ræ lize

Glitching devices for code execution

Niek Timmers
niek@raelize.com
@tieknimmers

Cristofaro Mune
cristofaro@raelize.com
@pulsoid

Goals

- Discuss how FI attacks to gain arbitrary code execution on devices
  - Regardless of CPU architecture
  - In absence of SW vulnerabilities
- Show how SW-based countermeasures can be entirely bypassed
- Discuss techniques that allow to loosen timing requirements

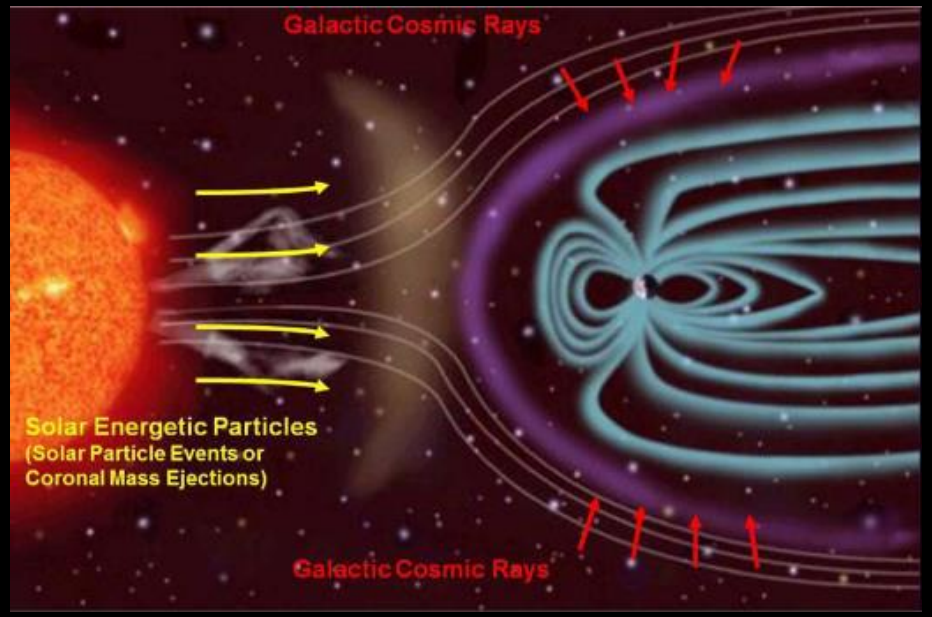
Contribute to the field

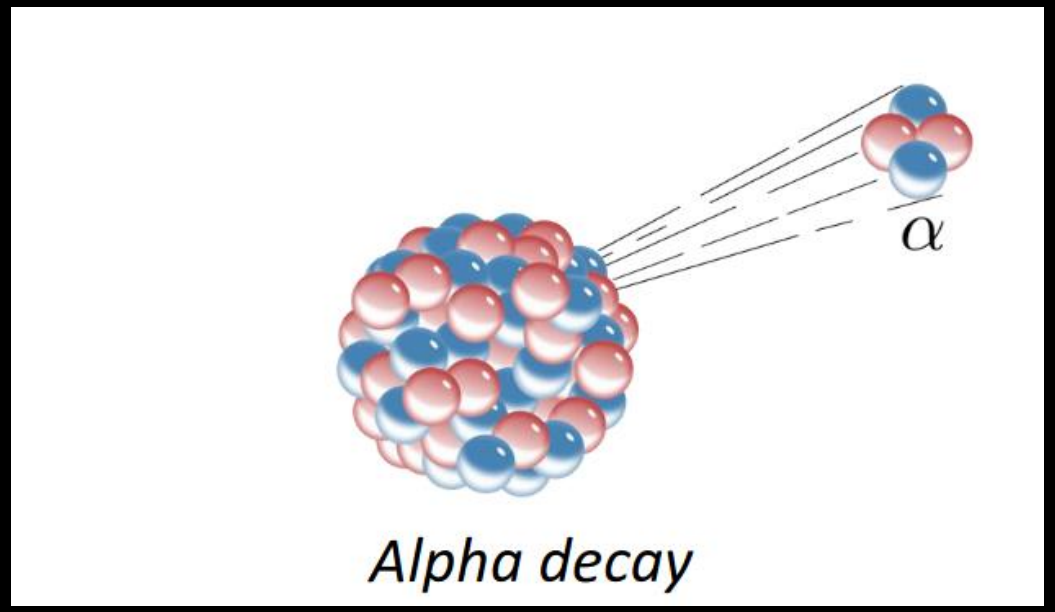
Agenda

- F(I)oundations
- FIeld systematization
- SW-based countermeasures
- Achieving PC control and countermeasures bypass
- Loosening time requirements

F(I)oundations.

(F)Inception: Natural Phenomena





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

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

First attacks (Academia): Differential Fault Analysis

- Boneh, DeMillo, Lipton – "On the Importance of Checking Computations (Extended abstract)" (1996)
  - No paper seems available nowadays
  - Referred by a Bihan-Shamir note (1996)
    - https://cryptome.org/jya/dfa.htm#Bellcore

- Now known as "Bellcore attack"
  - Single fault attack recovers RSA private key on CRT signature

First attacks (Hacking): Unlooper

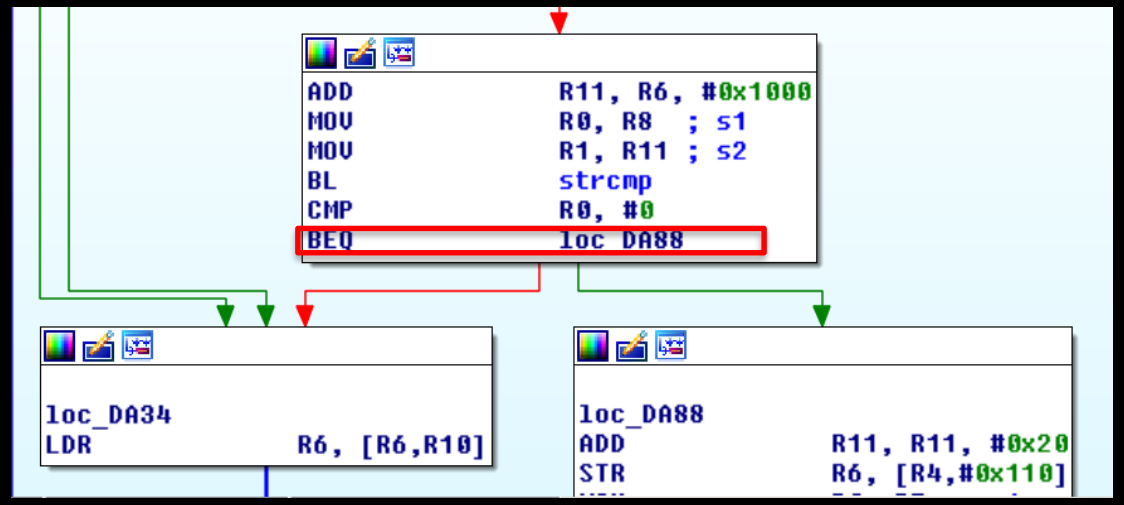
- Target: Pay-TV Smart Cards
  - Hacked smart cards were remote disabled
- Clock glitching
- Revive hacked smart cards!
  - “Jump out" of an infinite loop

Traditional attacks

Differential Fault Analysis (break crypto)

