



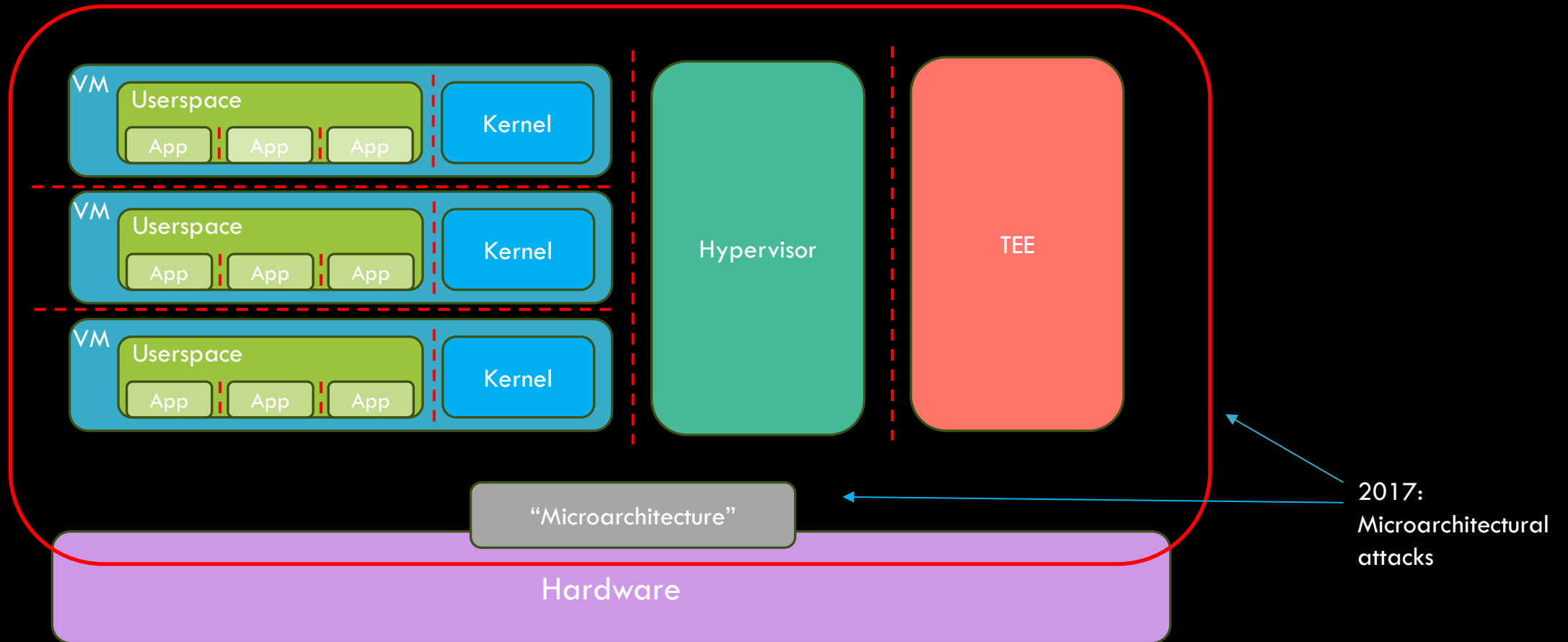
# *The Art of Fault Injection: Weird Machines all the way down*

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

Niek Timmers  
[niek@raelize.com](mailto:niek@raelize.com)  
[@tieknimmers](#)

Building up.

