DE LA RECHERCHE À L'INDUSTRIE

Méthode combinée d'Injection de faute et d'analyse Side-Channel temps réel pour contourner un Secure-Boot d'Android

Combined Fault Injection and Real-Time Side-Channel Analysis for Android Secure-Boot Bypassing

Clément Fanjas (1), Clément Gaine (1) (2),
Driss Aboulkassimi (1), **Simon Pontié** (1), Olivier Potin (2)

(1) CEA-Leti, SAS team, Gardanne France

(2) EMSE, SAS team, Gardanne France

Combined Fault Injection and Real-Time Side-Channel Analysis for Android Secure-Boot Bypassing

Authors: Clément Fanjas(1), Clément Gaine(1) (2), Driss Aboulkassimi(1), Simon Pontié(1), Olivier Potin(2)

(1) CEA-Leti, SAS team, Gardanne France

(2) EMSE, SAS team, Gardanne France

CARDIS Conference @CardisConf

Starting #CARDIS22 with the best paper award “Combined Fault Injection and Real-Time Side-Channel Analysis for Android Secure-Boot Bypassing” by Clément Fanjas, Clément Gaine, Driss Aboulkassimi, Simon Pontié, Olivier Pot;

Nov 8, 2022 · 9:53 AM UTC · TweetDeck

The experiments were done in the context of the EXFILES project.

1. Secure-Boot and Fault Injection

2. Target and methodology

3. Synchronization of hardware attacks

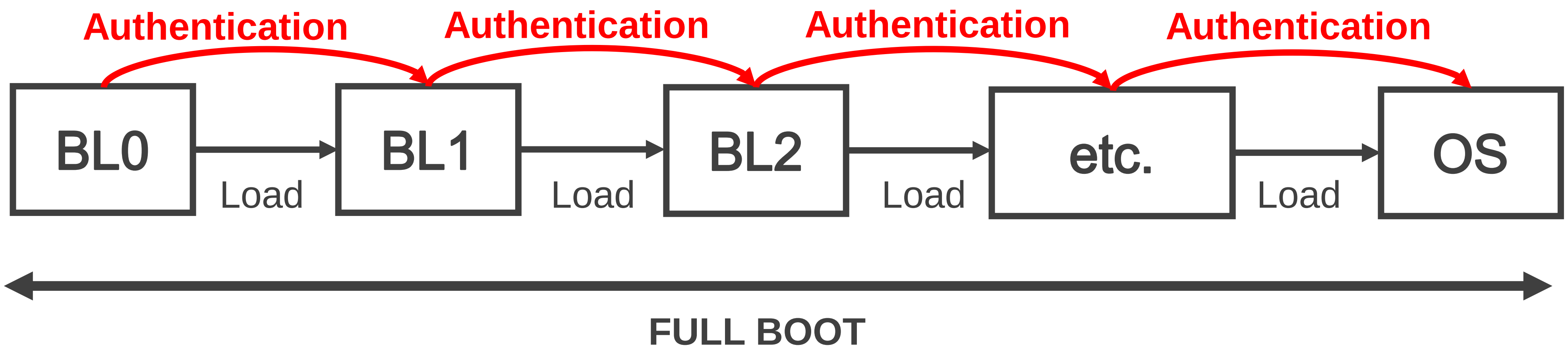
4. Combined Fault Injection and Real-Time SCA

What is a Secure Boot ?

➔ A security feature : it is a chain of trust where each high privilege program is authenticated before being executed.

Why it is important ?

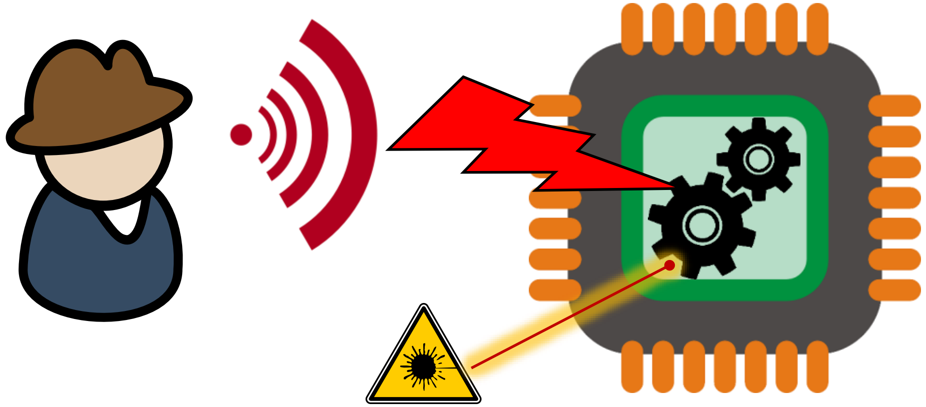
➔ It avoids running malicious program with high privilege.

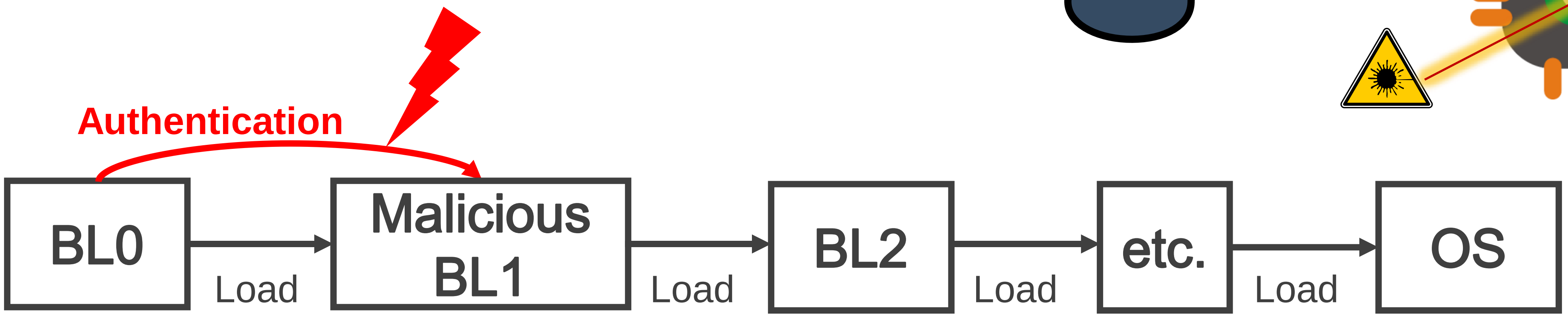


Fault Injection to bypass the Secure Boot

- Fault injection aims at disrupting the target behaviour.

- Fault injection methods already used against Secure Boot:
  - Optical (laser) [3]
  - **Electromagnetic (EMFI)** [4]
  - Glitch (clock and voltage) [1,2]





Bypassing the Secure Boot leads to a privilege escalation.

Hardware target :

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

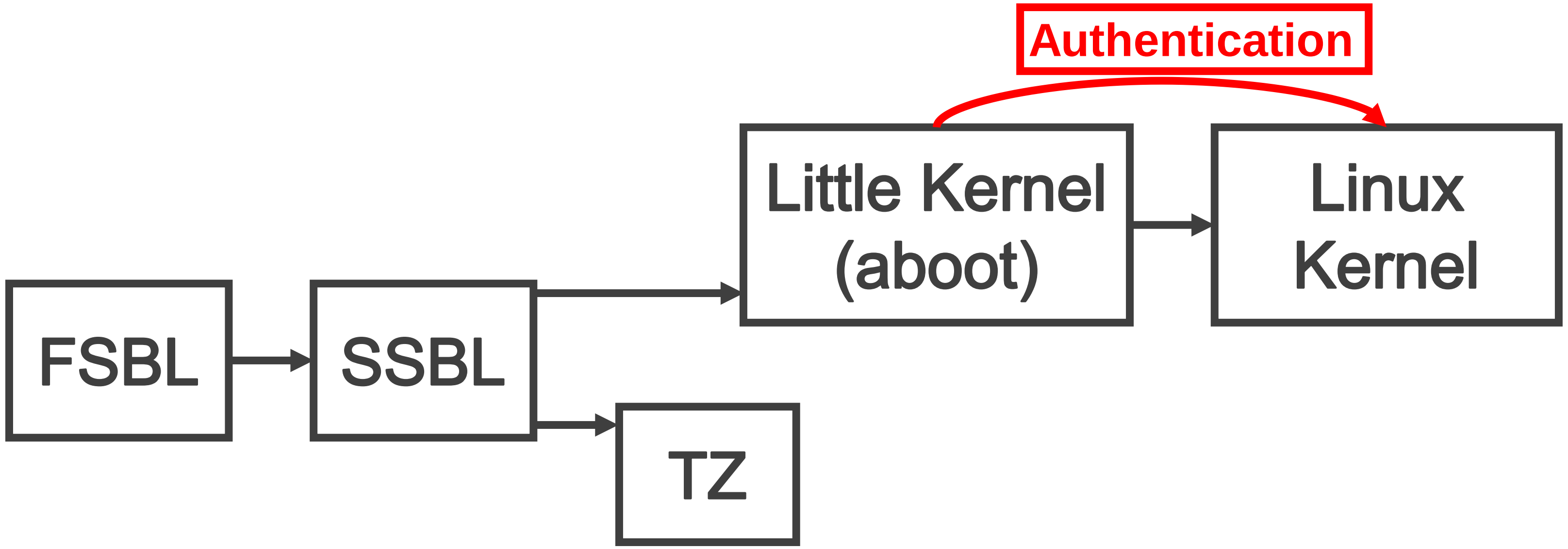
  - Previous work (**Gaine et al. 2020**):
    Using EMFI to skip an instruction is possible
  - Previous work (**Tasso et al. 2021**):
    Using EMFI to recover SIKE private key is possible

Clément Fanjas, Clément Gaine, Driss Aboulkassimi, Simon Pontié, Olivier Potin

Software target :

- Android Secure-Boot

  ➔ Linux kernel authentication



Our goal is to **bypass this authentication using EMFI** and **load a malicious Linux Kernel** on our target.

(A) Search an instruction skip vulnerability in the Linux Kernel authentication.

↓

(B) Identify the good EMFI parameters to skip an instruction.

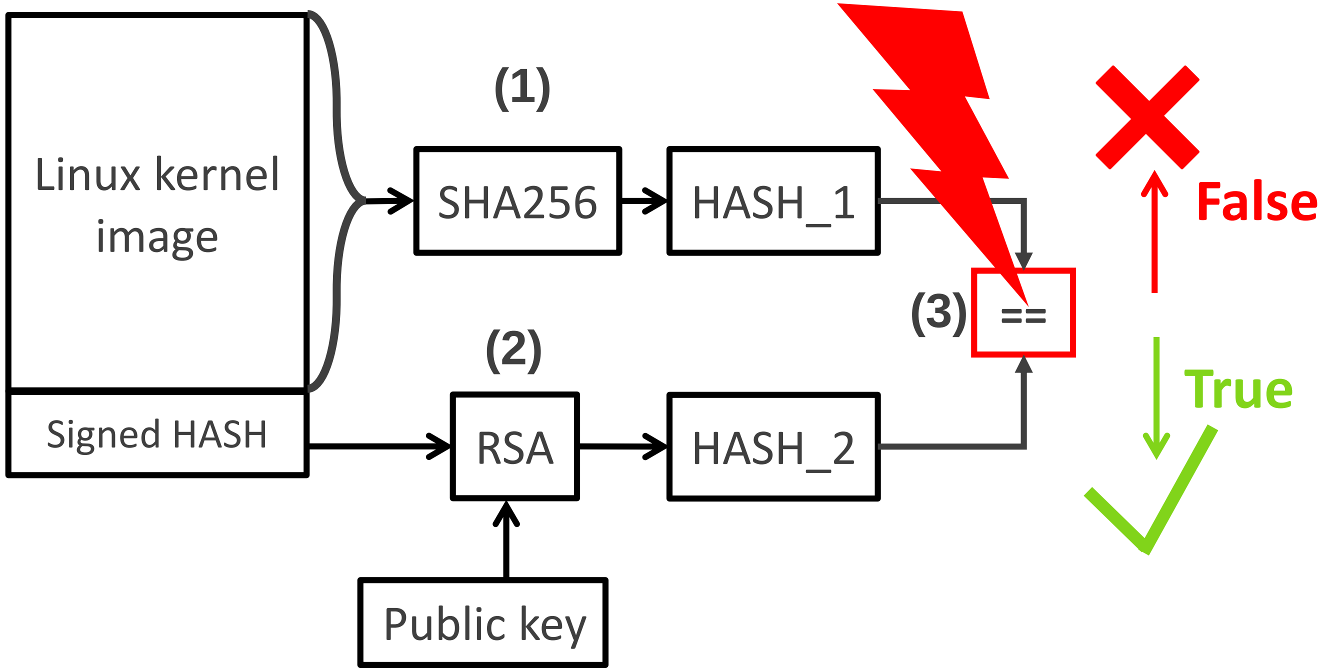
↓

(C) Find a way to synchronize the injection with the vulnerability.

