How to exploit EMFI to bypass the Secure-Boot of SoC

Driss ABOULKASSIMI | driss.aboulkassimi@cea.fr

Simon PONTIE | simon.pontie@cea.fr

Clément FANJAS | clement.fanjas@cea.fr

Olivier POTIN | potin@emse.fr

Une école de l'IMT

Agenda

1. Context & objective
2. EMFI experimental set-up
3. Hardware and Software targets
4. EMFI from µController to SoC
5. Synchronization: issue and solutions
6. Demo in video
7. Conclusion

Context and objective

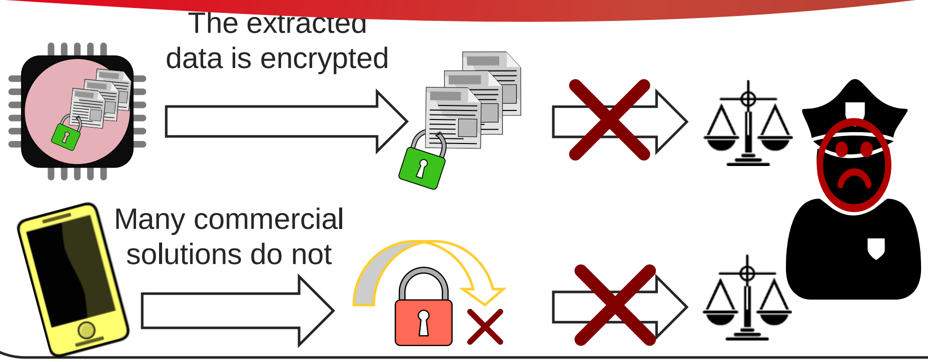
CONTEXT

Mobile phones is a key factor in criminal cases, intrusions, IP, security threats, and more.
The d[...]
may c[...]
is encrypted!

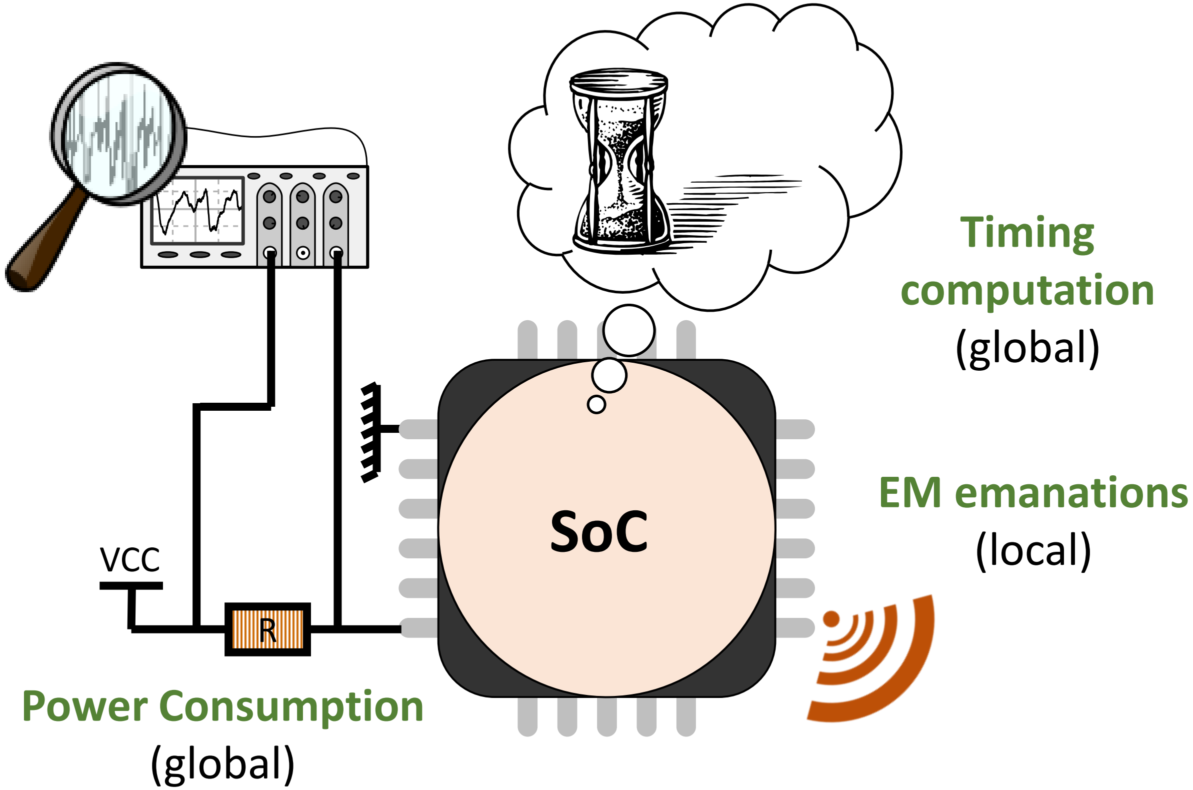
OBJECTIVE

How to bypass the encryption issue in
[...] e

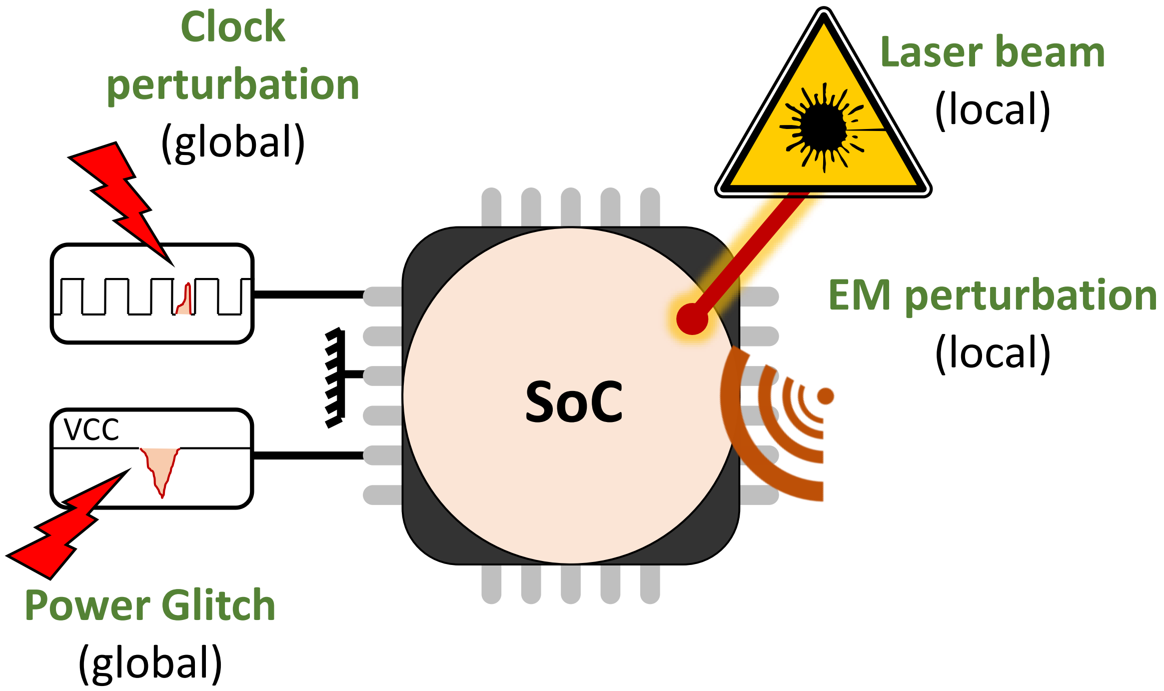
Security through encryption and security despite encryption



Hardware attacks



Side Channel Analysis (SCA)



Fault Injection (FI)

Physical access to the target is required: possible in all judiciary investigation cases

Hardware target: SoC complexe device/ packaging

Smartphone System-on-Chip on dev-board:
- CPU: quad-core ARM Cortex A53
- Maximum frequency:1.2GHz
- Running frequency during the boot: 800MHz

- Previous work (**Gaine et al. 2020**):
  Using EMFI is possible on SoC boar: by skipping instruction

- Previous work (**Fanjas et al. 2022**): => best student paper CARDIS'2022
  Combined EMFI & SCA attacks to bypass the secure boot of SoC

Software target: the secure boot

What is the Secure Boot ?

Chain of trust where each high privilege program is authenticated before being executed

Why it is important ?

To avoid the running of malicious program with high privilege

Software target: the secure boot mechanism

Boot

First Stage /Primary of Boot Loader (FSBL/PBL): executes the ROM code by:

- Loading the SSBL from the eMMC
- Authenticating the SSBL
- Executing the SSBL

SSBL/XBL

Second Stage of the Boot Loader (SSBL/XBL) for:

- Loading LK/Aboot from the eMMC
- Authenticating the LK/Aboot
- Executing the LK/Aboot

LK/Aboot

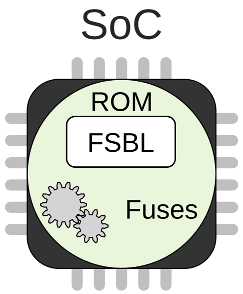
Third boot stage (LK/Aboot):

- Loading the Linux from the eMMC
- Authenticating the Linux Kernel
- Executing the the Linux Kernel