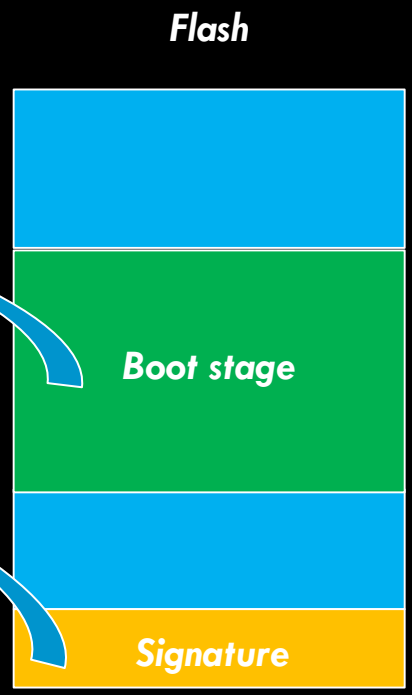
```
000c8420h: D0 EF AA FB 43 4D 33 85 45 F9 02 7F 50 3C 9F A8
000c8430h: 51 A3 40 8F 92 9D 38 F5 BC B6 DA 21 10 FF F3 D2
000c8440h: CD 0C 13 EC 5F 97 44 17 C4 A7 7E 3D 64 5D 19 73
000c8450h: 60 81 4F DC 22 2A 90 88 46 EE B8 14 DE 5E 0B DB
000c8460h: E0 32 3A 0A 49 06 24 5C C2 D3 AC 62 91 95 E4 79
000c8470h: E7 C8 37 6D 8D D5 4E A9 6C 56 F4 EA 65 7A AE 08
000c8480h: BA 78 25 2E 1C A6 B4 C6 E8 DD 74 1F 4B BD 8B 8A
000c8490h: 70 3E B5 66 48 03 F6 0E 61 35 57 B9 86 C1 1D 9E
```

Bypass checks



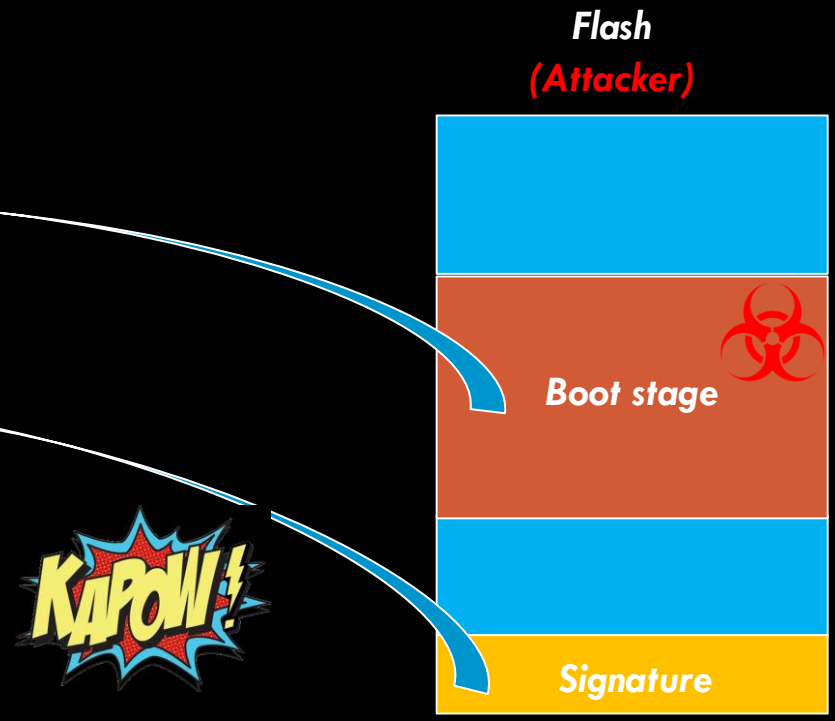
Secure boot

```
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     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 }
```



FI attack on 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     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 }
```



Why does it works?

- "Instruction skipping"
  - Glitch assumed to "*skip instructions*" → conditional instructions are not executed
  - Execution flow "falls through"

- Widely used description in academia and industry:
  - Dominated FI attack modeling for 30+ years.

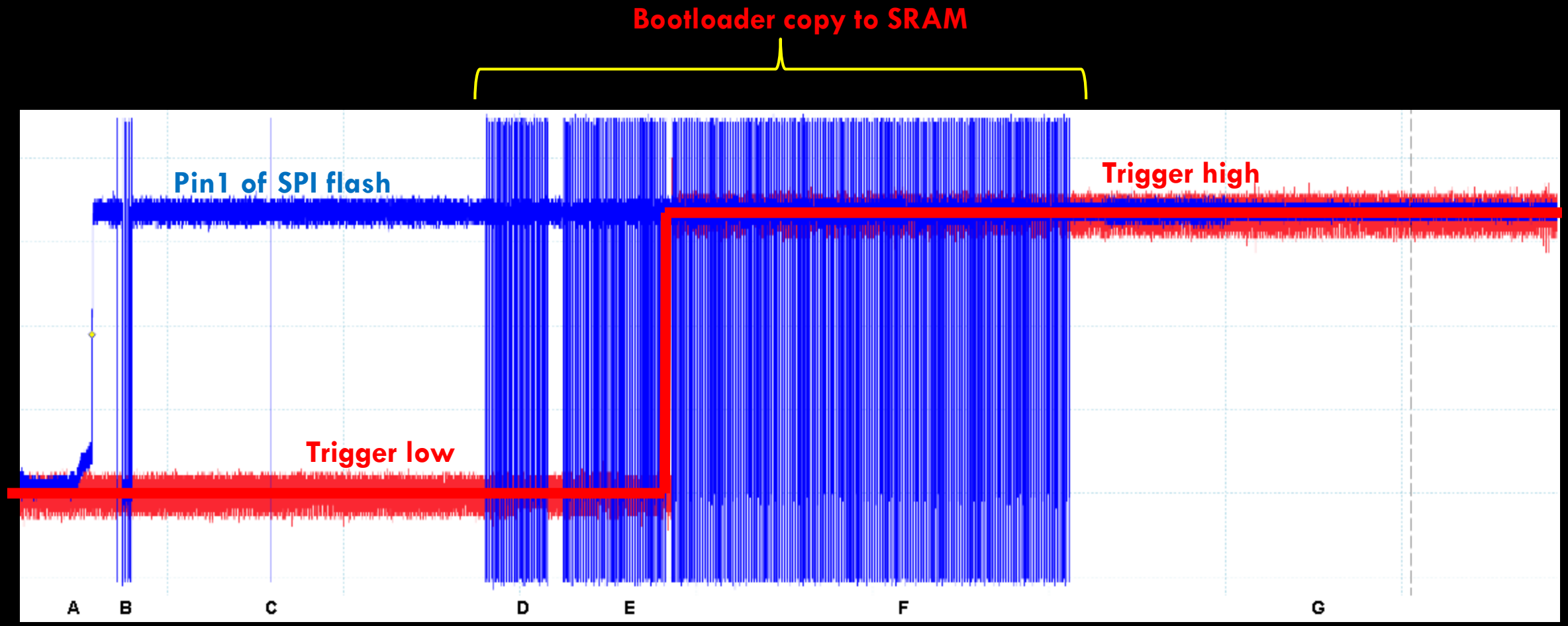
Our first Fault Model!

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 }
```