↓

(D) Use all these parameters to bypass the authentication with a combined attack.

(A) Vulnerability analysis
(B) EMFI parameters
(C) Synchronization
(D) Combined attack



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)

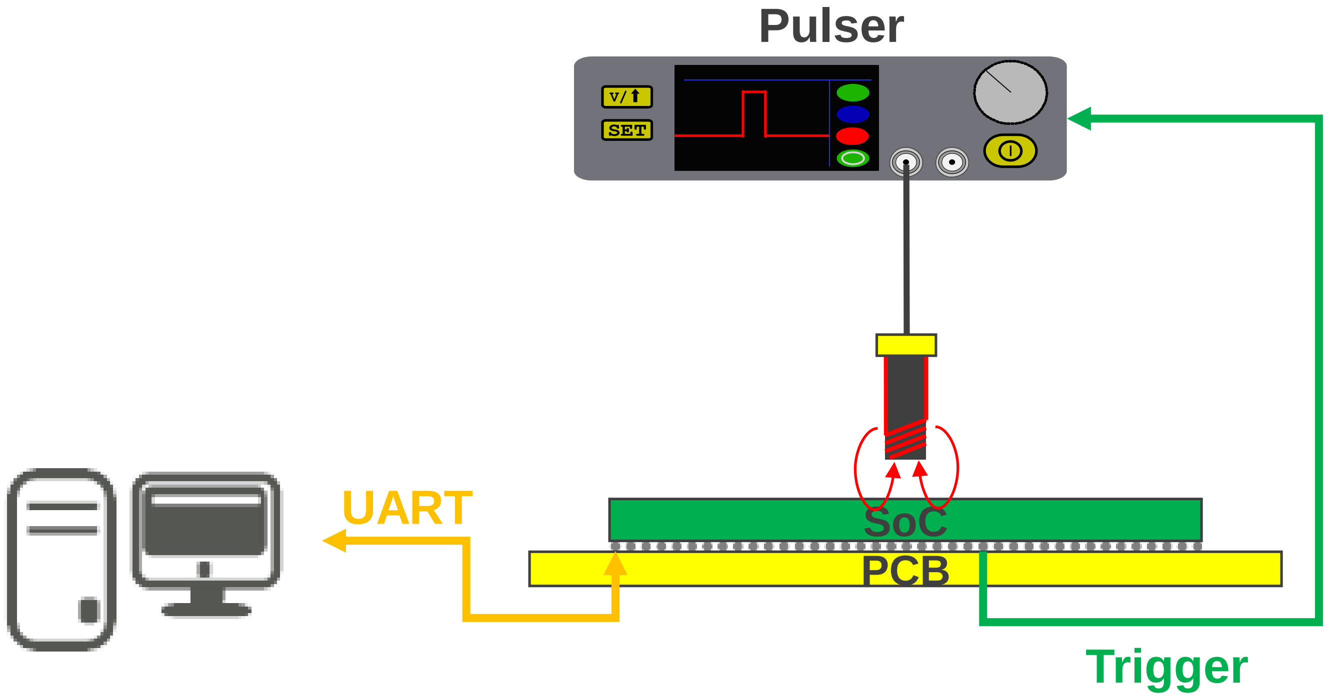
```
bl <memcmp>
clz r6, r0
lsr r6, r6, #5
```

Skipping this LSR allows to bypass the authentication

(A) Vulnerability analysis
(B) EMFI parameters
(C) Synchronization
(D) Combined attack

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: voltage and pulse width



(A) Vulnerability analysis
(B) EMFI parameters
(C) Synchronization
(D) Combined attack

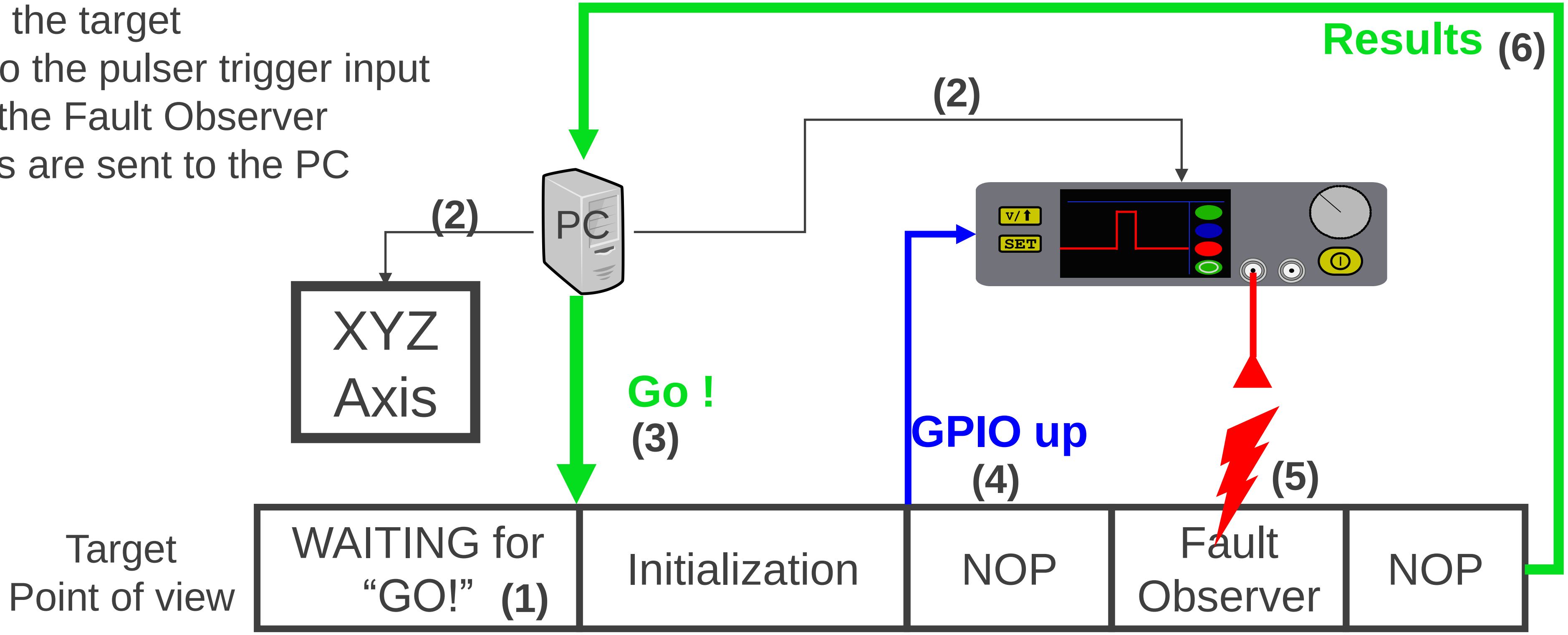
To find the good parameters we use a target without Secure-Boot.

We run a program with a known behaviour: we call it a ''Fault Observer''.

We observe the output of this program while injecting 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

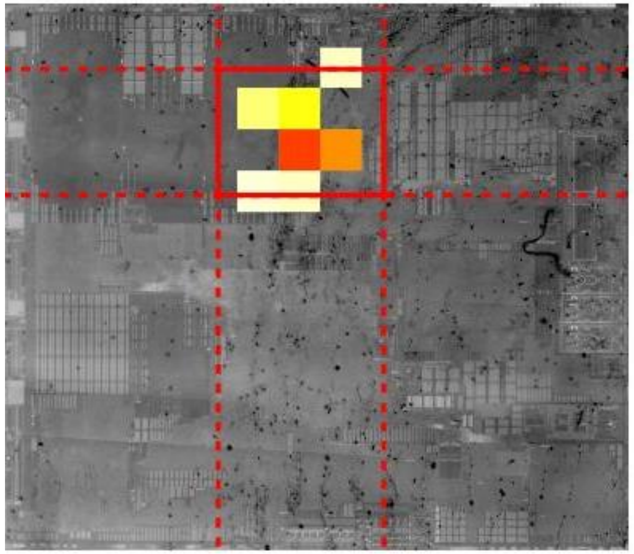


- ✓ (A) Vulnerability analysis
- **(B) EMFI parameters**
- (C) Synchronization
- (D) Combined attack

Fault model: Instruction skip

Faulty area with SoC IR imaging as background.
Pulse voltage = 400V
Pulse Width = 10ns

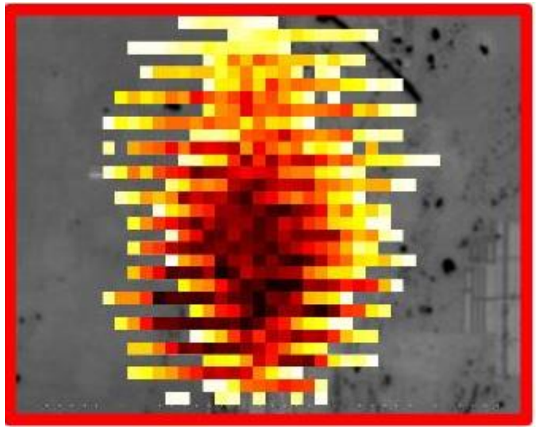
Step = 500µm
≈1/4 of the total SoC surface