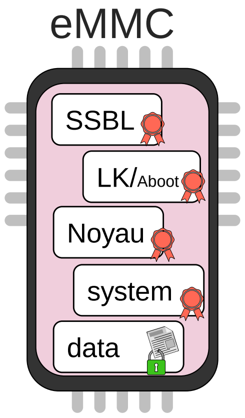
Linux Kernel

Android OS booting

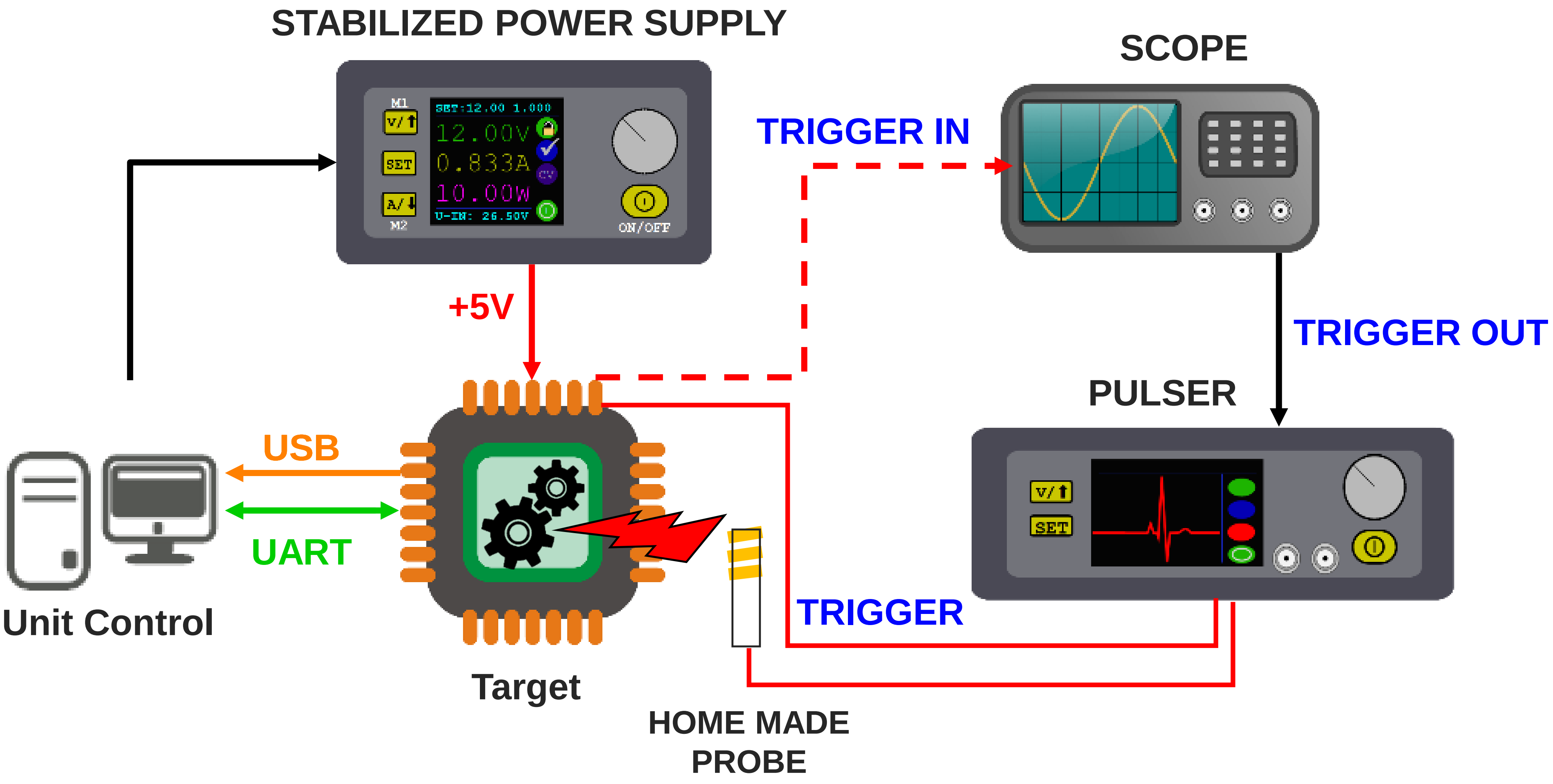
The FSBL/PBL code is located in the SoC ROM



Both codes are implemented in the L2 cache memory or in the RAM, The are loaded from an off-SoC memory



EMFI experimental set-up

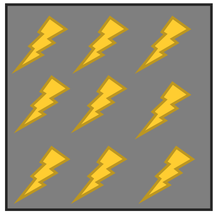


EMFI experimental set-up

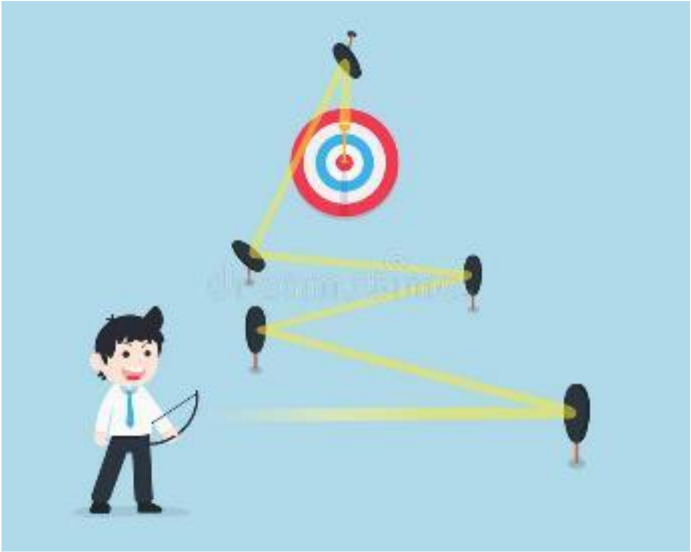
EMFI requests

Where to fire ?

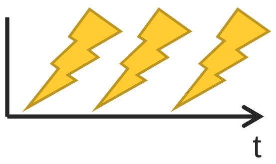
=> On the sensitive location on the to EMFI



When to fire?

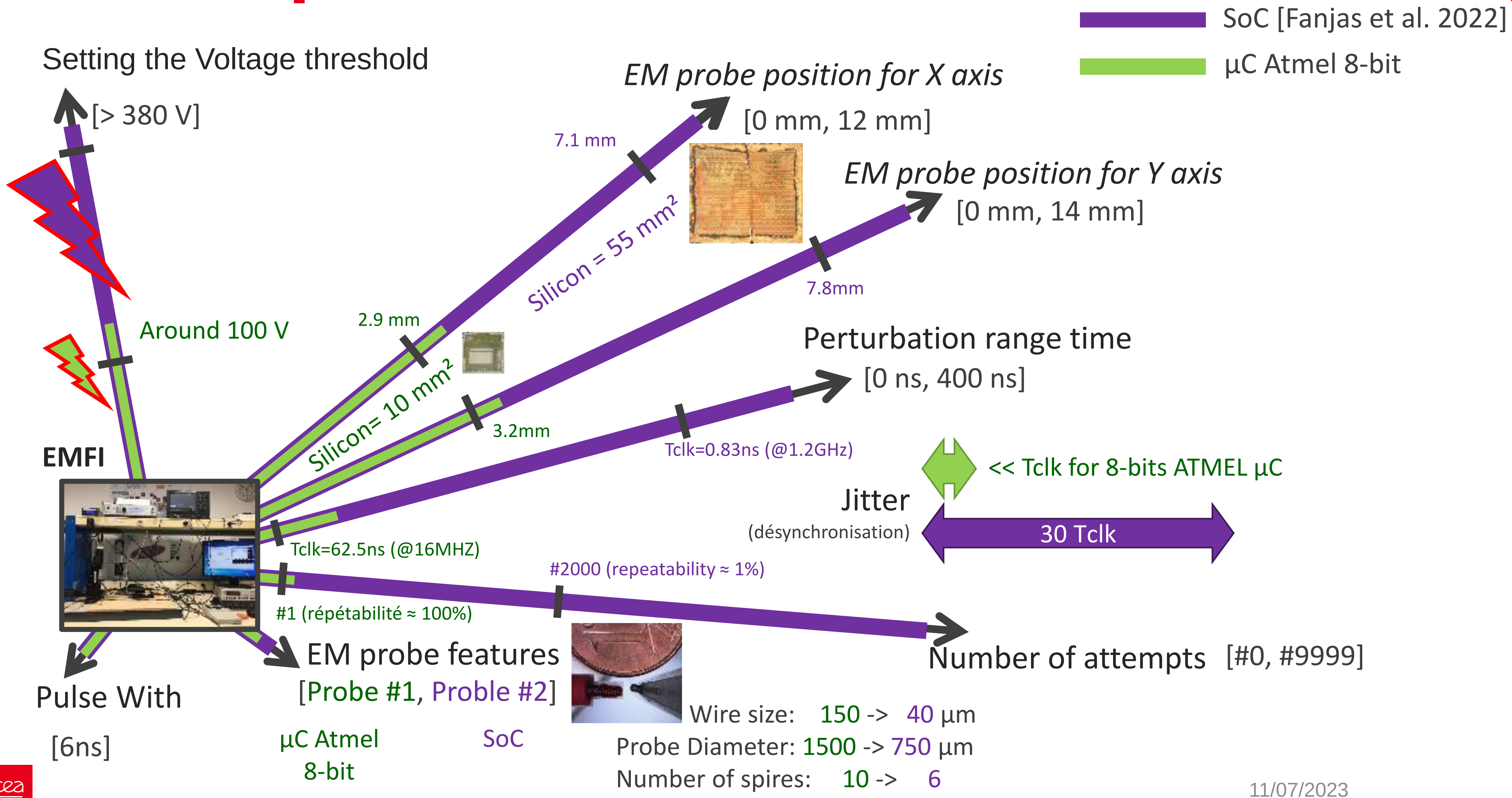


=> The synchronized timing with the targeted vulnerability ?



Both questions are applicable regardless of the target: µC or SoC

EMFI: from µController to SoC



EMFI Process on SoC in 4 steps

Step 1: Vulnerability analysis: identification of a vulnerable instruction in the authentication process

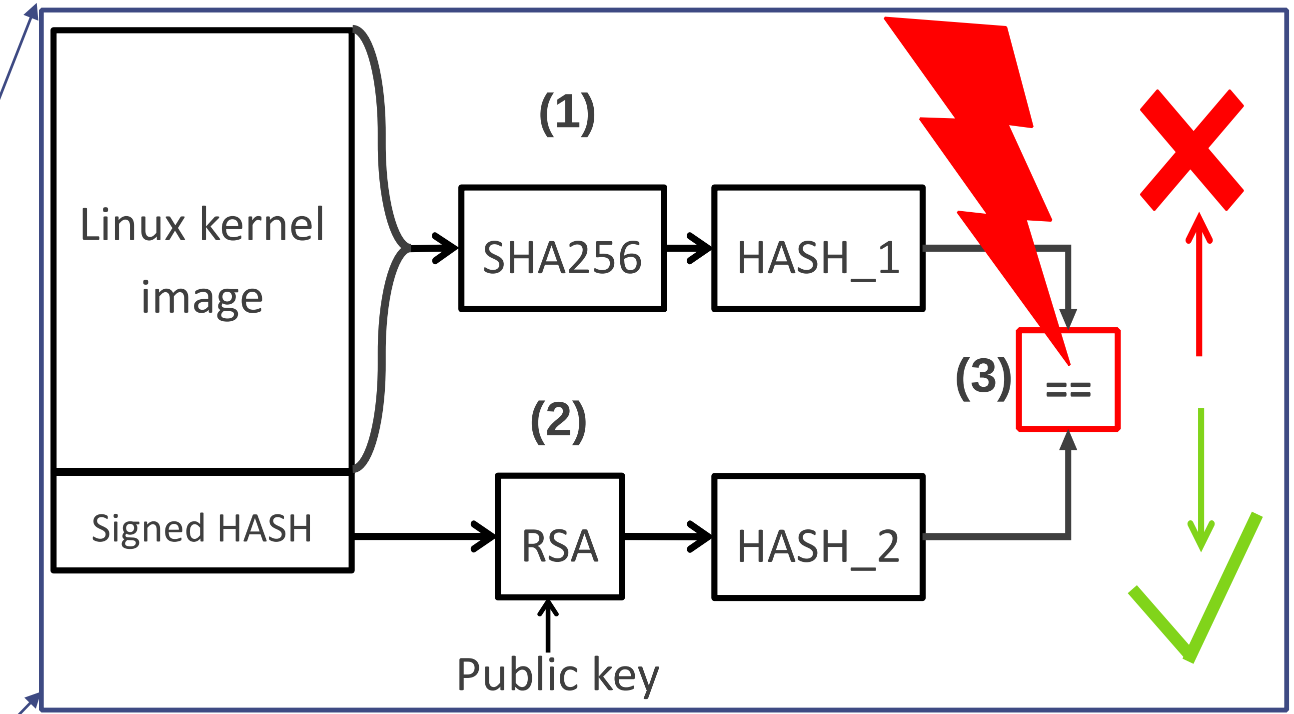
Step 2: Defining the parameters of the EMFI set-up

