

raelize

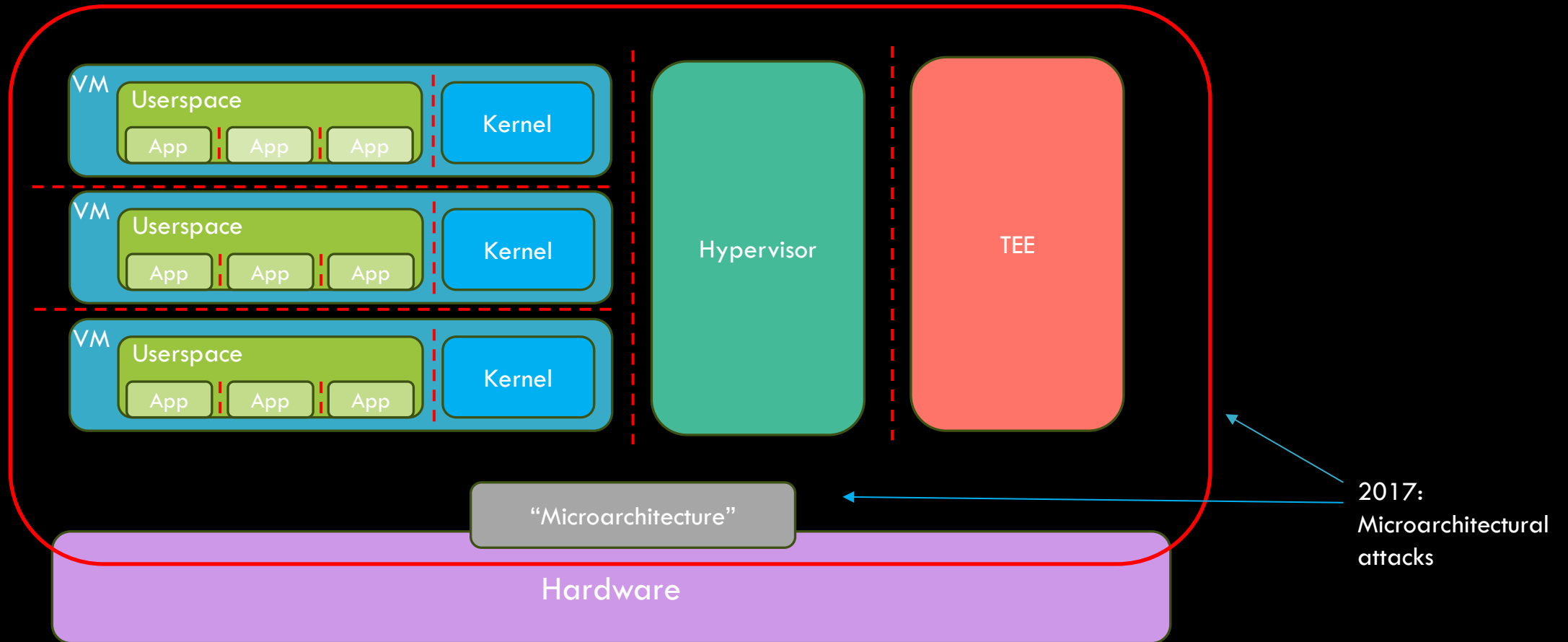


# *False Injections: Tales of Physics, Misconceptions and Weird Machines*

Cristofaro Mune  
[cristofaro@raelize.com](mailto:cristofaro@raelize.com)  
[@pulsoid](https://twitter.com/pulsoid)

Niek Timmers  
[niek@raelize.com](mailto:niek@raelize.com)  
[@tieknimmers](https://twitter.com/tieknimmers)

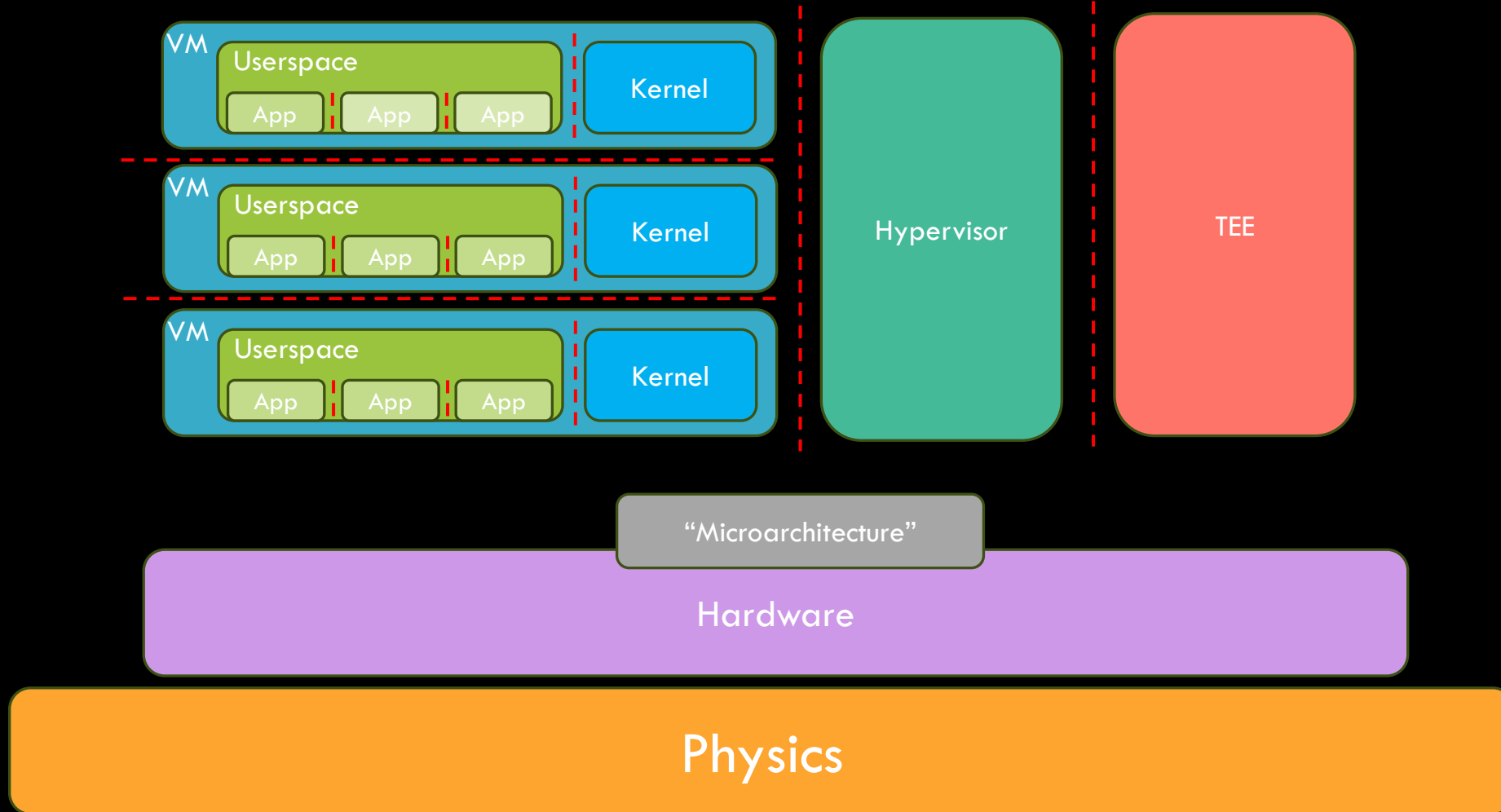
# Security boundaries



# Notes from Micro-architectural attacks [2017]

- Security models aren't **just** a Software (SW) thing
- Most of the Hardware (HW) has no **idea** of security boundaries:
  - unless factored in during design
- HW resources **shared** across security boundaries can be problematic
- It's **painful** to recover

# Are we missing anything?





# Walking on thin ice...

- The whole computing model assumes that:
  - the **right** logical values
  - are **correctly** represented
  - at the rising **edge**
  - of each **clock** cycle.
  - Everywhere
- That's why we have constraints on operating conditions (e.g. temperature range)

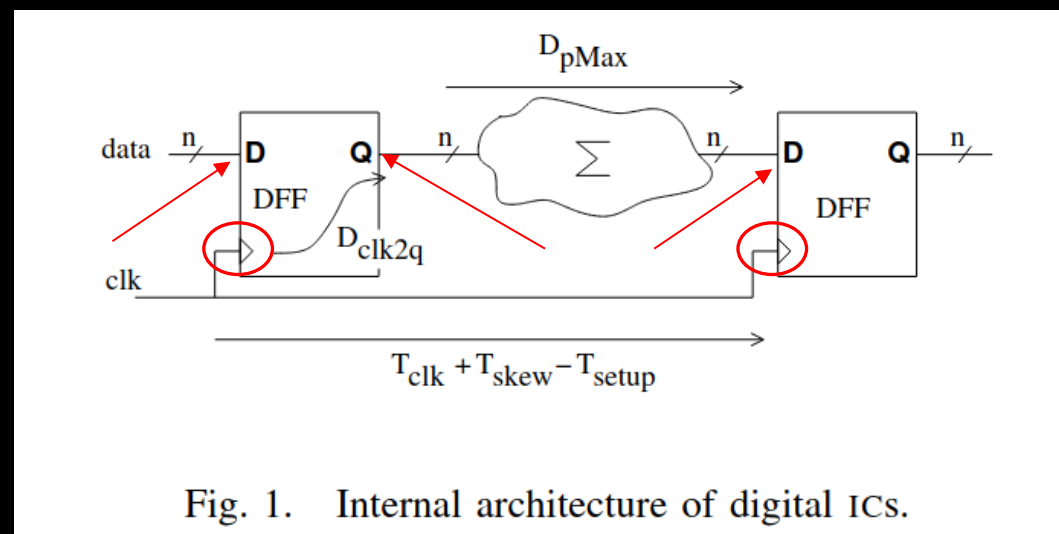
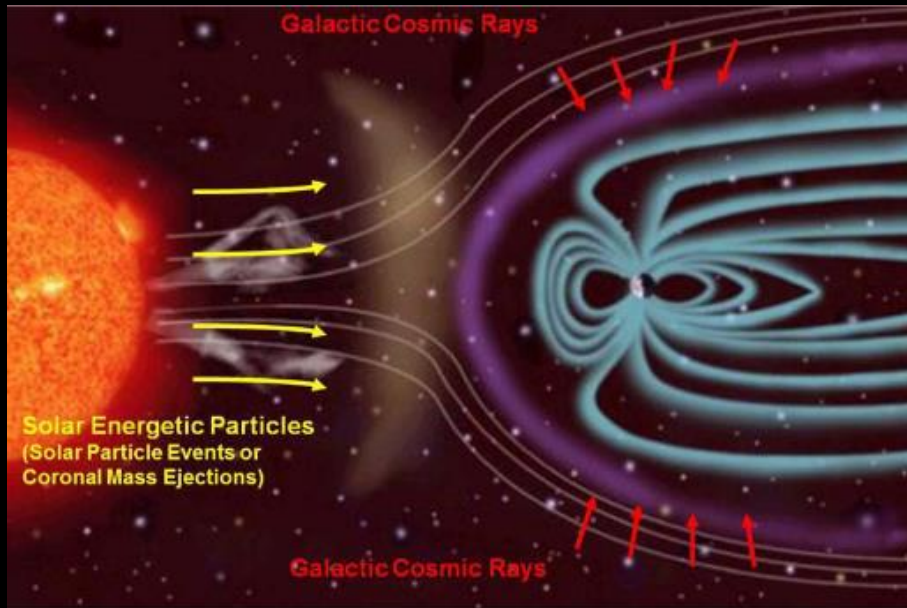


Fig. 1. Internal architecture of digital ICs.

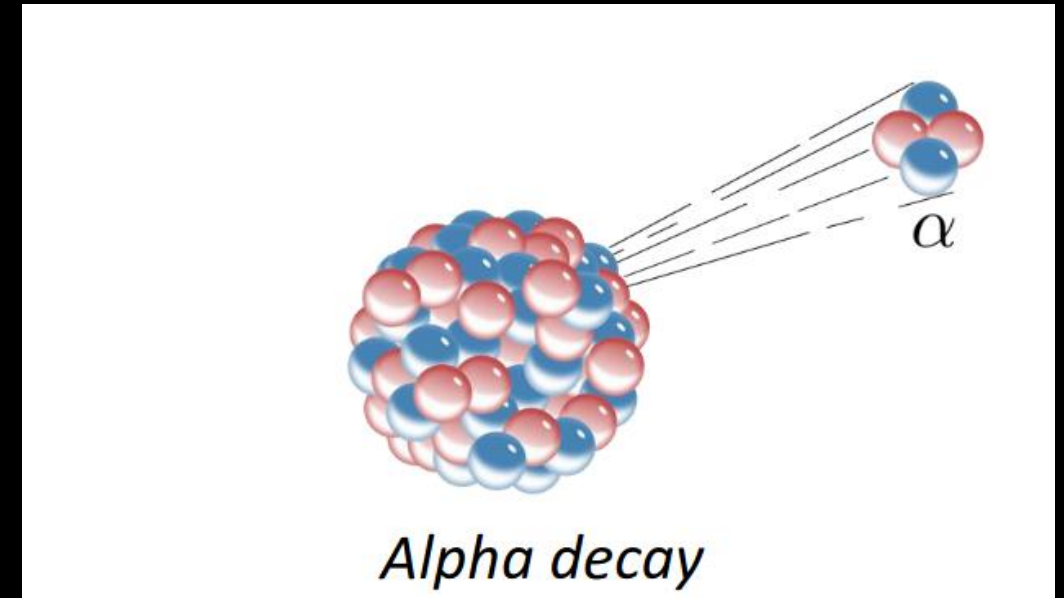
*Zussa et al – “Analysis of the fault injection mechanism related to negative and positive power supply glitches using an on-chip voltmeter” - [ZDRC2014]*