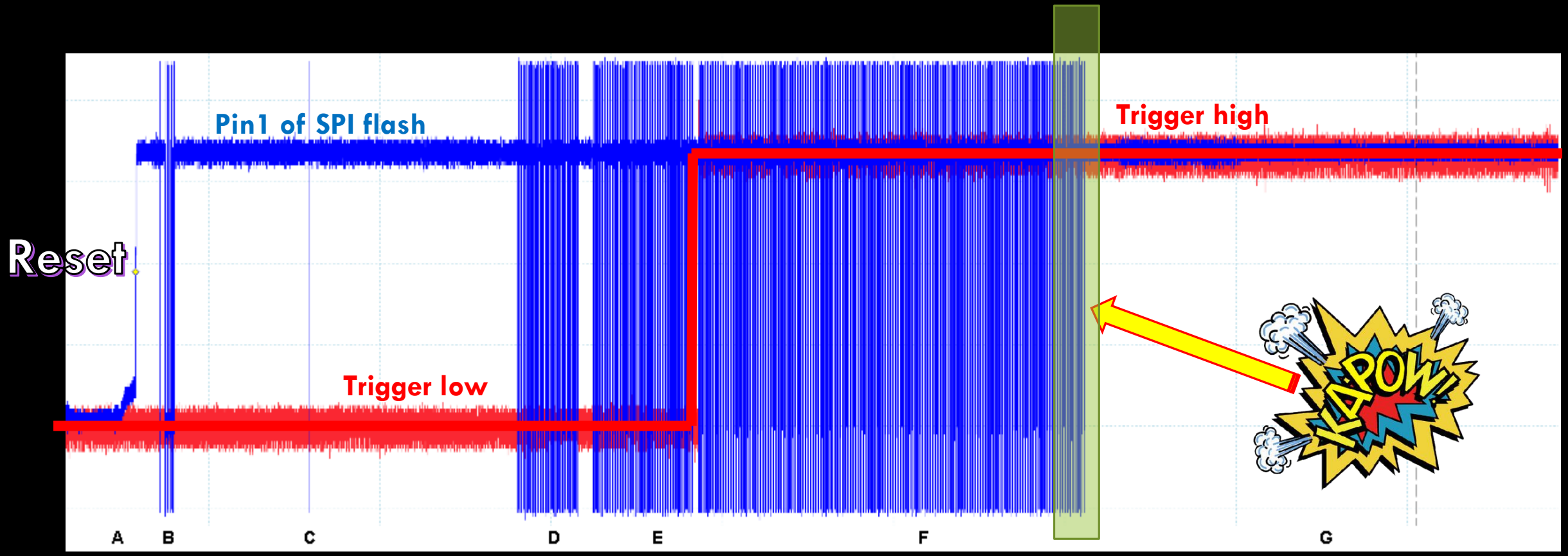
- "Instruction skipping" requires accurate timing
  - Synchronization with target often required
- Can be executed blindly:
  - i.e. no assumption on type of fault
  - "Glitch 'n pray"

Example: ESP32 Secure Boot bypass (1)



We use Flash communication for synchronization (triggering)

Example: ESP32 Secure Boot bypass (2)



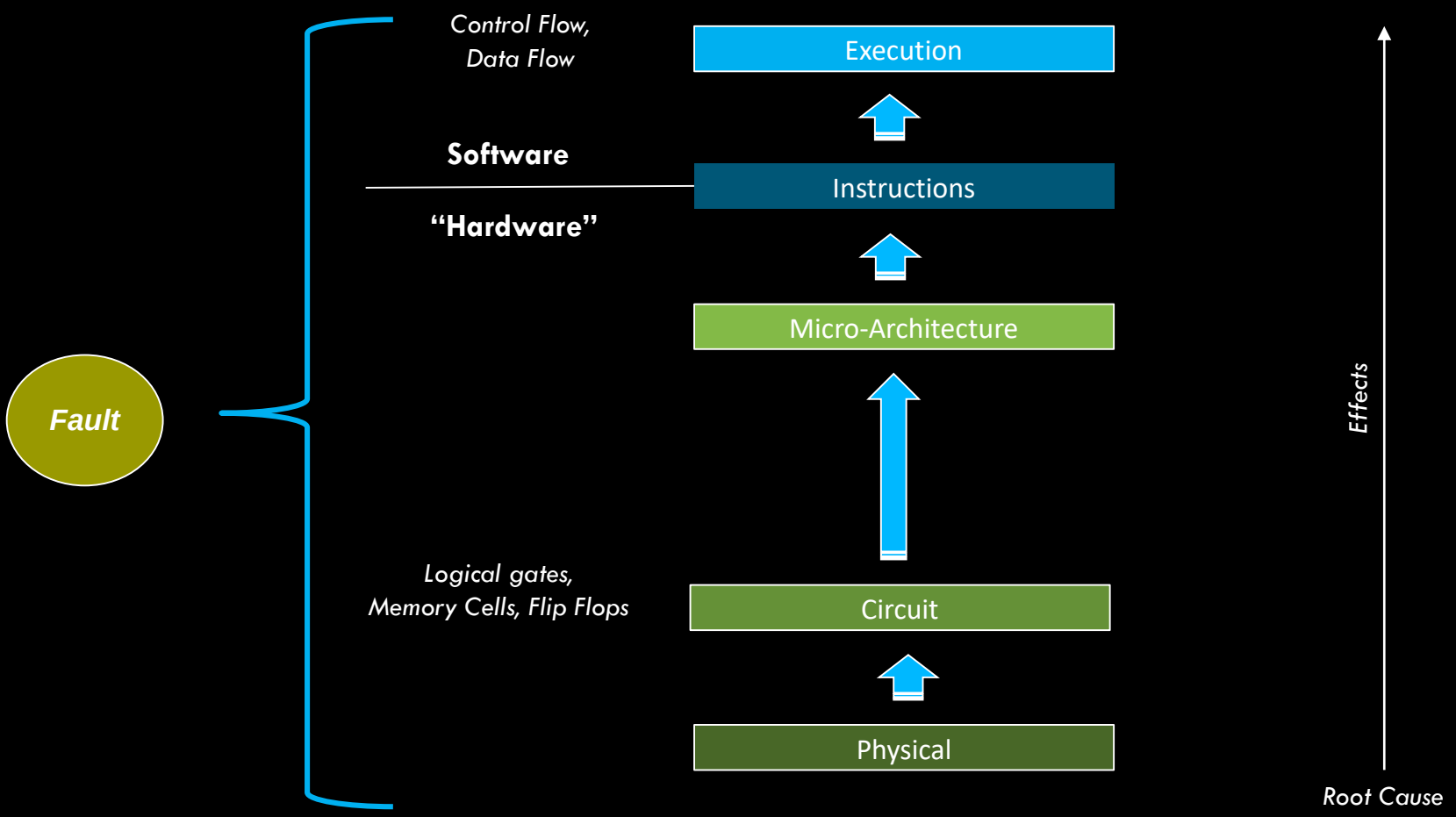
Glitch injected somewhere after the bootloader is copied.

No control whatsoever…

…is it just randomness and luck?