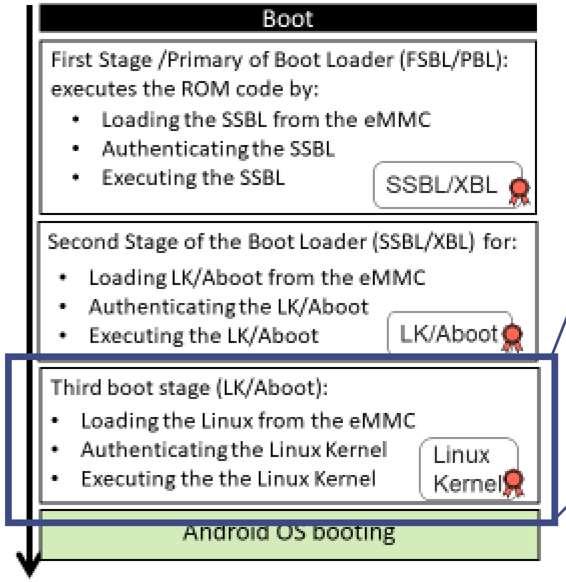
Step 3: Proposing solutions for synchronizing the vulnerability with the EM injection

Step 4: Implementing the EMFI attack

Vulnerability analysis : Linux Kernel Authentication

Step 1: vulnerability analysis
Step 2: EMFI parameters
Step 3: Attack synchronization
Step 4: EMFI implementation