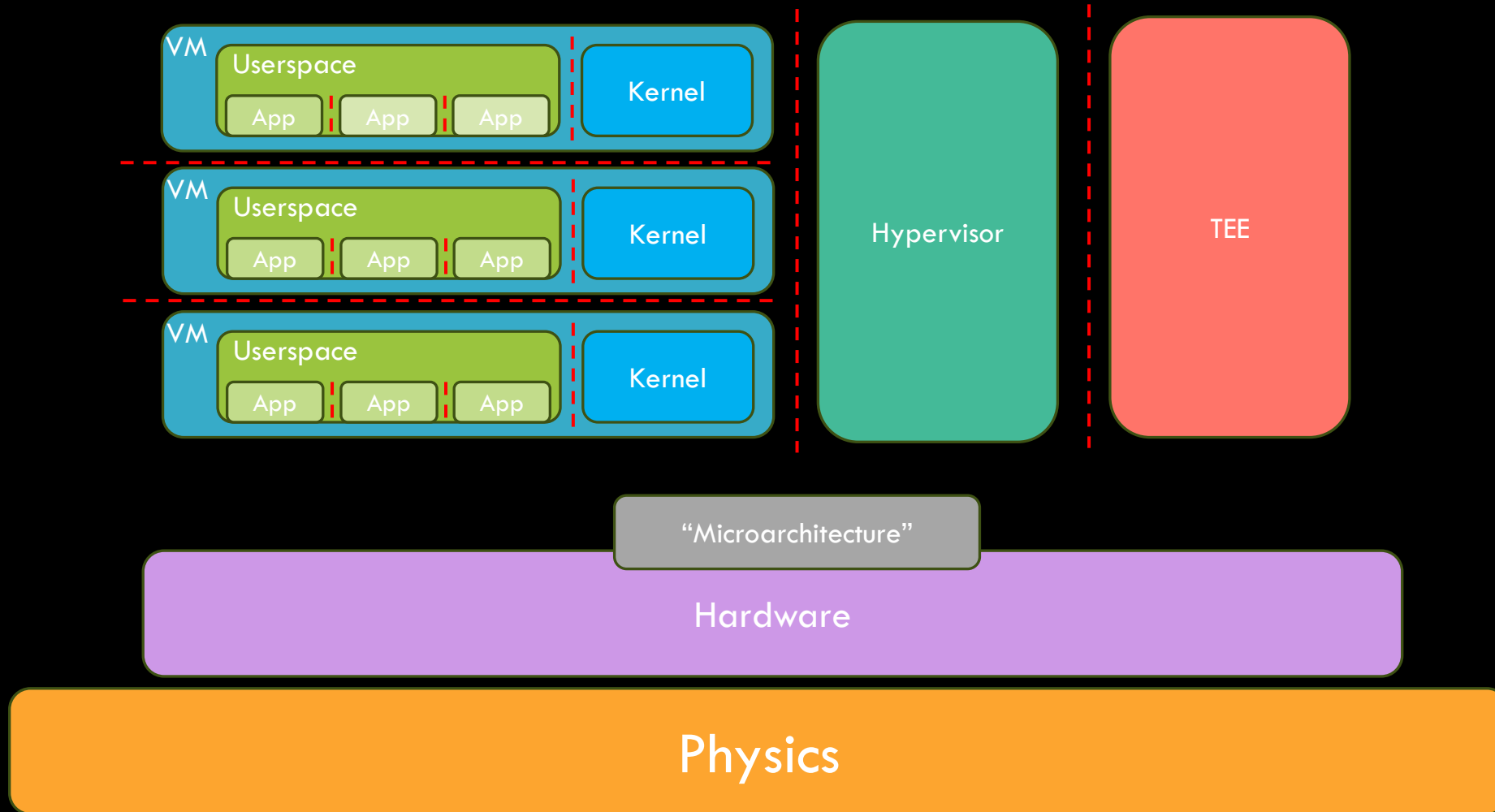
# Security boundaries



# Notes from Micro-architectural attacks [2017]

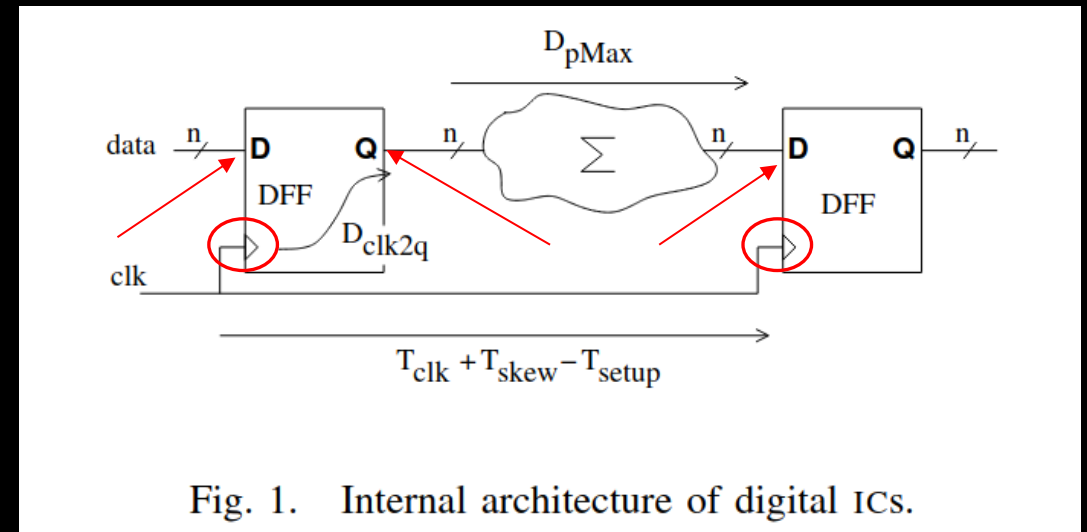
- Security models aren't **just** a Software (SW) thing
- Most of the Hardware (HW) has no **idea** of security boundaries:
  - unless factored in during design
- HW resources **shared** across security boundaries can be problematic
- It's **painful** to recover

# Are we STILL missing something?



# 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)



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

All the **computing** in the world relies upon...