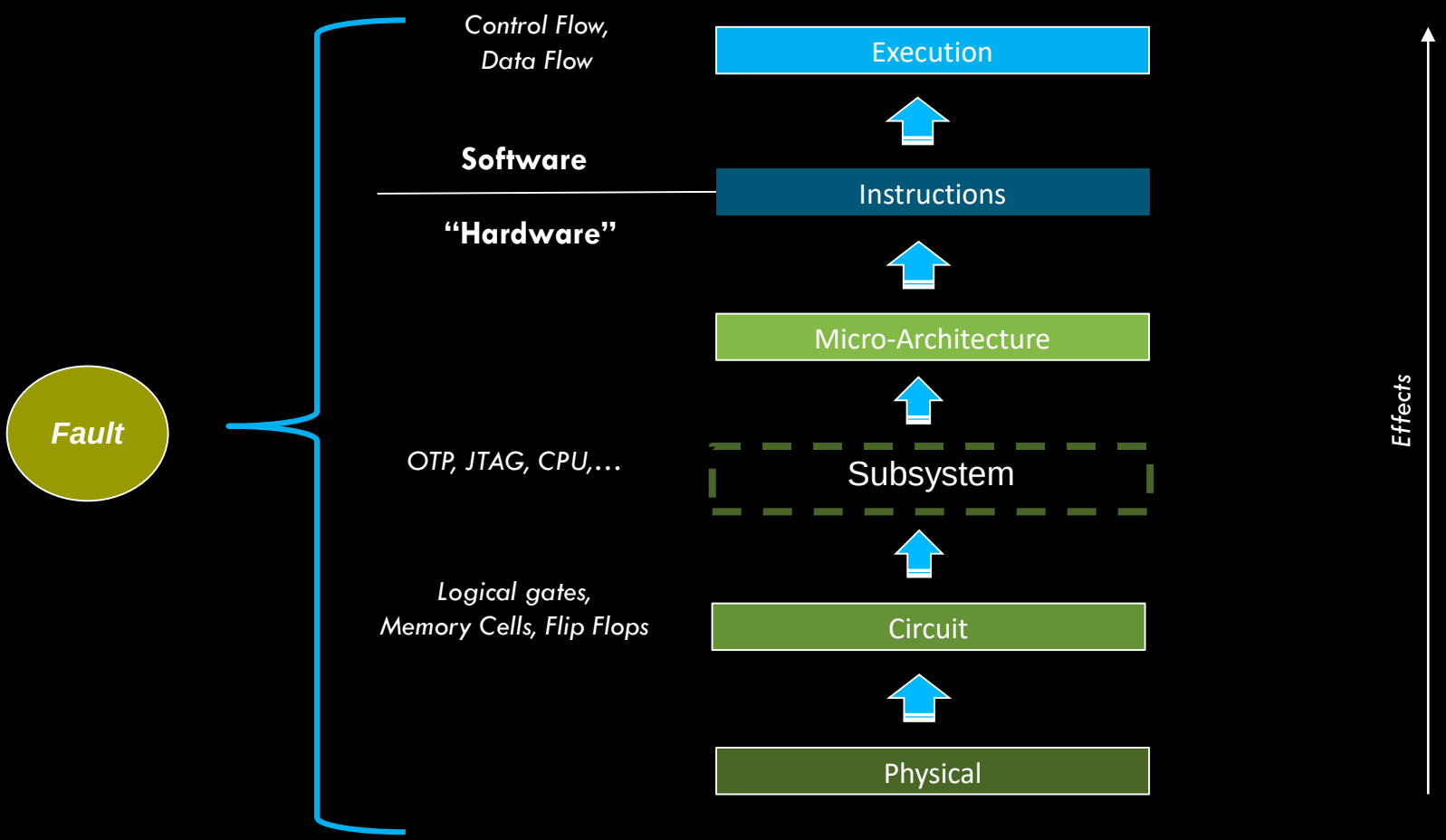
Science: FIeld systematization.

Effects of a fault



[2018]: Yuce, Schaumont, Witteman

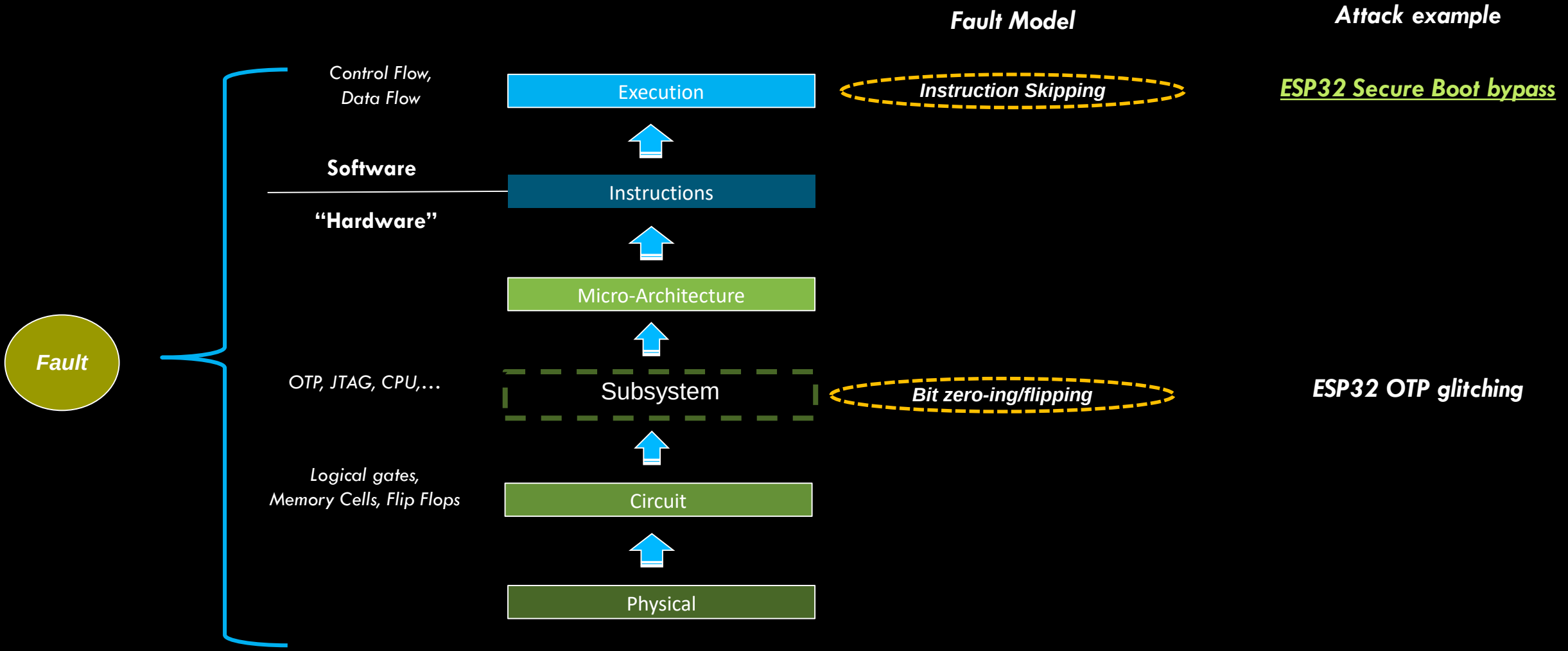
Effects of a fault++



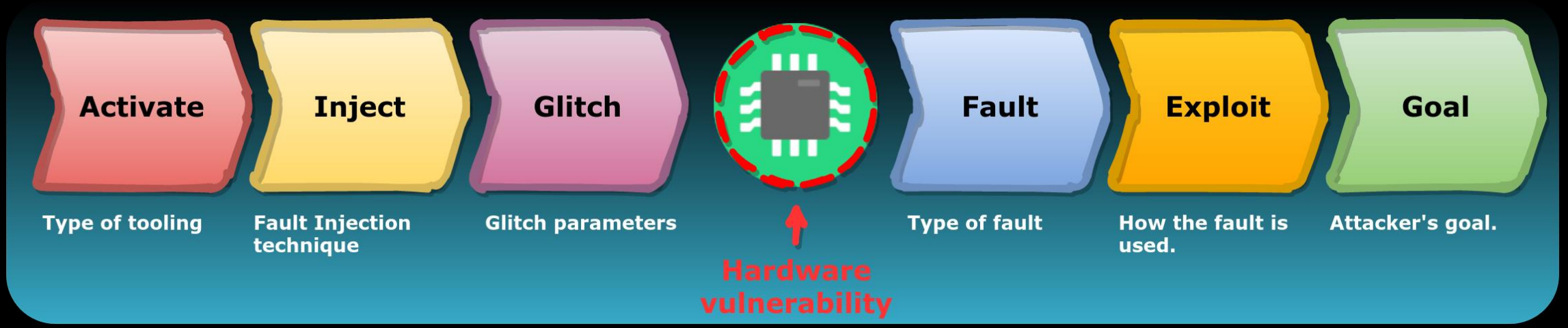
Fault Model

- A glitch may cause the system to misbehave in multiple manners
  - Not easily predictable…if predictable at all
- Multiple kind of faults may be generated
  - Not all the faults are interesting and can be used in an attack
- Fault model: defines the relevant set of faults
  - i.e. that can be leveraged into an exploit

