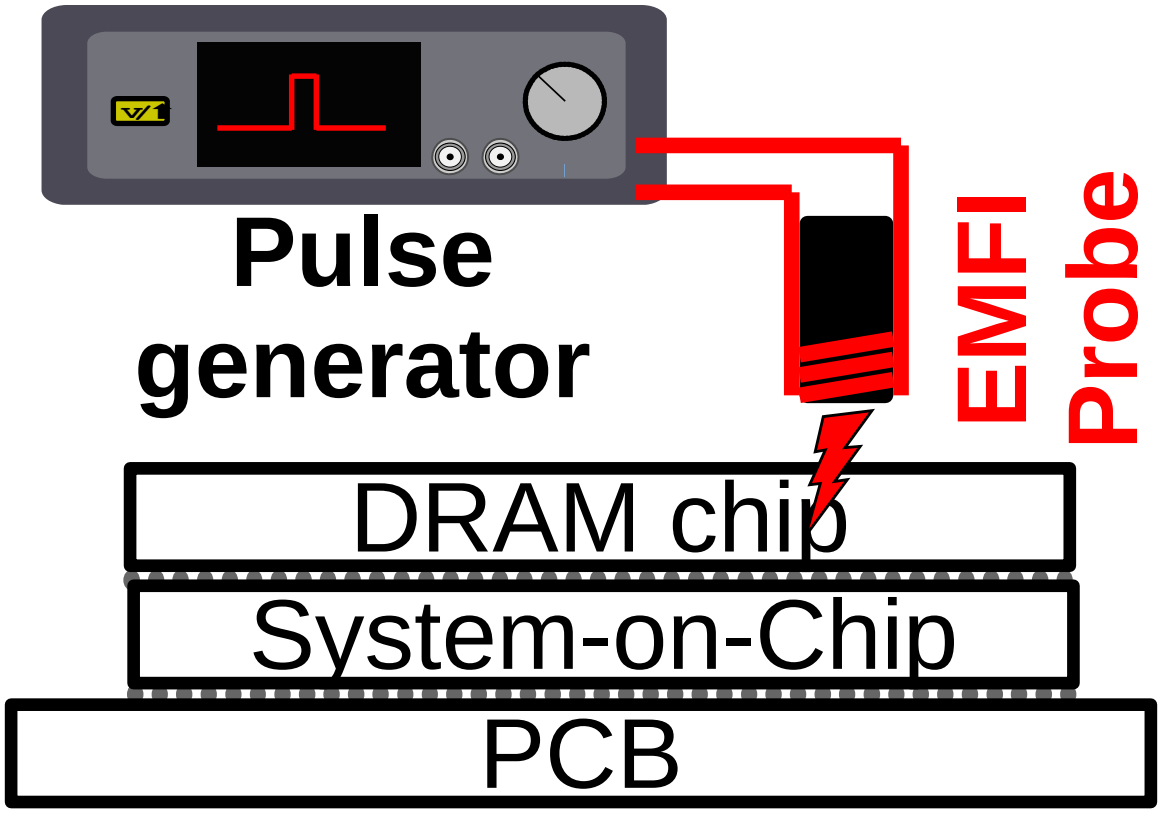
F4, F8

The different fault model can be identified by analysing the registers values that are sent to the host pc.