## What can go wrong?

# Examples: Natural Phenomena



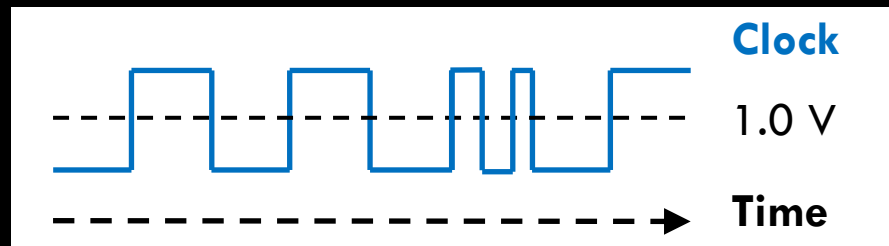
Ziegler, Lanford –“Effects of cosmic rays on computer memories”  
(1979)



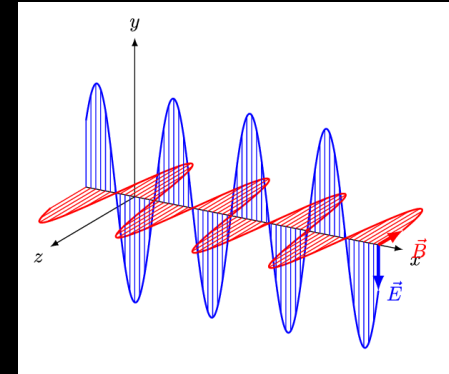
May, Woods –“Alpha-particle-induced soft errors in dynamic memories”  
(1979)

# Known (attack) techniques

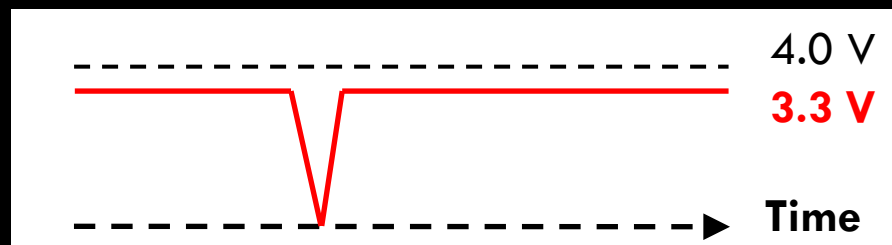
Clock



Electro-magnetic field



Voltage



Temperature



Laser ("Nexus-6" kitten)

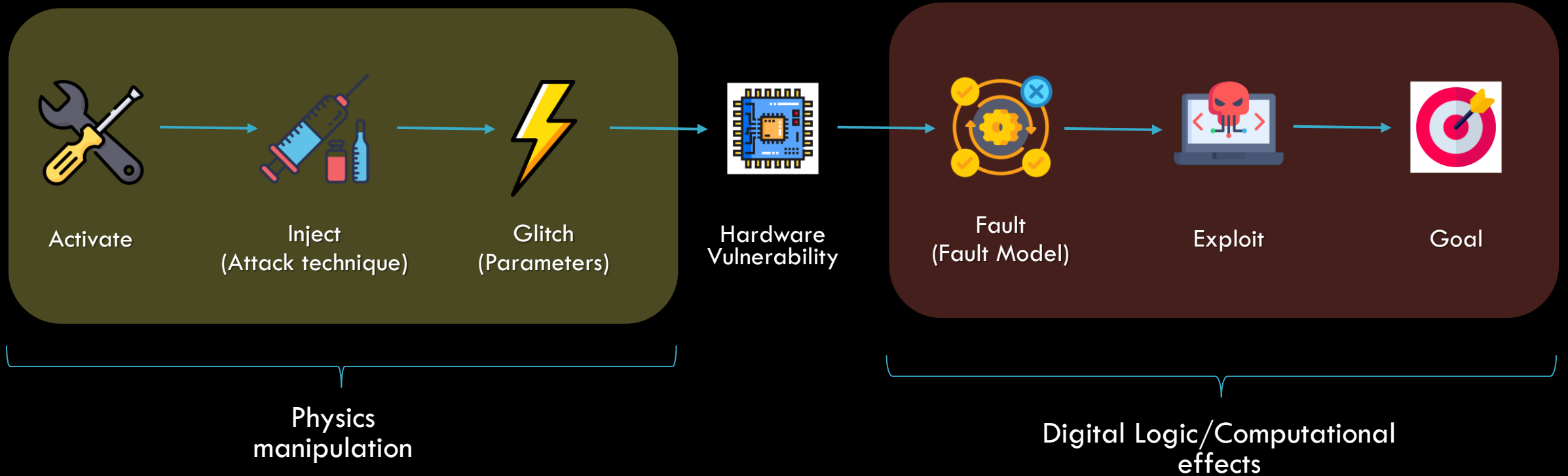


Interestingly...



Most of them involve transfer of **energy**

# Fault Injection Reference Model (FIRM)



# Physics and Computing

- The general relationship between Physics and Computing is mostly **unexplored** in CS:
  - R. P. Feynman – “Feynman Lectures on Computation” – Caltech lectures
- We are aware of “physical attacks”:
  - Mostly seen as a **computing** problem
- More in general, **physics** fundamentals are just seldomly discussed:
  - In academic papers, in the industry, as well as in the security community
  - Maybe “left as an exercise to the reader...”?

## Consequences

- Imprecise descriptions
- Perpetuation of beliefs
- Incorrect/sub-optimal modeling
- (Lack of) identification of **fundamental** problems

*How far does the rabbit hole go?*

# Goals

- Show examples of gaps in our approaches
  - We will be (mostly) using **Fault Injection** (FI) for our investigation
- Realize the potential of including physics in our **model** of computing:
  - We will be identifying threats, opportunities and attacks



Voltage glitch shape.

## Some widespread statements...

*“You want to affect a single instruction...”*

*“you have to glitch WHEN the instruction is being executed...”*

*“CPU is fast. You need to be very fast...”*

*“Hit within one single clock cycle...”*

*“Your glitch needs to be sharp...”*

# Assumptions?

- Fault is introduced by the glitch shape
  - regardless of target's physical parameters. (e.g.: Impedance, amount of stored energy, ...)
- Glitch effectiveness depends on its **shape/sharpness**
- **Precision** depends on sharpness
  - To be adjusted to CPU **speed**
- Glitch is somewhat **instantaneous**

## Physics has objections...

- Glitch effect cannot travel faster than light
- We need to consider **two** different times:
  - Time of glitch ( $T_g$ )
  - Time of fault ( $T_f$ )

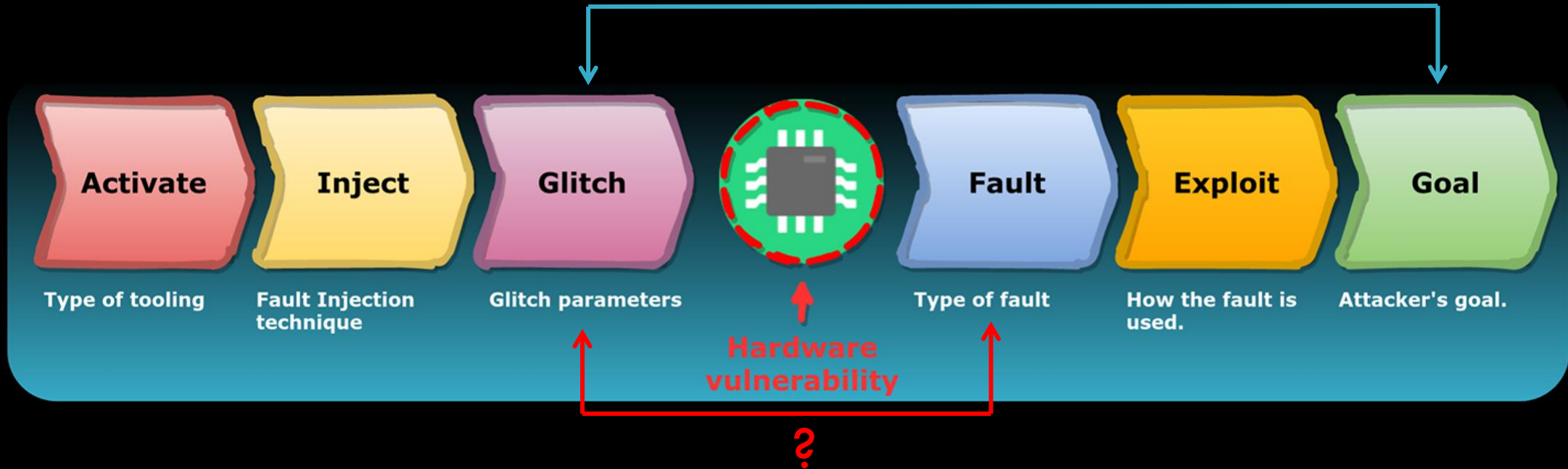


Glitch effect cannot be **instantaneous**

## In literature: Shaping the Glitch

- Effectiveness of glitch shape has been **investigated**:
  - “Shaping the Glitch: Optimizing Voltage Fault Injection Attacks” - Bozzato et al
  - Confirmed that arbitrarily shaped voltage glitches may be effective
- Tests performed targeting security protections preventing firmware dump
- No actual analysis of effects on CPU instructions execution

# Shaping the Glitch: Research approach



- Measured parameters shaping vs attack success
- No analysis on the underlying **physics** mechanisms that causes faults:
  - Reference to [\[ZDRC2014\]](#) (discussed later)

## Still many questions...

- Has a glitch to be **sharp** in order to affect a single instruction?
- Does a glitch need to be faster than a single clock cycle?
- Are multiple glitch **shapes** possible and effective in attackin CPU code execution?
  - Or are we just constrained to a single shape?

...**without** an answer (yet)

*Let's perform some experiments!*

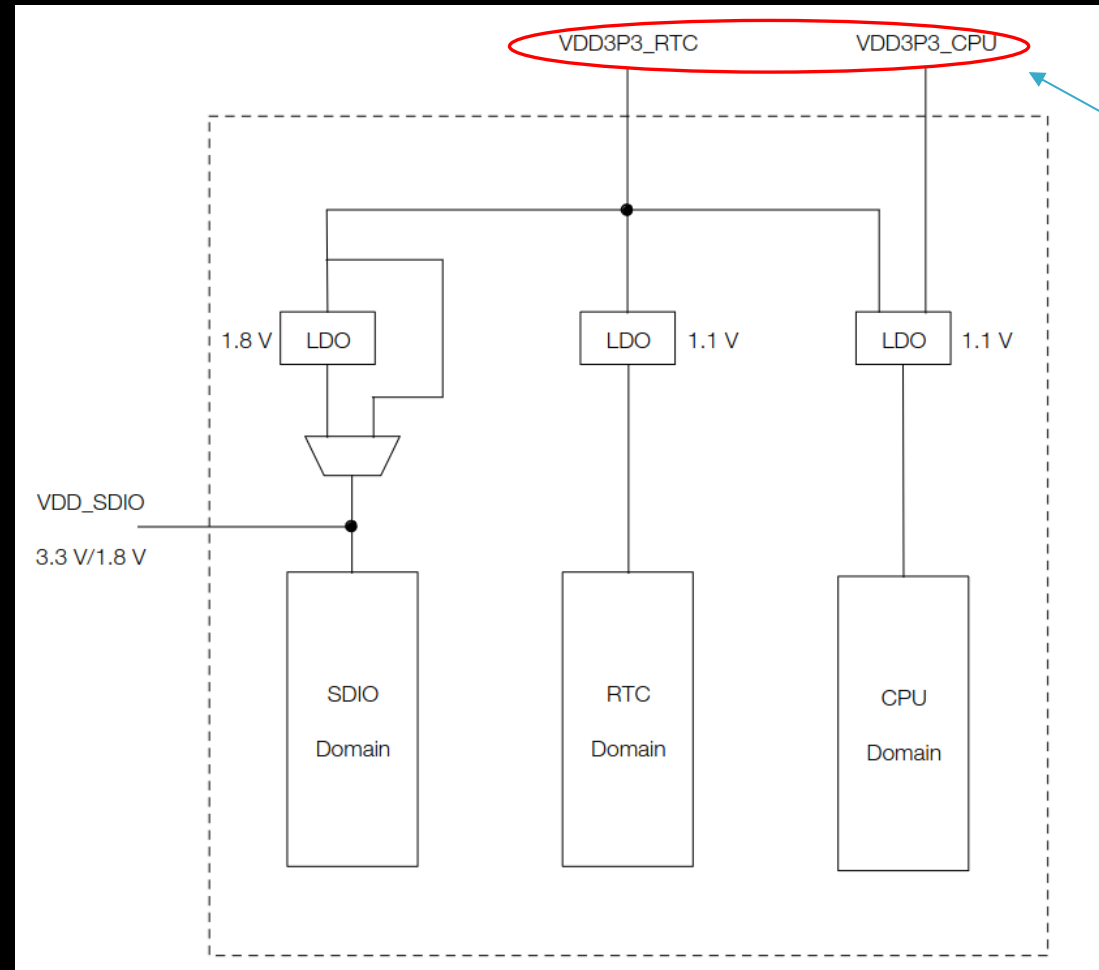


## Target: Espressif ESP32 (D0WDQ6)

- A feature-rich SoC with integrated Wi-Fi and Bluetooth connectivity
- Relevant (for us) features:
  - Clock speed: 80, 160, 240 MHz
  - Nominal voltage: 3.3 V
  - CPU architecture: Xtensa