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
```

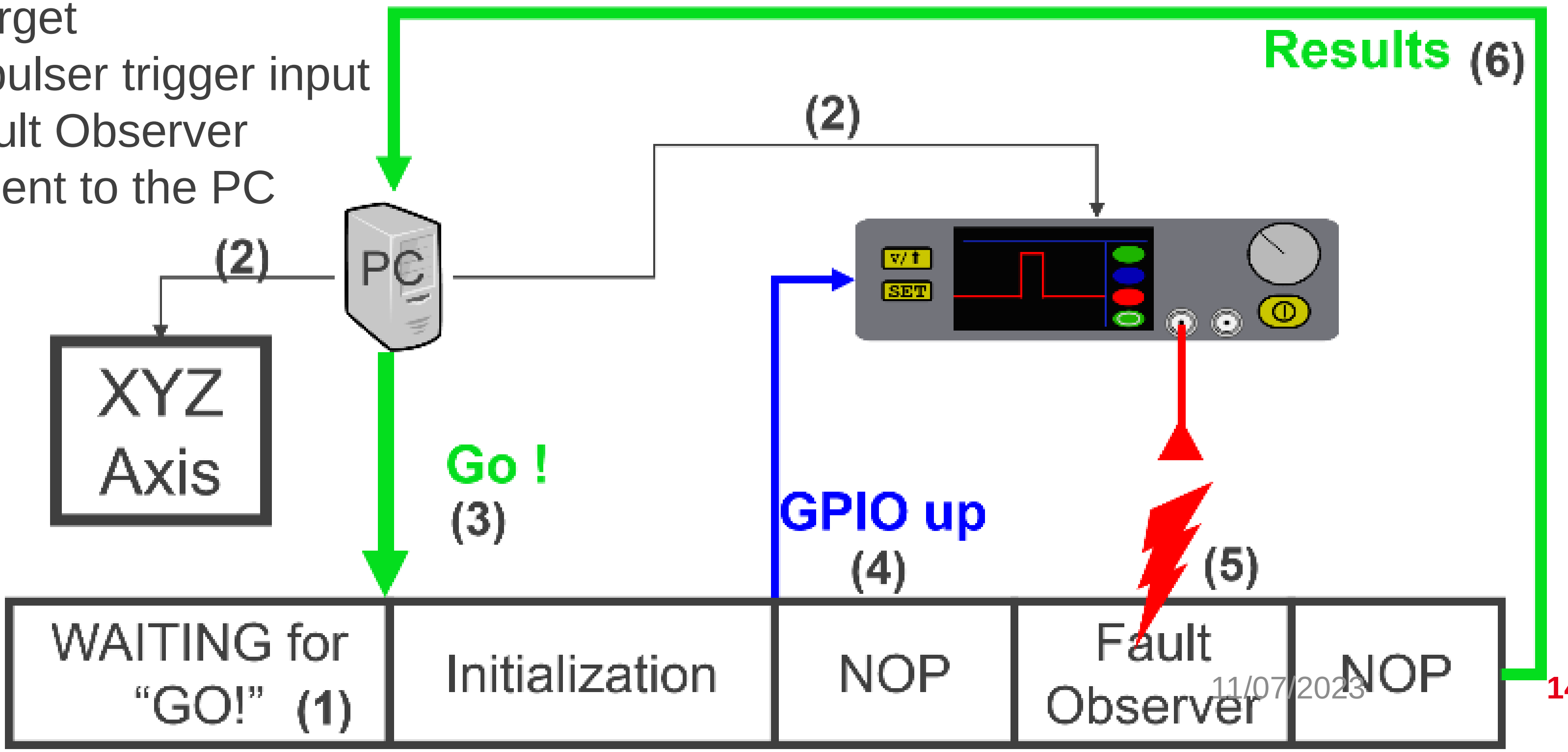
Skipping LSR allows to bypass the LK authentication

How to fire ?

- Step 1: *vulnerability analysis*
- *Step 2: EMFI parameters*
- *Step 3: Attack synchronization*
- *Step 4: EMFI implementation*

- **Executing a computation/program with a known result/output: ‘’Fault Observer”.**
- **Observing the output of this program: testing injection with different parameters**

Methodology:
(1) The target waits for an order
(2) The PC sets the pulse parameters and moves the XYZ axis
(3) The PC sends an order to the target
(4) The target rise a GPIO into the pulser trigger input
(5) A pulse is injected during the Fault Observer
(6) The Fault Observer results are sent to the PC

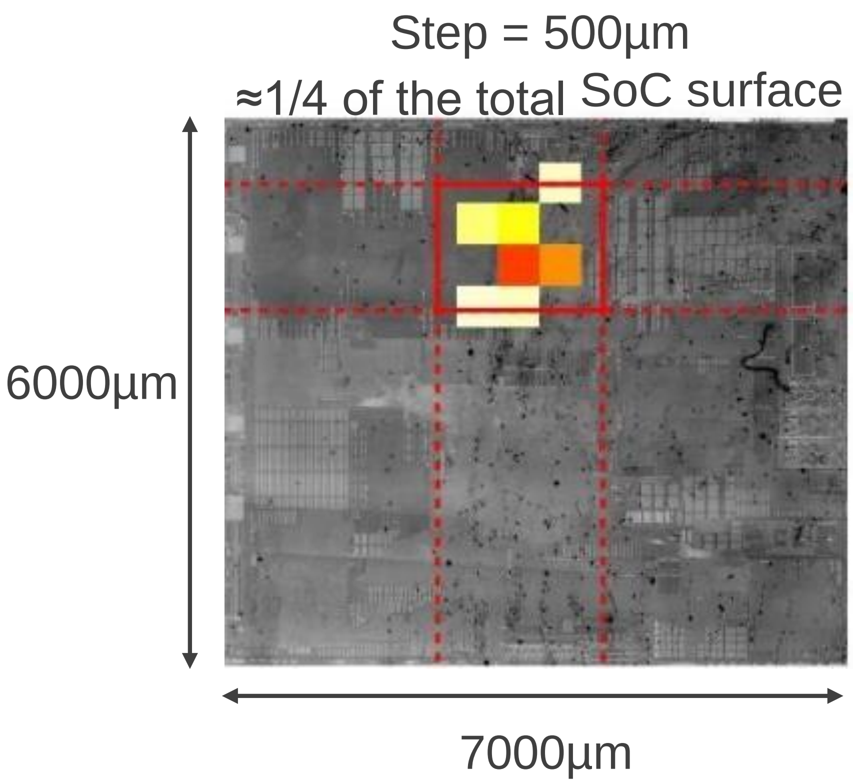


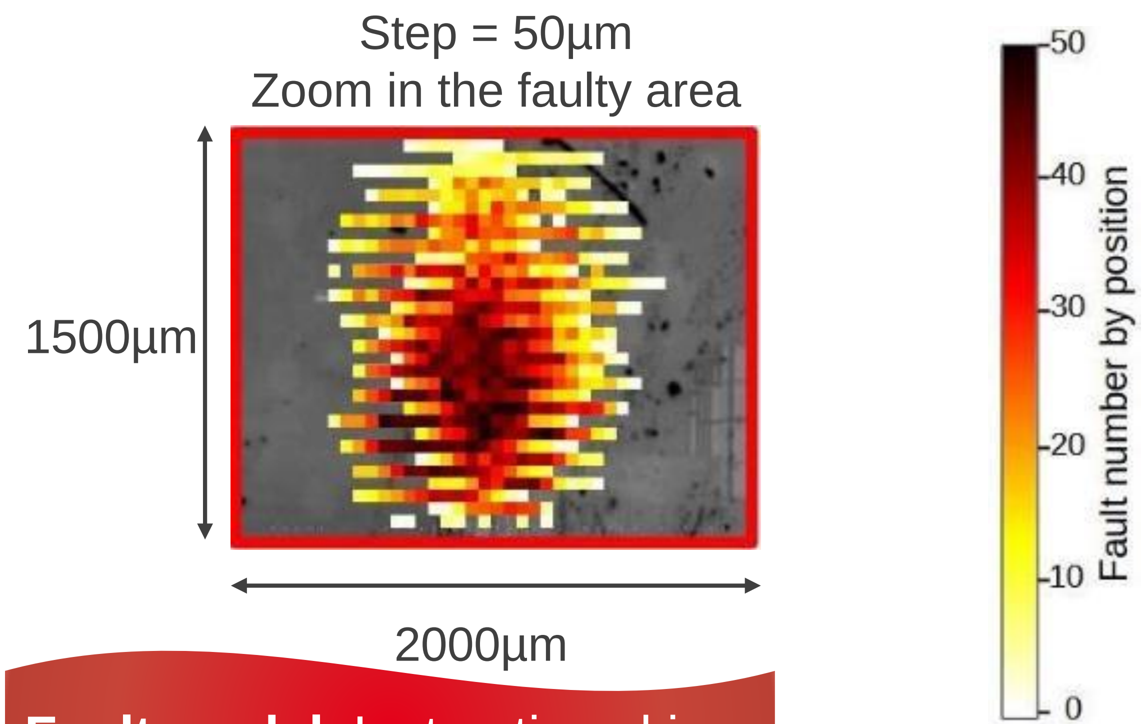
Where to fire ?

- Step 1: *vulnerability analysis*
- *Step 2: EMFI parameters*
- *Step 3: Attack synchronization*
- *Step 4: EMFI implementation*

Faults mapping area with SoC IR imaging as background.

- Pulse voltage = 400V
- Pulse Width = 10ns





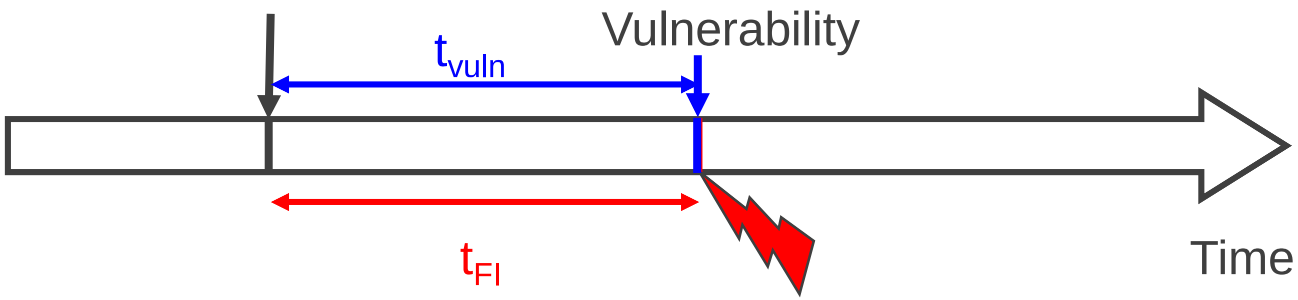
Fault model: Instruction skip

When to fire ?

- Step 1: vulnerability analysis
- Step 2: EMFI parameters
- Step 3: Attack synchronization
- Step 4: EMFI implementation

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.

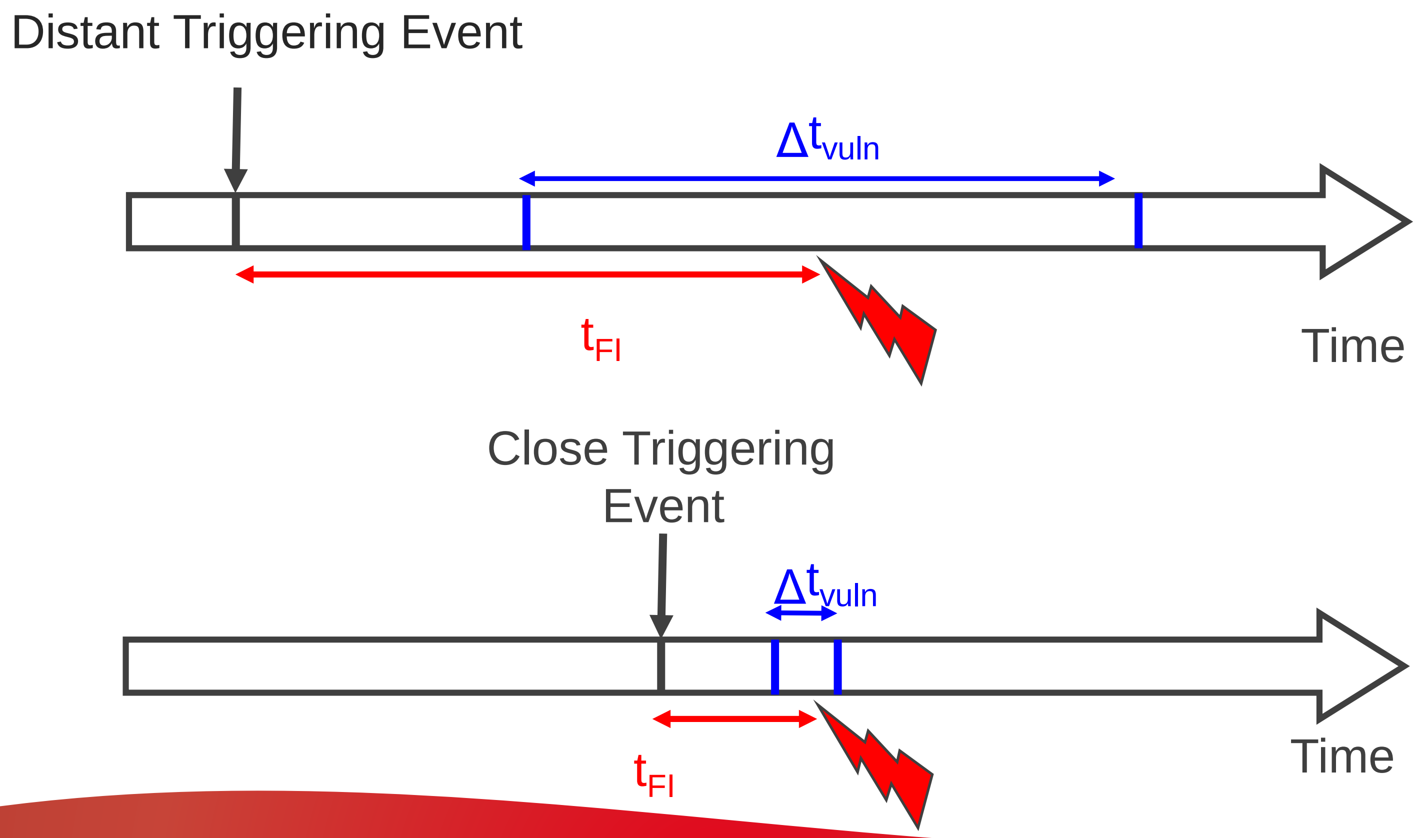


The attack is successful when $t_{vuln} = t_{FI}$
=> the injection and the vulnerability happen at the same time

When to fire ?

- Step 1: vulnerability analysis