Faults that do not enter in these faults models are classified in **F.9**

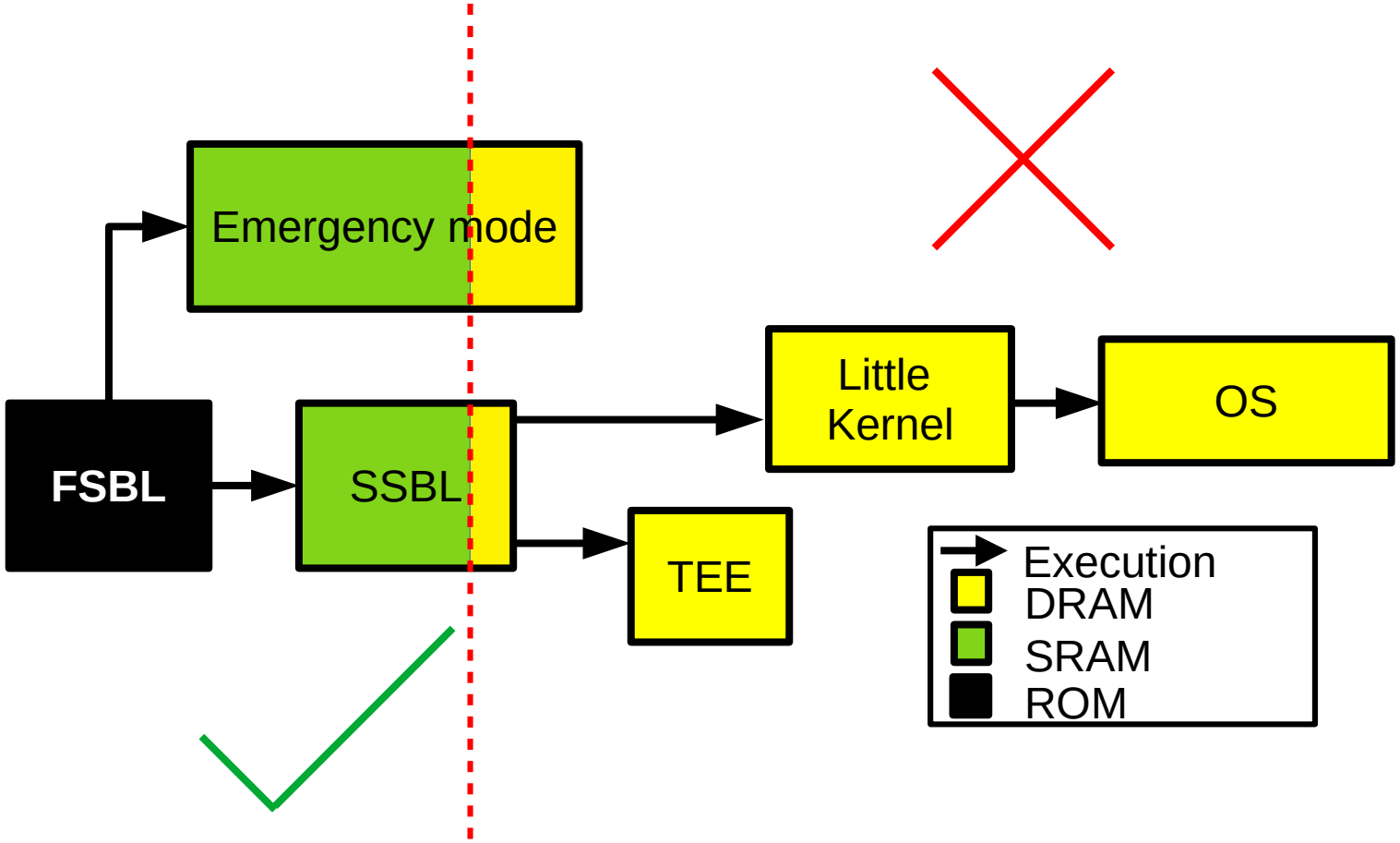
Package-on-Package SoC

Method 1: Conventional EMFI above the SoC with the PoP DRAM.
Reducing the target supply voltage might help injecting faults.



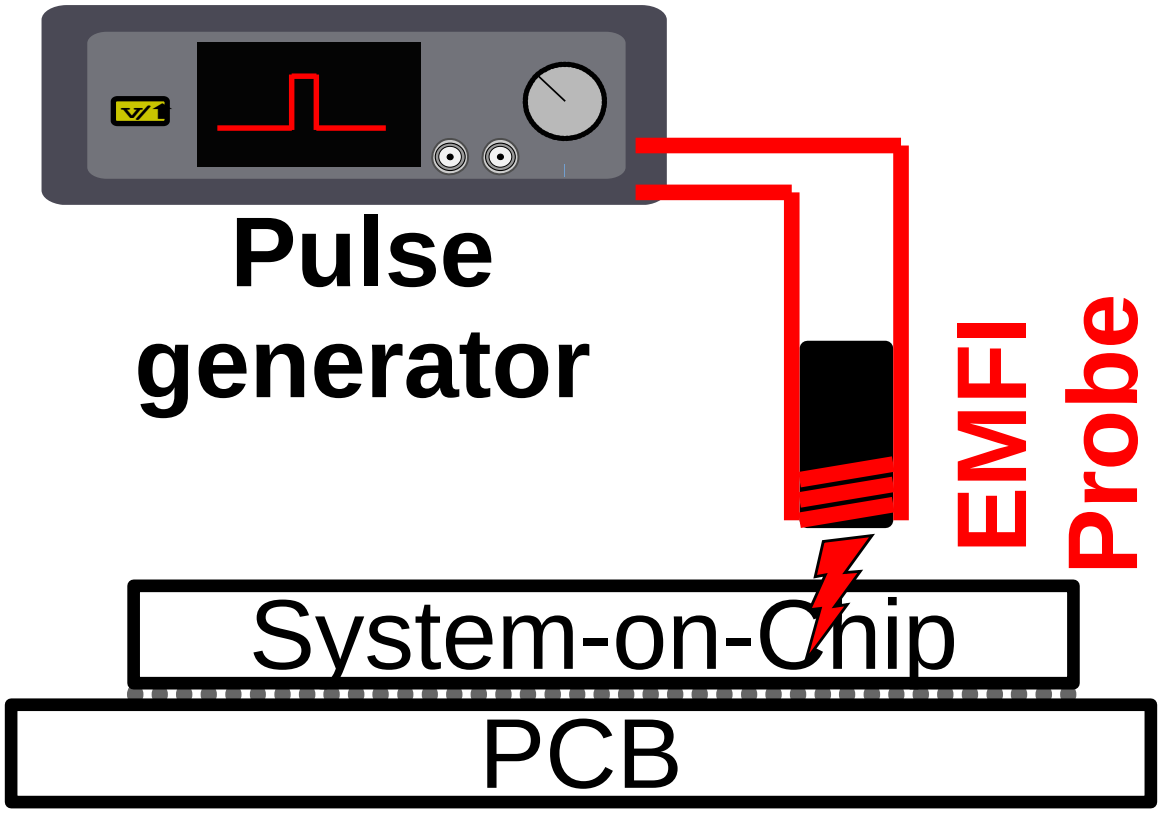
Package-on-Package SoC

Method 2: Conventional EMFI above the SoC without the PoP DRAM.
Vasselle et al. (2017) show that the first stages of the boot run correctly even if the DRAM has been removed.