*Sampling the **correct** data*

***Everywhere** (in billions of gates)*

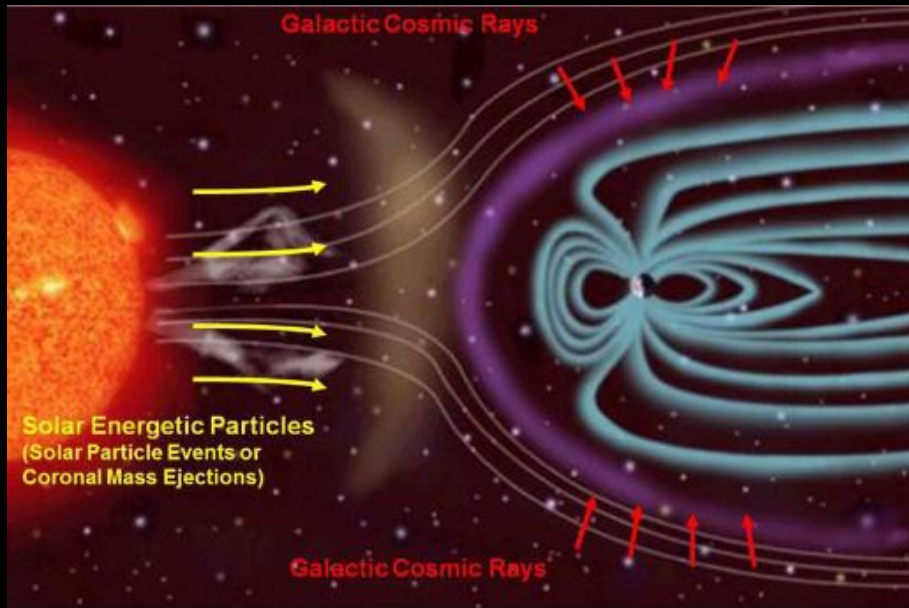
*A few billions of times per second*

***Every time.** Every single time.*

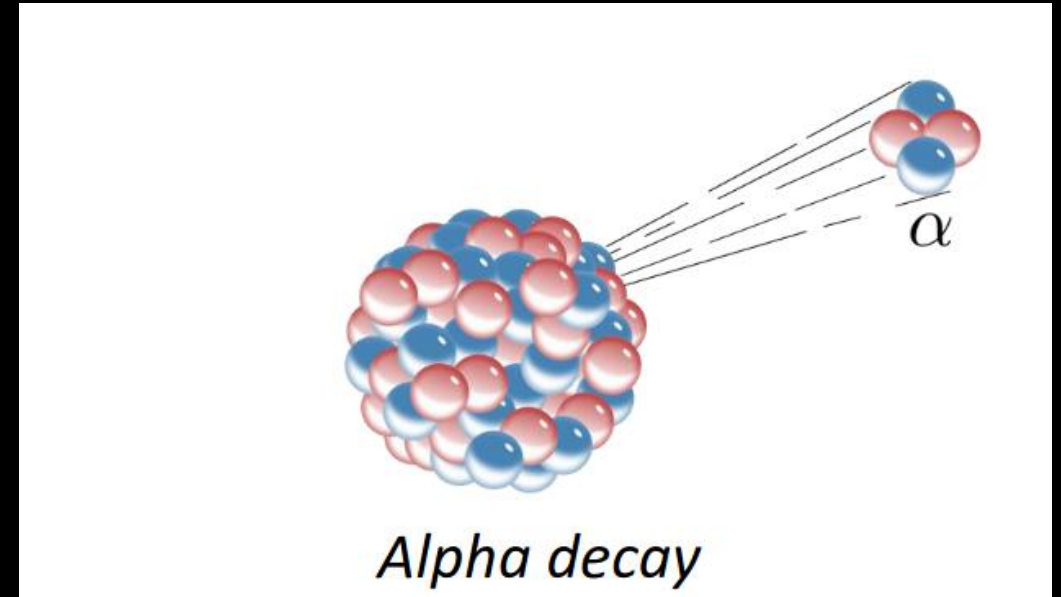
What can go wrong?



# Natural Phenomena



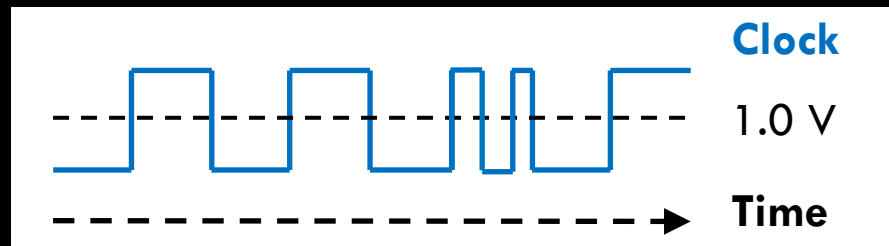
Ziegler, Lanford –“Effects of cosmic rays on computer memories”  
(1979)



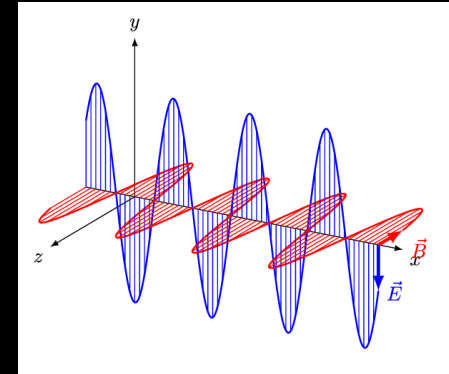
May, Woods –“Alpha-particle-induced soft errors in dynamic memories”  
(1979)

# Known (attack) techniques

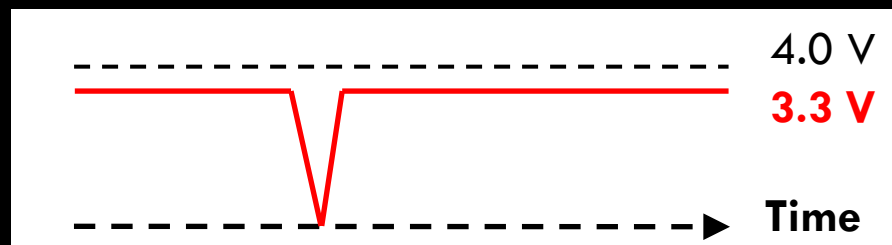
Clock



Electro-magnetic field



Voltage



Temperature



Laser ("Nexus-6" kitten)