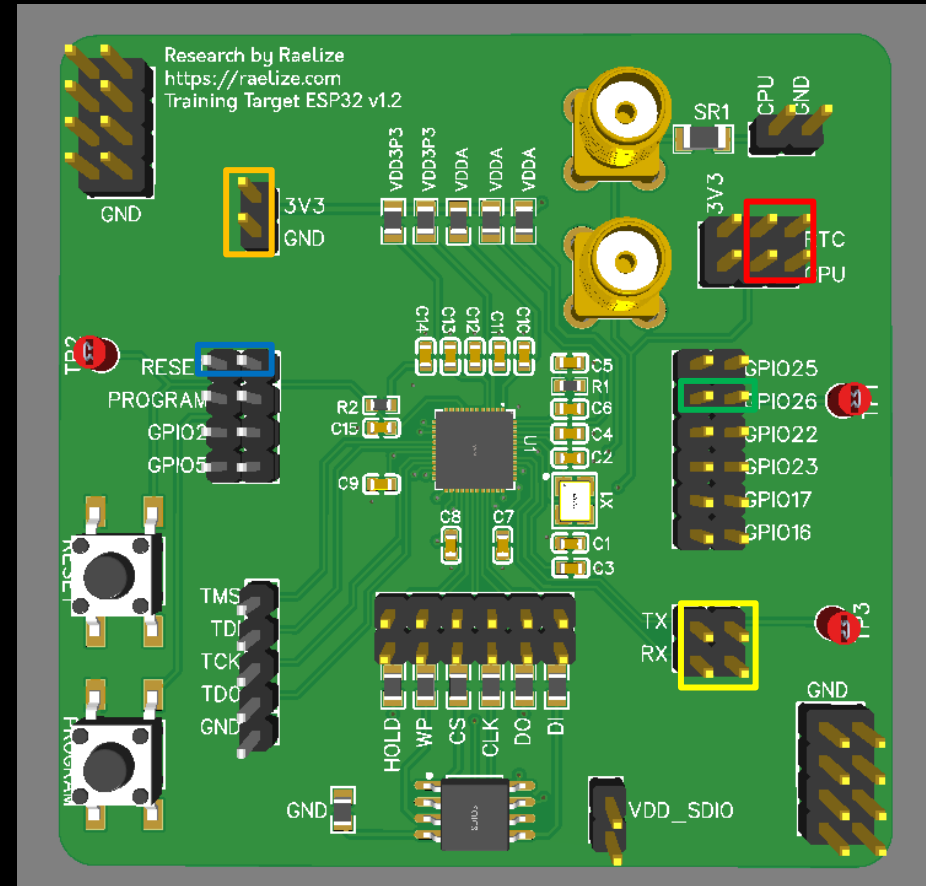
# Espressif ESP32 – Power Scheme



We glitch both CPU and RTC at the same time

# Raelize ESP32 Training Target v1.2

- Custom board for easy signal access during FI experiments:
  - **Reset**
  - **UART TX/RX**
  - **Trigger**
  - **VCC (main power @ 3.3V)**
    - used for subsystems other than CPU. E.g. Flash
  - **Voltage Glitch (CPU + RTC)**



# Generating a voltage glitch: techniques

- Original power source is retained:
  - Power line is pulled down to GND (“crowbar”)
  - Used by common hacking tools
- Original power source is replaced:
  - power supplied to the target is fully controlled in the experiment

*We are going to use the latter*

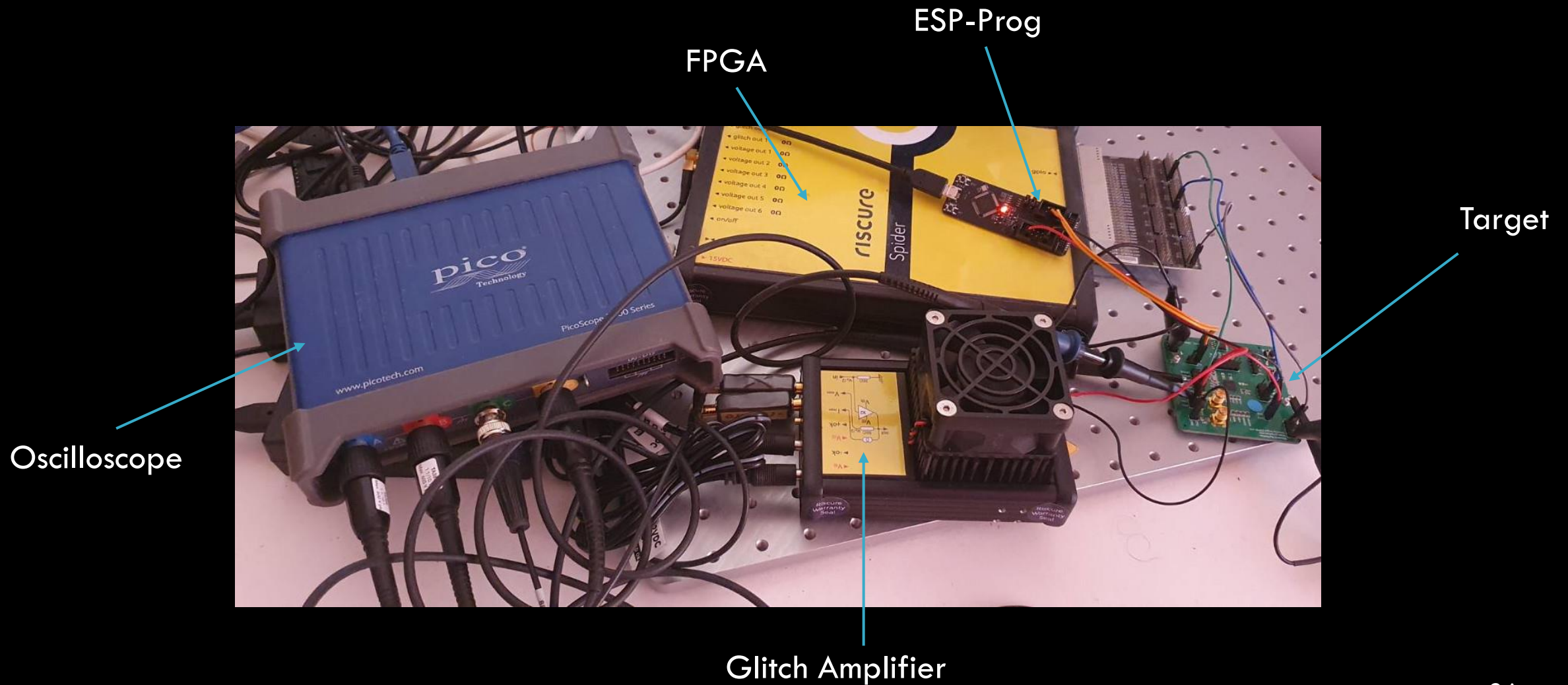
# Our setup

- Riscure Spider:
  - FPGA used for
    - Glitch generation
    - Glitch timing
    - Target reset
- Riscure Amplifier:
  - More stable glitch
- Espressif ESP-PROG:
  - Serial communications
  - Powering the main target power rail (3.3V)



\* Source: Riscure website

In real life...



# Target code: single add instruction

```
17 #define addi1 "addi.n a6, a6, 1;"
18
19 #define nop1 "nop;"
20 #define nop2 nop1 nop1
21 #define nop4 nop2 nop2
22 #define nop8 nop4 nop4
23 #define nop16 nop8 nop8
24 #define nop32 nop16 nop16
25 #define nop64 nop32 nop32
26 #define nop128 nop64 nop64
27 #define nop256 nop128 nop128
28 #define nop512 nop256 nop256
29 #define nop1024 nop512 nop512
30
31 ...
32
33 GPIO_OUTPUT_SET(26,1);
34
35 asm volatile (
36     ...                // Other regs initialization
37     "movi a6, 0;"
38     ...                // Other regs initialization
39     nop1024
40     addi1
41     nop1024
42     "mov %[counter], a6;"
43     : [counter] "=r" (counter)
44     :
45     : "a2", "a3", "a4", "a5", "a6", "a7", "a8", "a9", "a10", "a11", "a12", "a13", "a14"
46 );
47
48 GPIO_OUTPUT_SET(26,0);
49
50 esp_rom_printf("XXXX%08xYYYY%08xZZZZ\r\n", counter, counter);
```

Add instruction: adds 1 to a6

NOP Macros

Trigger (GPIO26): Up

1024 NOPs

Our Target: 1 add instruction

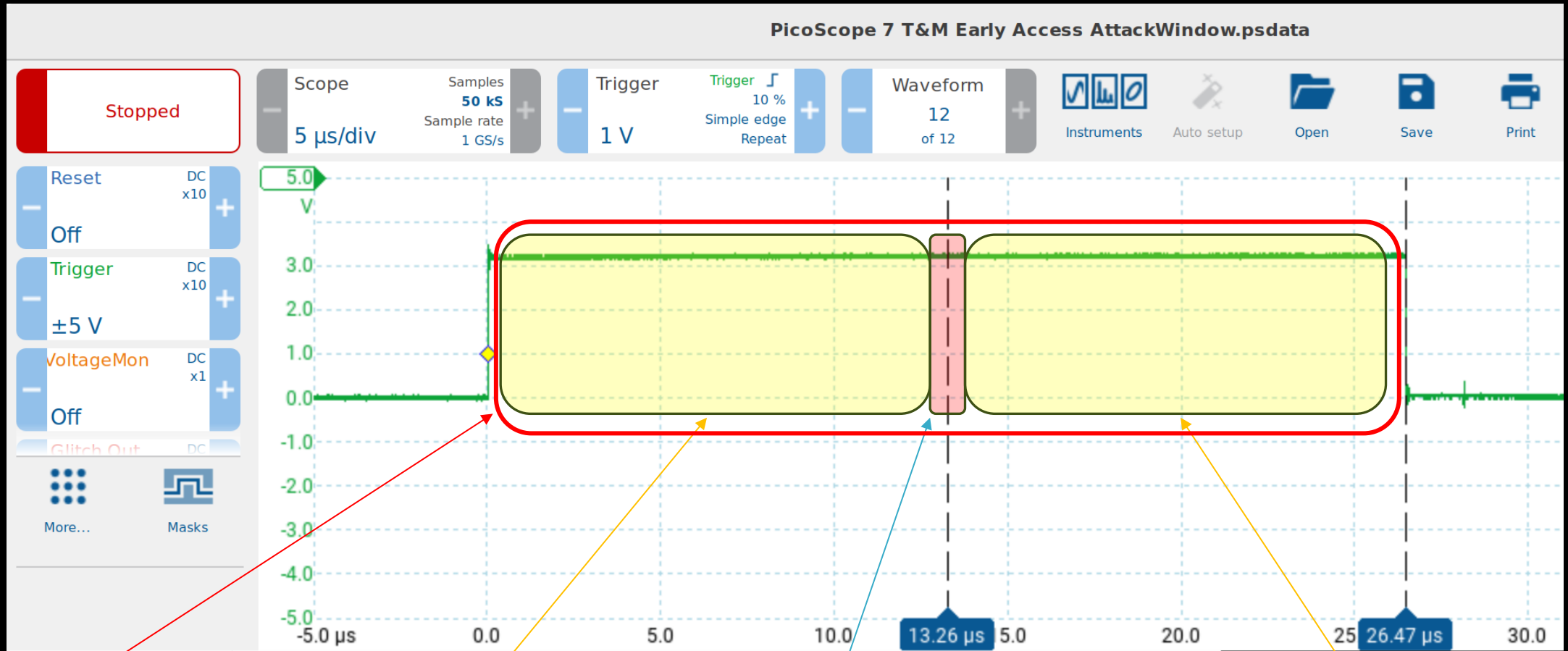
1024 NOPs

Trigger (GPIO26): Down

Print a6. Should be 1 (if not glitched)



# Attack Window



Attack Window

1024 NOPs

Target expected timing  
(approximate)

1024 NOPs



# Glitch parameters

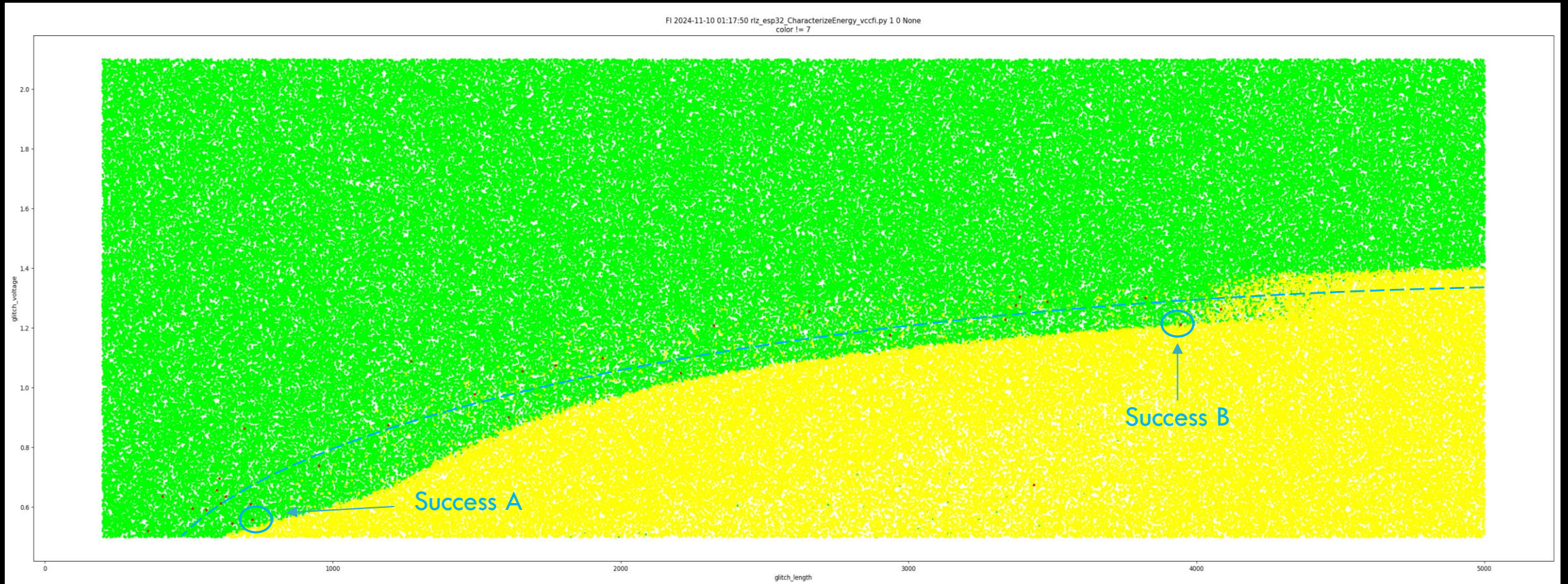


## FI campaign

- Normal voltage: *Fixed*. 2.1V
- Glitch delay: *Random*. Between 10us and 13.2us
- Glitch voltage: *Random*. Between 0.5V and 2.1V
- Glitch length: *Random*. Between 200ns and 5000ns
- Experiments: ~270k
- Success: **32** (0.01%)