6000µm

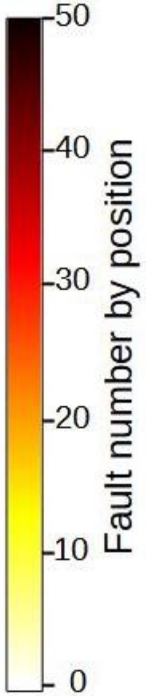
7000µm

Step = 50µm
Zoom in the faulty area



1500µm

2000µm

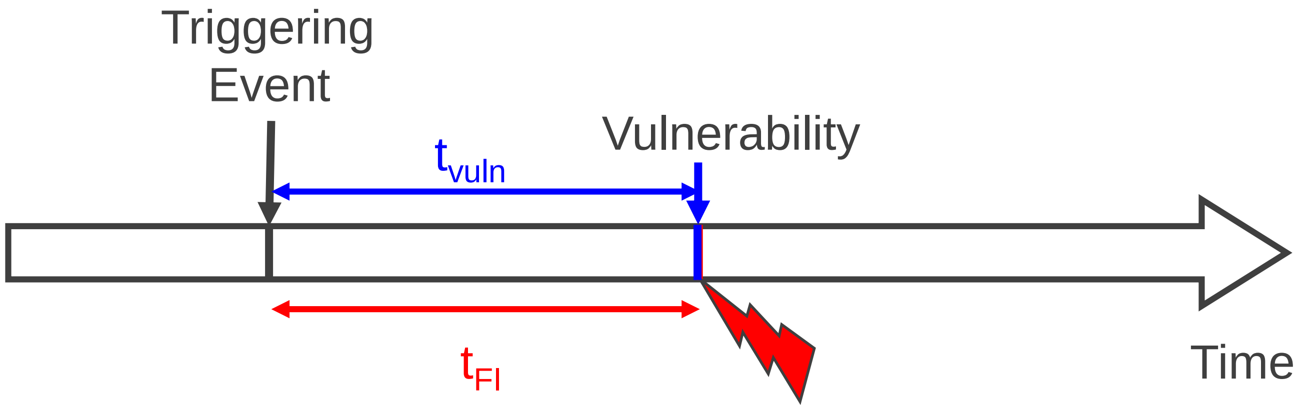


- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack

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

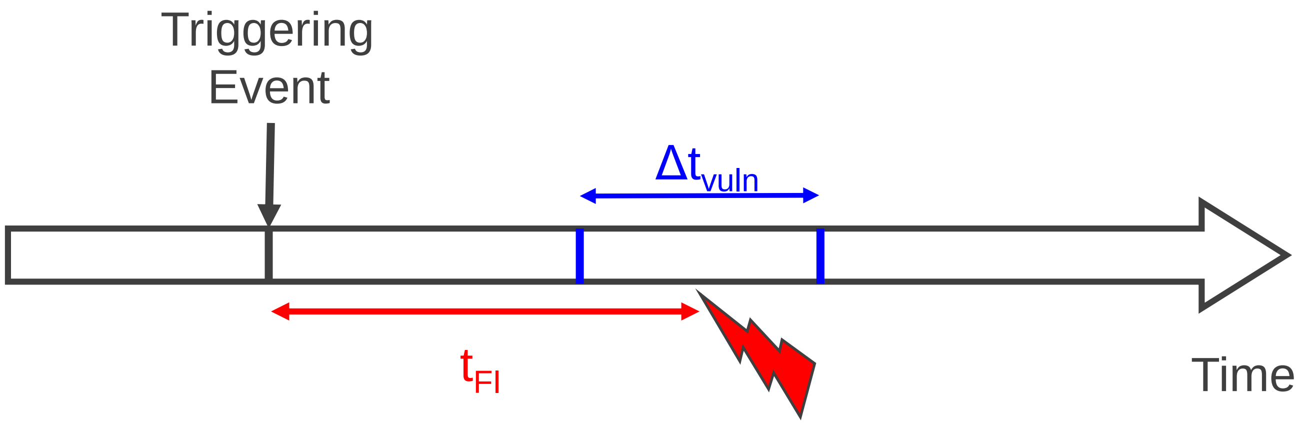
t_{vuln} = delay between the triggering event and the vulnerability.
t_{FI} = delay between the triggering event and the attack.

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



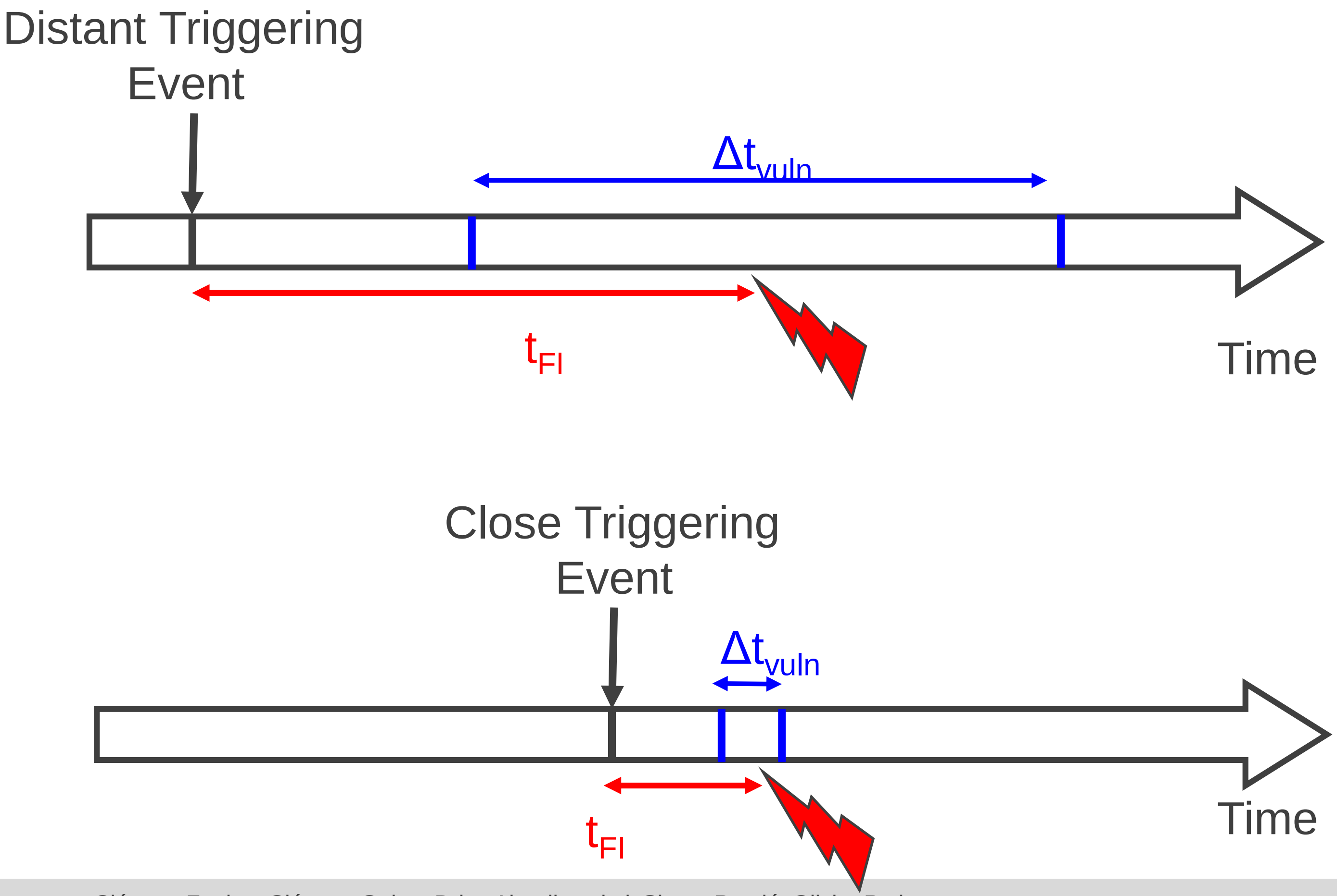
- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack

We set t_{FI}, but we have no influence on t_{vuln}.
t_{vuln} is confined in a temporal window Δt_{vuln} also called jitter.



- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack

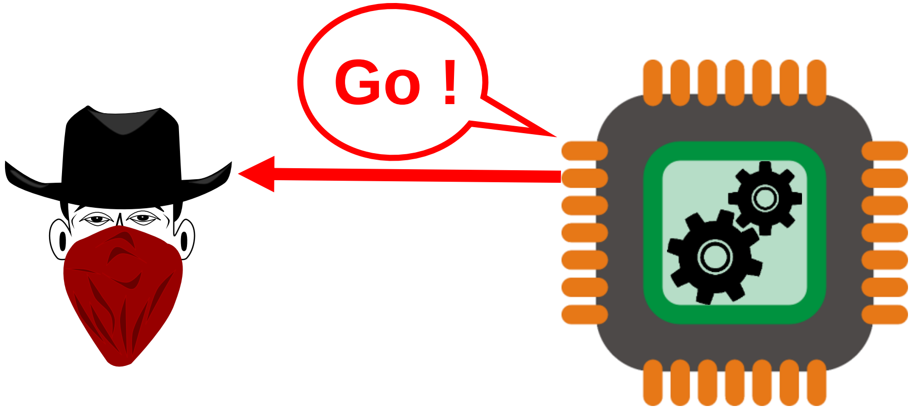
To maximize the attack success rate we need to reduce Δt_{vuln}. The best way to do it is to get the triggering event as close as possible to the vulnerability.



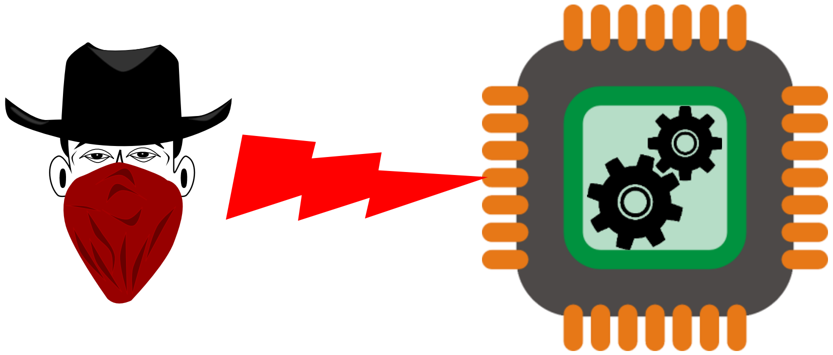
Solution 1: Trigger with fully controlled output such as GPIO

- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack

Step 1:



Step 2:

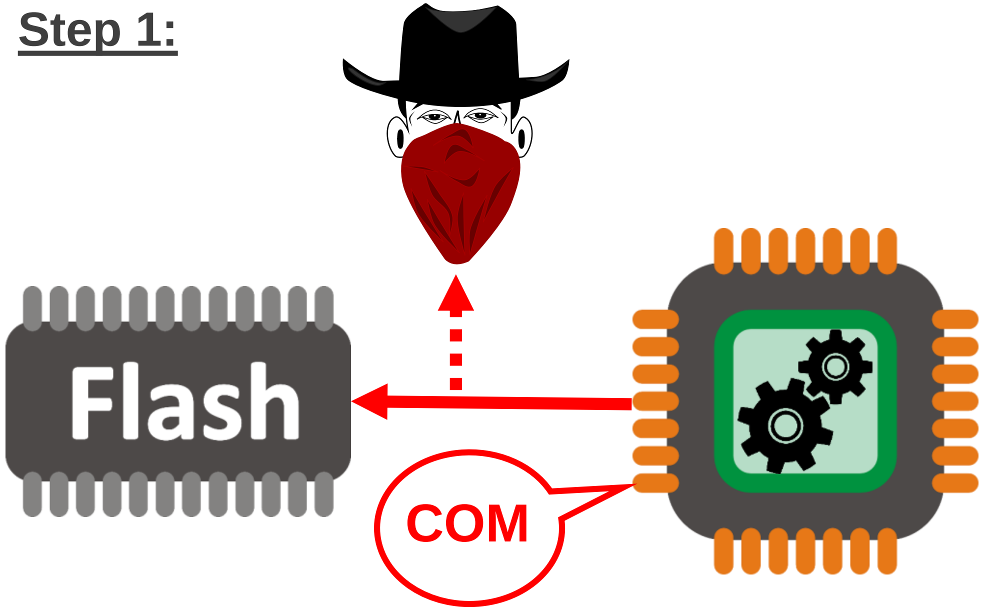


The attack is voluntarily triggered by the target.
The synchronization is optimal but it needs a high level of control over the target.