- Step 2: EMFI parameters
- Step 3: Attack synchronization
- Step 4: EMFI implementation

Δt_{vuln} : to be reduced to improve the success rate of the attack

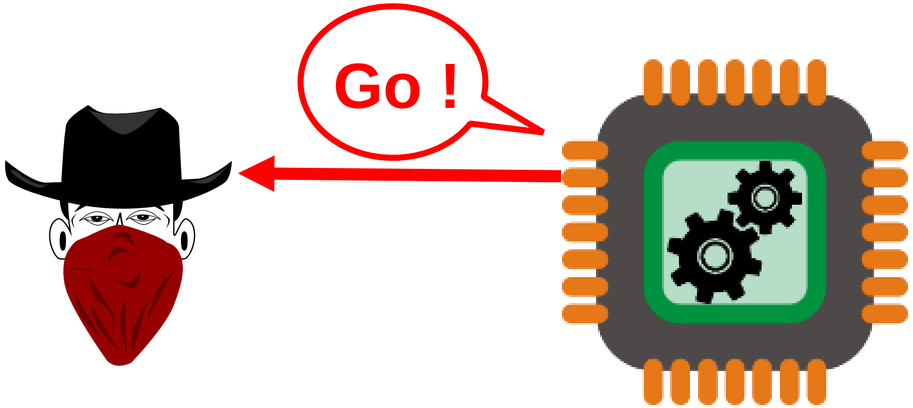


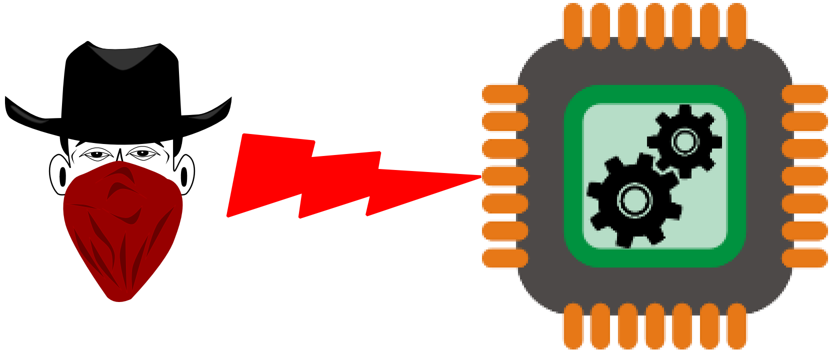
The trigger should be as close as possible to the vulnerability.

Synchronization solutions

- Step 1: vulnerability analysis
- Step 2: EMFI parameters
- Step 3: Attack synchronization
- Step 4: EMFI implementation

Solution 1: Trigger with fully controlled output such as GPIO
=> the attack is synchronized thanks to the signal implemented/captured in/from the target.



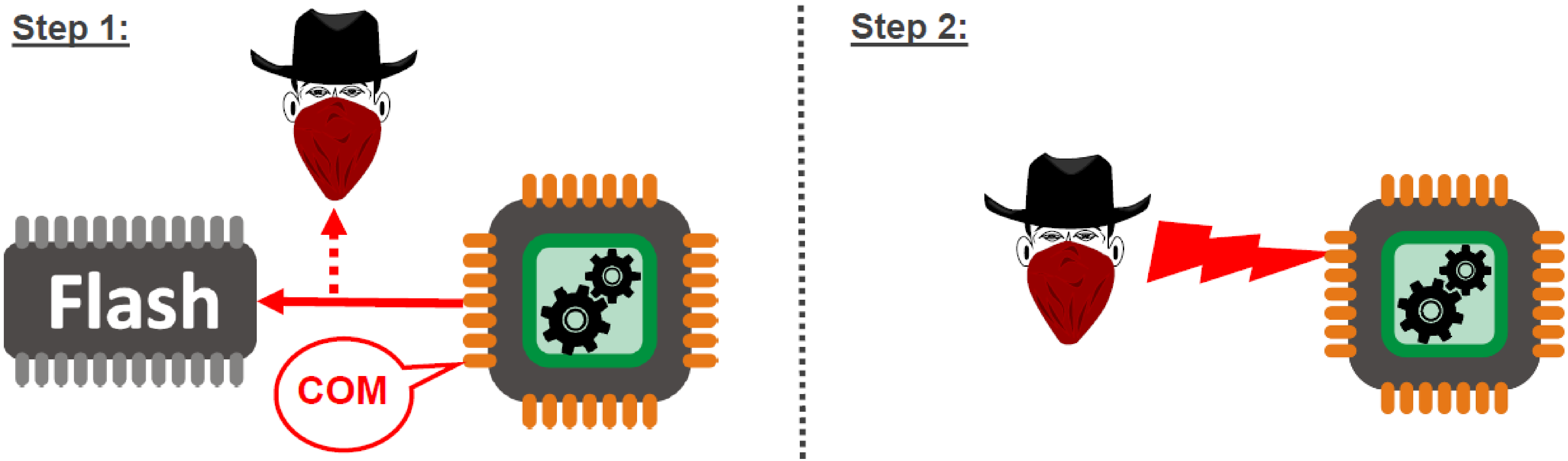


The synchronization is optimal but it needs a high level of control over the target

Synchronization solutions

- Step 1: vulnerability analysis
- Step 2: EMFI parameters
- Step 3: Attack synchronization
- Step 4: EMFI implementation

Solution 2: Trigger on uncontrolled I/O
=> the attack is synchronized thanks to the signal implemented/captured in/from the target.



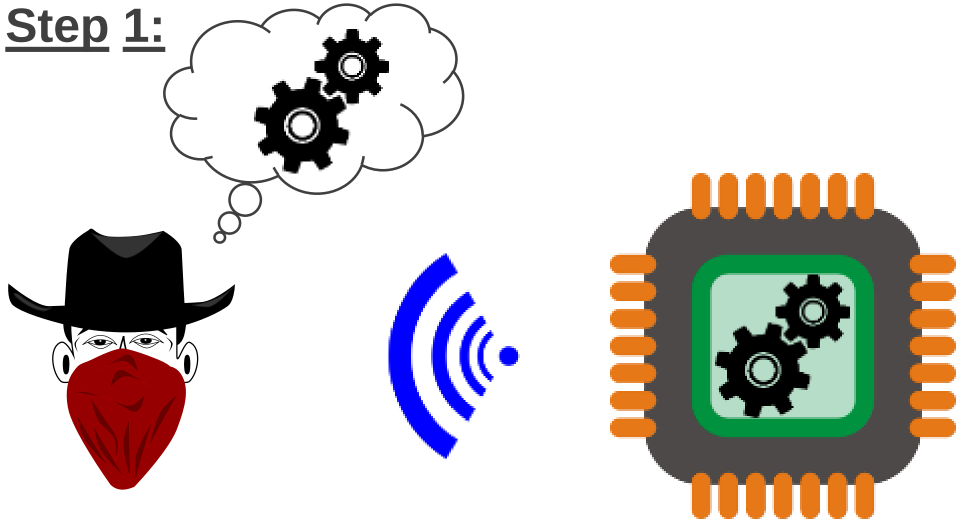
Not accurate and the fully control of the target is not required

Synchronization solutions

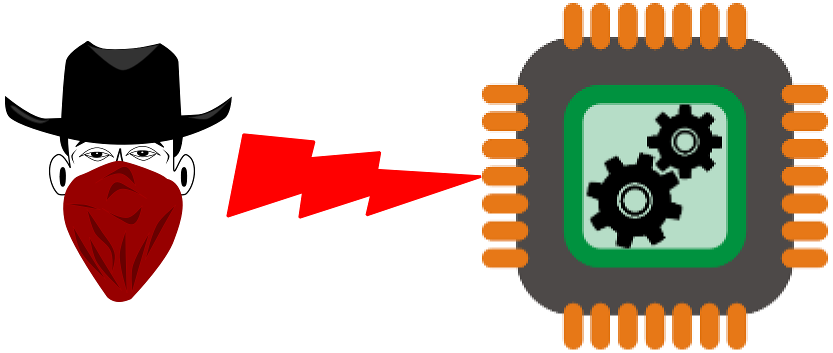
Step 1: vulnerability analysis
Step 2: EMFI parameters
Step 3: Attack synchronization
Step 4: EMFI implementation

Solution 3: Triggering on a Side-Channel event
=> The attacker has a great degree of freedom in the event choice

Step 1:



Step 2:

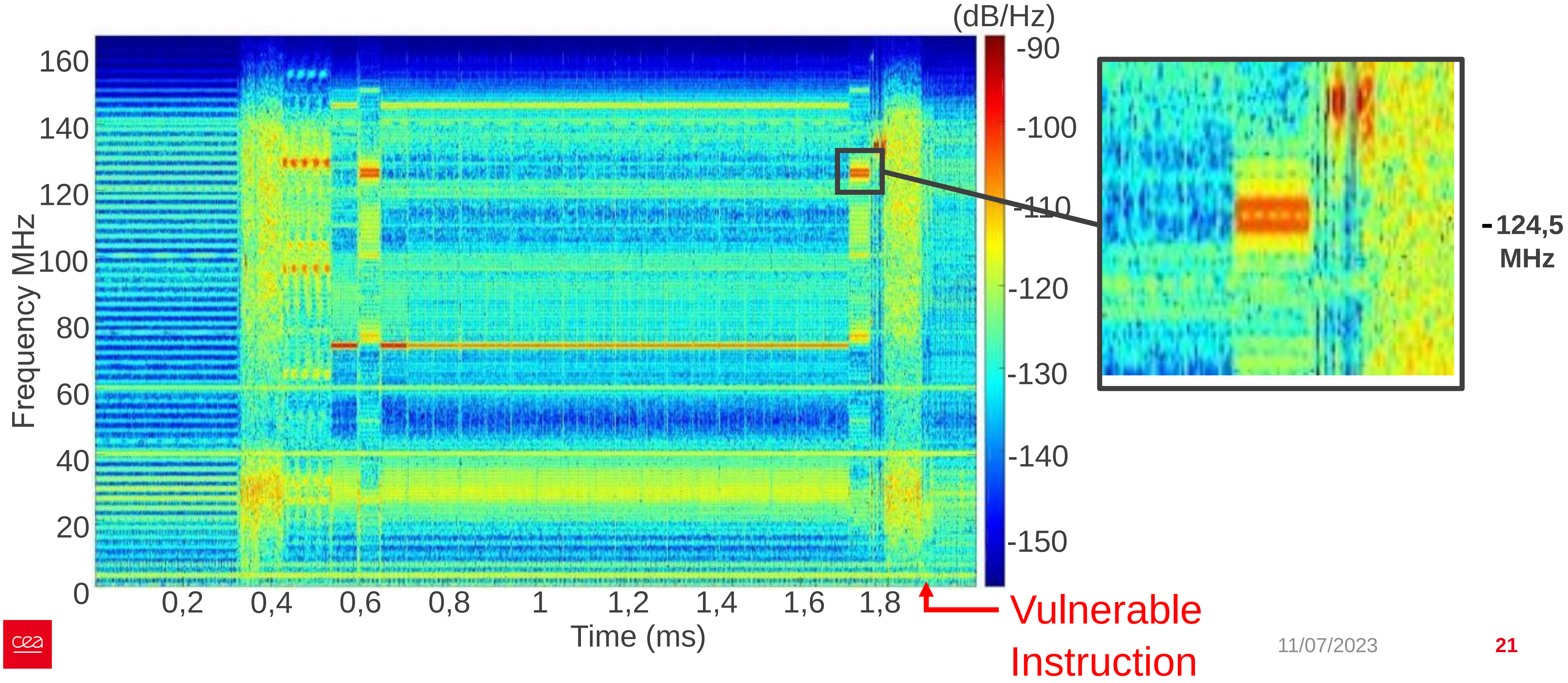


Real time analysis of EM signal is required: high frequency events analysis and detection is a real challenge