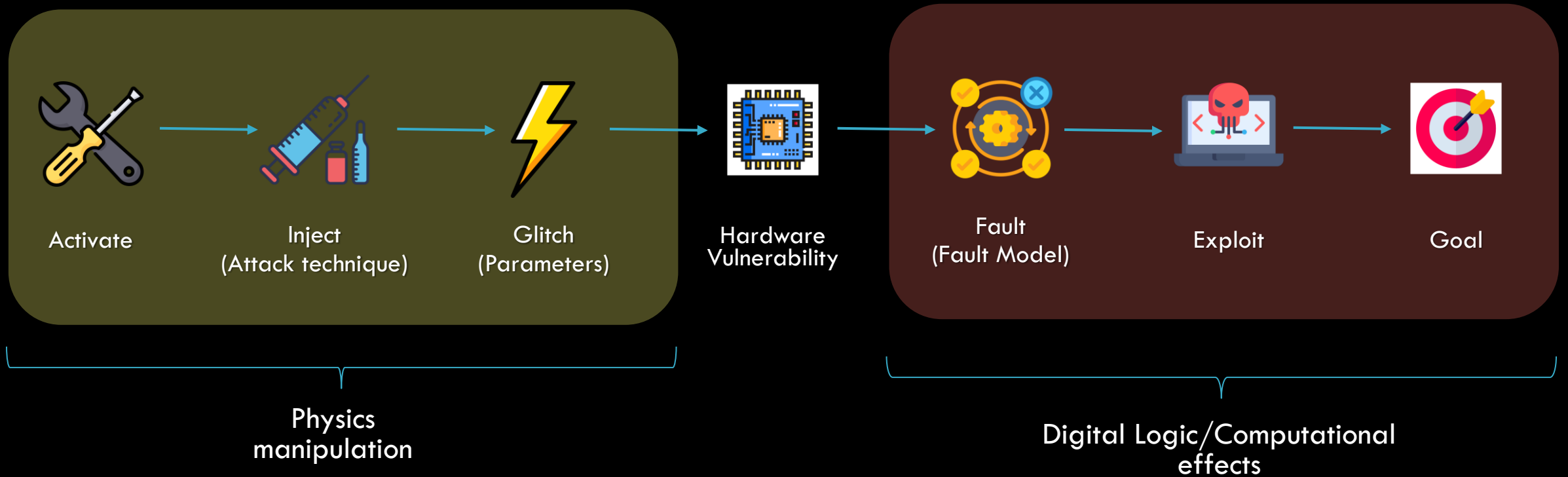


Interestingly...



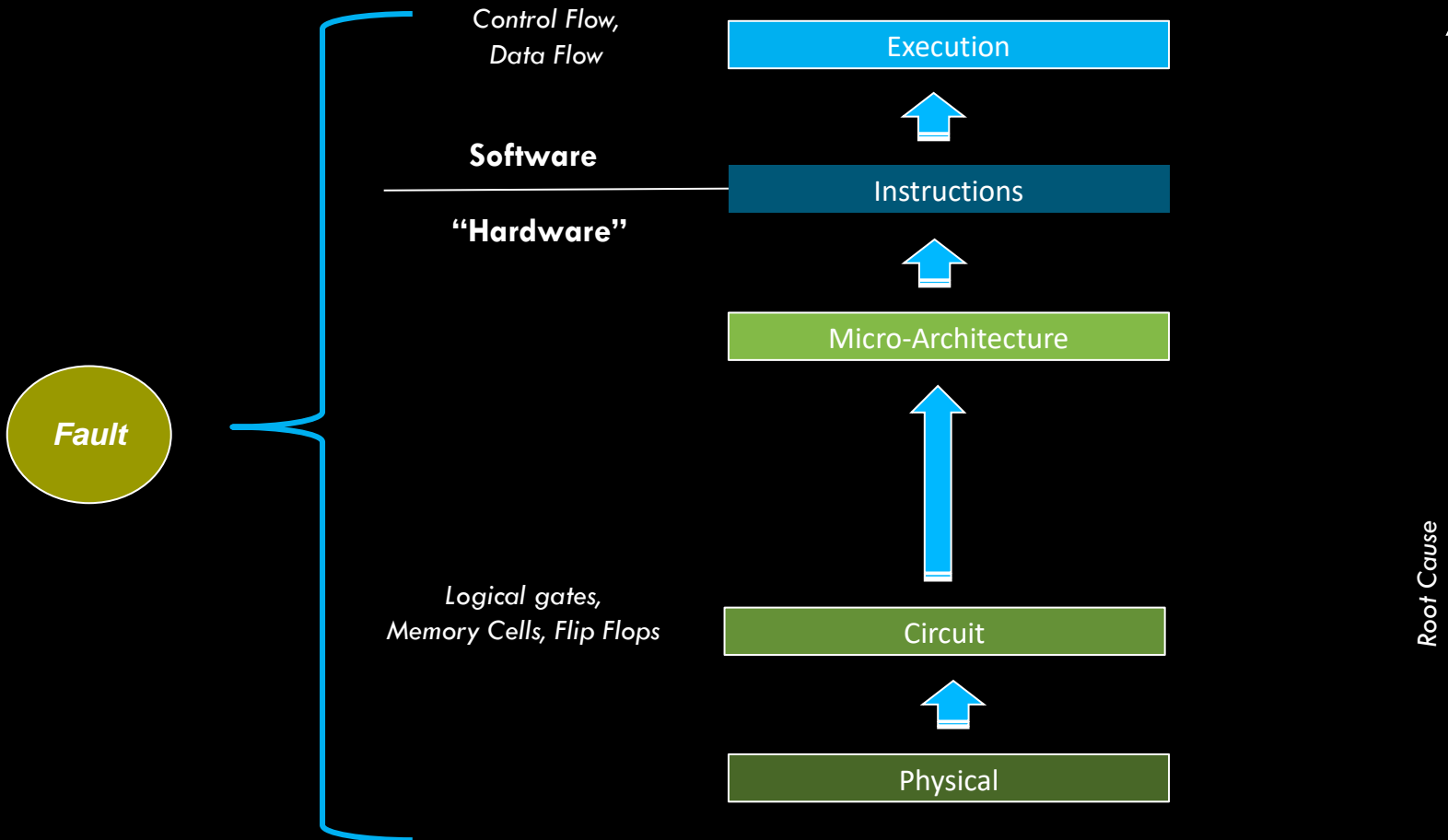
Most of them involve transfer of **energy**

# Fault Injection Reference Model (FIRM)



It's all our fault(s)!