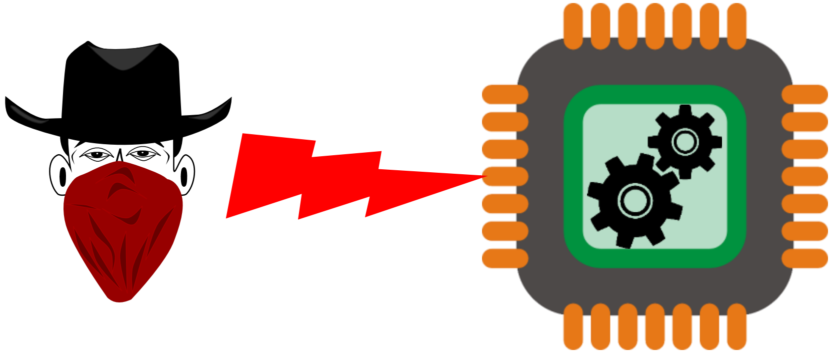
Solution 2: Trigger on uncontrolled I/O

- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack

Step 1:



Step 2:

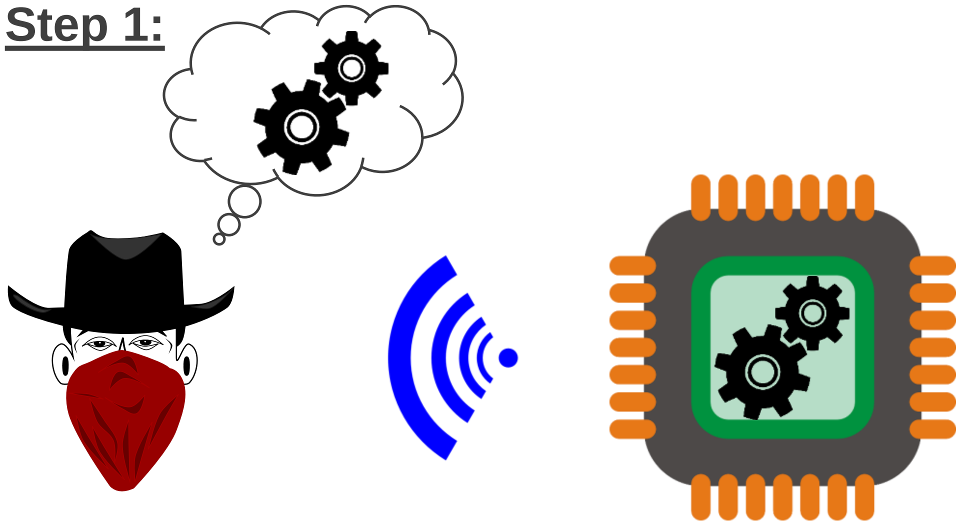


This kind of triggering event is not always accurate but there is no need to control the target.

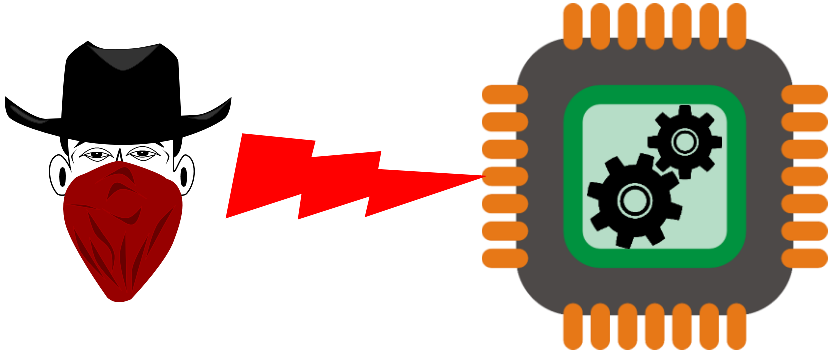
Solution 3: Triggering on a Side-Channel event

- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack

Step 1:



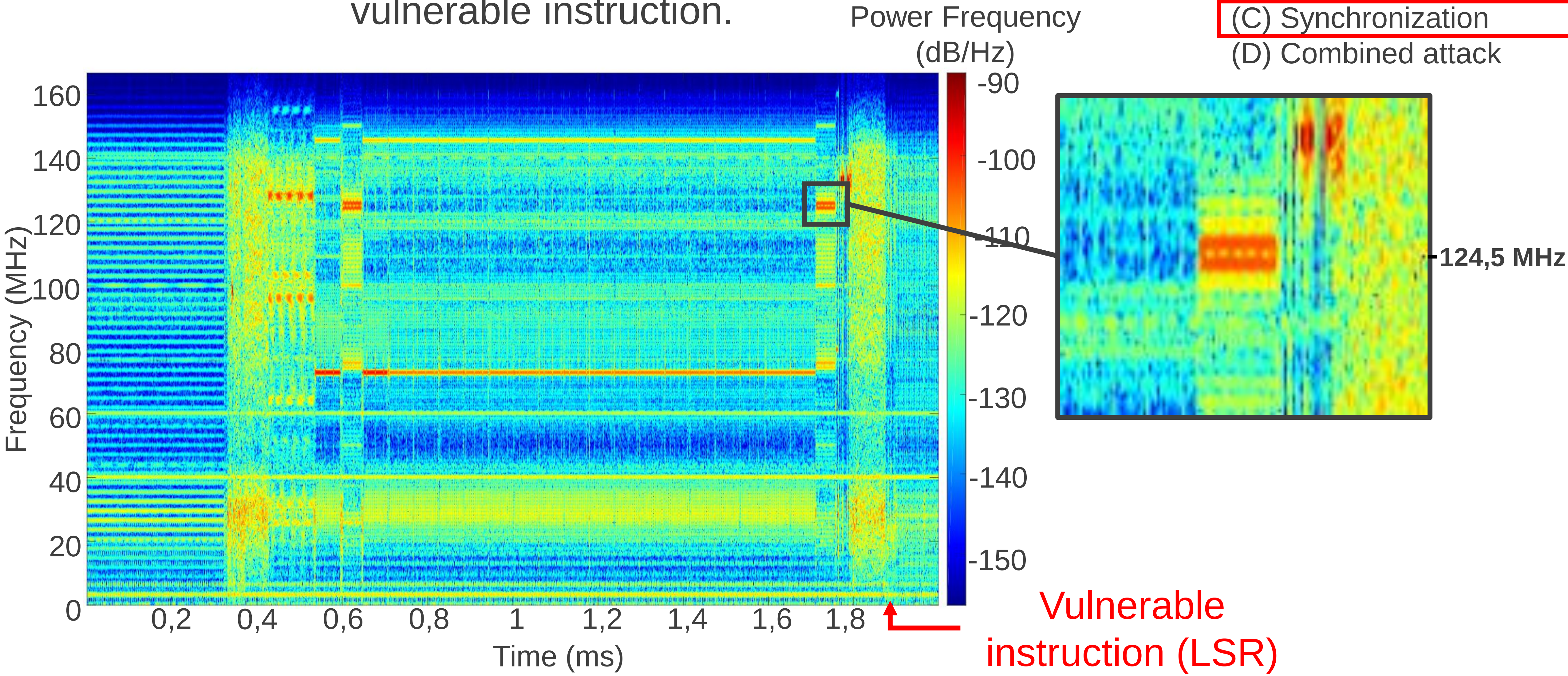
Step 2:



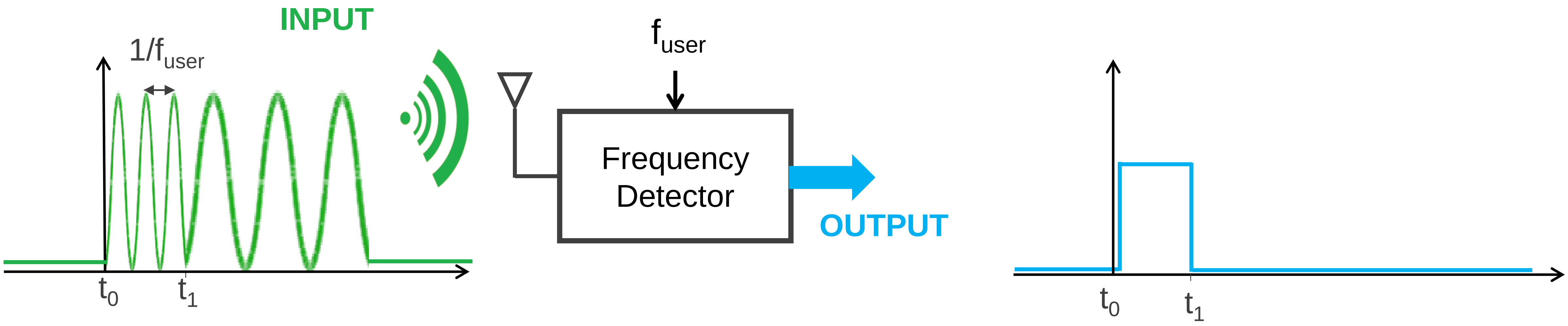
The attacker has a great degree of freedom in the event choice.
For Fault Injection it needs a real time analysis:
Analysis of high frequency events may be difficult

- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack

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



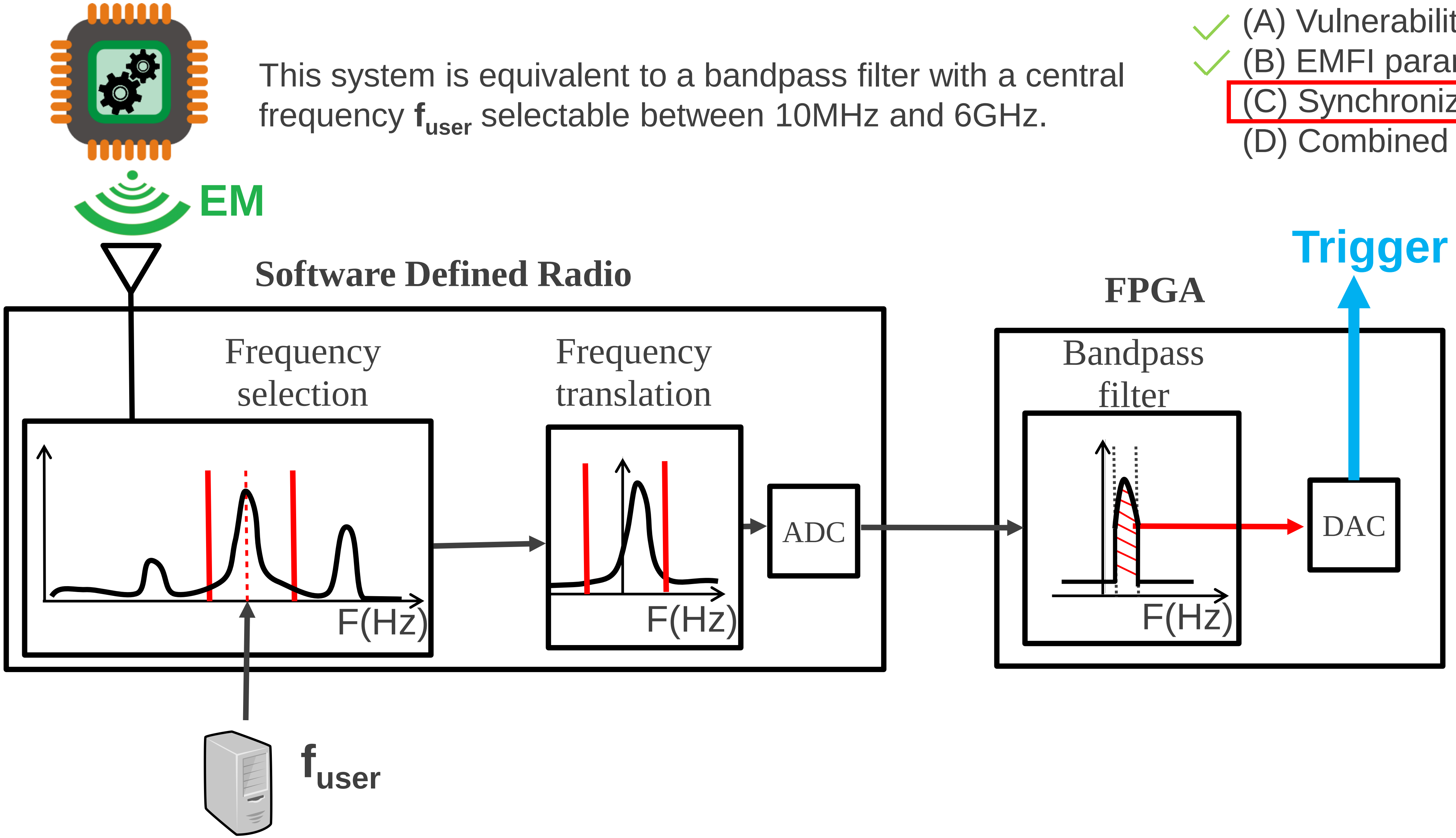
- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack



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

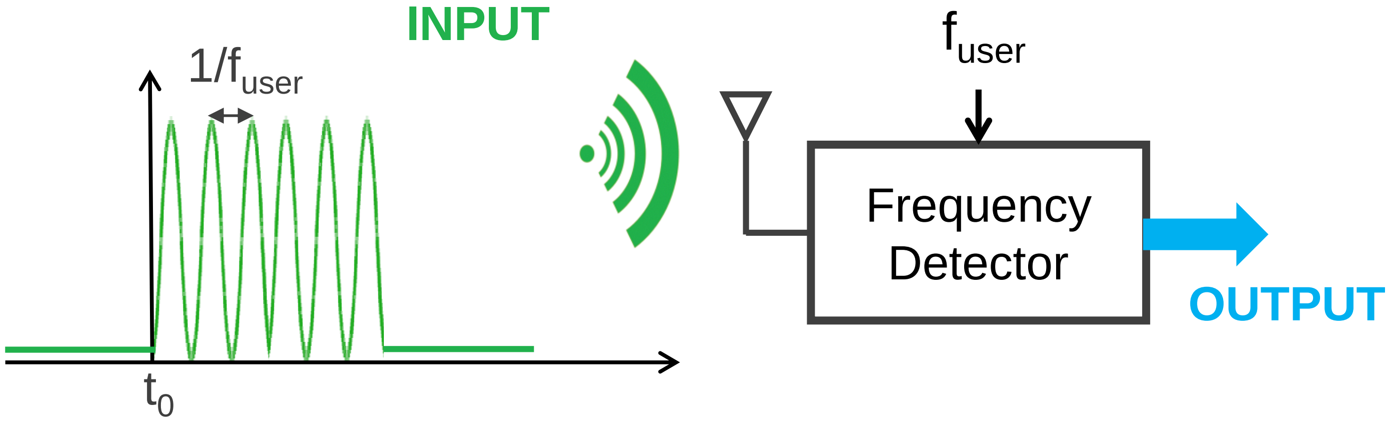
- (A) Vulnerability analysis
- (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack

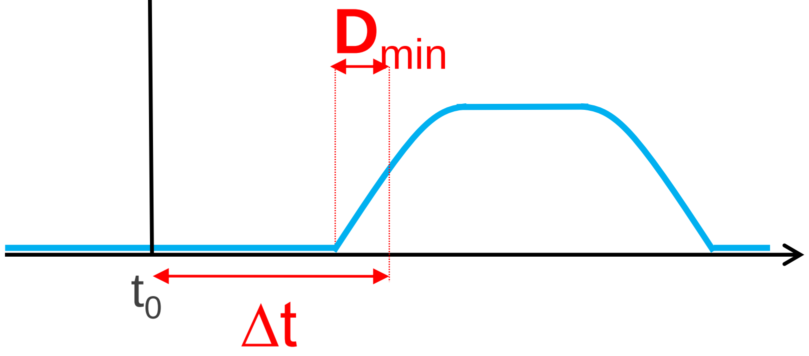
This system is equivalent to a bandpass filter with a central frequency $\mathbf{f_{user}}$ selectable between 10MHz and 6GHz.



Frequency detector performances

- (A) Vulnerability analysis ✓
- (B) EMFI parameters ✓
- **(C) Synchronization**
- (D) Combined attack



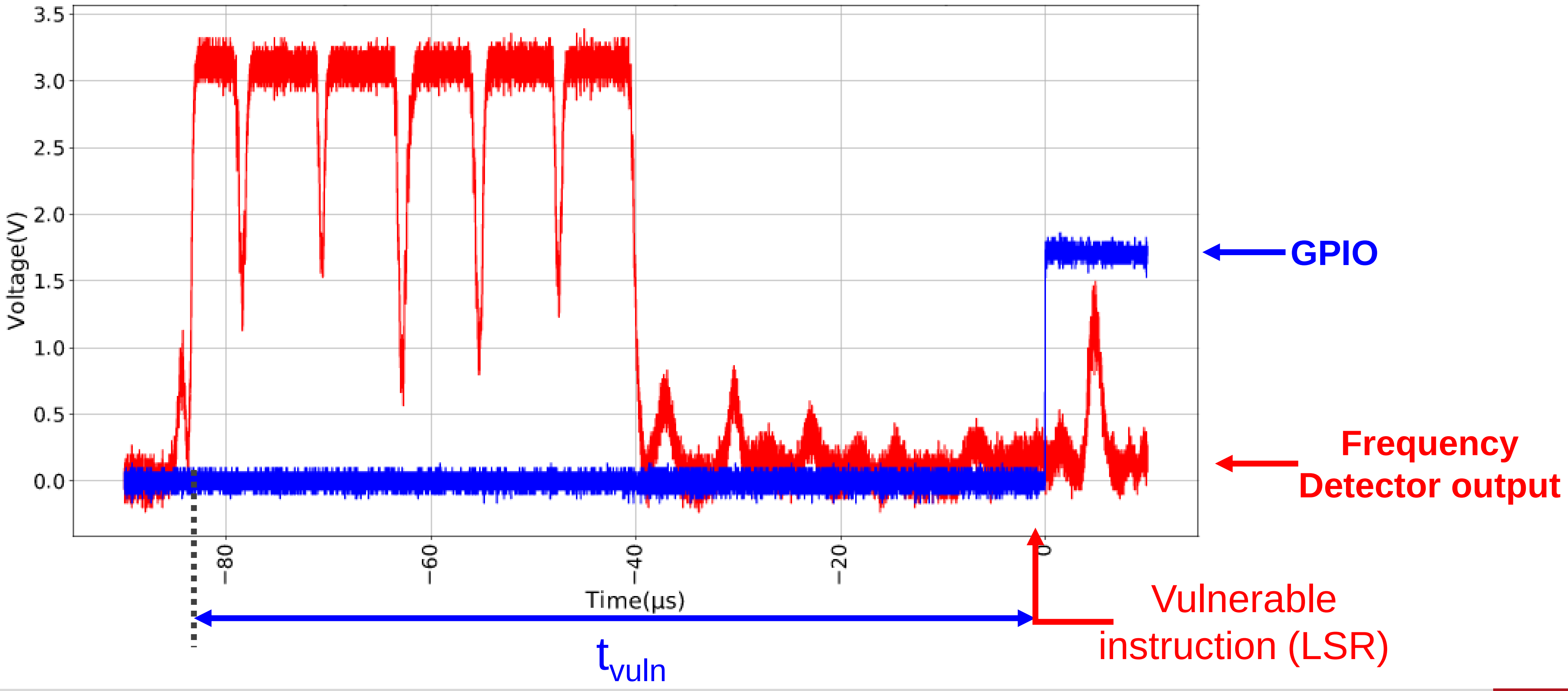


- Δt is the delay between the activation of f_{user} and the detection of f_{user}. This delay follows a normal distribution.
- In order to be detected, f_{user} needs to stay active during at least D_{min}.

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

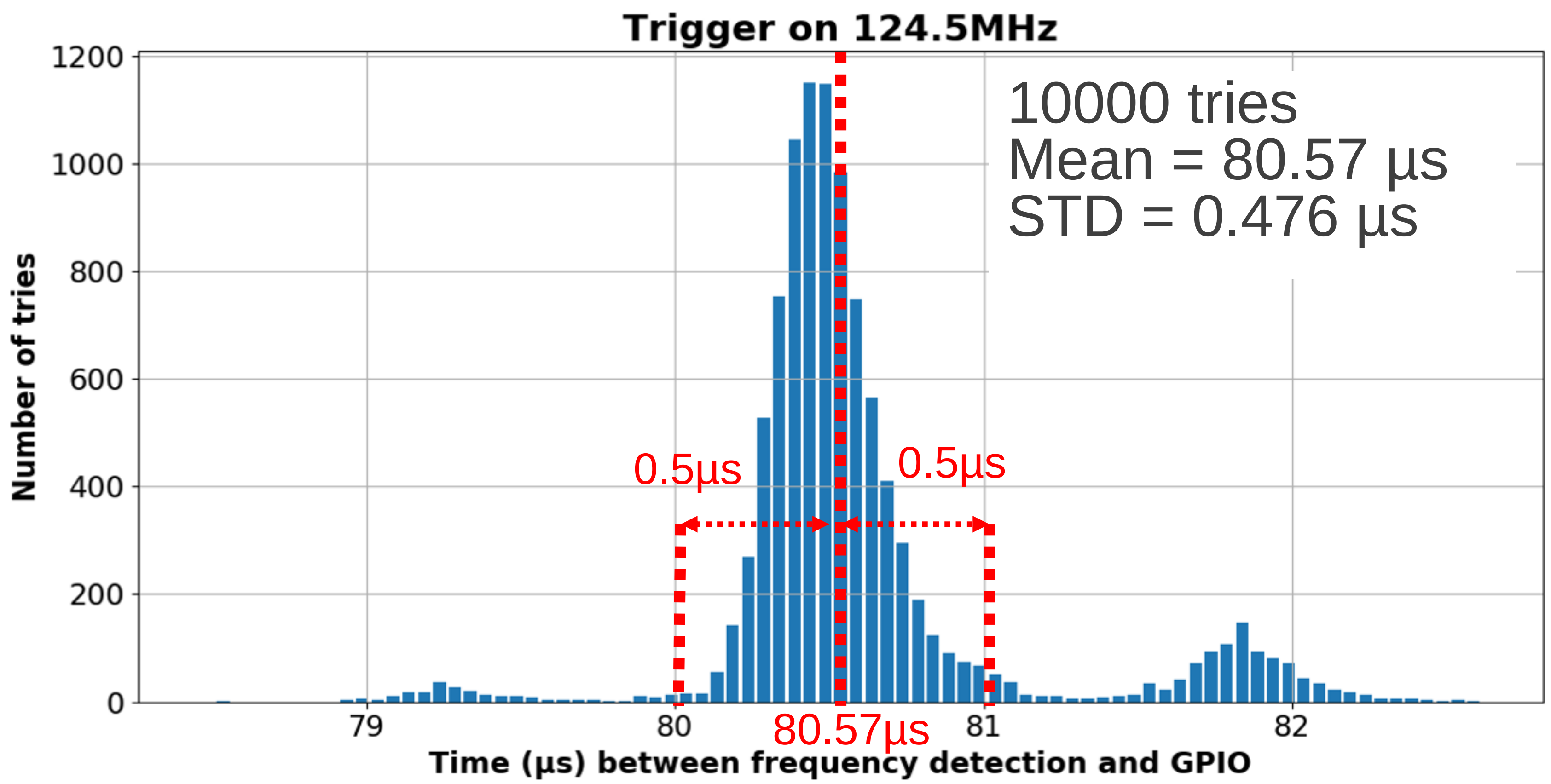
We use a modified Little Kernel which rises a GPIO just **after** the vulnerability. We set the frequency detector to trigger on the 124,5MHz frequency we identified before.

- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack



- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- **(C) Synchronization**
- (D) Combined attack

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



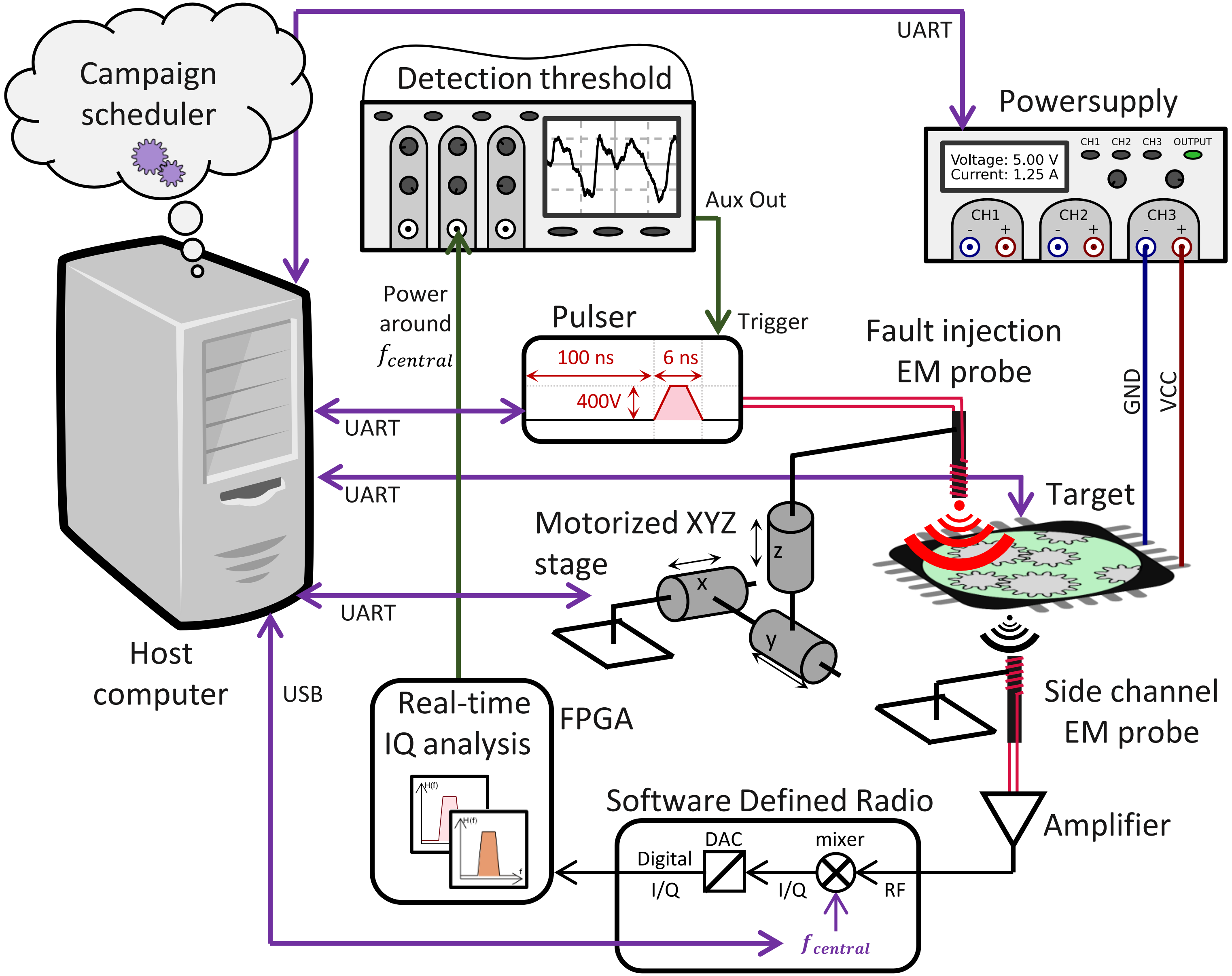
Clément Fanjas, Clément Gaine, Driss Aboulkassimi, Simon Pontié, Olivier Potin

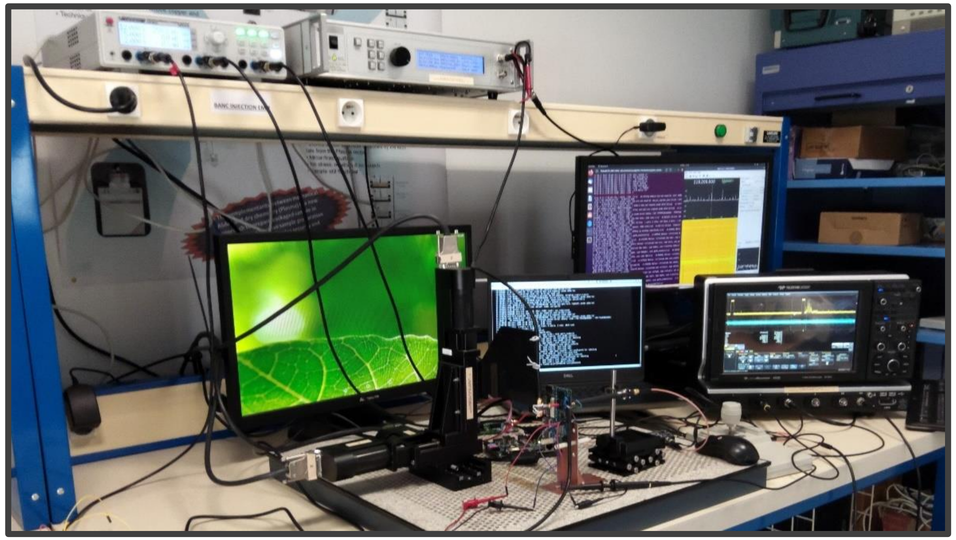
Comparison with other triggering events:

Triggering event	Mean value of vulnerability delay	STD value of vulnerability delay	Equivalent instruction number at 800MHz
Frequency detector	80,57µs	476ns	≈400 instructions
Trigger on UART character	113,8ms	2,06µs	≈1600 instructions
Board power-on	1,248s	5ms	≈4000000 instructions

Final experiment setup

- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- ✓ (C) Synchronization
- **(D) Combined attack**





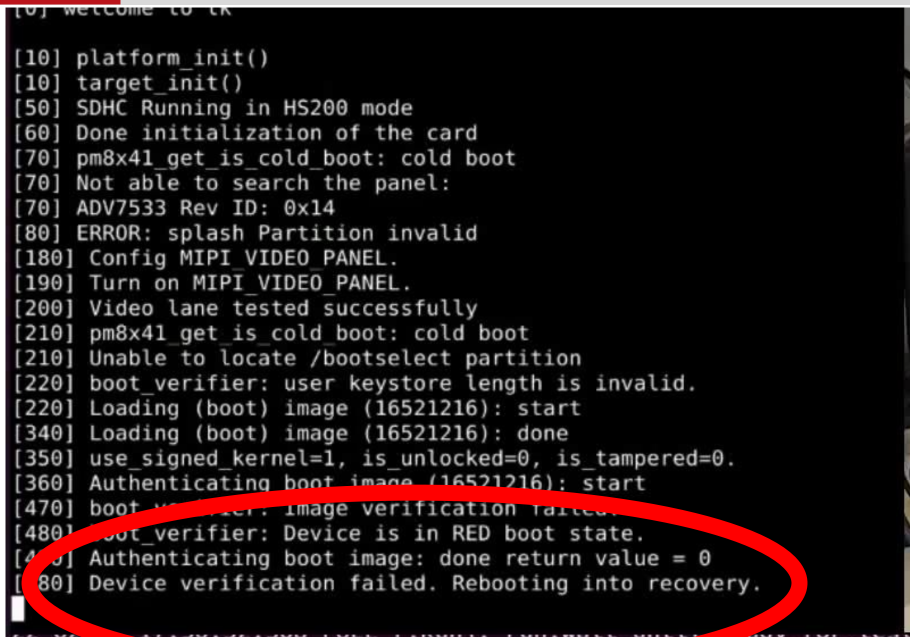
- ✓ (A) Vulnerability analysis
- ✓ (B) EMFI parameters
- ✓ (C) Synchronization
- (D) Combined attack

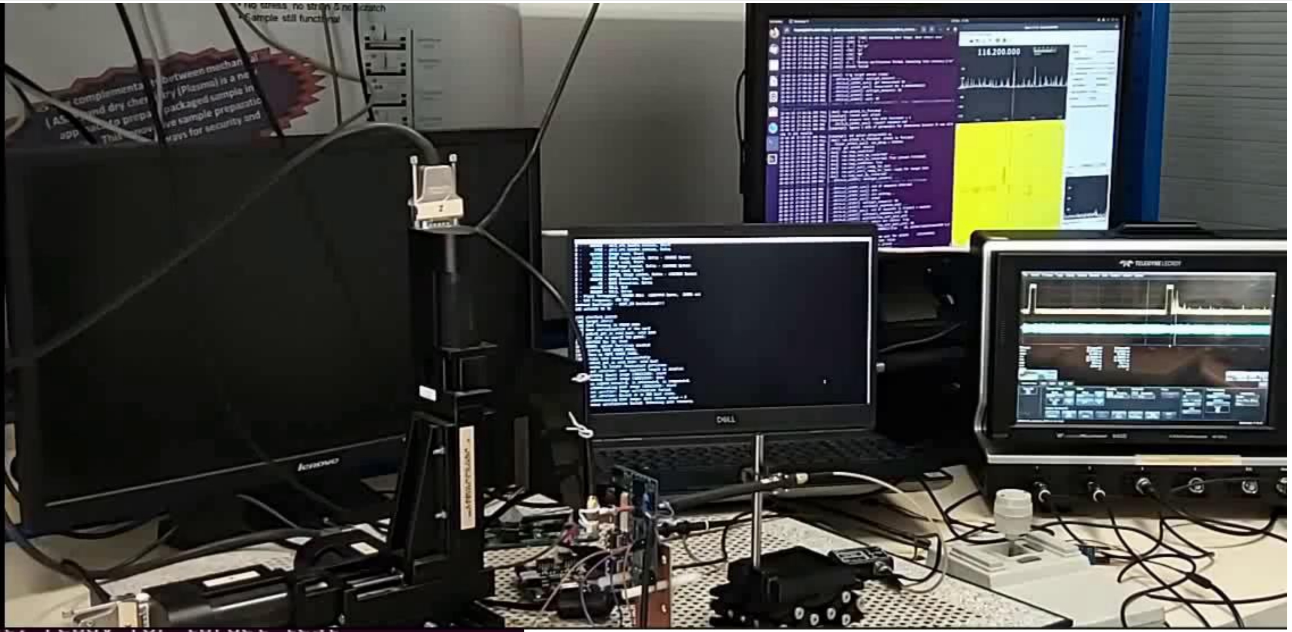
15000 attacks in 18h00:

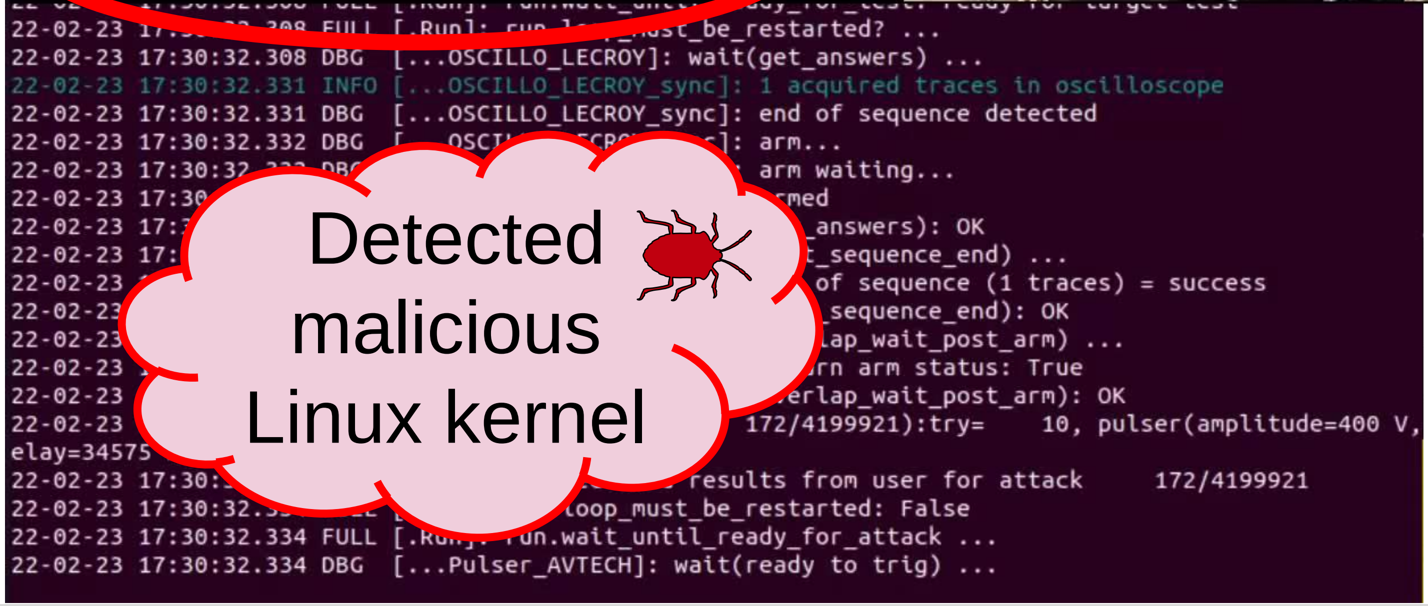
Scenarios	Results
Crash	7005 (46,777%)
No effect	7912 (52,75%)
Authentication bypass	83 (0,53%)

≈ 1 bypass every 15 minutes.