Package-on-Package SoC

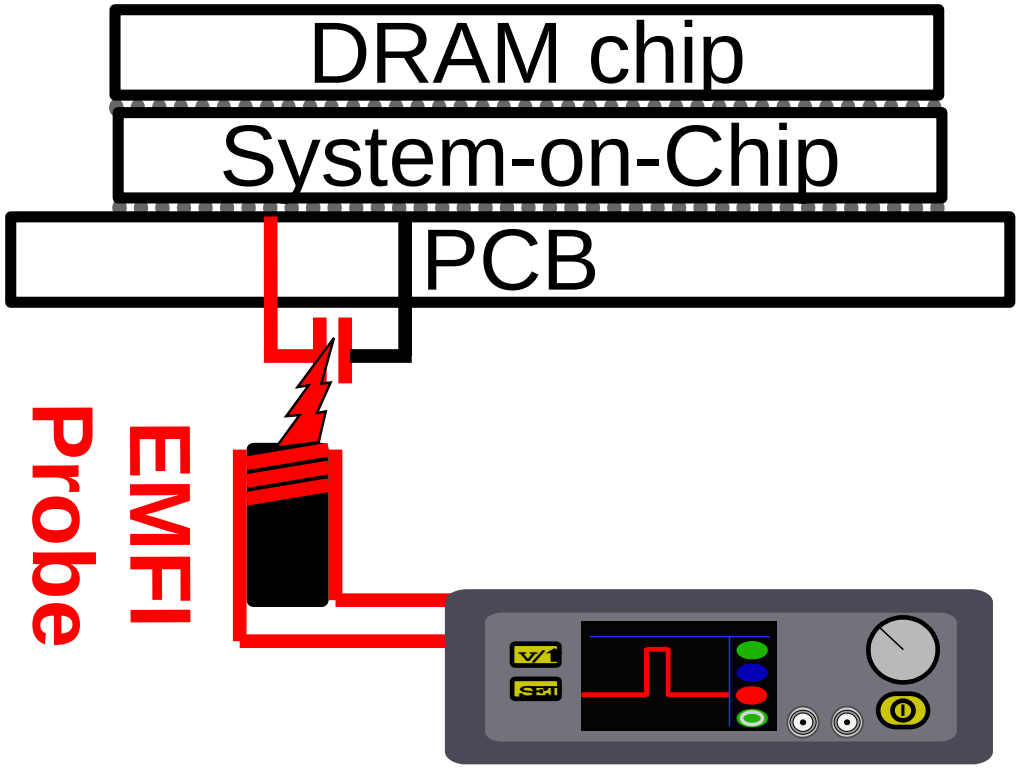
Method 2: Conventional EMFI above the SoC without the PoP DRAM.
Vasselle et al. (2017) show that the first stages of the boot run correctly even if the DRAM has been removed.



Package-on-Package SoC

Method 3: EM-induced and conventional voltage glitches

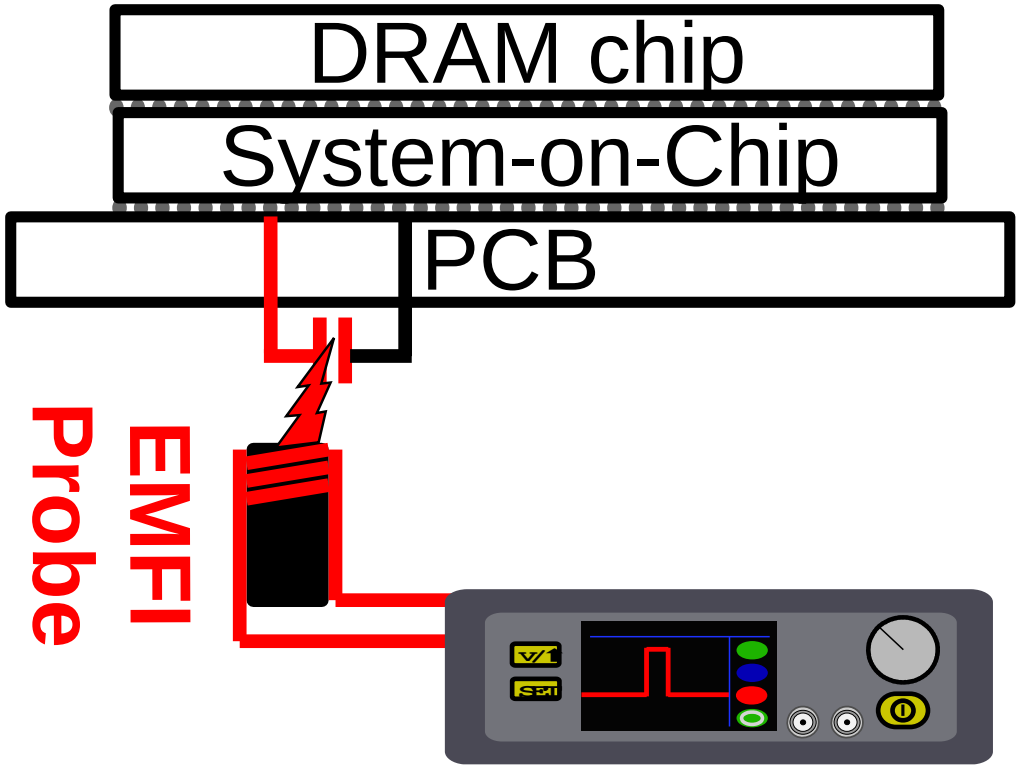
EM-induced glitches :
The idea is to induce voltage glitches on the target supply voltage line by using EM pulses.