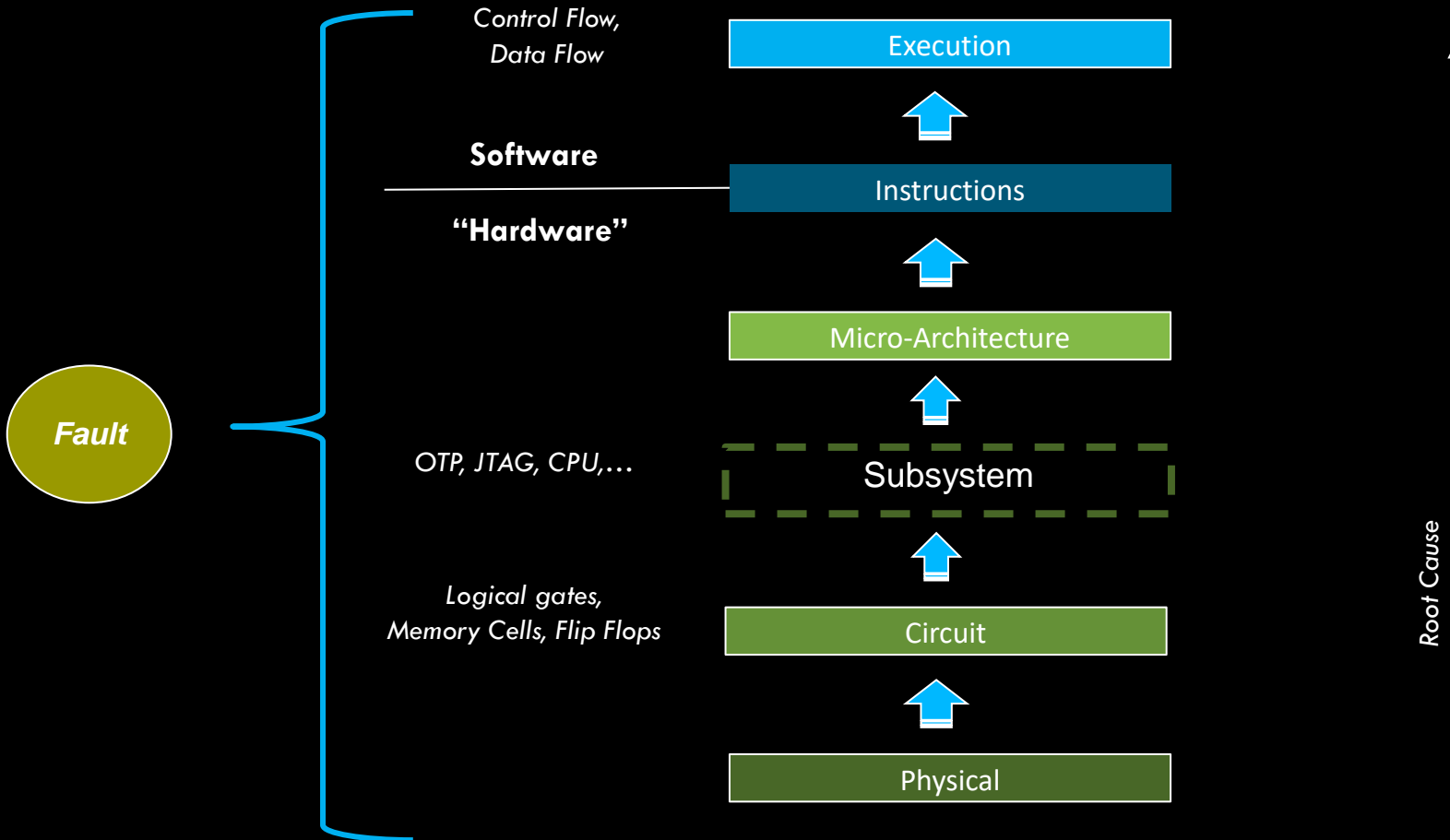
# A fault propagation model



# Notes

- Geared towards faults in software execution:
  - Not everything is instructions
- Attack against non-CPU subsystem 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





Modeling faults.

# 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 them to identify them
- Still, it may be possible to formally describe fault models geared toward **specific** attacks

# Guess how FI affects code execution...


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

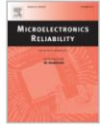
Home > CWE List > CWE-1332: Improper Handling of Faults that Lead to Instruction Skips (4.16)