Results



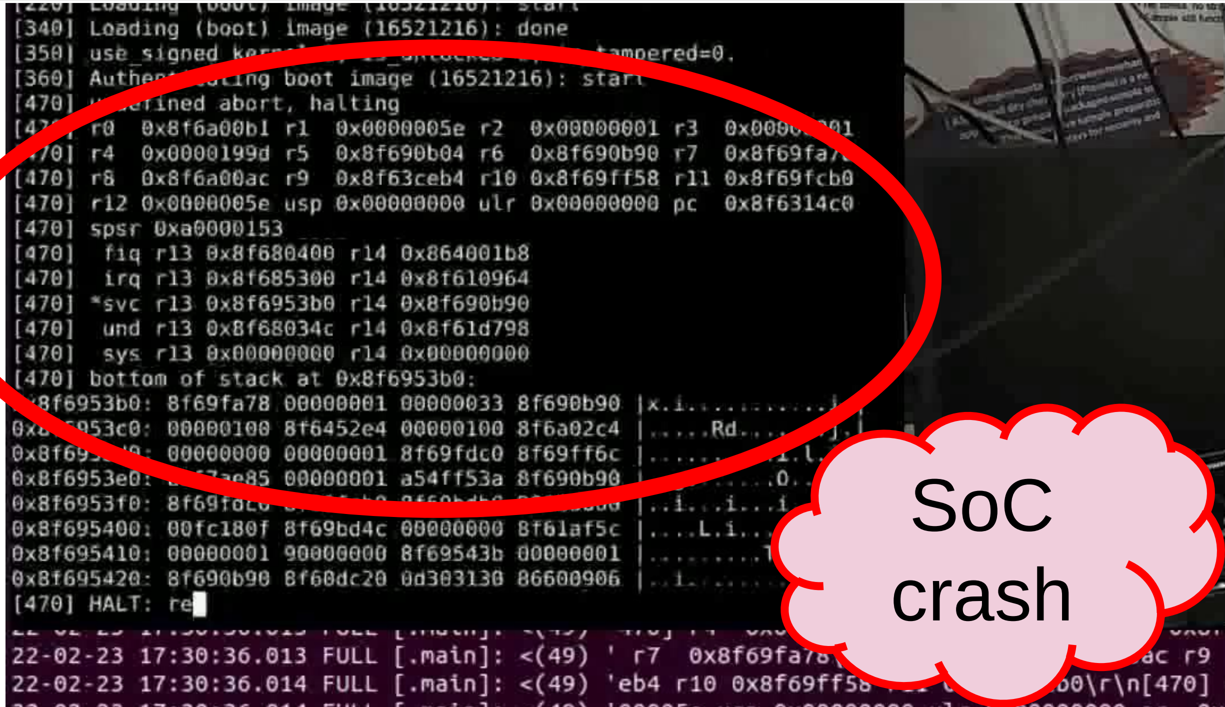


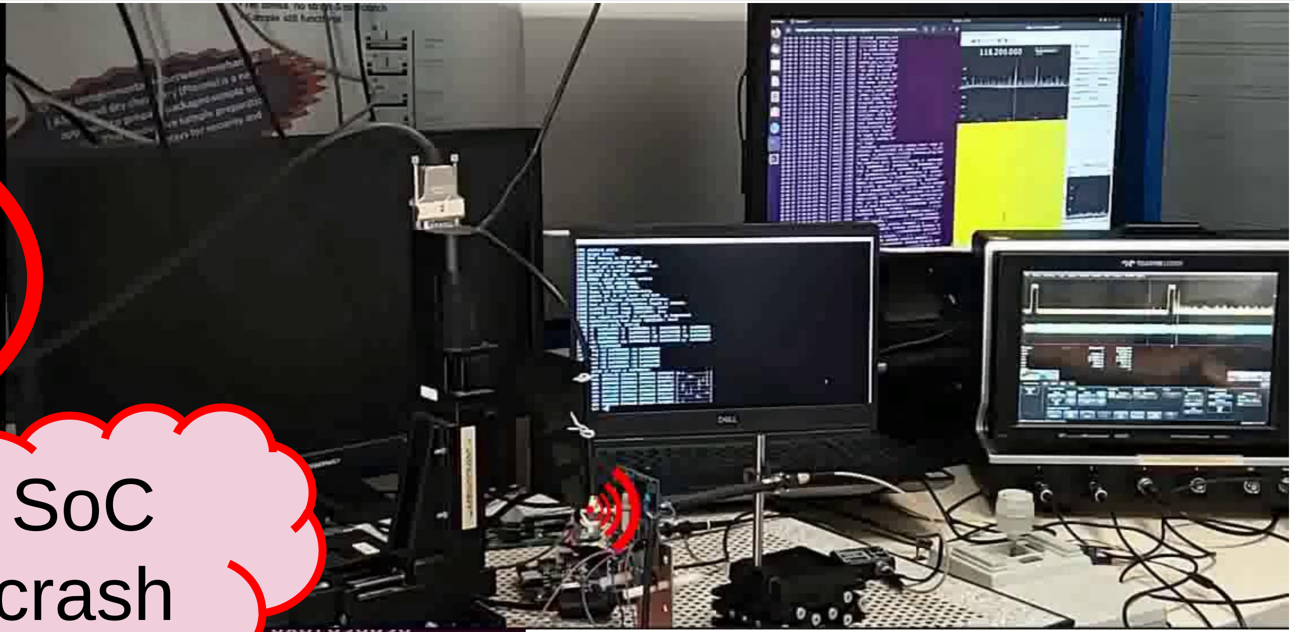


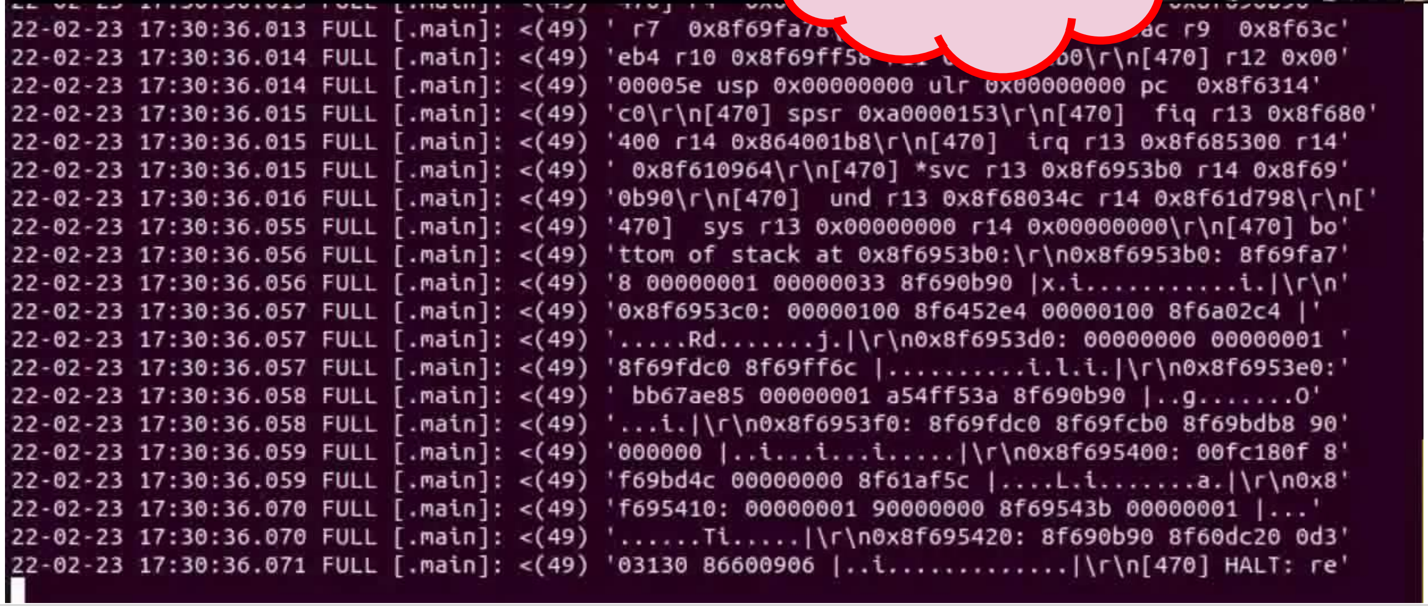
- **53%: detected malicious Linux kernel**
- 45%: timeout ou crash
- 1.67%: execution of malicious Linux kernel followed by a crash
- 0.53%: success, execution of malicious Linux kernel and Android OS.