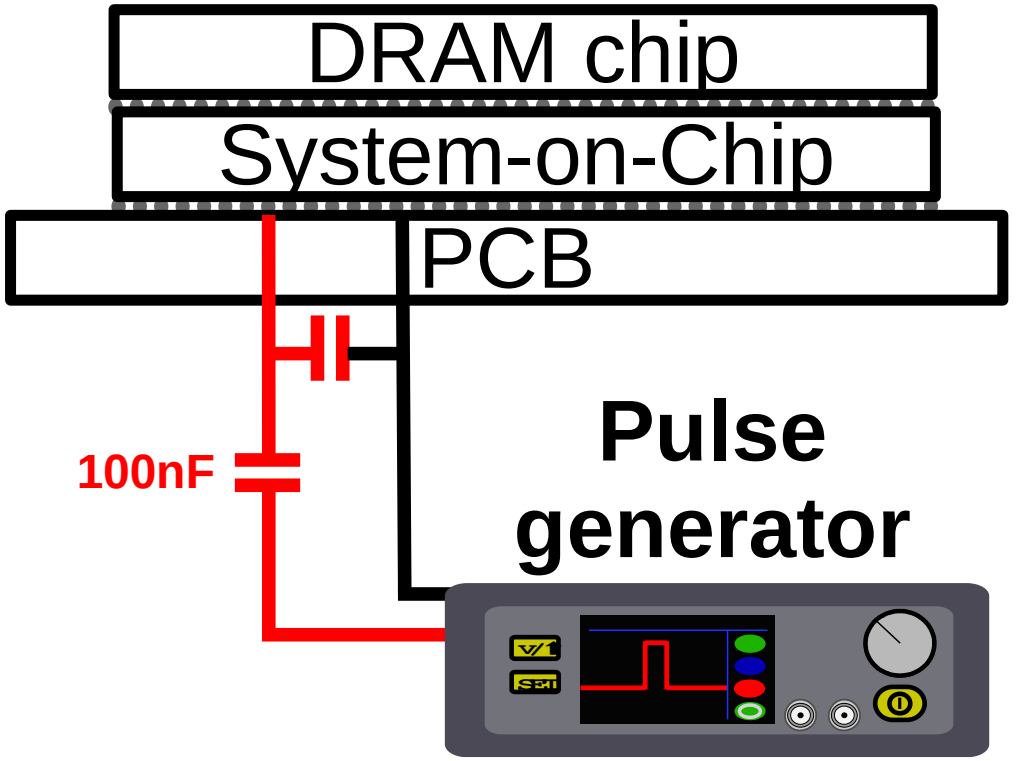
Package-on-Package SoC

Method 3: EM-induced and conventional voltage glitches

EM-induced glitches :
The idea is to induce voltage glitches on the target supply voltage rail by using EM pulses.

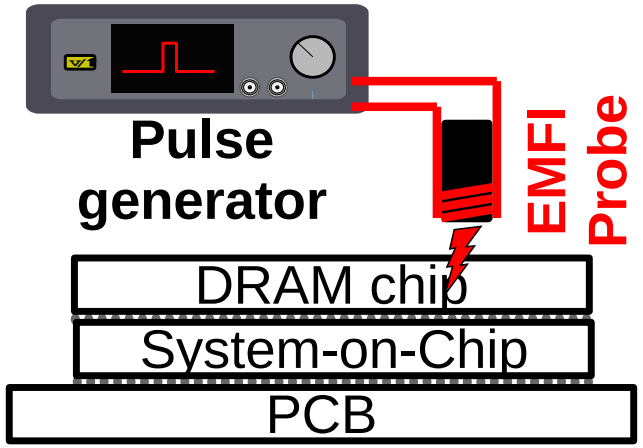


Conventional voltage glitches :
A negative pulse is injected in a 100nF capacitor soldered on the supply voltage rail.