Data **visualization** provides valuable information

# Distribution: glitch\_voltage vs glitch\_length



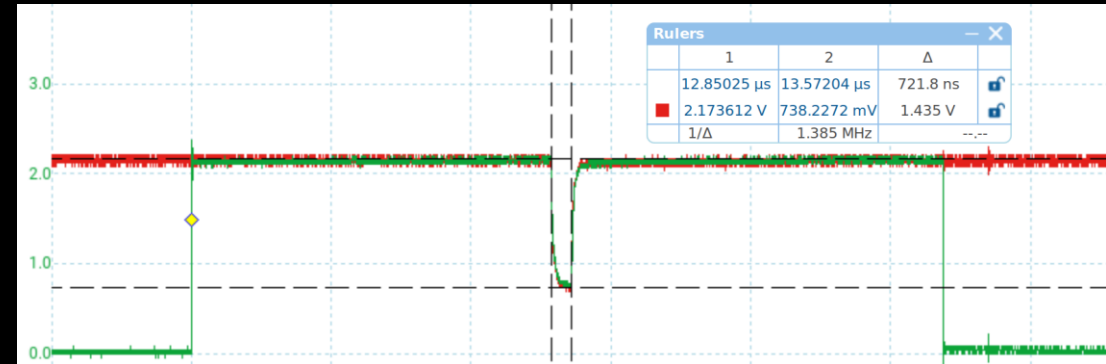
- **Green:** No effect
- **Yellow:** Garbage output/mute/reset
- **Red:** Successful glitch
- **Blue:** Comments

# Some interesting results

- Glitch A:

- Sharper, shorter, later
- *glitch\_voltage*: 0.558 V
- *glitch\_length*: 721 ns
- *glitch\_delay*: 12744 ns

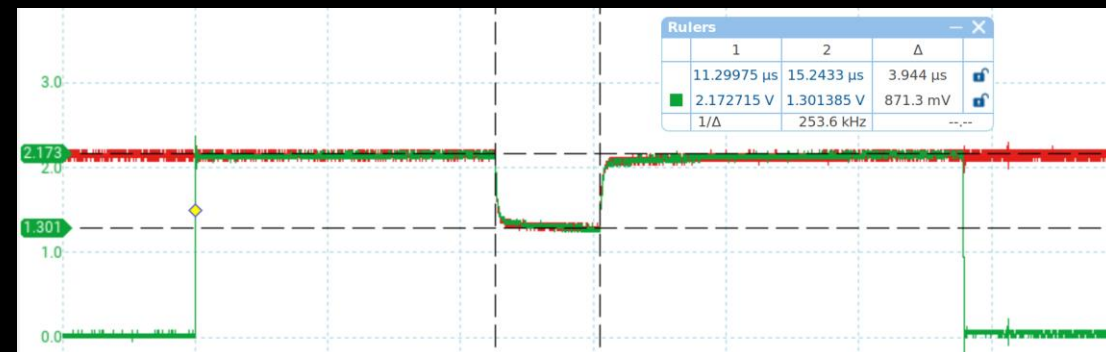
A



- Glitch B:

- Shallower, longer, earlier
- *glitch\_voltage*: 1.211 V
- *glitch\_length*: 3944 ns
- *glitch\_delay*: 11199 ns

B



- **Both** glitches are successful

## Sharpness vs CPU speed

- ESP32 CPU clock speed: Min 80 MHz  $\rightarrow$  1 clock cycle = 12.5 ns (or shorter)
- Successful glitch lengths:
  - Minimum: 200ns (16 times max clock cycle duration)
  - Maximum: 5000us (400 times max clock cycle duration)
- Our glitches are WAY longer than the duration of a single CPU clock cycle



## We have answers!

- Has a glitch to be **sharp** in order to affect a single instruction? **NO**
- Does a glitch need to be faster than single clock cycle duration? **NO**
- Are multiple glitch **shapes** possible? **YES**
  - Or are we ~~limited to a single shape?~~

A quite widespread belief is incorrect

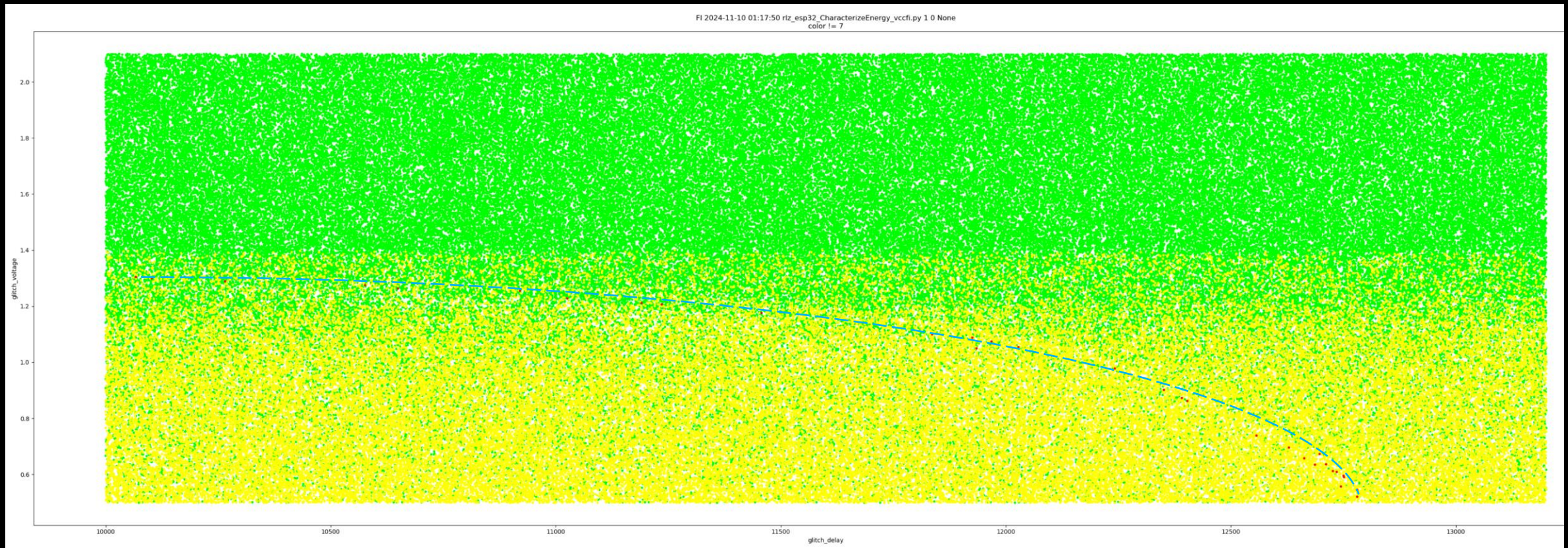
*Data analysis.*

# Patterns

- An interesting **relationship** between *glitch\_voltage* and *glitch\_length*:
  - Higher the *glitch\_voltage* → longer *glitch\_length*
- Glitches are mostly located along the green/yellow **border**
- Is there an actual curve profile?
  - Very likely, but it doesn't look great for this specific target
  - See the following...



# Distribution: *glitch\_voltage* vs *glitch\_delay*



- **Green:** No effect
- **Yellow:** Garbage output/mute/reset
- **Red:** Successful glitch
- **Dashed blue:** Just a marker stroking a “curve”

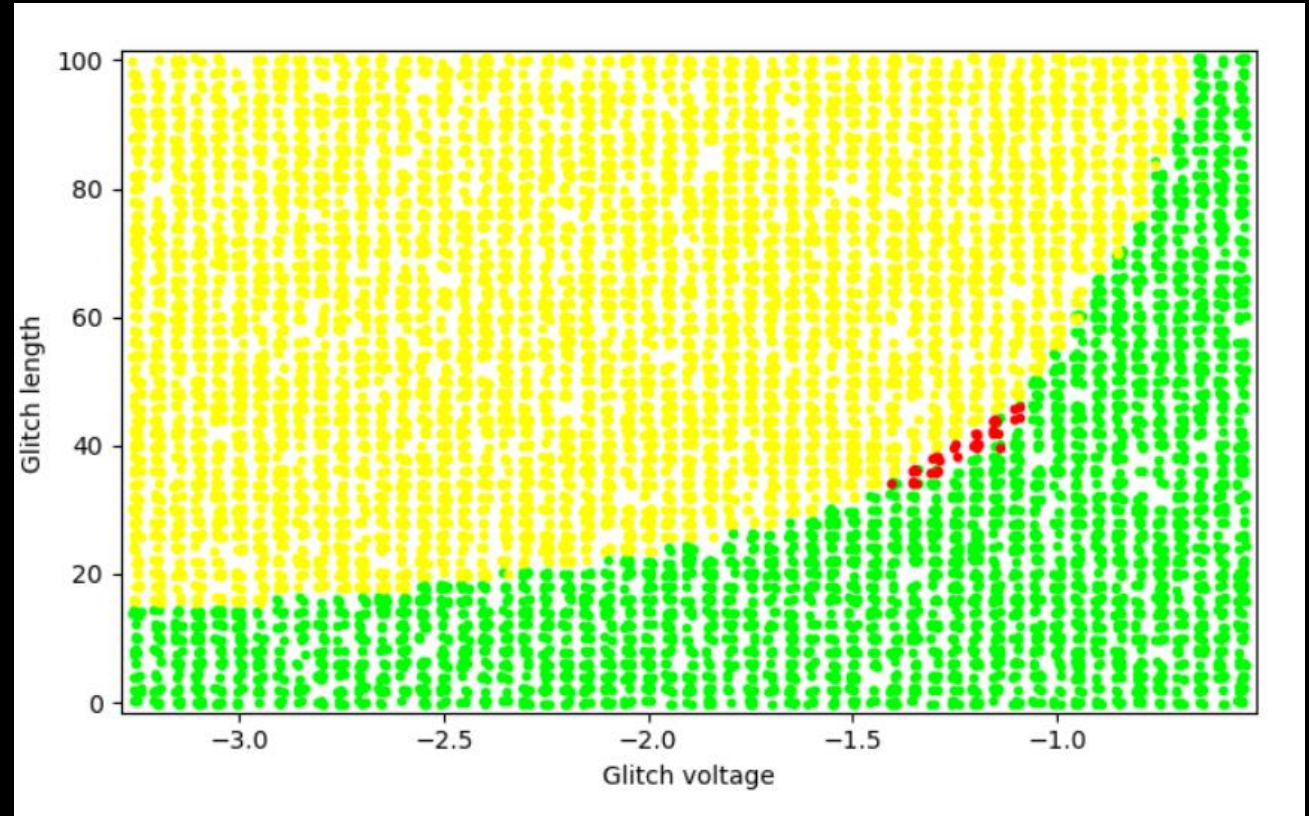
## More patterns...

- Another apparent **relationship** between *glitch\_voltage* and *glitch\_delay*:
  - Higher *glitch\_voltage*  $\rightarrow$  lower *glitch\_delay* (i.e. start glitching earlier)
- Successful glitches seem to align on some kind of curve
- The pattern also proves that we are glitching our target: a **single** add instruction
  - A corruption of the NOP sleds (into an `addi1`) would have created a time-independent distribution of successes (reds)



# Is this chip special?

- Not at all.
- Such **patterns** are common and they are present for almost all chips
  - See a clearer one on the right
- Yet...they are unknown to the most.



*Yuce et al. - "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation"*

\* Glitch Voltage plotted as the deviation from normal\_voltage

# Why?

- They just rarely surface in literature:
  - Both in academia and security community works
- Identification requires:
  - Varying glitch voltage/length:
    - most glitchers out there only glitch to GND
  - Data visualization and analysis
    - Still rarely used in FI attacks
    - some research mostly focuses on getting to success and increasing the success rate  
😊

# Why do these patterns exist?

- A thorough investigation is still lacking in the public domain:
  - To the best of my knowledge 😊
- Whenever reported, they are usually not accompanied by physics modeling:
  - E.g. The [paper](#) we got the example pattern from.