Un contournement de Secure Boot toutes les 15 minutes

Results

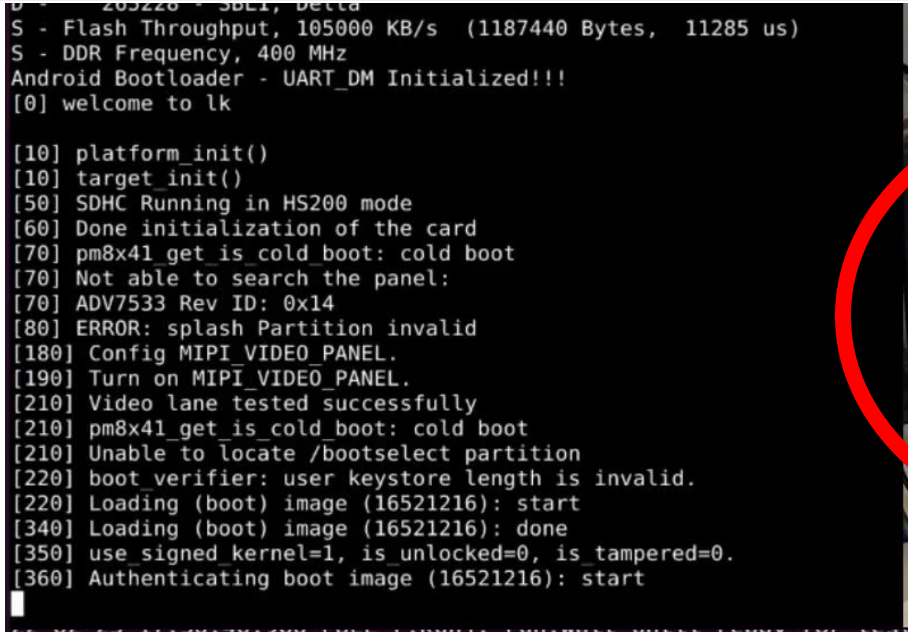




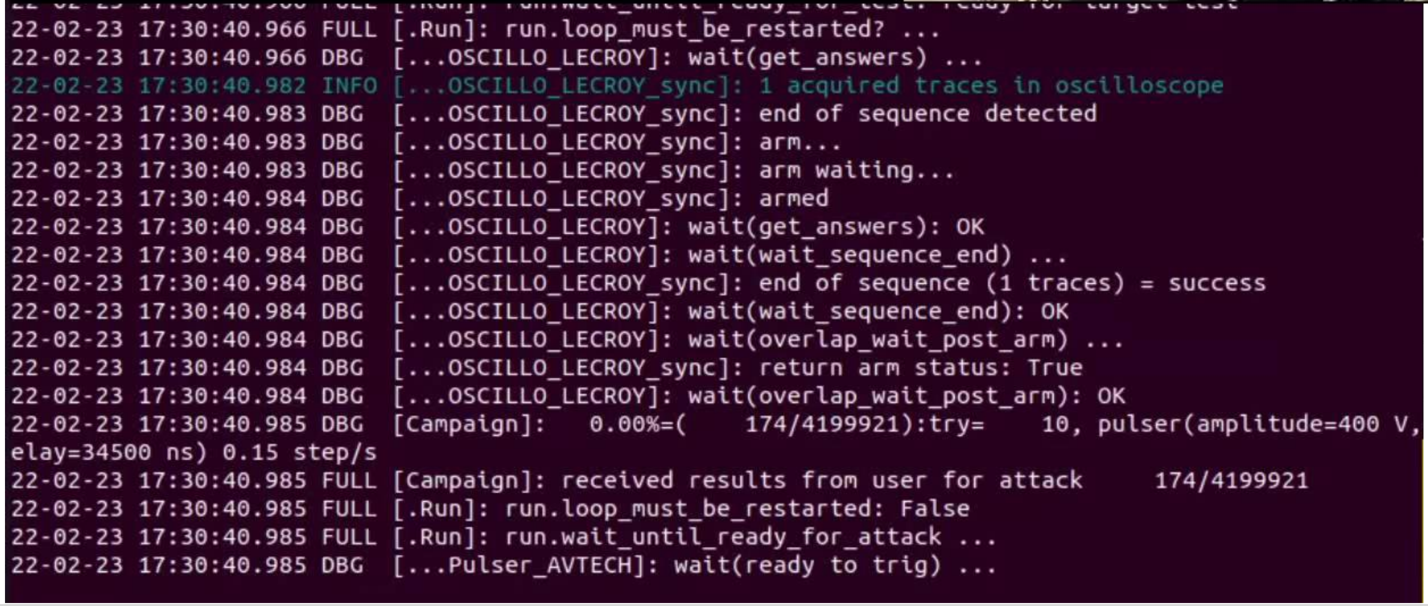


- 53%: detected malicious Linux kernel
- **45%:** timeout ou **crash**
- 1.67%: execution of malicious Linux kernel followed by a crash
- 0.53%: success, execution of malicious Linux kernel and Android OS.

Un contournement de Secure Boot toutes les 15 minutes

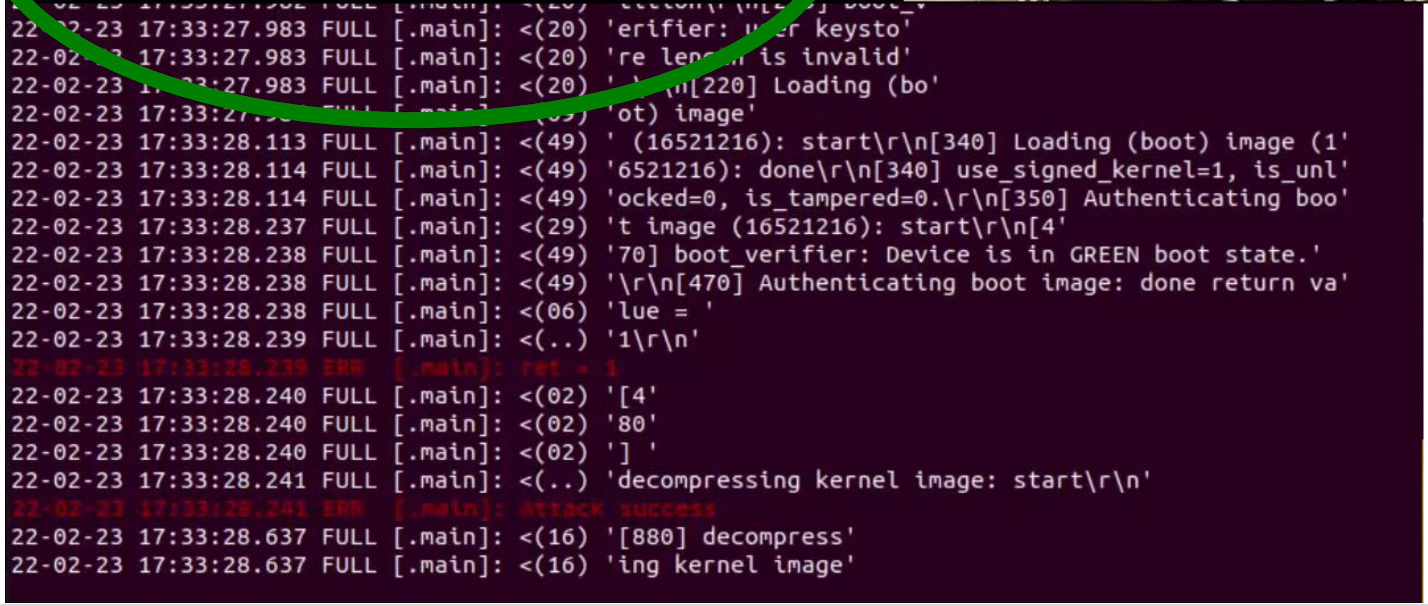


SoC crash: Blue screen



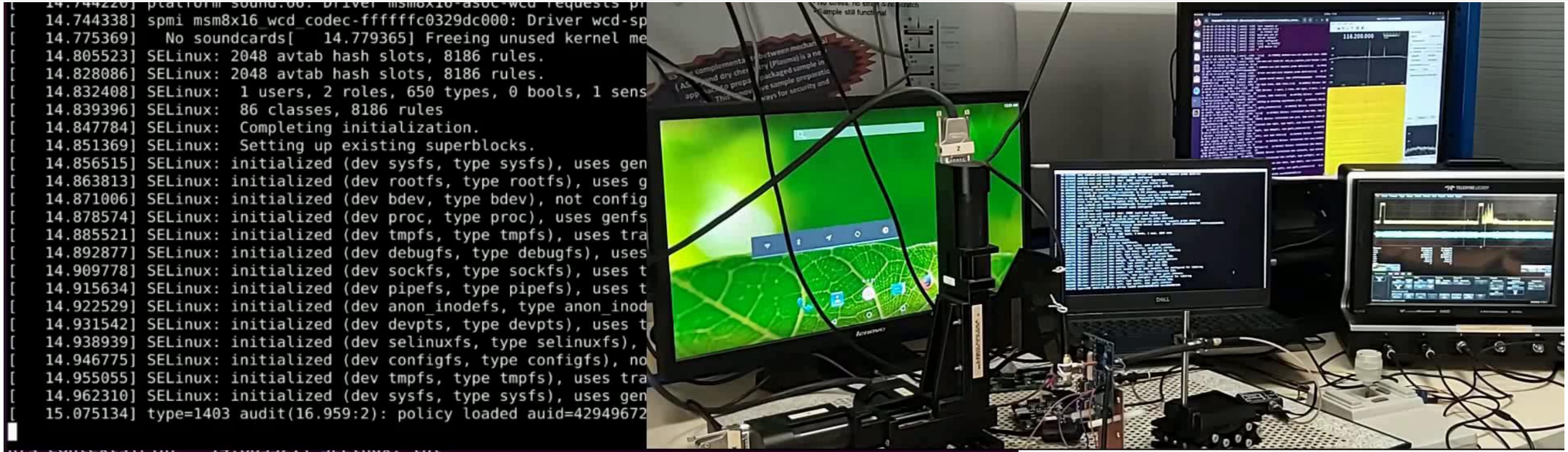
- 53%: detected malicious Linux kernel
- **45%: timeout** ou crash
- 1.67%: execution of malicious Linux kernel followed by a crash
- 0.53%: success, execution of malicious Linux kernel and Android OS.

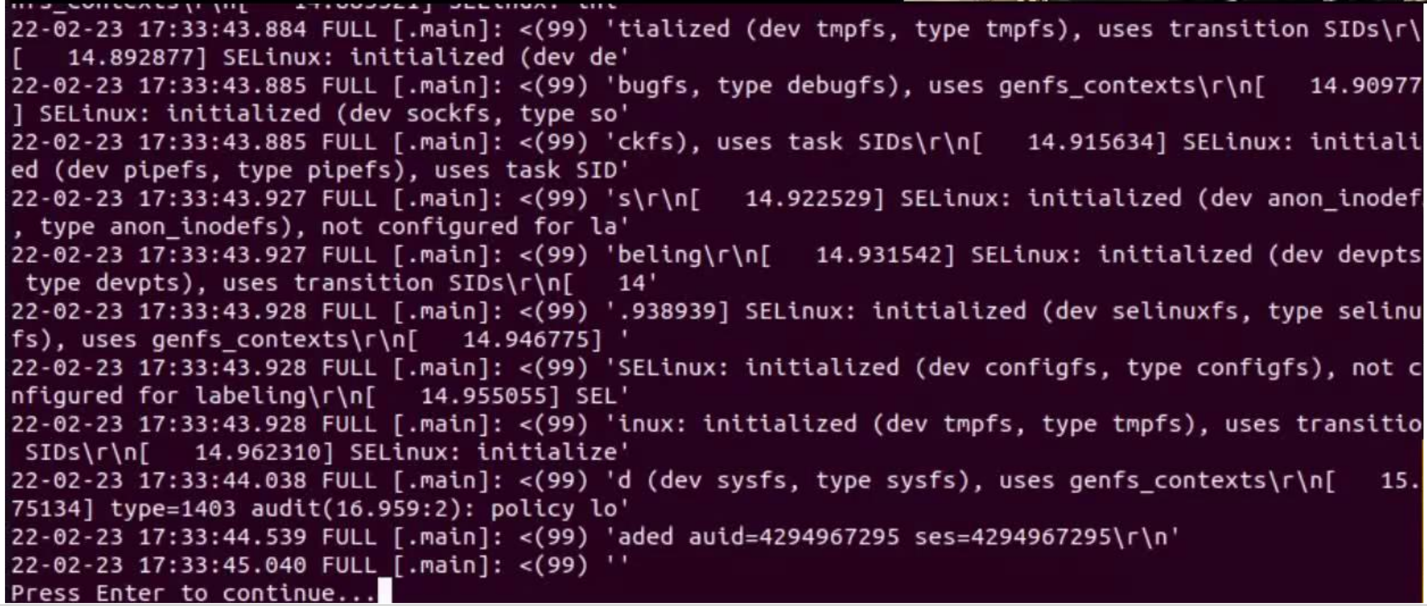
Un contournement de Secure Boot toutes les 15 minutes



- 53%: detected malicious Linux kernel
- 45%: timeout ou crash
- 1.67%: execution of malicious Linux kernel followed by a crash
- **0.53%: success, execution of malicious Linux kernel and Android OS.**

Un contournement de Secure Boot toutes les 15 minutes





- 53%: malicious Linux kernel
- 45%: timeout ou crash
- 1.67%: execution of malicious Linux kernel followed by a crash
- **0.53%: success, execution of malicious Linux kernel and Android OS.**

Un contournement de Secure Boot toutes les 15 minutes

- (A) Vulnerability analysis
- (B) EMFI parameters
- (C) Synchronization
- (D) Combined attack

Conclusion

- We identified a vulnerability to fault injection in the Secure Boot of our target.
- We present a new synchronization method to trigger hardware attacks on high frequency event.
- By using this synchronization method we successfully synchronized a fault injection with the vulnerability identified, bypassing the Linux Kernel authentication step of our target Secure Boot.

Prospect

- Use the same methodology on other targets:
    - ➔ Hardware: smartphones SoC
    - ➔ Software: previous Secure Boot steps
- Improving the hardware of our frequency detector

Thank you for your attention.

References

Secure Boot attacks:

1. The forgotten threat of voltage glitching: A case study on nvidia tegra x2 socs. (2021)

Otto Bittner, Thilo Krachenfels, Andreas Galauner, and Jean-Pierre Seifert

2. Controlling pc on arm using fault injection (2016)

Niek Timmers, Albert Spruyt, and Marc Witteman

3. Laser-induced fault injection on smartphone bypassing the secure boot (2017)

Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien Ermeneux

4. BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection (2017)

Ang Cui and Rick Housley