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

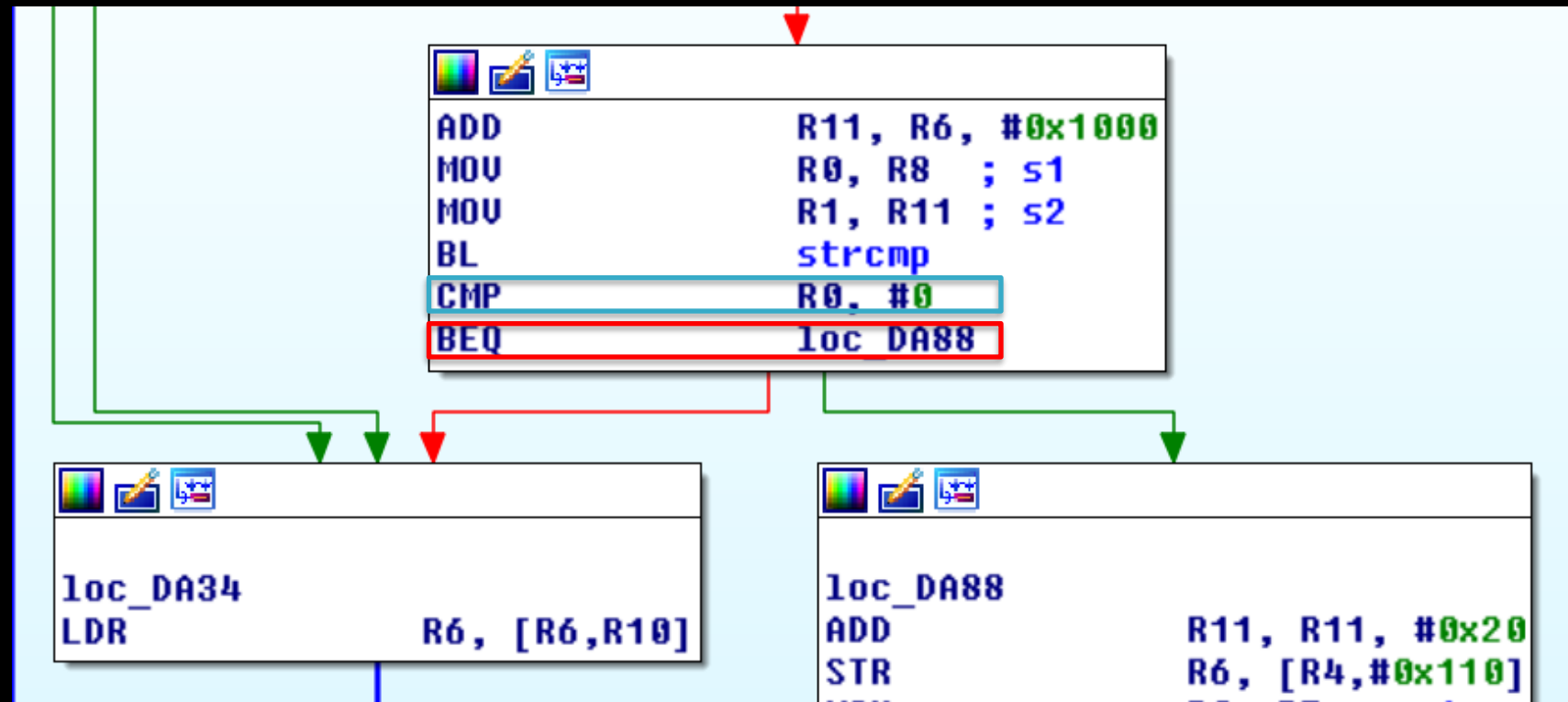
- The most common description of FI effects (fault model) on CPU execution:
  - Been with us for at least 3 decades 😊
- First attacks mostly targeted security relevant **decisions**
  - Smart Card pin authentication
  - Signature checks
  - ...

*“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
- Commonly advised and implemented in FI-resistant targets
- They reduce attack success rate:
  - Multiple glitch required
  - Attack timing 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



# 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?

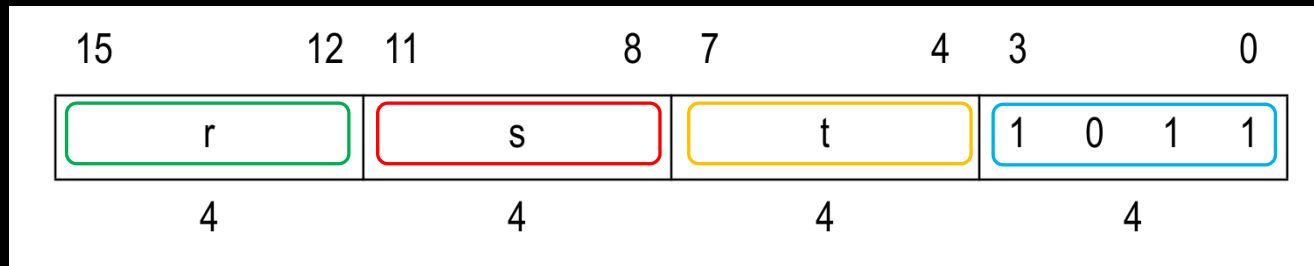
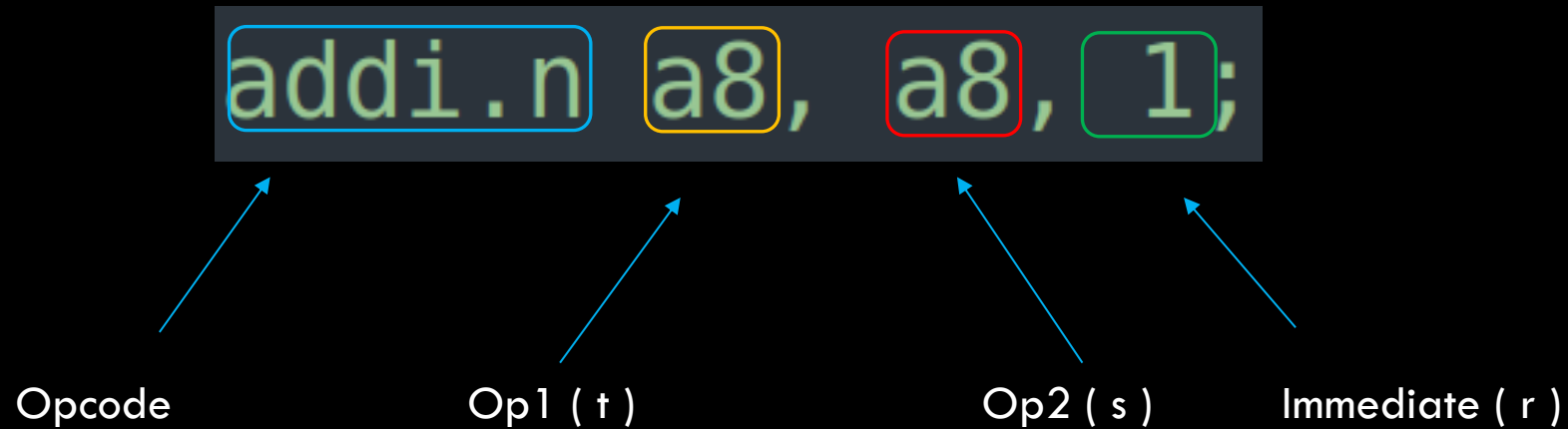
# Some hints

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

- Glitches are most likely **corrupting** instructions
- “Instruction corruption” explains 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 skipping...does NOT exist

- Well, it *MAY* still exist...:
  - But we have a better explanation now for it
- Instruction is likely corrupted to become a **NOP-equivalent** instruction:
  - i.e. an instruction with no relevant side-effects
  - Examples:
    - `orr r0, r0, r0`
    - `add r0, r0, #0`
    - ...