Package-on-Package SoC

Method 1: Conventional EMFI above the SoC with the PoP DRAM :



Nominal supply voltage : 950mV
Operational limit : 730mV

F1, F2, … F9 ⇒ Fault models identified

	$\frac{F.1}{F_{tot}}$	$\frac{F.2}{F_{tot}}$	$\frac{F.3}{F_{tot}}$	$\frac{F.4}{F_{tot}}$	$\frac{F.5}{F_{tot}}$	$\frac{F.6}{F_{tot}}$	$\frac{F.7}{F_{tot}}$	$\frac{F.8}{F_{tot}}$	$\frac{F.9}{F_{tot}}$	$\frac{F_{tot}}{Total}$	Number of experiment (Total)
950mV XY map	No Faults	No Faults	No Faults	No Faults	No Faults	No Faults	No Faults	No Faults	No Faults	No Faults	260100
730mV XY map	**93.3 %**	0.60 %	0.65 %	0.03 %	3.22 %	1.61 %	0.35 %	0 %	0.2 %	1.11 %	1125000

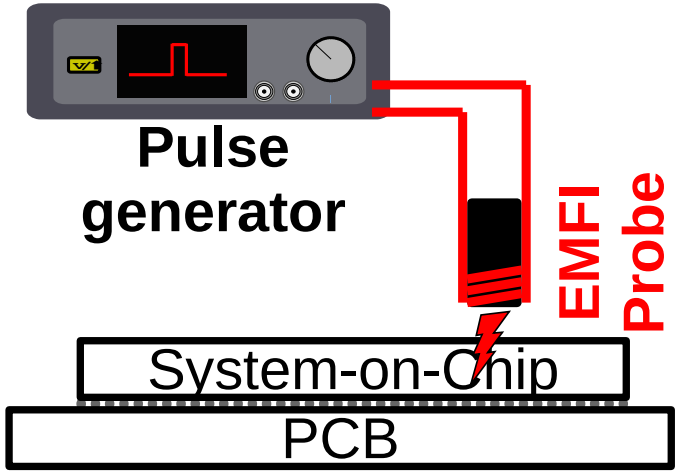
- Faults are injected when reducing the supply voltage to its operational limit.
- However, faults are injected even during test and when no EM pulses are injected.
- The ratio faults/fault_during_test stays the same for all pulse amplitude and XY position.

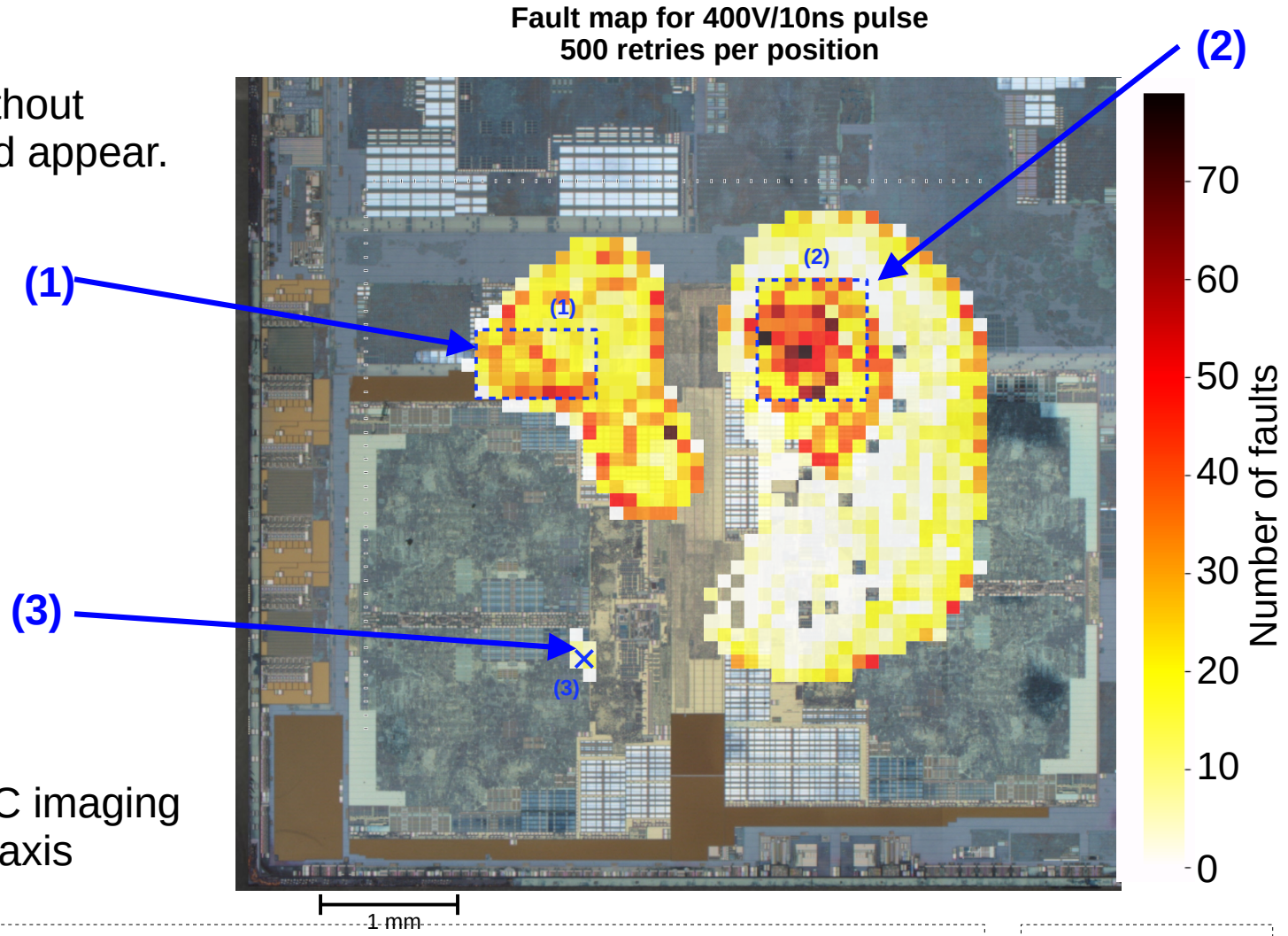
The faults injected are caused by the underpowering and not by the EMFI