## Current understanding

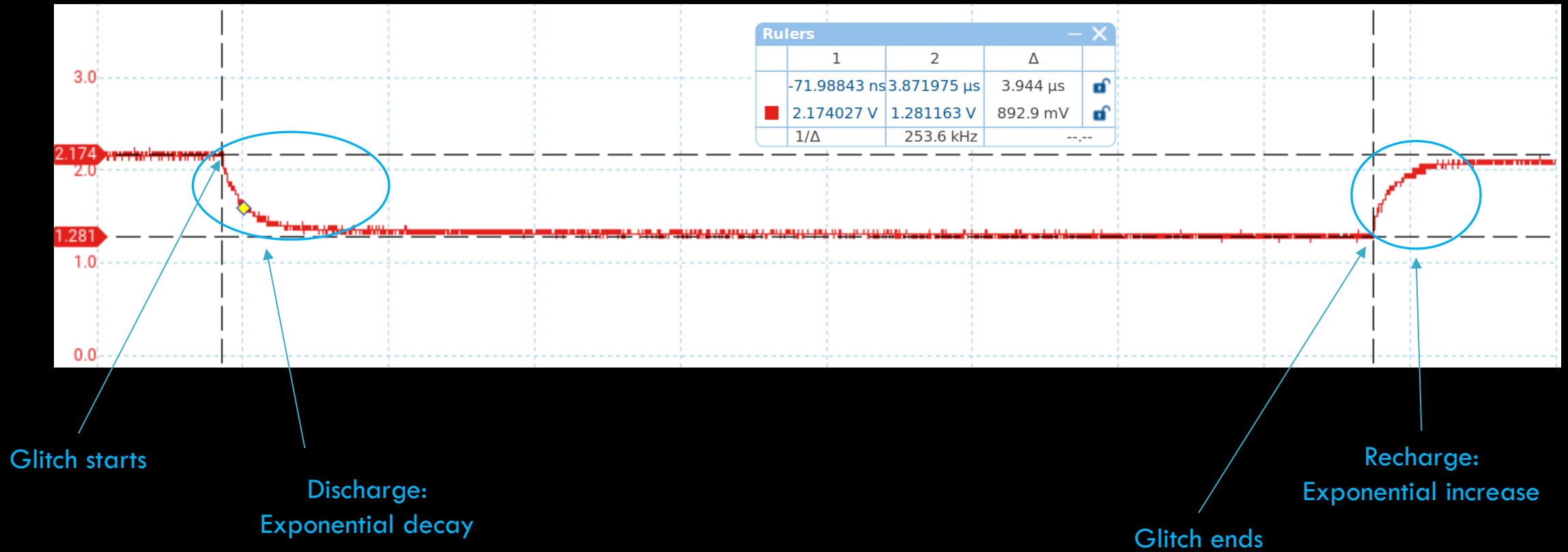
- Voltage glitches are caused by **setup time violations**:
  - See [\[1\]](#), [\[2\]](#), [\[3\]](#), [\[4\]](#)
- In a nutshell, lower voltages increases propagation time...and wrong values are sampled
- Totally valid. But it may be challenging to explain the patterns...
  - E.g. why does *glitch\_length* **matter** at?

Are we missing something **important**?

## Idea: **Energy-based** interpretation

- Lowering the voltage deprives the target of energy:
  - i.e. we **discharge** our target over time
  - The amount of energy depends on both *glitch\_voltage* and *glitch\_time*
- The internal voltages drop as well
- Below a certain level  $V_f$ , representing logical “1” is not possible anymore.

# Glitch profile





# Implications

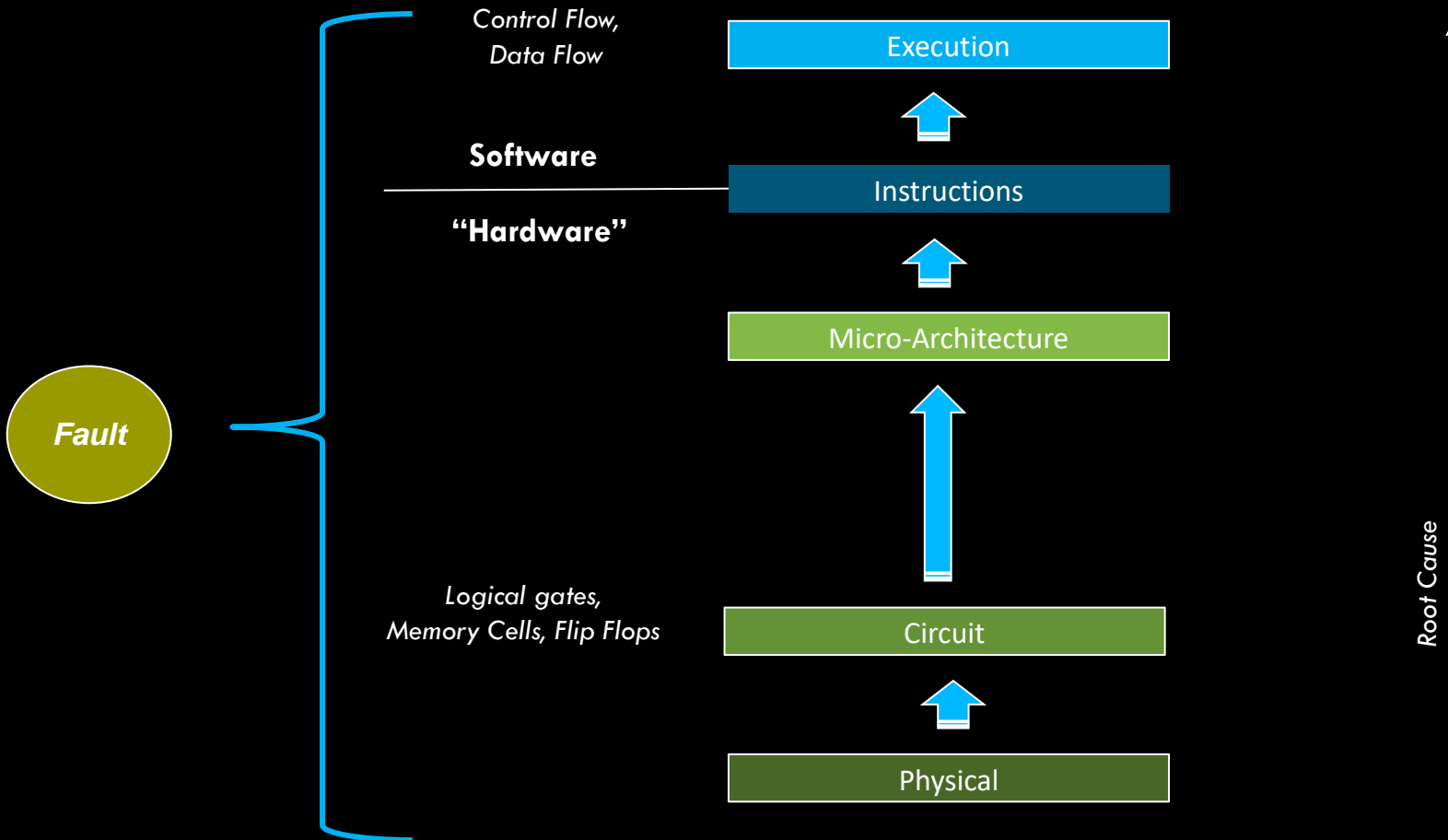
- Different glitch shapes are always possible:
  - Requirement: internal voltage must drop below  $V_f$  at the right time
- It is possible to perform attacks with **very shallow and very long** glitches:
  - **Yes**. We have been using them 😊
  - This may help bypassing hardware **countermeasures** (e.g. glitch detector, brownout detectors,..)

# Summary

- Widespread **beliefs** found to be incorrect
- Physics **modeling** in paper is rare
- Parameter space **visualization** is rare
- Some interesting **patterns** and features are:
  - Not discussed
  - Challenging to explain with the current interpretation
- We may be missing on some **fundamental** understanding...
- ..as well as some **powerful** attacks.

Modeling faults.

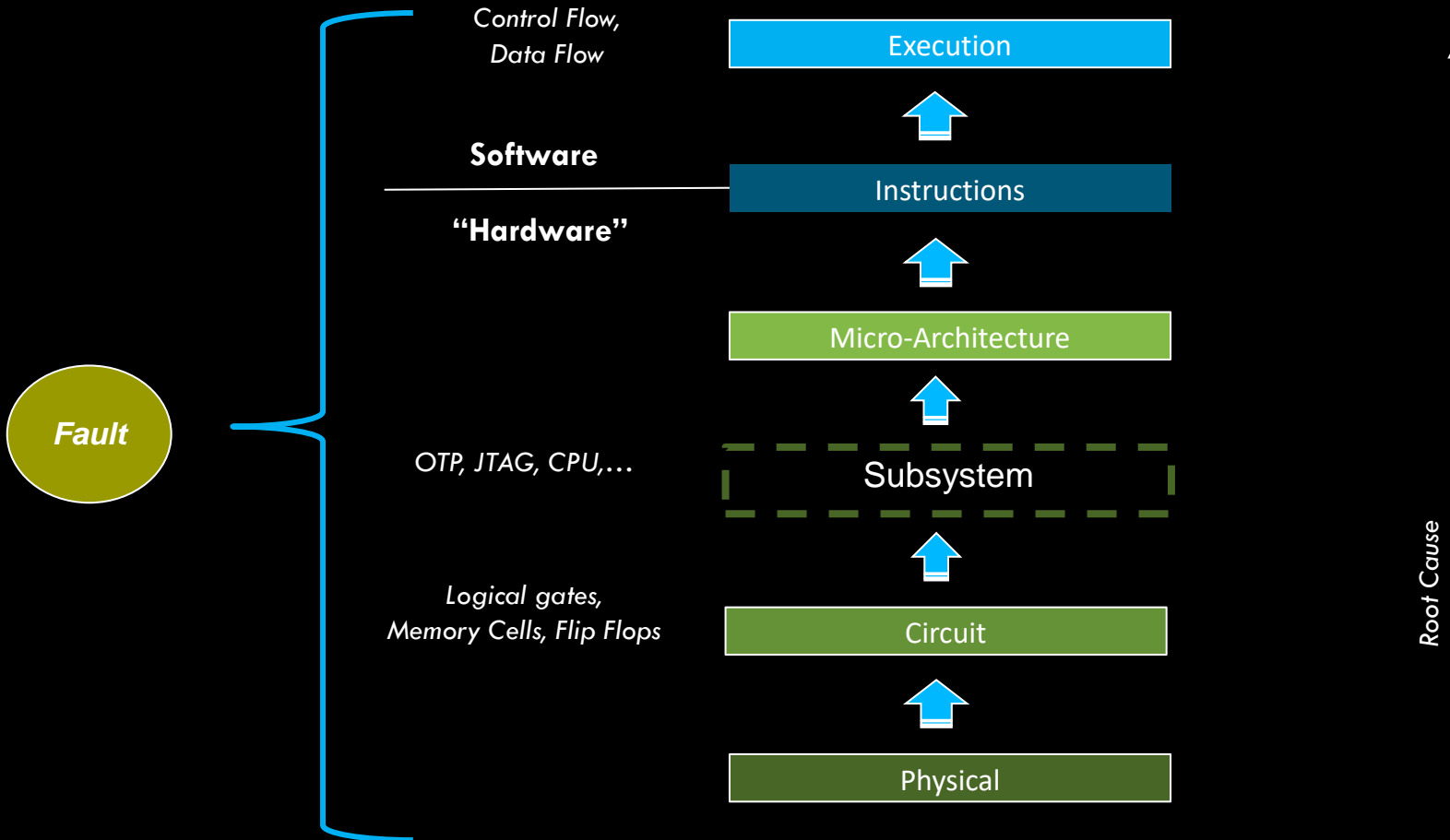
# A fault propagation model



# Notes

- Geared towards faults in software execution:
  - Not everything is instructions
- Attack against non-CPU subsystem (i.e. pure hardware implementations) do not easily fit:
  - JTAG
  - OTP
  - RNGs
  - ...
- Example:
  - [Hardwear.io USA 2022 - "Breaking SoC Security by Glitching OTP Data Transfers" \[Raelize\]](#)

# Let's extend it



# The observer's challenge

ALL the faults introduced in  
a system

Faults that CAN be observed

Faults that ARE being observed

Faults useful for an attack



# Notes

- Describing all the faults actually introduced in a system is possibly infeasible:
  - We need to observe a fault (or its consequences) in order to be able to acknowledge a fault has occurred
- Still, it may be possible to formally describe fault models geared toward **specific** attacks



A widespread model (or misconception?) .