Weird machines...  
out of Data transfers.



# Instruction corruption

- Glitches may corrupt instructions (examples on ARM32)

- **Single bit** corruptions

```
add  x0, x1, x3    = 1000101100000001100000000000100000
add  x0, x1, x2    = 1000101100000001000000000000100000
```

- **Multi bit** corruptions

```
ldr  x0, [sp, #32] = 1111100101000000000010011111000000
str  x0, [x0, #32] = 1111100100000000000010000000000000
```

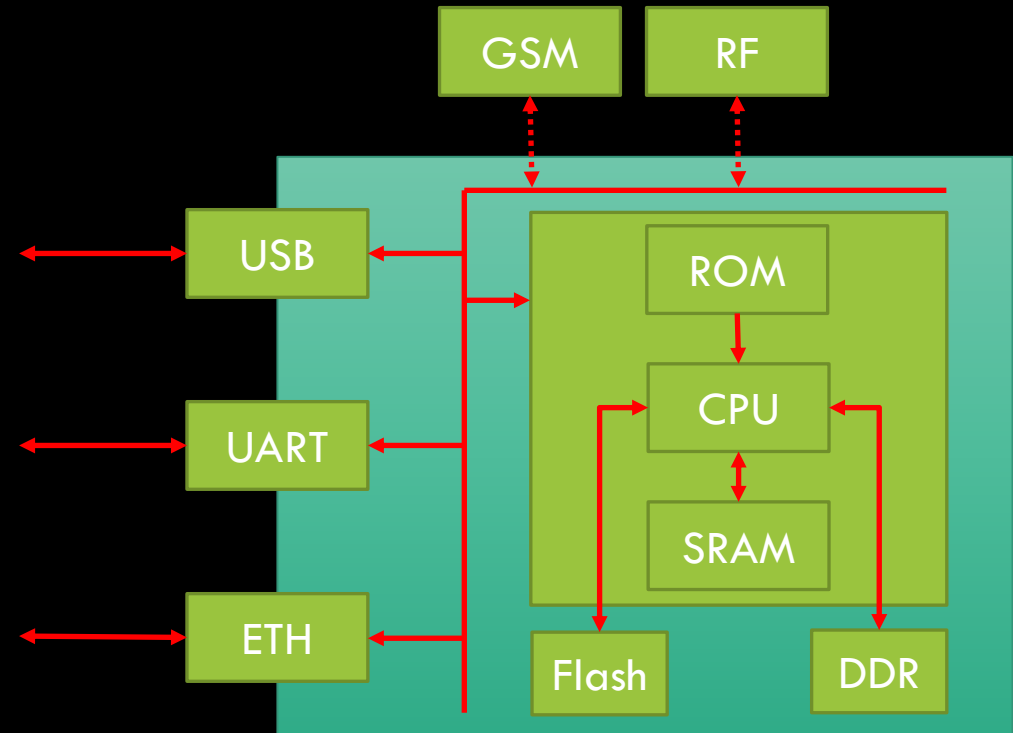
- Most chips are affected by this fault model

- 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

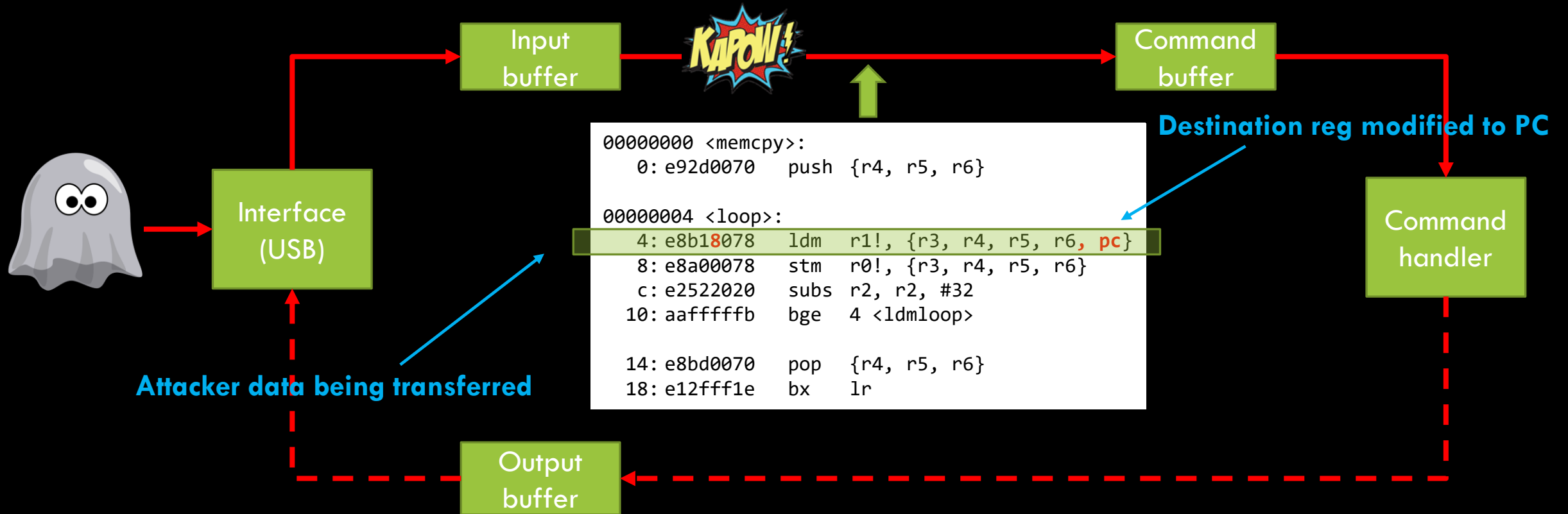
## memcpy()

- 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...