Fault Models and attacks

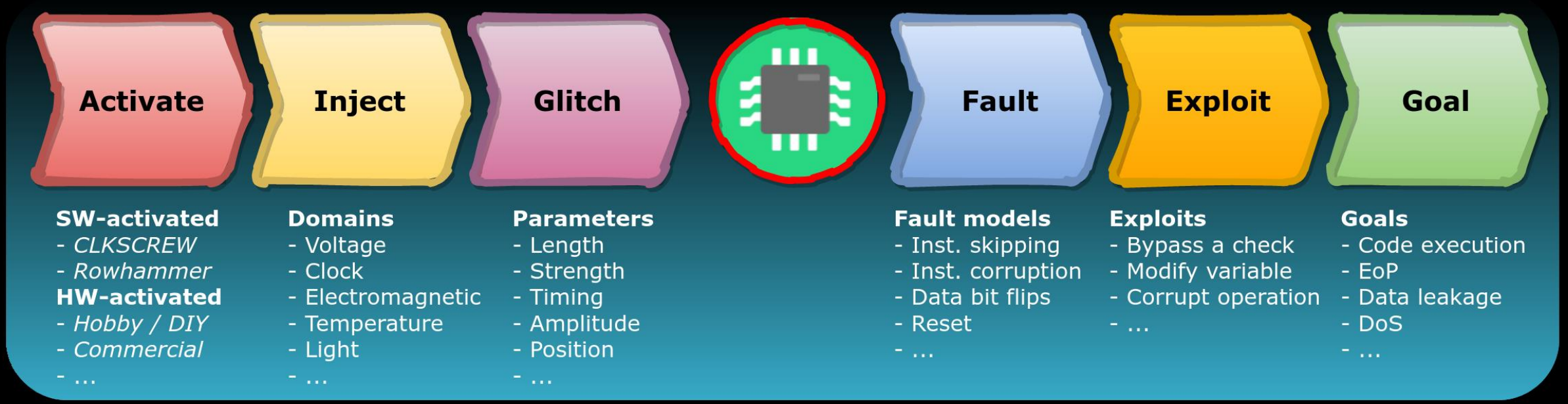


Fault Injection Reference Model (FIRM)



Modeling an FI attack.

More in details…



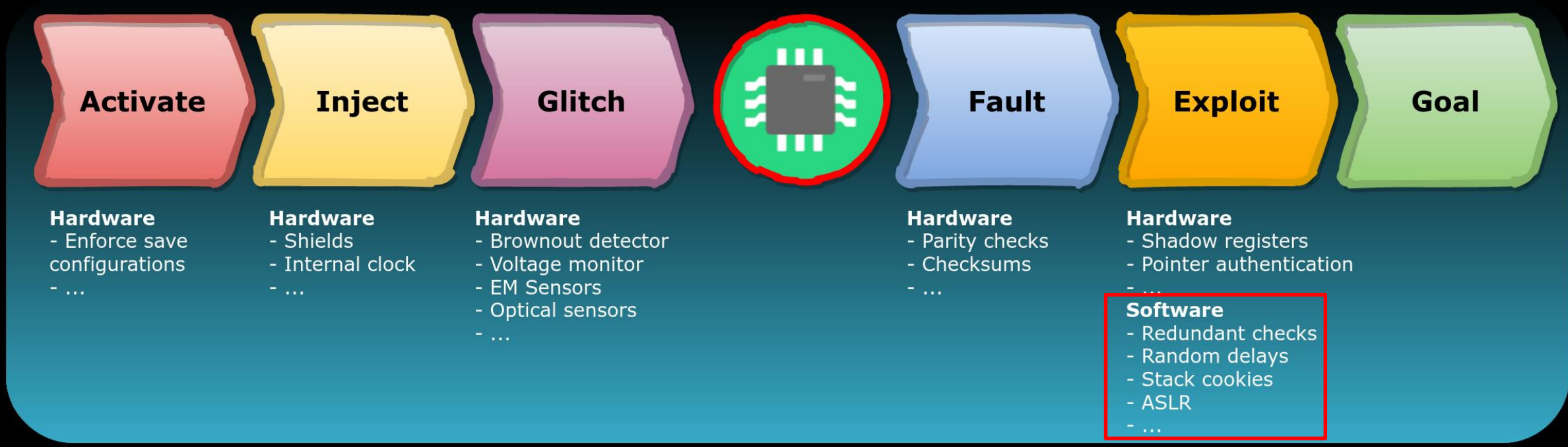
More information: https://raelize.com/posts/raelize-fi-reference-model/

Key points

- FI vulnerability: sensitiveness to a FI technique
  - E.G. Target may be vulnerable to EM but not to voltage glitching.
- A FI vulnerability always occurs in hardware
  - Software only concurs to its exploitability
- The same vulnerability may yield different faults
- Effects of a single fault may fall within multiple fault models.
- Different fault models yield different attacks

SW-based countermeasures.

FI countermeasures overview



SW-based 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 }
```

- Checks are performed multiple times
- Assumption:
  - A glitch is required for every check

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 }
```

- Random delays are introduced around critical checks
- Location in time is not fixed anymore
- Assumption:
  - A glitch must “hit” a specific point in time

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
  - Target synchronoziation more difficult

Untold(?) assumptions…

- SW-based countermeasures…require SW to be executed:
  - E.g. multiple checks

- Attack is expected to:
  - target specific checks (strong attack "locality")
  - be very precise in time for hitting specific instructions:
    - E.g. sharp glitches, multiple triggering…

Instruction skipping fault model assumed!

What if…we switch to another fault model?

Instruction corruption.