# Guess how FI affects code execution...


**CWE** Common Weakness Enumeration  
*A community-developed list of SW & HW weaknesses that can become vulnerabilities*

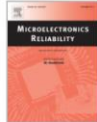
Home > CWE List > CWE-1332: Improper Handling of Faults that Lead to Instruction Skips (4.16)

Home | About | CWE List | Mapping | Top-N Lis

**CWE-1332: Improper Handling of Faults that Lead to Instruction Skips**

Weakness ID: 1332  
**Vulnerability Mapping: ALLOWED**  
Abstraction: Base

 **Microelectronics Reliability**  
Volume 121, June 2021, 114133



---

## Experimental analysis of the electromagnetic instruction skip fault model and consequences for software countermeasures

 **Microelectronics Reliability**  
Volume 155, April 2024, 115370

---

Research paper

## Software countermeasures against the multiple instructions skip fault model

Formal verification of a software countermeasure against instruction skip attacks

Nicolas Moro<sup>1,2</sup>, Karine Heydemann<sup>1</sup>, Emmanuelle Encrenaz<sup>1</sup>, and Bruno Robisson<sup>2</sup>

<sup>1</sup>Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, 75005 Paris, France

firstname.lastname@lip6.fr

<sup>2</sup>CEA, CEA-Tech PACA, LSAS, 13541 Gardanne, France

firstname.lastname@cea.fr

February 24, 2014

# Instruction skipping

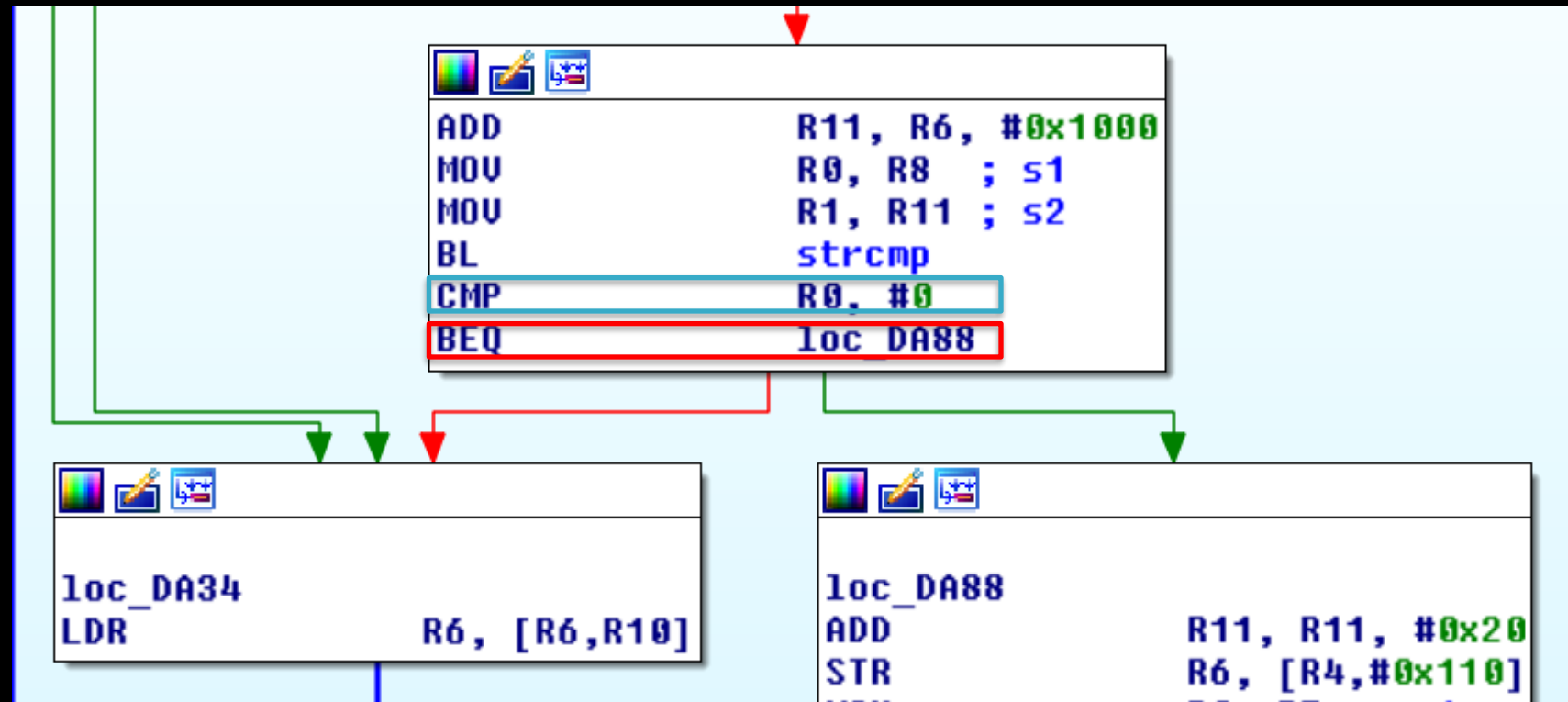
- The most common fault model for describing FI effect on CPU execution:
  - Been with us for at least 3 decades 😊
- First attacks mostly targeted security relevant **decisions**
  - Smart Card pin authentication
  - Signature checks
  - ...

*“It is as if...we skipped that instruction”*

# Typical attacks

- Targets:
  - **Conditionals:**
    - To “skip” the compare instruction
  - Function calls:
    - To “skip” the execution of a security relevant function
  - Infinite loops:
    - To “skip” the current instruction and fall into the next one
- This requires **precise** targeting of specific instructions:
  - Strong **timing** requirements
  - Potential targets are easy to **predict**

# Example



# Notes

- “Instruction skipping” models fault at the instruction execution level
- The **original** program continues to be executed
  - We just take an unintended branch in a decision
- Hard to jump at arbitrary locations

# Attack execution

```
1  int load_exec_next_boot_stage() {
2
3      // Destination addresses in SRAM
4      uint32_t img_addr = 0xd0000000;
5      uint32_t sig_addr = 0xd1000000;
6
7      // Copy next stage image from Flash to SRAM
8      load_next_stage_img(img_addr);
9
10     // Copy signature from Flash to SRAM
11     load_next_stage_signature(sig_addr);
12
13     if (verify_signature(img_addr, sig_addr)) {
14
15         // Wrong signature. Reset system
16         reset_SOC();
17     }
18
19     // Signature valid. Exec next stage code
20     exec_stage(img_addr);
21 }
```

- “Instruction skipping”  
requires accurate **timing**
- Can be executed  
blindly:
  - i.e. no assumption on  
type of fault
  - “Glitch ‘n **pray**”

# SW countermeasures: Multiple checks

```
1 int load_exec_next_boot_stage() {
2
3     // Destination addresses in SRAM
4     uint32_t img_addr = 0xd0000000;
5     uint32_t sig_addr = 0xd1000000;
6
7     // Copy next stage image from Flash to SRAM
8     load_next_stage_img(img_addr);
9
10    // Copy signature from Flash to SRAM
11    load_next_stage_signature(sig_addr);
12
13    if (verify_signature(img_addr, sig_addr)) {
14        reset_SOC();
15    }
16
17    if (verify_signature(img_addr, sig_addr)) {
18        reset_SOC();
19    }
20
21    if (verify_signature(img_addr, sig_addr)) {
22        reset_SOC();
23    }
24
25    // Signature valid. Exec next stage code
26    exec_stage(img_addr);
27 }
```

- Attack assumption:
  - A glitch is required for **every** check
  - One instruction, one glitch
- Mitigation: Perform multiple checks



# SW countermeasures: Making synchronization harder

```
1 int load_exec_next_boot_stage() {  
2  
3     // Destination addresses in SRAM  
4     uint32_t img_addr = 0xd0000000;  
5     uint32_t sig_addr = 0xd1000000;  
6  
7     // Copy next stage image from Flash to SRAM  
8     load_next_stage_img(img_addr);  
9  
10    // Copy signature from Flash to SRAM  
11    load_next_stage_signature(sig_addr);  
12  
13    random_delay();  
14  
15    if (verify_signature(img_addr, sig_addr)) {  
16        reset_SOC();  
17    }  
18  
19    random_delay();  
20  
21    if (verify_signature(img_addr, sig_addr)) {  
22        reset_SOC();  
23    }  
24  
25    random_delay();  
26  
27    if (verify_signature(img_addr, sig_addr)) {  
28        reset_SOC();  
29    }  
30  
31    random_delay();  
32  
33    // Signature valid. Exec next stage code  
34    exec_stage(img_addr);  
35 }
```

- Attack assumption:
  - A glitch must “hit” that instruction at a specific point in **time**
- Mitigation:
  - Random delays are introduced around critical checks

# Observations

- SW-based countermeasures are widely used in the **industry** and **academia**
  - Multiple checks and random delays are two prominent examples
  - Additional countermeasures available
  - Example: Riscure [whitepaper](#)
- Commonly advised and implemented in FI-resistant targets
- They reduce attack success rate:
  - Multiple glitch required
  - Target synchronization more difficult

## A few common beliefs

- “Software is vulnerable to FI”:
  - Wrong. Hardware is.
- Source code **reviews** for fault injections are considered a proper tool for spotting “FI vulnerabilities”:
  - We will understand why that is not the case, shortly

# Untold assumption

Instruction **skipping** is the relevant fault model

Is that true?

# Test code: Counter (unrolled loop)

```
...  
#define addi1 "addi.n a8, a8, 1;"  
#define addi2 addi1 addi1  
#define addi4 addi2 addi2  
#define addi8 addi4 addi4  
#define addi16 addi8 addi8  
#define addi32 addi16 addi16  
#define addi64 addi32 addi32  
#define addi128 addi64 addi64  
#define addi256 addi128 addi128  
#define addi512 addi256 addi256  
#define addi1024 addi512 addi512
```

```
...  
uint32_t counter = 0;
```

```
GPIO_OUTPUT_SET(26,1);
```

```
asm volatile (  
    "movi a8, 0;"  
    addi1024  
    "mov %[counter], a8;"  
    : [counter] "=r" (counter)  
    :  
    : "a8"  
);
```

```
GPIO_OUTPUT_SET(26,0);
```

```
esp_rom_printf("XXXX%08xYYYY%08xZZZZ\n", counter, counter);  
...  
...
```

Add instruction: adds 1

Macros

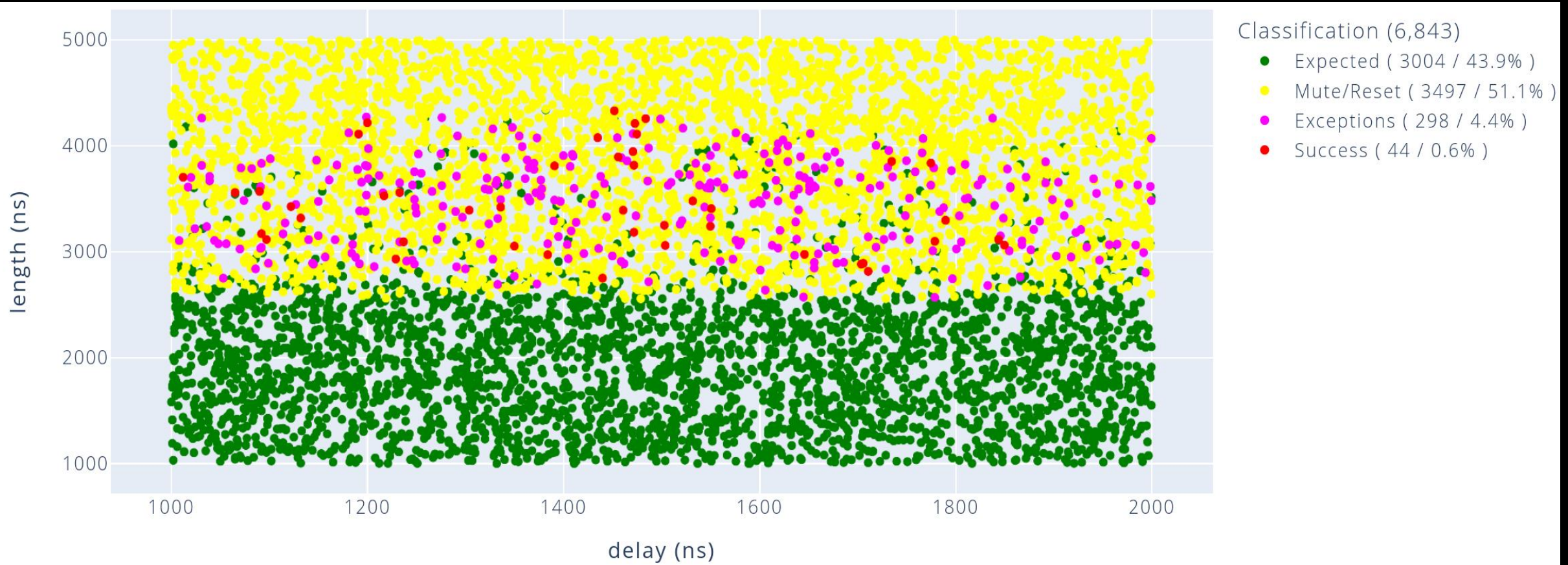
Target code

1024 add instructions (Unrolled loop)

Trigger (GPIO26): Up

Trigger (GPIO26): Down

# Results



# Data analysis (1)

AMOUNT	COLOR	DELAYMIN	DELAYMAX	LENGTHMIN	LENGTHMAX	RESPONSE
filter data	R					
11	R	1090	1850	2815	4331	XXXX000003ffYYYY000003ffZZZZ
5	R	1191	1233	2931	4218	XXXX3ffe417aYYYY3ffe417aZZZZ
4	R	1735	1790	3098	3853	XXXX3ffe414eYYYY3ffe414eZZZZ
4	R	1012	1391	2972	3811	XXXX000003feYYYY000003feZZZZ
3	R	1435	1844	2975	4077	XXXX00000401YYYY00000401ZZZZ
3	R	1471	1475	3946	4211	XXXX00000407YYYY00000407ZZZZ
2	R	1461	1472	3392	3817	XXXX00000408YYYY00000408ZZZZ
2	R	1065	1092	3170	3559	XXXX800812edYYYY800812edZZZZ

Instruction skipping



# Something weird...

AMOUNT	COLOR	DELAYMIN	DELAYMAX	LENGTHMIN	LENGTHMAX	RESPONSE
filter data	R					
11	R	1090	1850	2815	4331	XXXX000003ffYYYY000003ffZZZZ
5	R	1191	1233	2931	4218	XXXX3ffe417aYYYY3ffe417aZZZZ
4	R	1735	1790	3098	3853	XXXX3ffe414eYYYY3ffe414eZZZZ
4	R	1012	1391	2972	3811	XXXX000003feYYYY000003feZZZZ
3	R	1435	1844	2975	4077	XXXX00000401YYYY00000401ZZZZ
3	R	1471	1475	3946	4211	XXXX00000407YYYY00000407ZZZZ
2	R	1461	1472	3392	3817	XXXX00000408YYYY00000408ZZZZ
2	R	1065	1092	3170	3559	XXXX800812edYYYY800812edZZZZ

How do we explain these results with instruction skipping?

...and weirder...