Package-on-Package SoC

Method 2: Conventional EMFI above the SoC without the PoP DRAM.

When injecting faults above the SoC without DRAM, 3 areas where faults are injected appear.





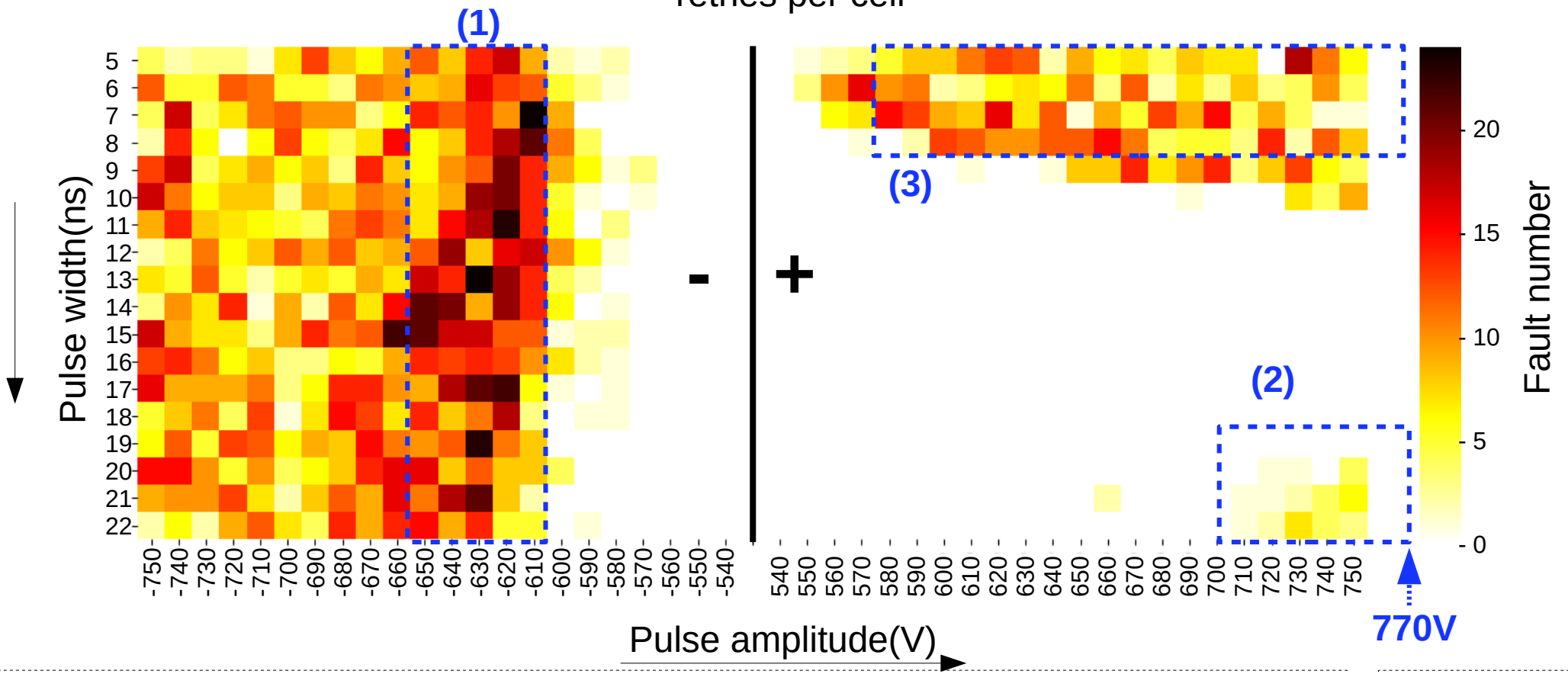
XY Fault Map over the SoC imaging
100um step, fixed Z axis