Synchronization by Frequency Detection: EM Side Channel Analysis based

- *Step 1: vulnerability analysis*
- *Step 2: EMFI parameters*
- *Step 3: Attack synchronization*
- *Step 4: EMFI implementation*

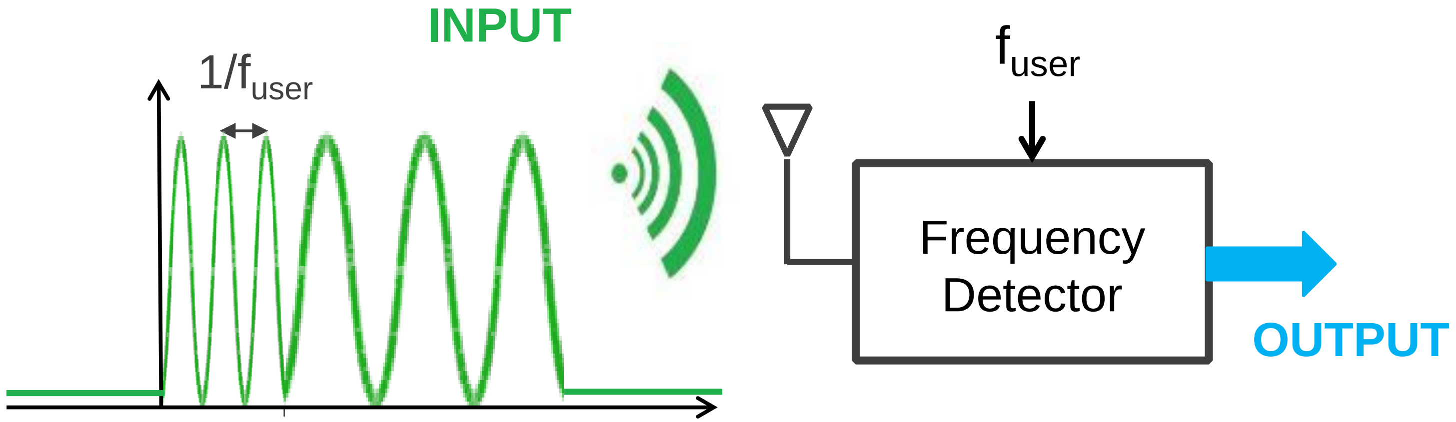
Step1: Offline spectrogram analysis of the EM emanations
=> Secure Boot Sequence before the Linux Kernel authentication

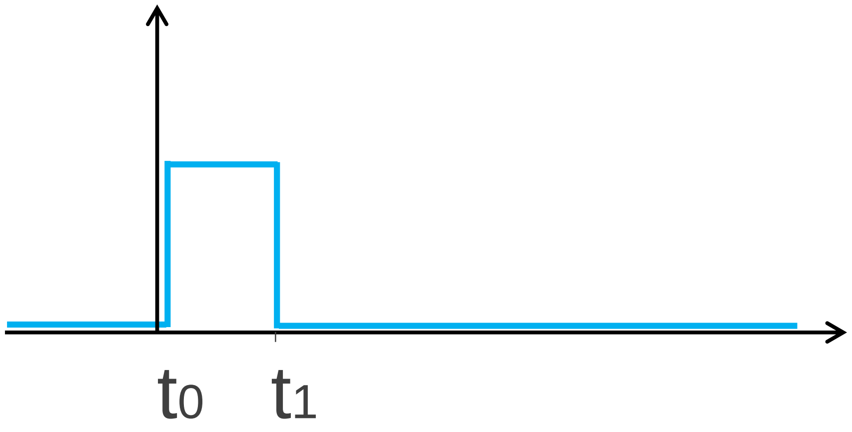


Synchronization by Frequency Detection: EM Side Channel Analysis based

Step 1: *vulnerability analysis*
Step 2: *EMFI parameters*
Step 3: Attack synchronization
Step 4: EMFI implementation

Step2: trigger Generation based on Characteristic Frequency Detection

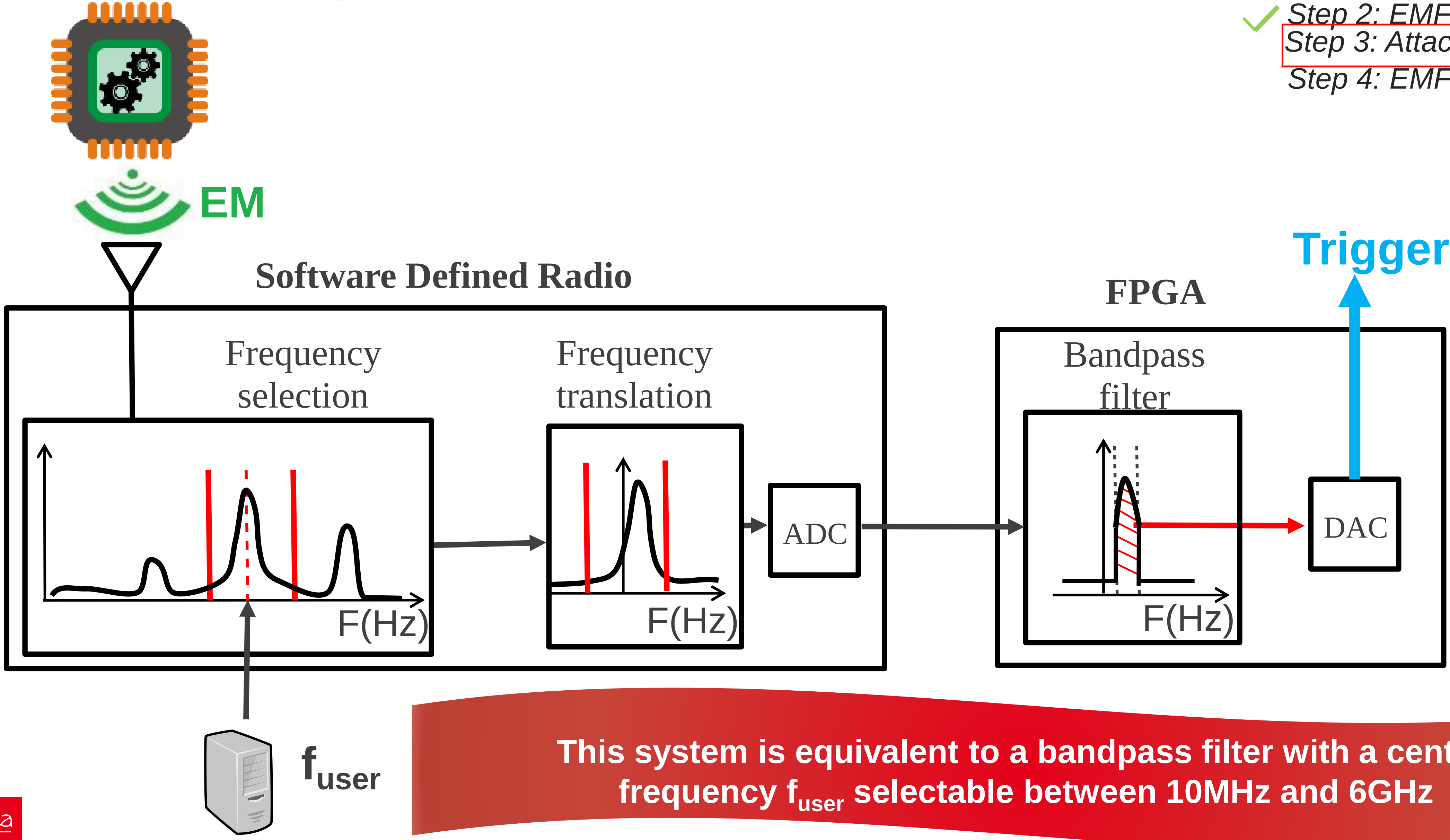




After the f_{user} detection it is needed to generate the trigger signal

Synchronization by Frequency Detection: EM Side Channel Analysis based

- *Step 1: vulnerability analysis*
- *Step 2: EMFI parameters*
- *Step 3: Attack synchronization*
- *Step 4: EMFI implementation*

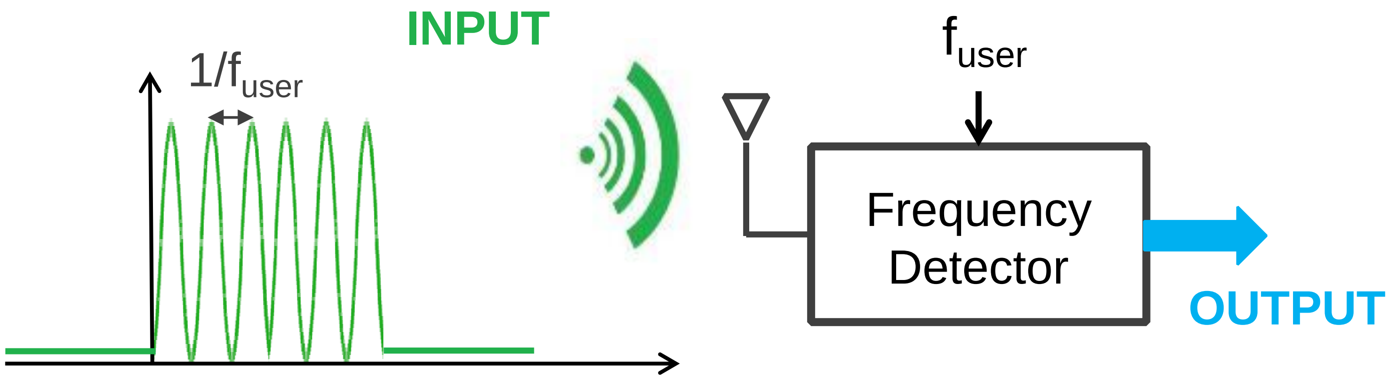


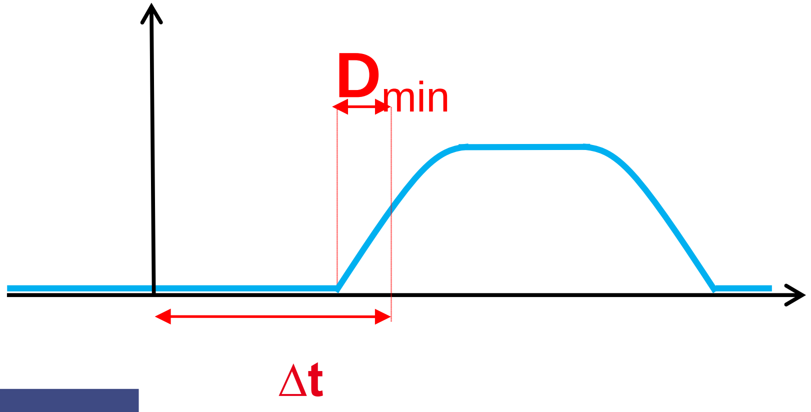
This system is equivalent to a bandpass filter with a central frequency f_{user} selectable between 10MHz and 6GHz

Synchronization by Frequency Detection: EM Side Channel Analysis based

Step 1: vulnerability analysis
Step 2: EMFI parameters
Step 3: Attack synchronization
Step 4: EMFI implementation

Δ**t** is the delay between the activation of f_{user} and its detection





	Results
Δt_{avg}	2.5 µs
Δt_{std}	60 ns
D_{min}	450 ns

