S4: Fault Models for improved attacks and defenses

Cristofaro Mune
(c.mune@pulse-sec.com)
@pulsoid

SILM Summer School, INRIA (2019)

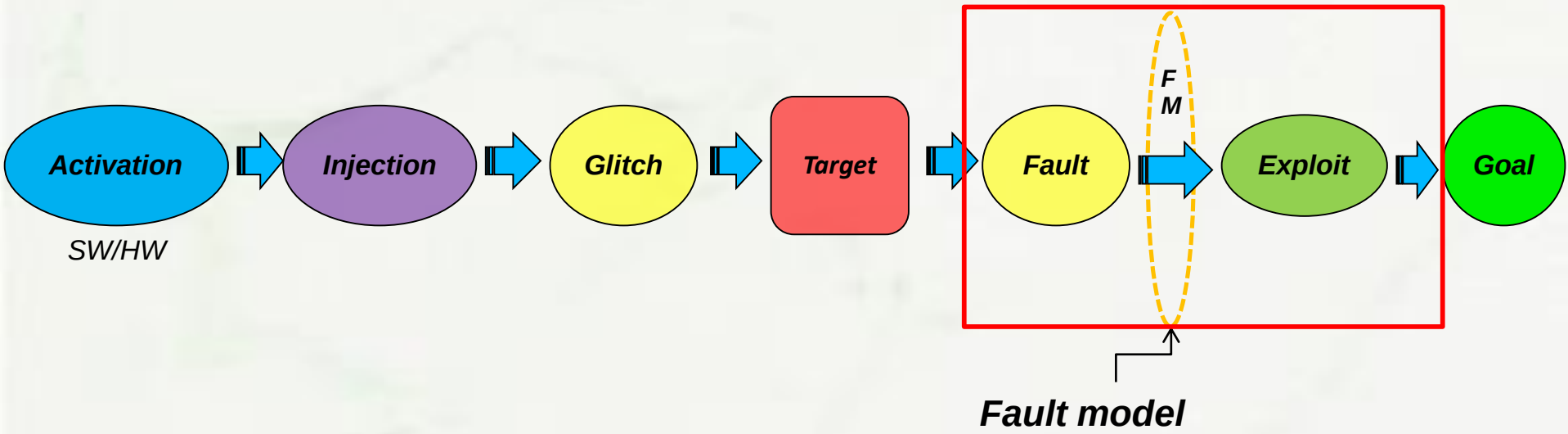
Case study:

Secure Boot

We are going to…

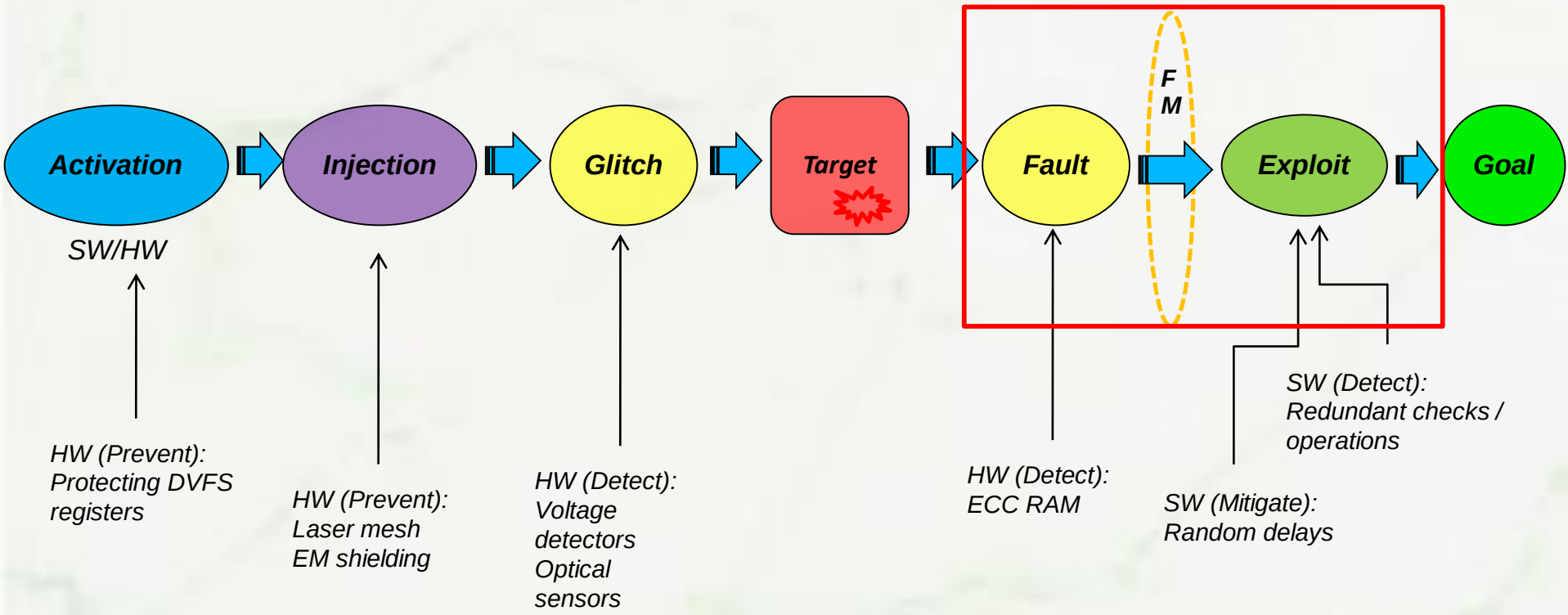
- Evaluate multiple FI Secure Boot attacks:
  - Independently of Injection technique
- Goal: Bypass Secure Secure Boot
  - i.e. a boot stage with a wrong signature is validated and executed
- Each attack using a *different:*
  - *Fault Model*
  - *Exploit*
- For each attack, we evaluate:
  - what can be achieved
  - which countermeasures may (or may not) be effective