Instruction corruption

- Glitches may corrupt instructions (examples on ARM32)
  - Single bit corruptions

```
add   x0, x1, x3    = 10001011000000110000000000100000
add   x0, x1, x2    = 10001011000000010000000000100000
```

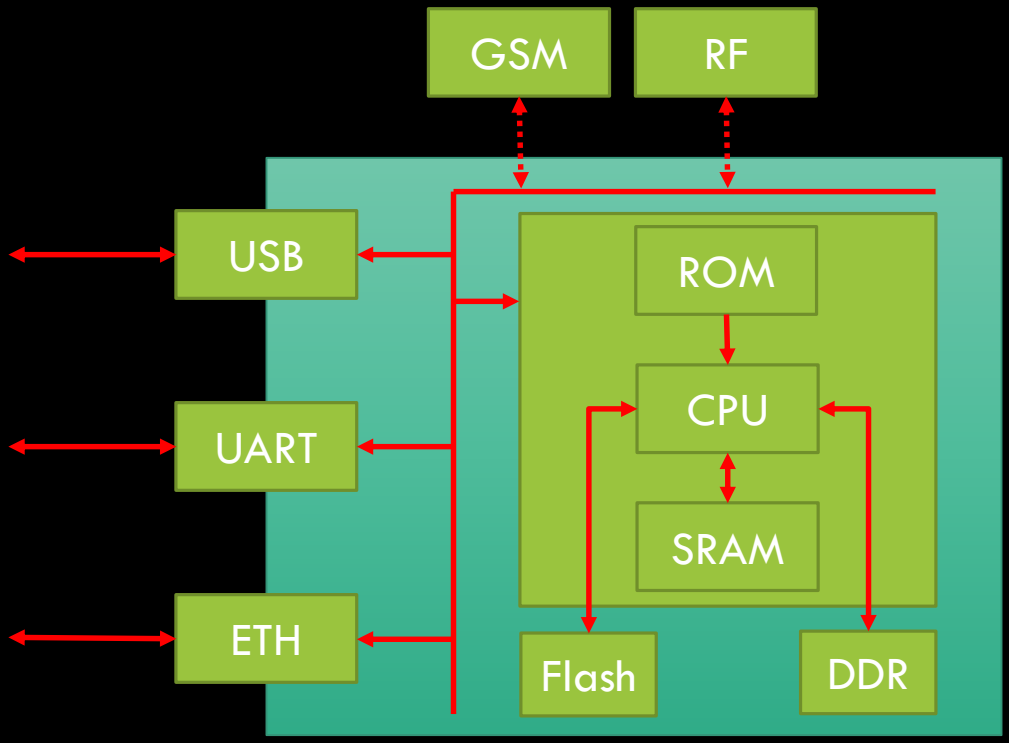
  - Multi bit corruptions

```
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



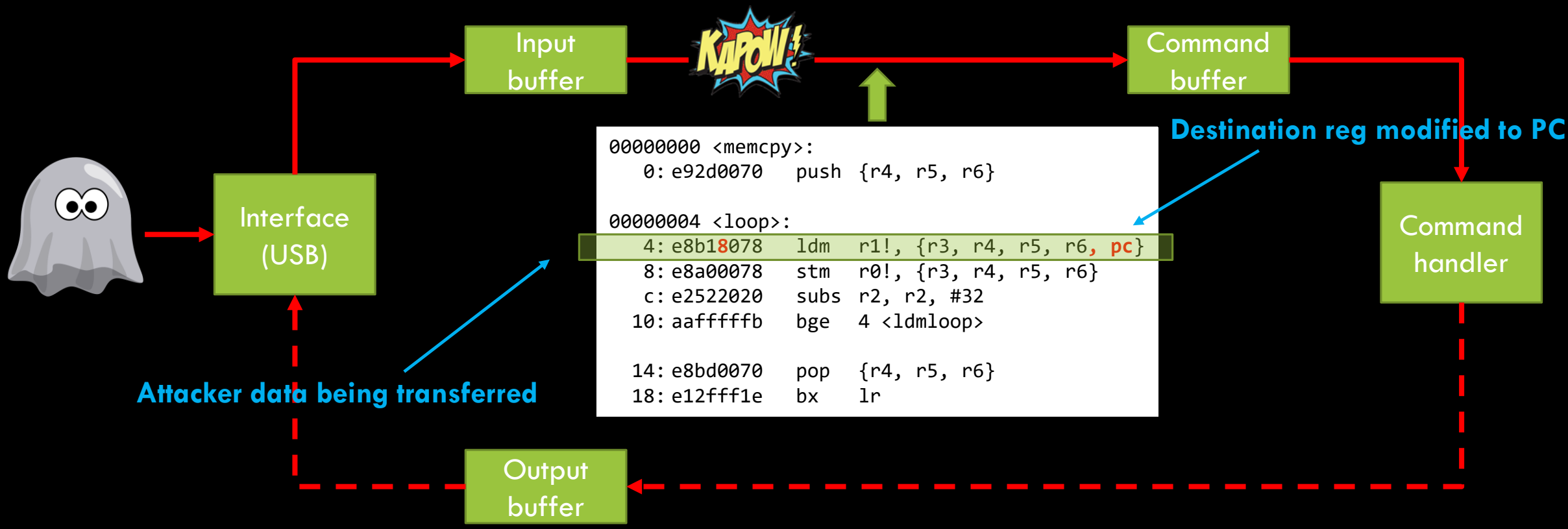
Transferred data may be under attacker's control

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…

Example: USB data transfer



PC set to attacker data. Control flow directly hijacked!

Attack summary (ARM32)

- Corrupt instruction
- Modify load instruction operands (destination register)
- Directly addressable PC is set to attacker controlled value

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

Nice! Does it work on other architectures?

Definitely!

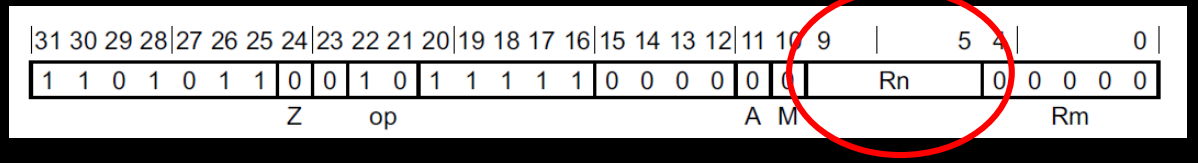
- We identified multiple variants and techniques
- Yield arbitrary code execution:
  - from controlled data only
  - By corrupting instruction destination registers
- Sufficiently generic to work across multiple architectures
- Examples:
  - Corrupting stored PC (in regs) or SP
  - Hijacking jump/call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

More interesting examples in our research!

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

```
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
```

- Glitch RET instruction into RET x6 or RET x7:
  - Equivalently glitch ldr x6, … to ldr x30, …

PC hijacked from controlled data.

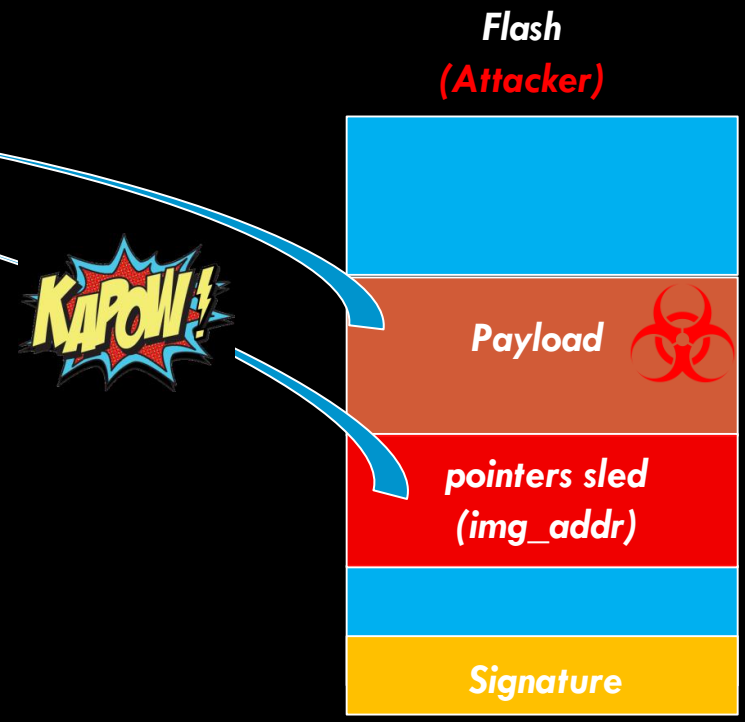
“Instruction corruption”: Recipe for success

- Identify data transfers you control
- Set your transfer payload to a sled of pointers
  - Point to your shellcode location
- Glitch during ANY memcpy
- PC control

A stack overflow…without SW vulns ☺

Attacking Secure Boot

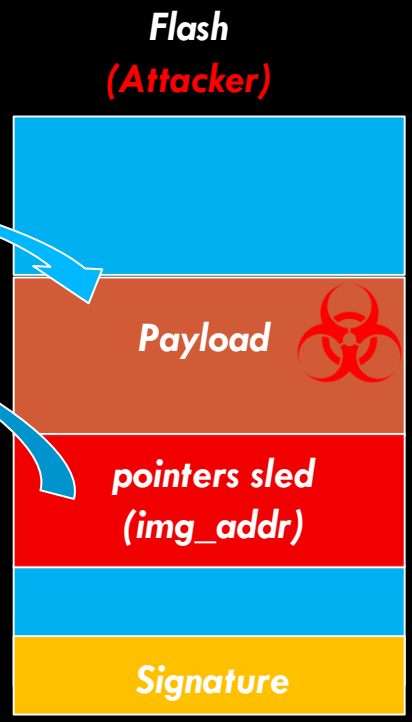
```
1  int load_exec_next_boot_stage() {
2
3      // Destination addresses in SRAM
4      uint32_t img_addr = 0xd0000000;
5      uint32_t sig_addr = 0xd1000000;
6
7      // Copy next stage image from Flash to SRAM
8      load_next_stage_img(img_addr);
9
10     // Copy signature from Flash to SRAM
11     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 }
```



- 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 }
```