AMOUNT	COLOR	DELAYMIN	DELAYMAX	LENGTHMIN	LENGTHMAX	RESPONSE
filter data	R					
11	R	1090	1850	2815	4331	XXXX000003ffYYYY000003ffZZZZ
5	R	1191	1233	2931	4218	XXXX3ffe417aYYYY3ffe417aZZZZ
4	R	1735	1790	3098	3853	XXXX3ffe414eYYYY3ffe414eZZZZ
4	R	1012	1391	2972	3811	XXXX000003feYYYY000003feZZZZ
3	R	1435	1844	2975	4077	XXXX00000401YYYY00000401ZZZZ
3	R	1471	1475	3946	4211	XXXX00000407YYYY00000407ZZZZ
2	R	1461	1472	3392	3817	XXXX00000408YYYY00000408ZZZZ
2	R	1065	1092	3170	3559	XXXX800812edYYYY800812edZZZZ

What are the values in these responses?

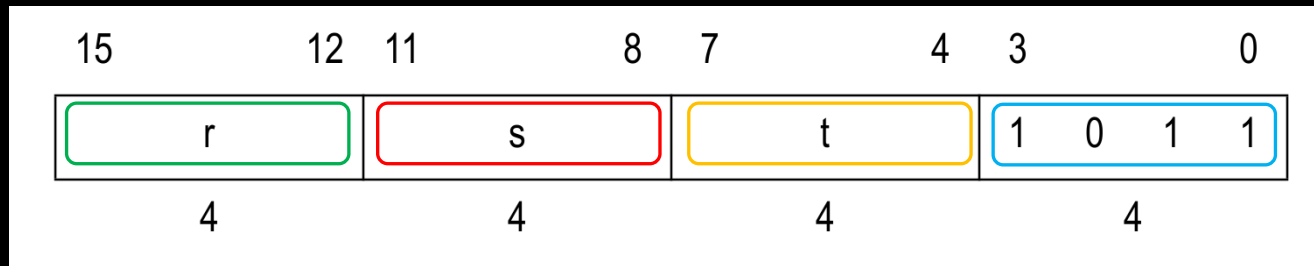
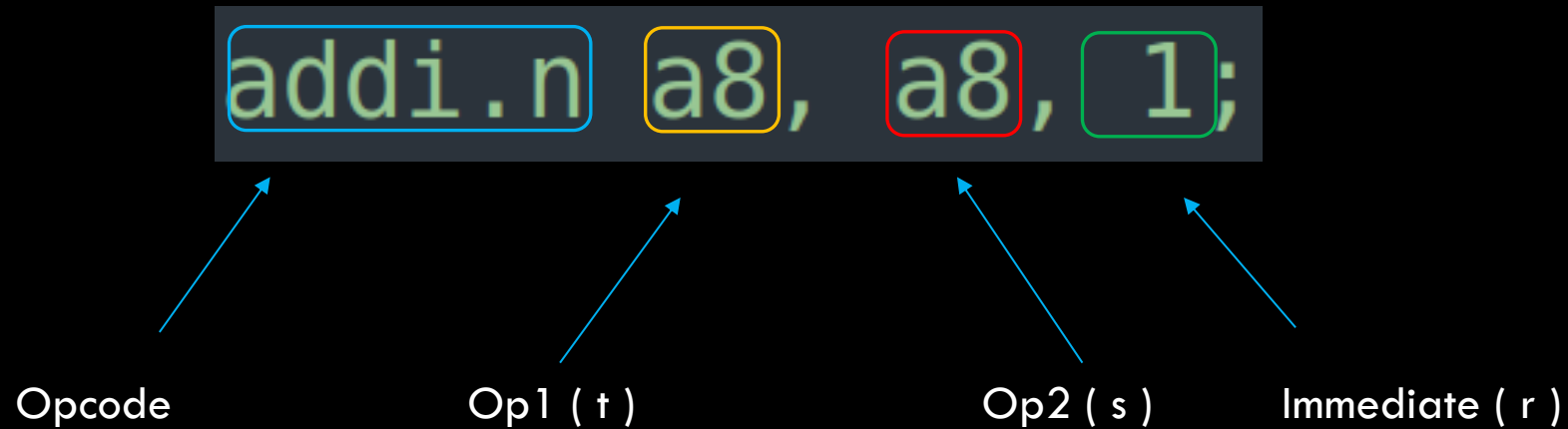
# Some hints

Table 1-2. Embedded Memory Address Mapping

Bus Type	Boundary Address		Size	Target	Comment
	Low Address	High Address			
Data	0x3FF8_0000	0x3FF8_1FFF	8 KB	RTC FAST Memory	PRO_CPU Only
	0x3FF8_2000	0x3FF8_FFFF	56 KB	Reserved	-
Data	0x3FF9_0000	0x3FF9_FFFF	64 KB	Internal ROM 1	-
	0x3FFA_0000	0x3FFA_DFFF	56 KB	Reserved	-
Data	0x3FFA_E000	0x3FFD_FFFF	200 KB	Internal SRAM 2	DMA
Data	0x3FFE_0000	0x3FFF_FFFF	128 KB	Internal SRAM 1	DMA

A memory address? how?

## Our instruction (+ encoding)



What could be happening?

# Occam's razor

- Our glitches are most likely **corrupting** instructions
- This fault model alone is able to explain all the responses we see
  - Responses slightly above 0x400 → Immediate corruption
  - Responses containing a memory address → Source register corruption
  - Responses below 0x400 (i.e. “instruction skipping”)
    - Instruction is mutated into one without side effects. E.g: `addi.n a8, a8, 0`
- Also all the **exceptions** can be explained!

# Instruction fetch / Bus transfers?

- Some studies suggest instruction corruption may occur during the fetch phase of an instruction
- Or, more specifically, in the transfer from memory to CPU over busses
- This seem to apply to multiple techniques (e.g. Voltage and EM)
- Some relevant work:
  - Microcontrollers: “Building fault models for microcontrollers” – A. Spruyt [2012] - Voltage Fault Injection
  - ARM Cortex M: Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller [2014] – EMFI
  - ARM Cortex A8 (1 Ghz): Exploring Effects of Electromagnetic Fault Injection on a 32-bit High Speed Embedded Device Microprocessor - T. Hummel [2014] - EMFI
    - This research shows a strong time-dependence for affecting specific registers of an instruction
  - ARM Cortex A72 (1.5 – 1.8 Ghz): Faults in Our Bus: Novel Bus Fault Attack to Break ARM TrustZone – Mishra et al. [2024] - EMFI

Weird machines...  
out of Data transfers.

# Arbitrary code execution (ARM32)

## Controlling PC on ARM using Fault Injection

Niek Timmers

*Riscure – Security Lab*

Albert Spruyt

*Riscure – Security Lab*

Marc Witteman

*Riscure – Security Lab*

- Our research (2016) is the first (or one of?) leveraging the **instruction corruption** fault model for actual attacks:
  - Note: the idea of FI corrupting instructions may have been previously hinted in others' research and discussion
- Introduces powerful attack: PC control on AArch32



# Instruction corruption

- Glitches may corrupt instructions (examples on ARM32)

- **Single bit** corruptions

```
add  x0, x1, x3    = 100010110000000110000000000100000
add  x0, x1, x2    = 100010110000000100000000000100000
```

- **Multi bit** corruptions

```
ldr  x0, [sp, #32] = 11111001010000000001001111100000
str  x0, [x0, #32] = 11111001000000000001000000000000
```

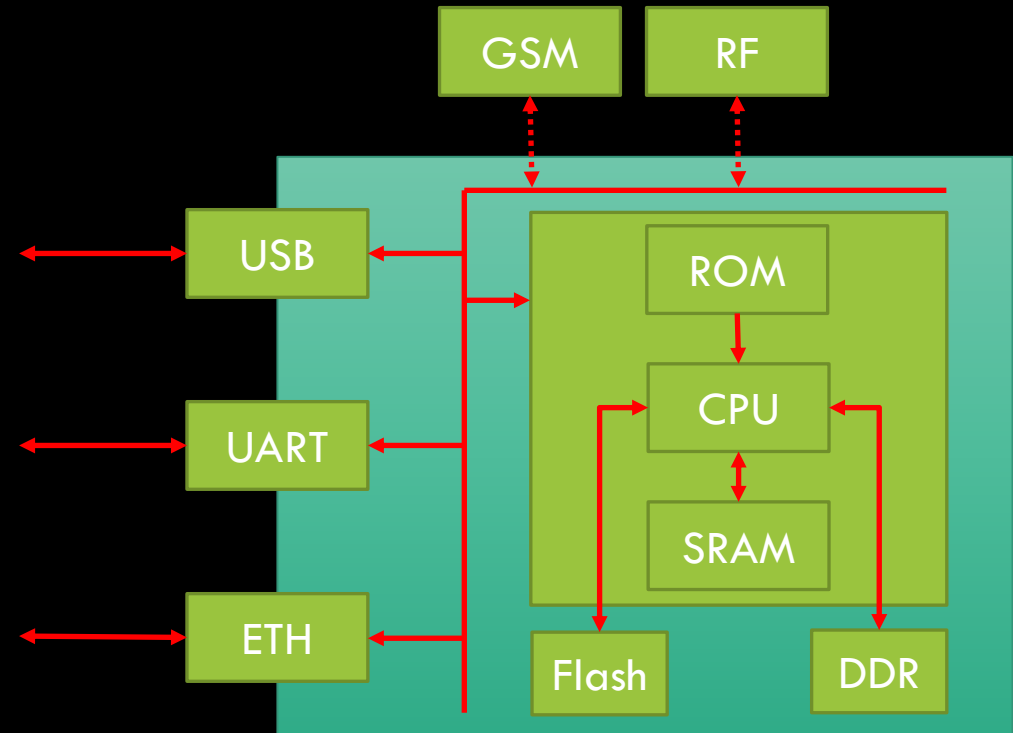
- Most chips are affected by this fault model

- Which bits can be controlled, and how, depends on the target, ...

- As software is modified; any **software security model** breaks

# Data transfers are a great target

- All devices **transfer data**
- From memory to memory
- Using **external** interfaces



Transferred **data** may be under attacker's control

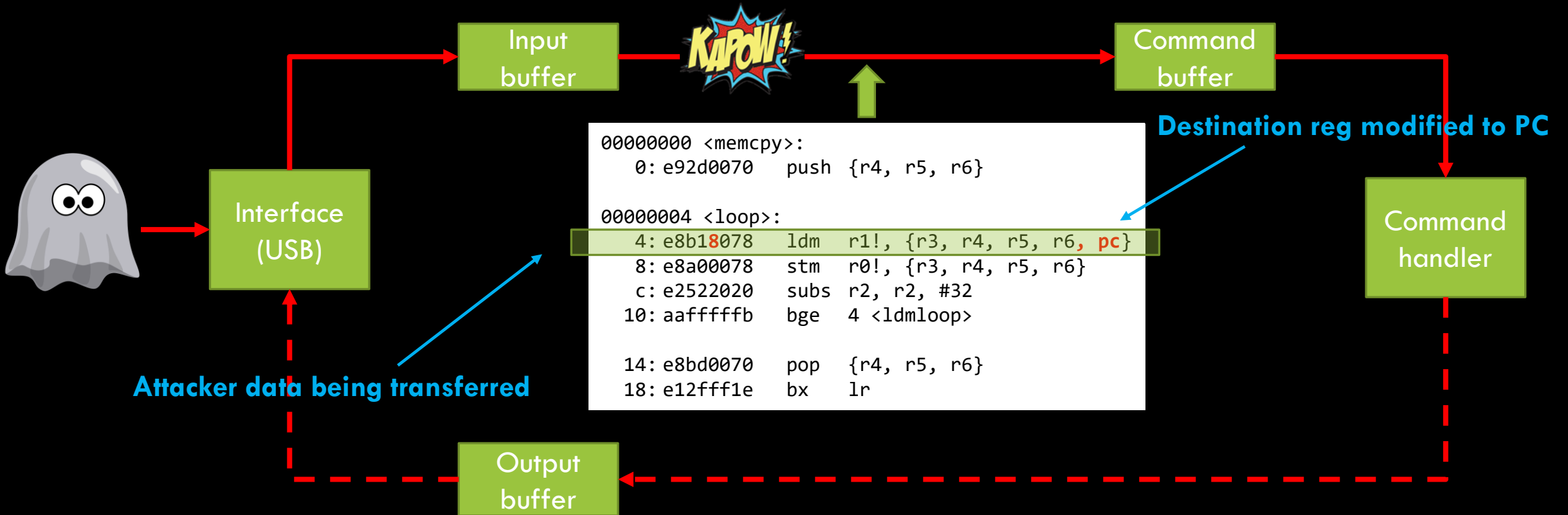
## memcpy()

- It's **everywhere**.
- **SW** security: Parameters are typically checked (**dest**, **src** and **n**)
- Transferred content itself **not** considered security critical

*Let's **use it** as a Fault Injection target...*

PC control with Instruction corruption (ARM32).