Package-on-Package SoC

Method 3: EM-induced and conventional voltage glitches

EM-induced glitches:

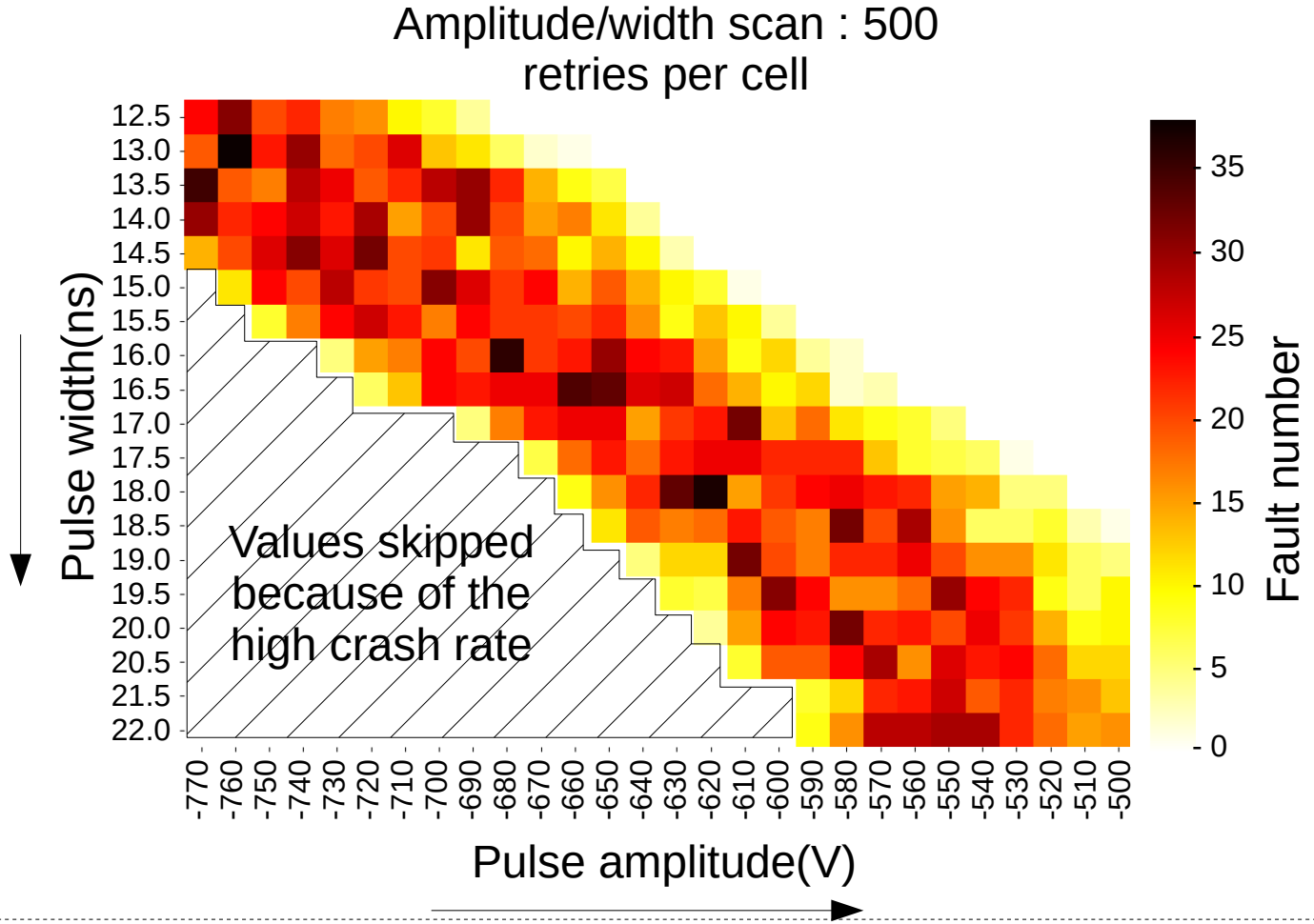
Amplitude/width scan : 100 retries per cell