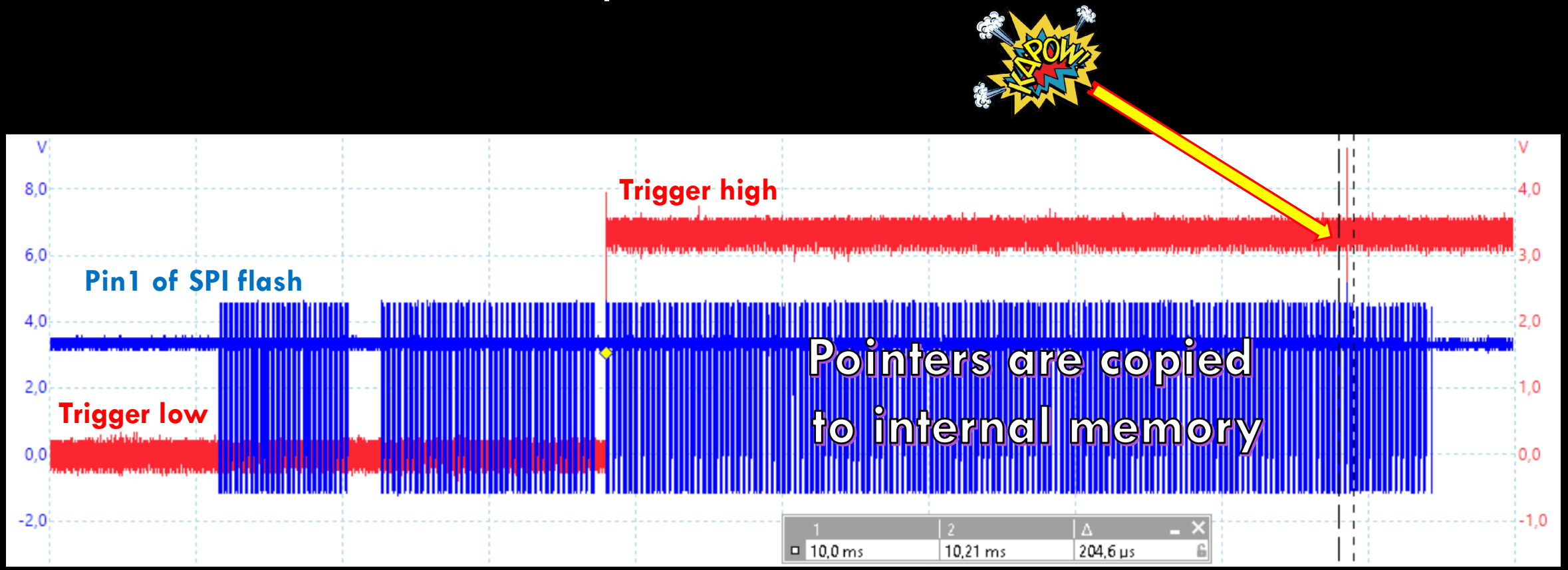
- PC value set to img_addr
- Control flow hijacked
- SW-based countermeasures not executed

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

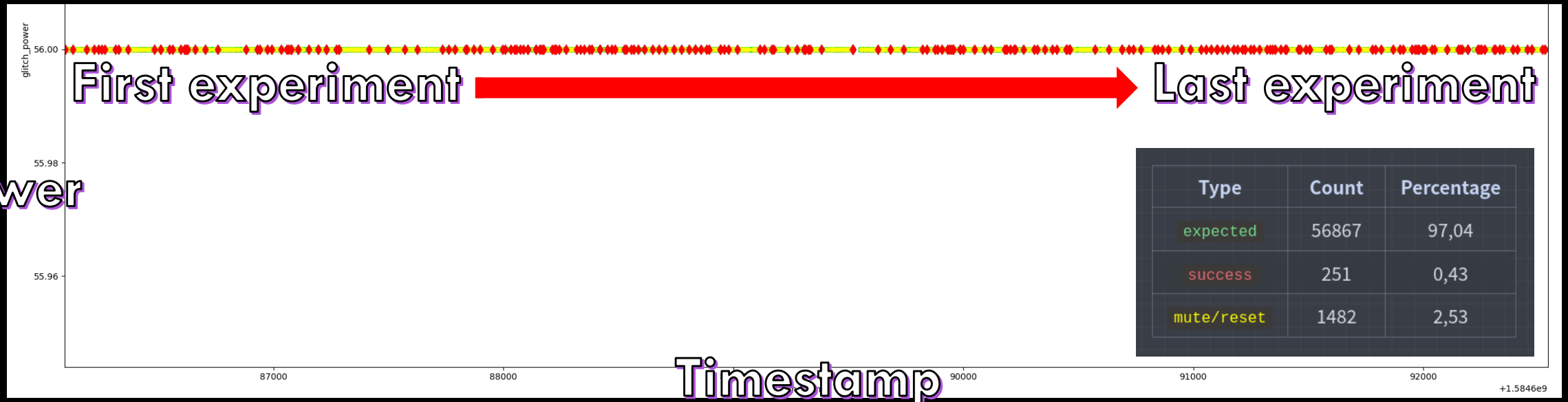
Very hard to protect against. Applicable to FI-resistant targets.

Example: ESP32 PC control



We inject glitches in a small ‘attack window’ while the bootloader is being copied.

Success



Type	Count	Percentage
expected	56867	97,04
success	251	0,43
mute/reset	1482	2,53

We achieve a success rate of 2 successful glitches per minute where we load an arbitrary value into the program counter.