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

# A peculiar attack

- The attack uses data...but it does not target ANY **parser**
- Targets instruction decoding
- Leverages addressable PC for ARM32:
  - i.e. PC is a generic registers itself and can be explicitly assigned
- Execution flows **outside** the original program

Extension to multiple architectures...



# Our research

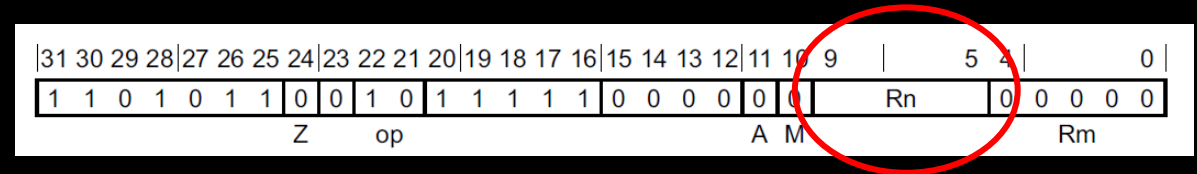
- We identified **multiple** variants and techniques
- Yield **arbitrary** code execution:
  - from controlled **data** only
  - By corrupting instruction destination registers
- Sufficiently generic to work across multiple architectures
- Examples:
  - Corrupting stored PC (in regs) or SP
  - Hijacking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

## Data scope

- Attacker data may **linger** in registers, across function boundaries
- Even if out of scope, the data is still available
- An attack may still be possible, at a later point in the execution flow

Attack example.

## “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

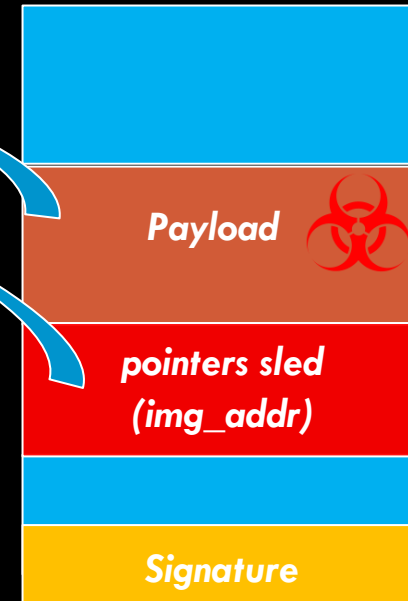
A stack overflow...without SW vulns 😊

# 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)**



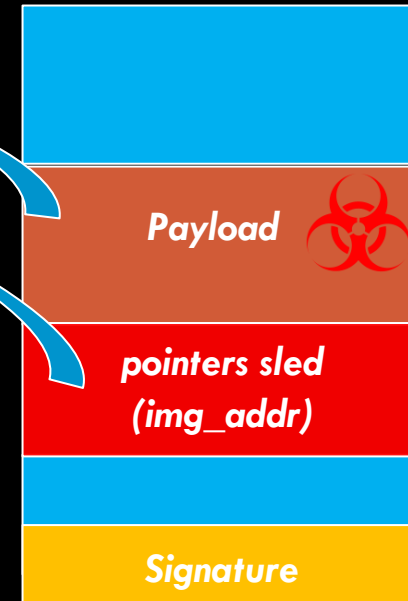
- Payload loaded at img\_addr
- Pointer sled after payload
- Glitch during pointer sled transfer

# 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

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

# Weird machines ingredients

- Memory not executable?
  - We can always start a ROP chain (we have PC control!)
- We can combine multiple glitches
  - if they are sufficiently separated in time
- E.g. We could do ROP and...
  - jump to any infinite loop in the code, at a convenient location
  - jump to our memcpy multiple times and transfer our payload multiple times
  - ...

Some open questions.

- Are “instruction skipping” faults a proper subset of “instruction corruption faults”?:
  - i.e. Can “instruction skipping” always be explained by instruction corruptions?
- Can experiments be **designed** to actually prove/disprove the above?

## 2

- For the first time the actual data being transferred becomes security relevant
  - Memcpy() is **agnostic** w.r.t to the transferred data
- Could we **distinguish** good data from attacker data?...without any knowledge of data structure?
  - If so, how?

### 3

- Can we prove that on all architectures, PC control can be obtained by corrupting a limited number of instructions?
- Can we limit attack opportunities by sanitizing data that goes out of scope?
  - E.g. should we clean temporary registers when exiting a function? What would be the impact?

## 4

- How many other relevant fault models are we missing?
- Can we somehow classify/identify them?
  - E.g:
    - given a single function and attacker-controlled data, can we formally describe all the ways to achieve PC control?
- Can we design an **ISA** such as that instruction corruption becomes much less relevant or much harder?
  - E.g. hamming distance in instruction encoding?

Conclusion.



## Final considerations

- Perturbations at the physical level can lead to unusual attacks
- Abstracting hardware/physics away is not always a good idea
- FI makes data-based attacks possible on software, without any parser being involved
- Languages and systems are typically not designed to withstand recent fault injection attacks
- We may benefit from a **holistic** view of systems security

raelize

Thank you! Any questions!?

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

Niek Timmers  
[niek@raelize.com](mailto:niek@raelize.com)  
[@tieknimmers](#)