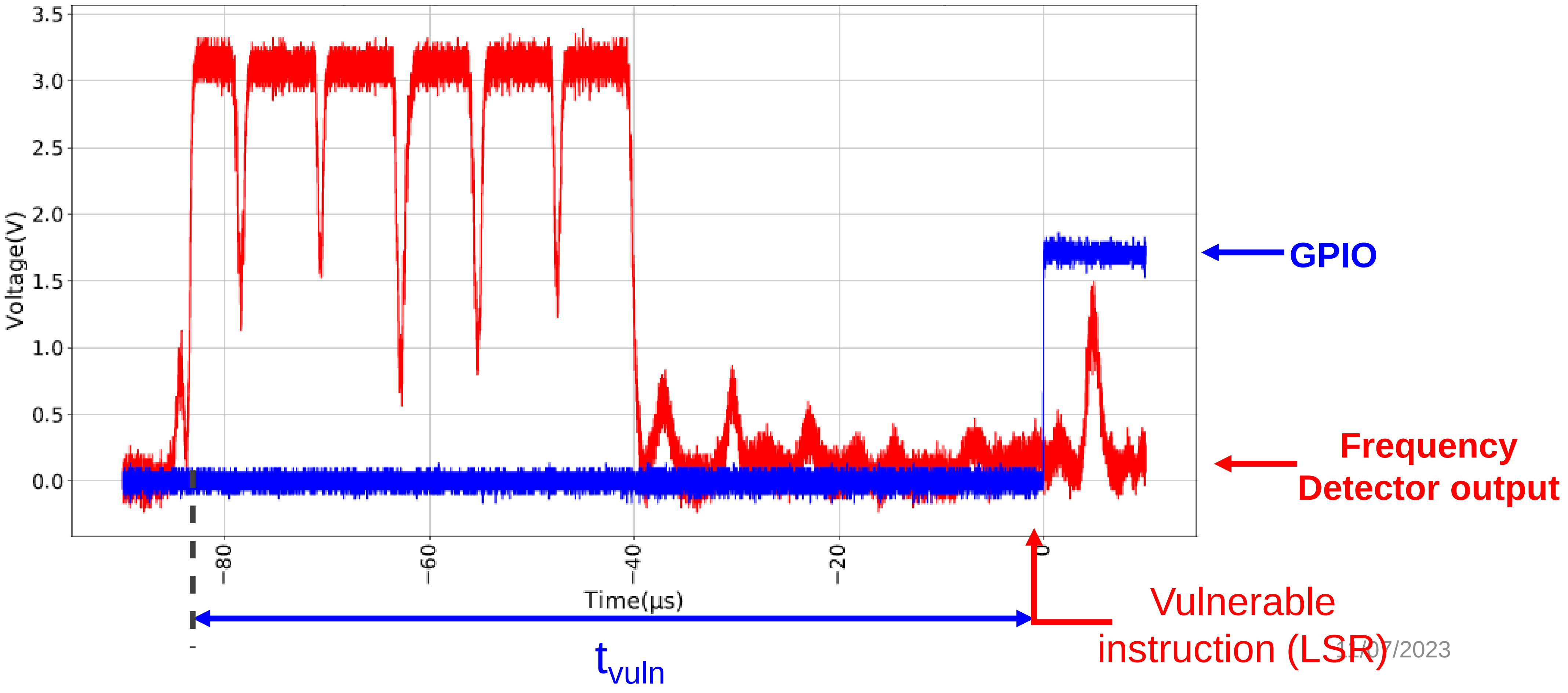
f_{user} needs to stay active during at least **450 ns** to be detected

Synchronization by Frequency Detection: EM Side Channel Analysis based

- Step 1: vulnerability analysis
- Step 2: EMFI parameters
- Step 3: Attack synchronization
- Step 4: EMFI implementation

Delay measurement between the frequency detector output and the vulnerability

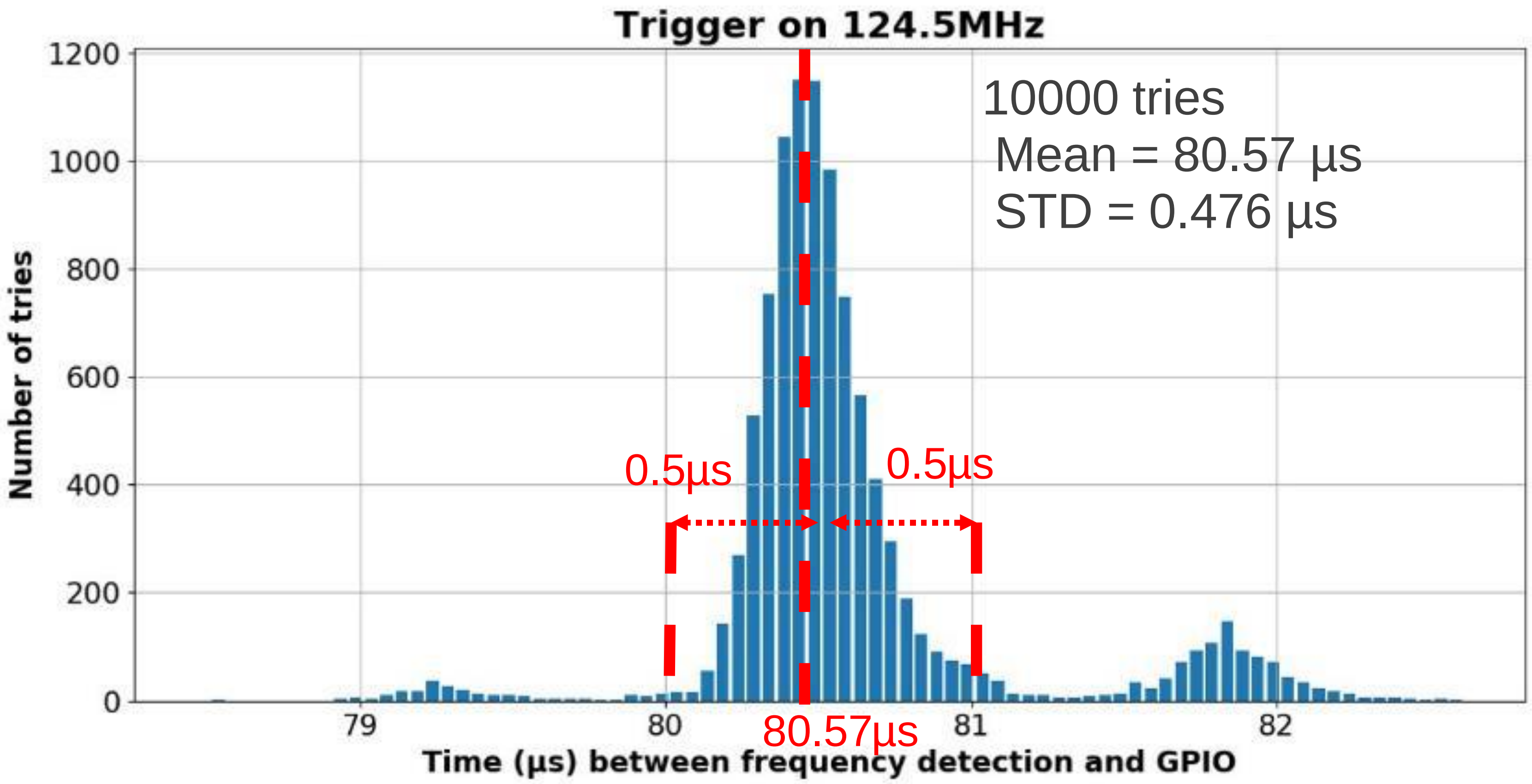
- We use a modified Little Kernel which rises a GPIO just after the vulnerability.
- We set the frequency detector to trigger on the 124,5MHz frequency we identified before.



Synchronization by Frequency Detection: EM Side Channel Analysis based

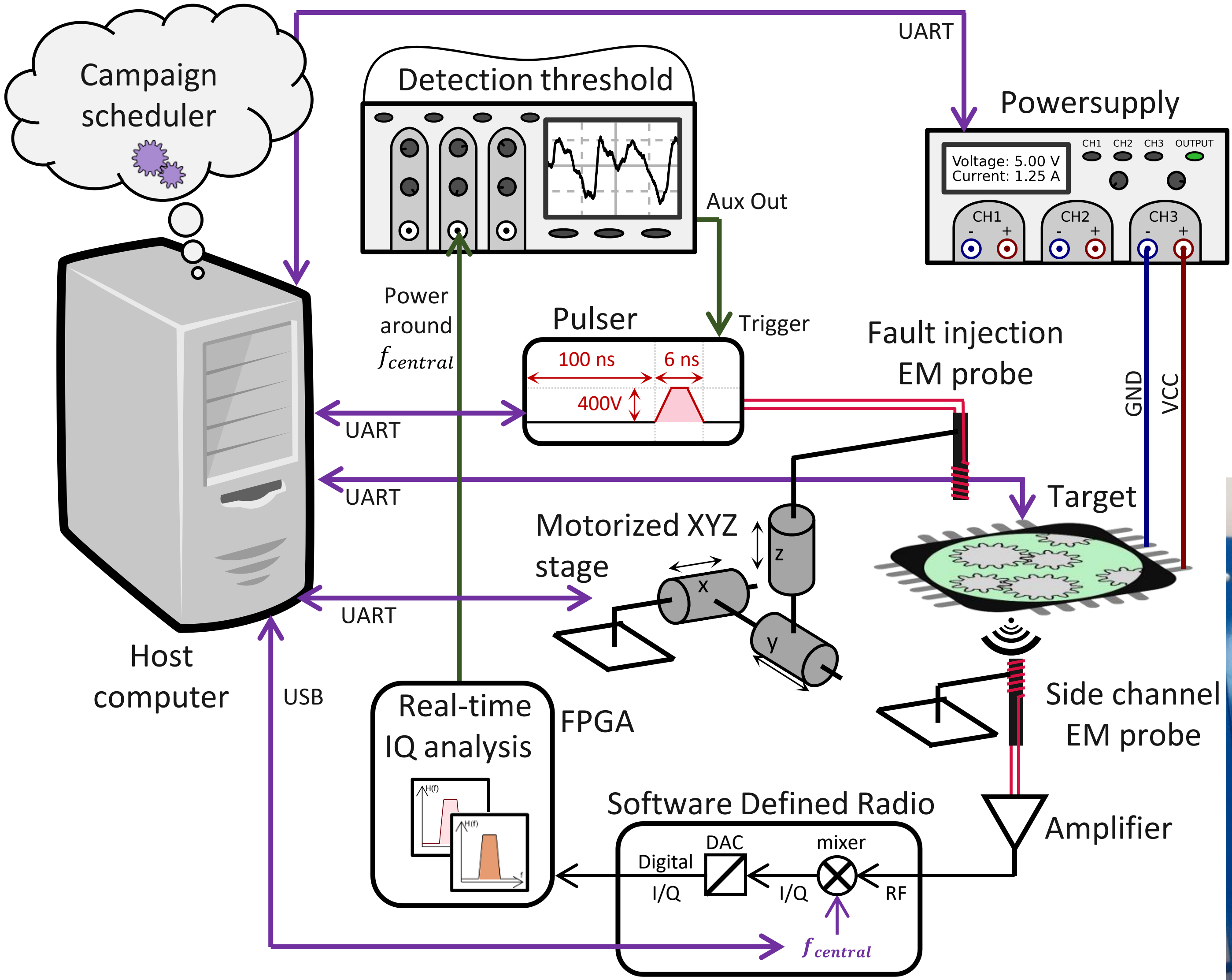
- Step 1: vulnerability analysis
- Step 2: EMFI parameters
- Step 3: Attack synchronization
- Step 4: EMFI implementation

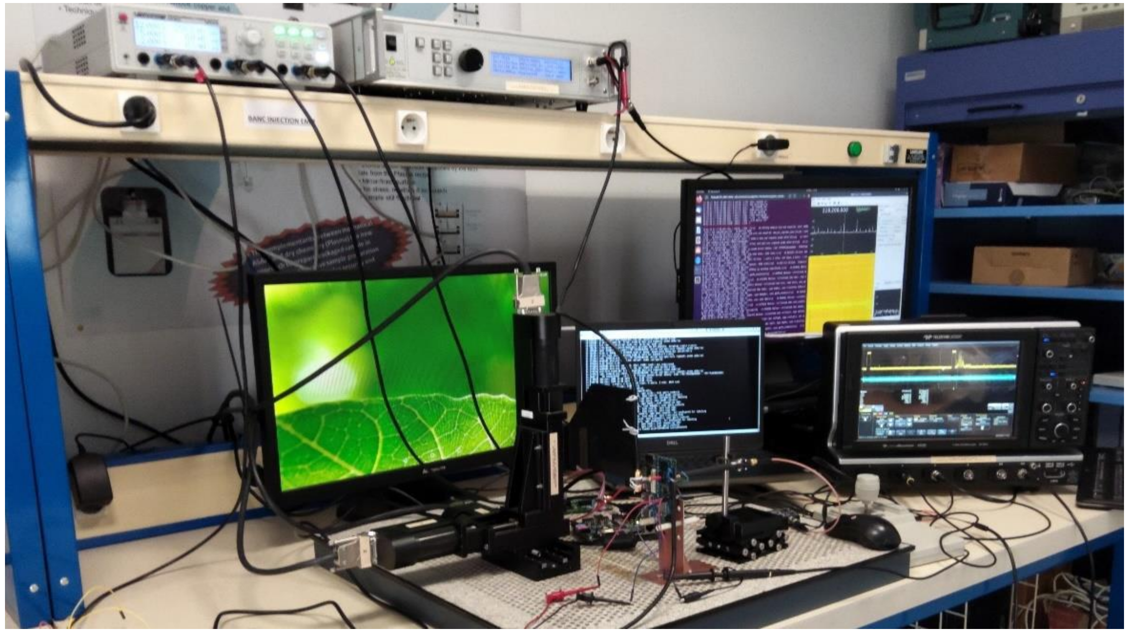
Delay measurement between the frequency detector output and the vulnerability: measures performed 10000 times to identify the mean delay and the jitter



Implementation of EMFI

- ✓ *Step 1: vulnerability analysis*
- ✓ *Step 2: EMFI parameters*
- ✓ *Step 3: Attack synchronization*
- *Step 4: EMFI implementation*





Results

- Step 1: vulnerability analysis
- Step 2: EMFI parameters
- Step 3: Attack synchronization
- Step 4: EMFI implementation

15000 attacks performed in 18 hours

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

≈ 1 bypass every 15 minutes

Conclusion

Where to fire:

- Identification of the sensitive area to EMFI
- Identification of the vulnerability in the Secure Boot Software process