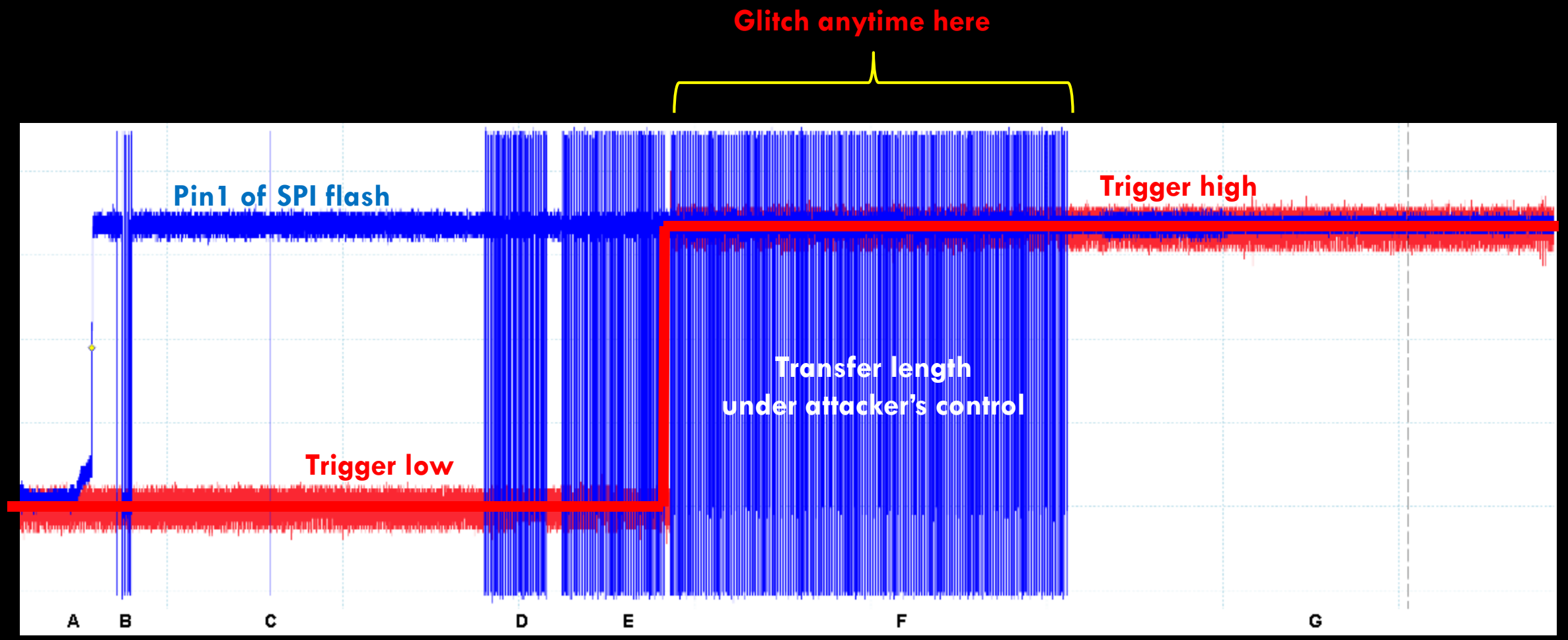
Loosening time constraints.

Observations

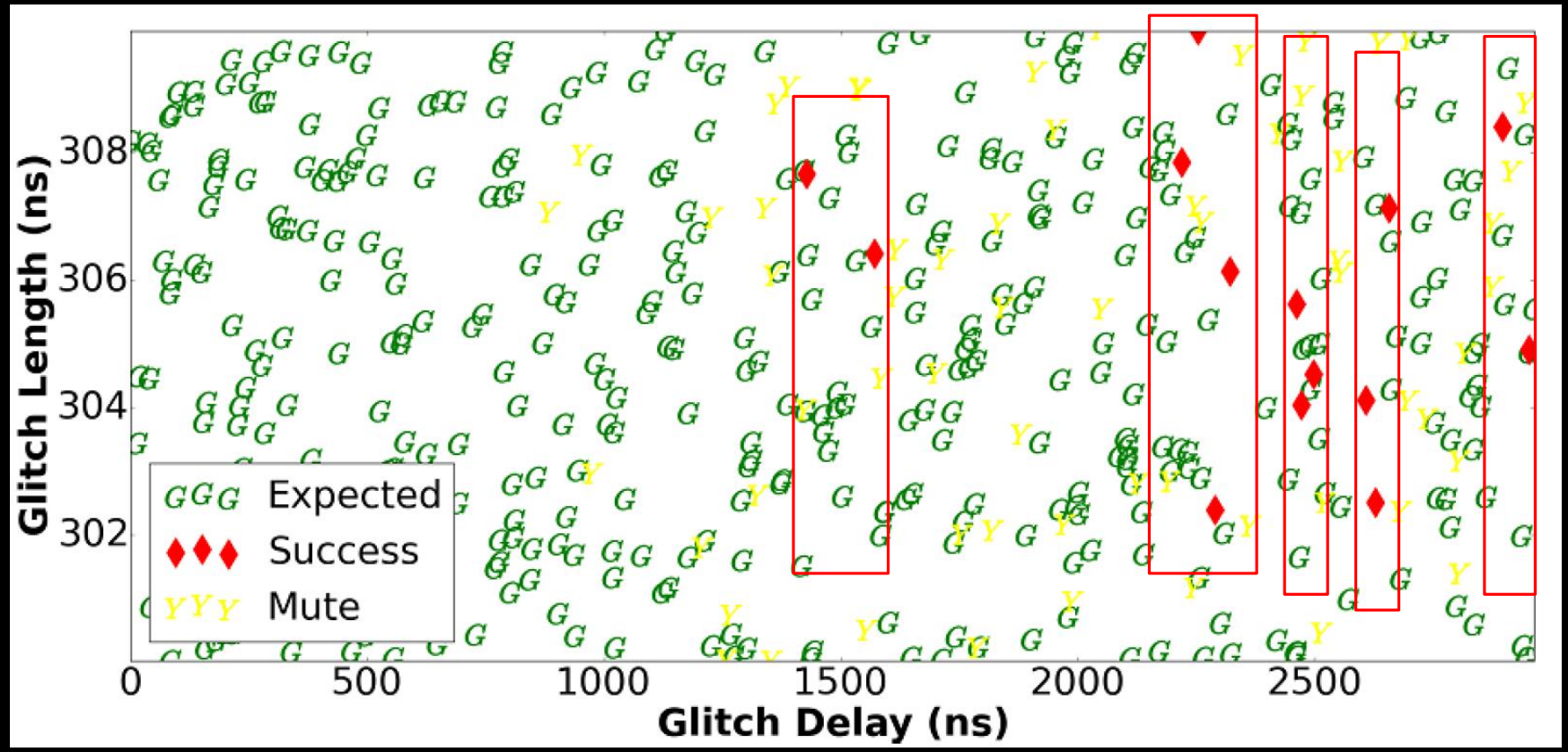
- memcpy():
  - Loads data into registers
  - Control flow loops (depend on transfer size)
    - Sometimes under attacker's control
- Glitch modifies load instruction register
- Control flow hijacked at function exit

A very large number of opportunities.

Glitch precision not required



Multiple data transfers



Linux kernel PC control with userspace data.

“Time is on my side”

- Large time windows for glitching
- ANY data transfer with controlled data…can be a target
- We can have very loose synchronization with the target
- Precise triggering often not with required

(Almost) triggerless attacks.

Conclusion.

Final considerations

- FI SW based countermeasures are historically based on “instruction skipping”
- Instruction corruption attacks are very interesting for code execution
  - May yield direct PC control
  - Only require data control
  - Any data transfer can be a target
  - May allow for loose target synchronization and easier setups
- Modern strong targets may still be vulnerable to “Instruction corruption” attacks:
  - SW-based countermeasures may be bypassed

rælize

Thank you! Any questions!?

Niek Timmers
niek@raelize.com
@tieknimmers

Cristofaro Mune
cristofaro@raelize.com
@pulsoid