Package-on-Package SoC

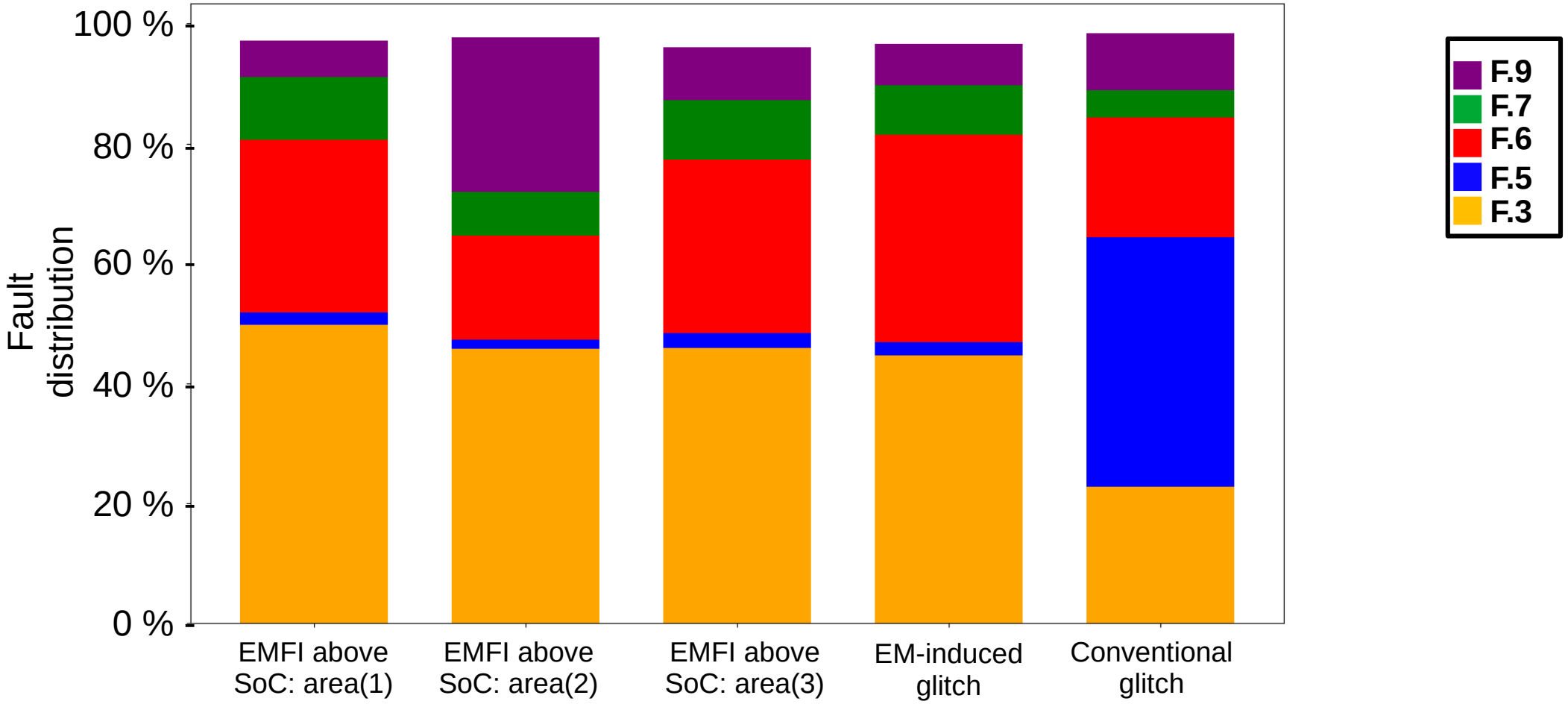
Method 3: EM-induced and conventional voltage glitches

Conventional voltage glitches:



Package-on-Package SoC

Fault distribution for each Fault Injection method



F1-F9 ⇒ Fault models that can impact the the code under attack

Package-on-Package SoC : Conclusion

Package-on-Package SoC ⇒ Shield against EMFI

- Presentation of 3 solutions to inject faults in a SoC despite the PoP DRAM:
  - EMFI above the SoC, the target DRAM has to be removed as presented by **Vasselle et al. (2017)**.
  - EM-induced glitches ⇒ new method which consists in injecting EM pulses above the decoupling capacitors to induce voltage glitches on the cores supply voltage rail.
  - Conventional voltage glitch

- Comparison of the 3 methods :

EMFI ≈ EM-induced glitch ≠ Voltage glitch

Conclusion

Problematic of Fault Injection attacks on System-on-Chip explored:

- **Attack synchronization**
  ⇒ Presentation of a Side-Channel based methodology to synchronize a Fault Injection upon the activation of a specific frequency
- **Vulnerability identification and timing**
  ⇒ Presentation of a methodology to identify vulnerabilities to Fault-Injection in black-box
- **Target Packaging (Package-on-Package)**
  ⇒ Exploration of methods to inject faults in a SoC using a PoP DRAM
  - Adaptation of an existing method : remove the DRAM
  - Conventional voltage glitch
  - New method to induce voltage glitches using EM pulses : **EM-induced glitches**

- Fault Injection parameters identification in black-box⇒ Next work

Conclusion

Problematic of Fault Injection attacks on System-on-Chip explored:

- **Attack synchronization**
  ⇒ CARDIS2022 (Best student paper) :
  'Combined Fault Injection and Real-Time Side-Channel Analysis for Android Secure-Boot Bypassing'
  ⇒ Presentation of this work at JAIF2022
- **Vulnerability identification and timing**
  ⇒ DFT2023 :
  'Exploration of System-on-Chip Secure-Boot Vulnerability to Fault-Injection by Side-Channel Analysis'
- **Target Packaging (Package-on-Package)**
  ⇒ Paper submitted to Usenix Security 2024 : eprint soon

Questions ?

Clément Fanjas

clement.fanjas@cea.fr

Thank you for your attention