When to fire:

- new synchronization method to trigger hardware attacks based on the detection of high frequency event
- By using this synchronization method we successfully synchronized a fault injection with the vulnerability:

The bypass of Linux Kernel authentication is possible by EMFI

Discussion around where to fire

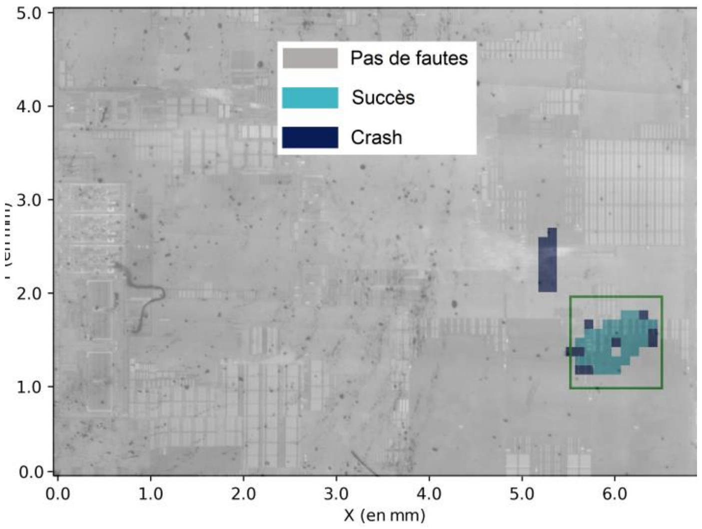
From the white-box

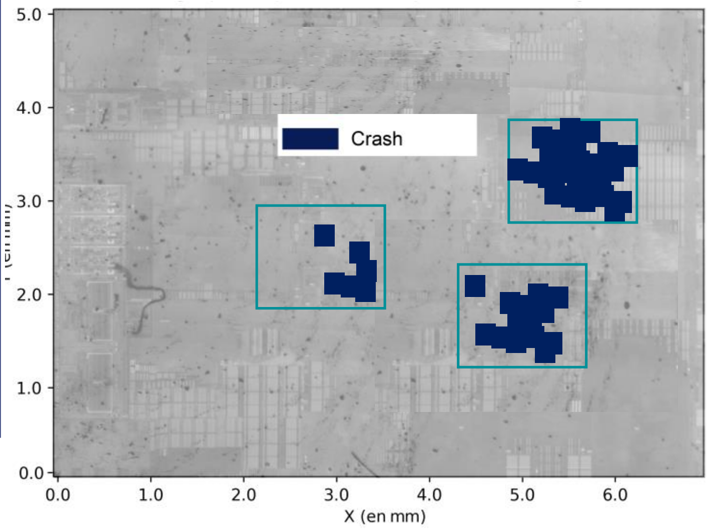
To the black-box

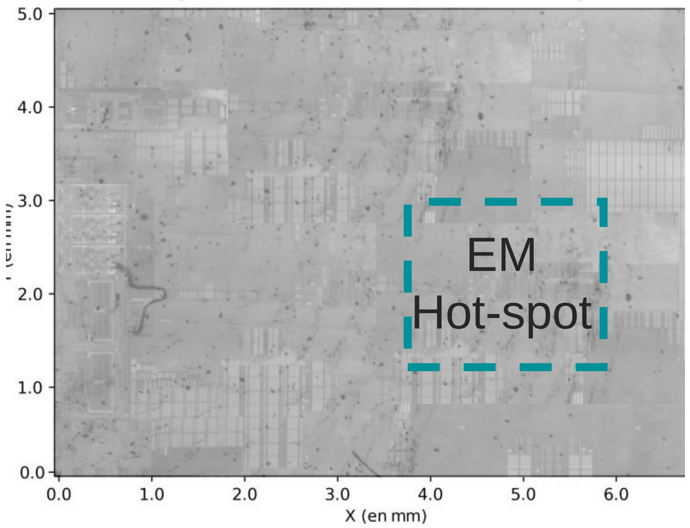
If the fully access to the target is possible: Executing a "fault observer" will provide a significate support for EMFI implementation

Identifying the crash area by utilizing maximum power and a large probe

Measuring the EM activities to identifying the hotspots or active points of the target gives a good understanding of the sensitive electromagnetic (EM) area.





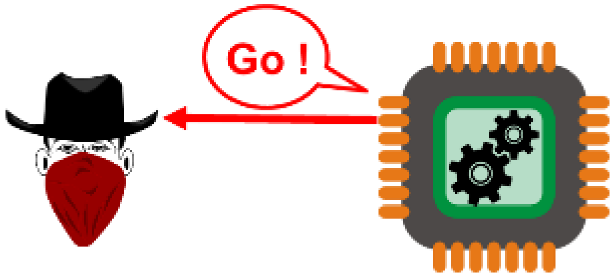


Perspective/ongoing: when to fire

From the white-box

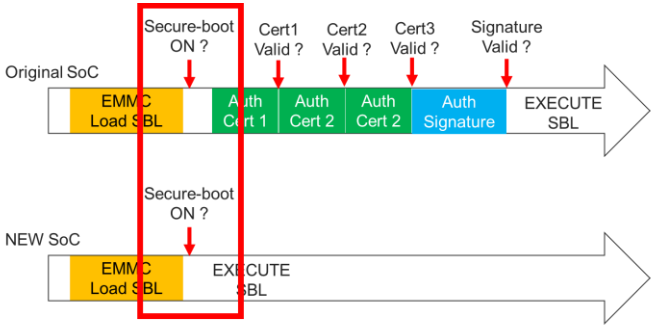
To the black-box

Trigger with fully controlled output such as GPIO



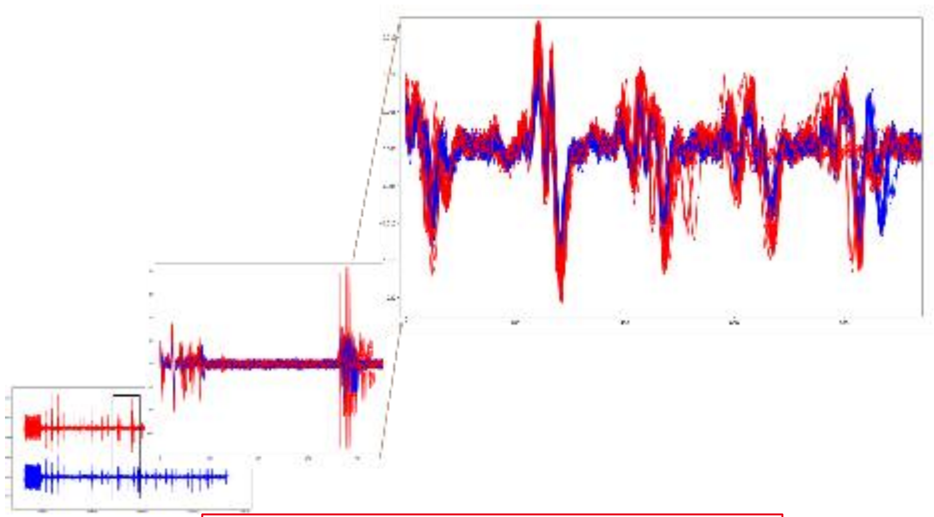
In controlled environment

Comparing the EM activities of SB: enabled vs disabled



Requests an open target

Analysis the EM activities of malicious program vs the original one: defining the range time corresponding to the reject of the code by the target



Needs to flash the target

We have a funded PhD offer:

Do not hesitate to contact me for any requests at:
driss.aboulkassimi@cea.fr

Thank you for your attention

Driss ABOULKASSIMI