# Example: USB data transfer (ARM32)



PC set to attacker data. Control flow **directly** hijacked

# We regularly use this technique...

- [2016]: Bypassing Secure Boot (non-encrypted)
  - [Bypassing Secure Boot using Fault Injection @ BlackHat EU 2016](#)
- [2017]: Escalating privileges from user to kernel in Linux
  - [Escalating Privileges in Linux Using Voltage Fault Injection @ FDTC 2017](#) (academic paper)
  - [R00ting the Unexploitable using Hardware Fault Injection @ BlueHat v17](#)
- [2019]: Bypassing encrypted secure boot
  - [Hardening Secure Boot on Embedded Devices @ Blue Hat IL 2019](#)
- [2019]: Taking control of an AUTOSAR based ECU
  - [Attacking AUTOSAR using Software and Hardware Attacks @ escar USA 2019](#)

...as of today 😊

- [2019]: Generalizing the technique to (all?) other architectures
  - [Using Fault Injection to Turn Data Transfers into Arbitrary Execution @ PoC 2019](#)
- [2020]: Bypassed an encrypted Secure Boot (data transfers over DMA) leveraging SRAM persistence
  - [Look mum, no key! Bypassing Encrypted Secure Boot @ hardwear.io 2020](#)
- [2024]: Bypassing encrypted secure boot:
  - using attacker-derived data (and not directly controlled)
  - Manipulating ciphertext to force plaintext's CRC32 to a controlled value
  - Getting PC control by glitching a single instruction
  - [Breaking Espressif's ESP32 V3: Program Counter Control with Computed Values using Fault Injection @ Usenix WOOT'24](#)  
(academic paper)

# Works on multiple architectures

- We identified **multiple** variants and techniques
- Yield **arbitrary** code execution:
  - from controlled **data** only
  - By corrupting instruction destination registers
- Sufficiently generic to work across multiple architectures
- Examples:
  - Corrupting stored PC (in regs) or SP
  - Hijacking jump/call (through registers)
  - Corrupting callee saved regs (across function calls)

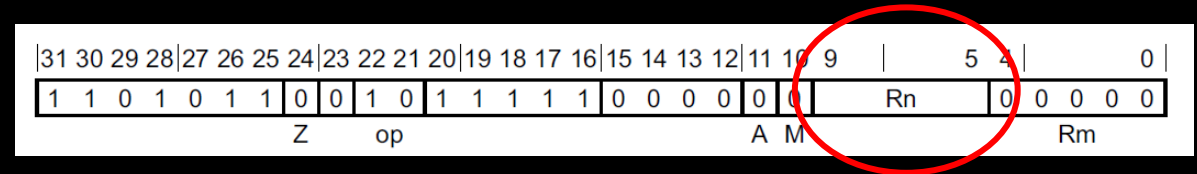
More details [here](#)



## Example: ARMv8 **RET** instruction

- Used for returning from a function call.
  - Return address stored in register (default X30)

- It has the following encoding:



- **RET** instruction can encode any register (**x0** to **x30**)

# Real world example

- Google Bionic's (LIBC) **memcpy**
- Copying 16 bytes executes the following code:
  - Source data resides in **x6** and **x7**
  - Source data is not wiped before **RET**
- Glitch **RET** instruction into **RET x6** or **RET x7**:
  - Equivalently glitch **ldr x6, ...** to **ldr x30, ...**



```
memcpy:
0:8b020024 add x4, x1, x2
4:8b020005 add x5, x0, x2
8:f100405f cmp x2, #0x10
c:54000229 b.ls 50 <memcpy+0x50>
...
50:f100205f cmp x2, #0x8
54:540000e3 b.cc 70 <memcpy+0x70>
58:f9400026 ldr x6, [x1]
5c:f85f8087 ldur x7, [x4, #-8]
60:f9000006 str x6, [x0]
64:f81f80a7 stur x7, [x5, #-8]
68:d65f03c0 ret
```

PC hijacked from controlled data.

## “Instruction corruption”: Recipe for success

- Identify data transfers you **control**
- Send sled of **pointers**
  - E.g. Point to your shellcode location
- Glitch during **ANY** memcpy
- PC control

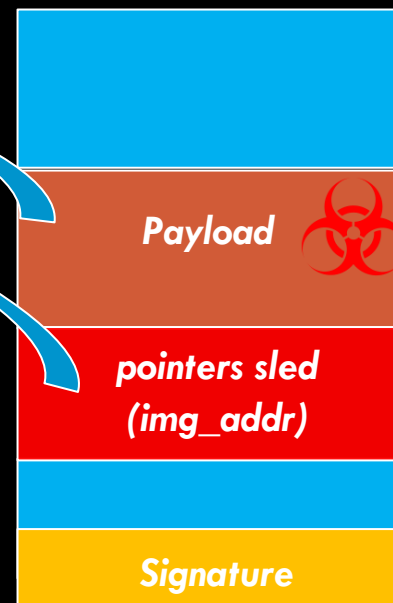
A stack overflow...without SW vulns 😊

# Attacking Secure Boot

```
1 int load_exec_next_boot_stage() {
2
3     // Destination addresses in SRAM
4     uint32_t img_addr = 0xd0000000;
5     uint32_t sig_addr = 0xd1000000;
6
7     // Copy next stage image from Flash to SRAM
8     load_next_stage_img(img_addr);
9
10    // Copy signature from Flash to SRAM
11    load_next_stage_signature(sig_addr);
12
13    random_delay();
14
15    if (verify_signature(img_addr, sig_addr)) {
16        reset_SOC();
17    }
18
19    random_delay();
20
21    if (verify_signature(img_addr, sig_addr)) {
22        reset_SOC();
23    }
24
25    random_delay();
26
27    if (verify_signature(img_addr, sig_addr)) {
28        reset_SOC();
29    }
30
31    random_delay();
32
33    // Signature valid. Exec next stage code
34    exec_stage(img_addr);
35 }
```



**Flash**  
**(Attacker)**



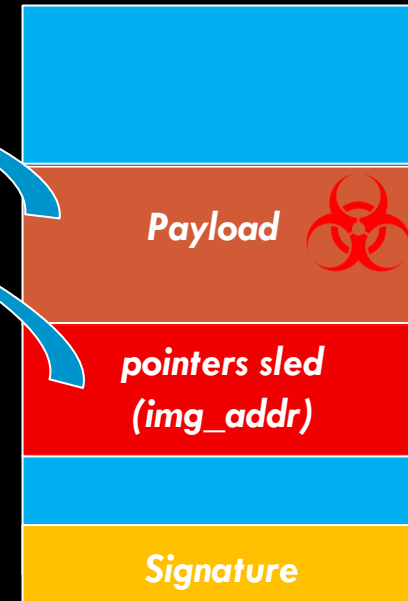
- Payload loaded at img\_addr
- Pointer sled after payload
- Glitch during pointer sled transfer

# SW-based countermeasures bypass

```
1 int load_exec_next_boot_stage() {  
2  
3     // Destination addresses in SRAM  
4     uint32_t img_addr = 0xd0000000;  
5     uint32_t sig_addr = 0xd1000000;  
6  
7     // Copy next stage image from Flash to SRAM  
8     load_next_stage_img(img_addr);  
9  
10    *img_addr();  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35 }
```



**Flash**  
**(Attacker)**



- PC value set to `img_addr`
- Control flow hijacked
- SW-based countermeasures **not** executed

## Key points

- SW-based countermeasures completely **ineffective**:
  - Countermeasures code not executed
- The attack:
  - does NOT target checks. Is unrelated to checks location (weak locality)
  - Can target ANY data transfer before SW checks
- **ROM** control flow hijacked:
  - Instruction “skipping” only yields bootloader-level access

Very **hard** to protect against. Applicable to FI-resistant targets.

# Observations

- FI SW countermeasures have been designed with an implicit fault model **assumption**
- Comes from a partial/**incorrect** understanding of FI effects on CPU code execution
- This leaves room to powerful attacks:
  - SW-based countermeasures are mostly ineffective
  - Exploit **mitigation** countermeasures may be applicable

PoC: ESP32.



# Test code: Pointers “sled” copy

- We set a specific pointer in Flash:
  - 0x4005a980: ROM code printing “Falling back...”

```
GPIO_OUTPUT_SET(26,1);
```

Trigger (GPIO26): Up

```
memcpy(buffer_A, (uint32_t *) (0x3f400000 + 0x100), sizeof(buffer_A));
```

Copy pointers from Flash to SRAM

```
memcpy(buffer_B, buffer_A, sizeof(buffer_B));
```

Copy pointers from SRAM to SRAM

```
memcpy(buffer_A, (uint32_t *) (0x3f400000 + 0x100), sizeof(buffer_A));
```

```
memcpy(buffer_B, buffer_A, sizeof(buffer_B));
```

```
memcpy(buffer_A, (uint32_t *) (0x3f400000 + 0x100), sizeof(buffer_A));
```

```
memcpy(buffer_B, buffer_A, sizeof(buffer_B));
```

```
memcpy(buffer_A, (uint32_t *) (0x3f400000 + 0x100), sizeof(buffer_A));
```

```
memcpy(buffer_B, buffer_A, sizeof(buffer_B));
```

Repeat 4 times

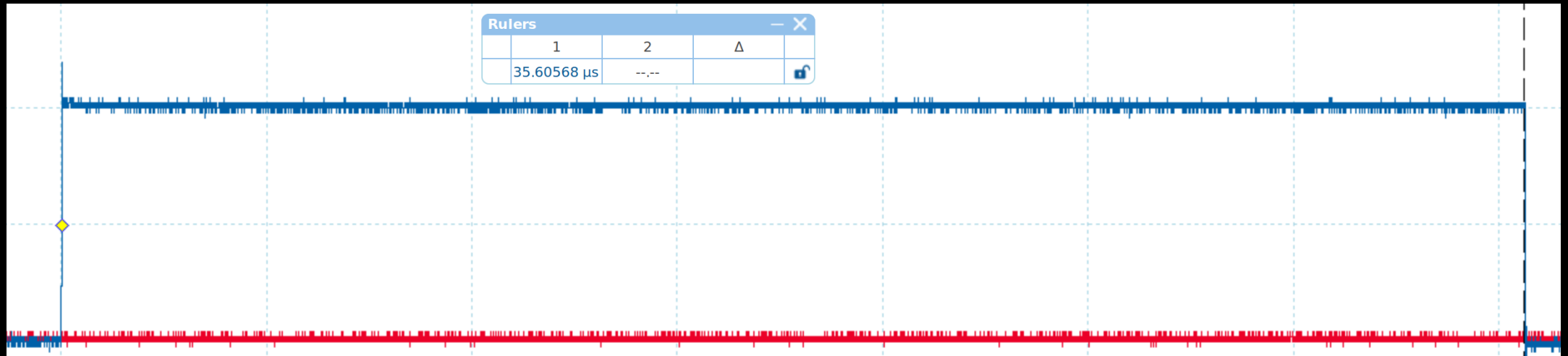
```
GPIO_OUTPUT_SET(26,0);
```

Trigger (GPIO26): Down

```
esp_rom_printf("AAAA%08xBBBB%08xCCCC\r\n", *(uint32_t *) (buffer_A), *(uint32_t *) (buffer_B));
```

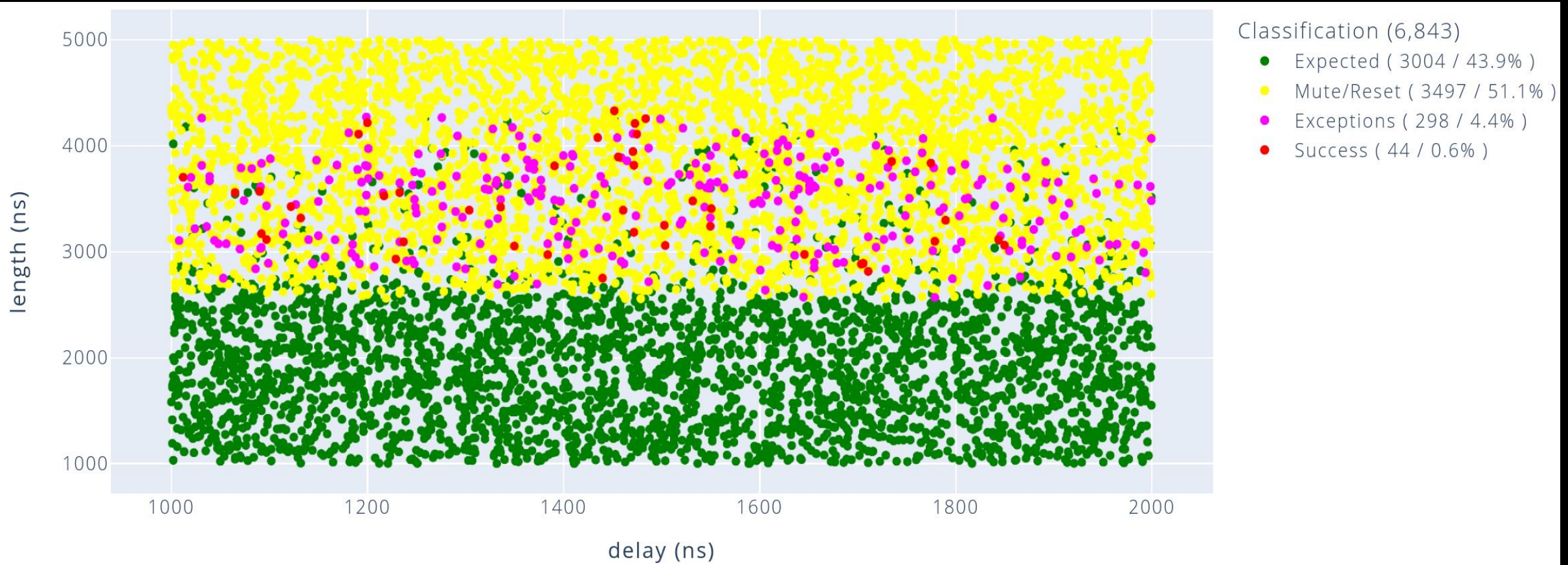
PC control → Jump to pointer → Print “Falling...”

# Trigger



Our **Attack** window:  $\sim 35.60\mu\text{s}$

# Results



Conclusion.

## Final considerations

- Identified some gaps in our approach towards **Physics** and Computing
- Concrete **impact** in our understanding of the security of modern digital systems.
- Mostly due to Physics (+ its modeling and approach) not being part of the regular Computing discussion.
- WE may be missing on:
  - A **holistic** view of systems security
  - Understanding of critical scientific fundamentals
  - Understanding of threats
  - ...and powerful attacks

raelize

Thank you! Any questions!?

Cristofaro Mune  
[cristofaro@raelize.com](mailto:cristofaro@raelize.com)  
[@pulsoid](#)