Focus



- We focus on *how to use different faults and Fault Models*
- Out of scope: *How the fault if is injected*
  - Irrelevant (for our purposes)

Countermeasures



- *Focus:*
  - Fault Model and Exploit dependent countermeasures

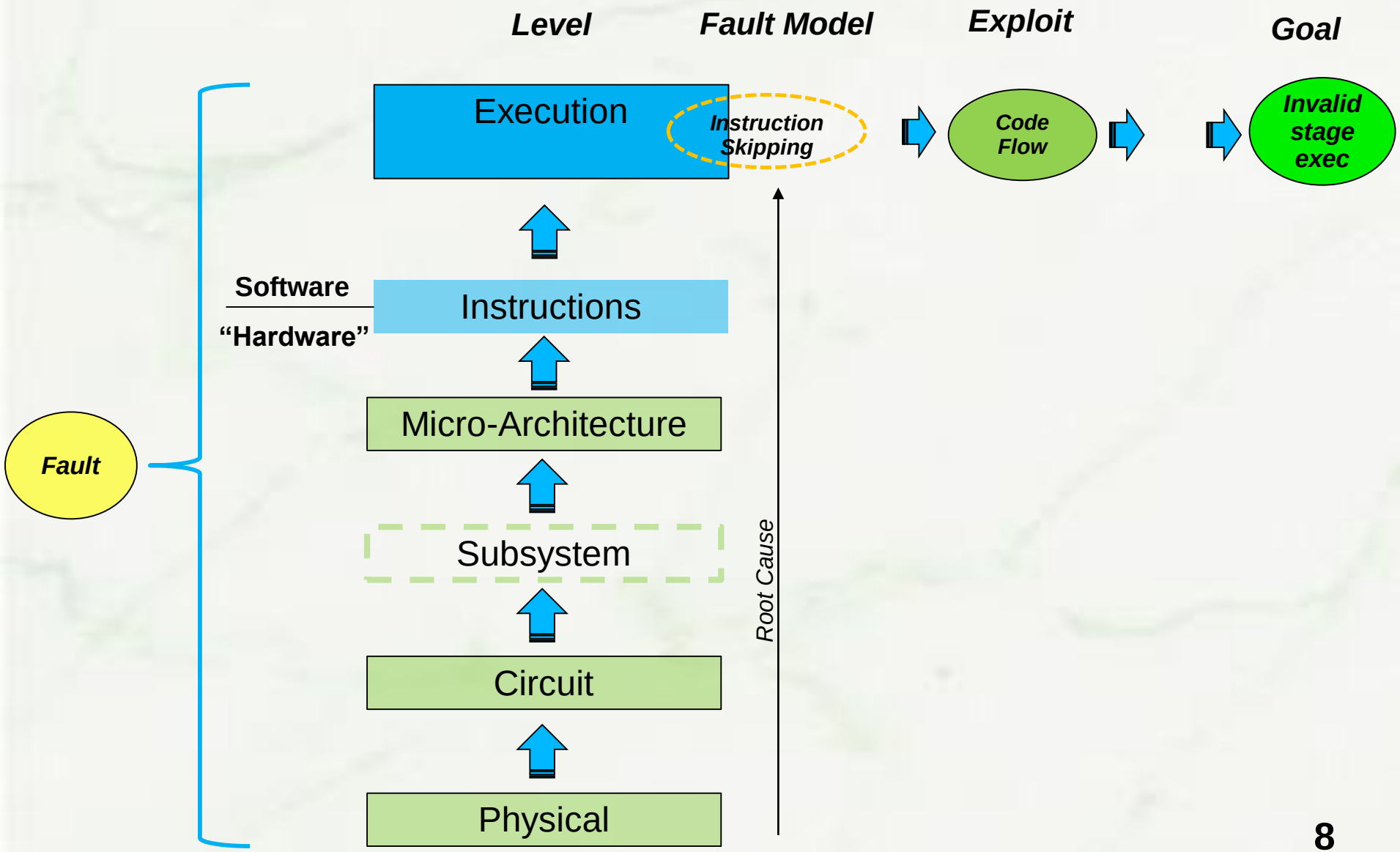
Secure Boot:

“Instruction Skipping”

Textbook attack

- Already covered by Niek!
  - *see session 2*
- Fault Model: *“You are able to selectively skip instructions”*
  - Located at “Execution” Level
- Exploit:
  1. *“Target conditionals”*
  2. *“Affect Code Flow”*
  3. *Wrong decision on boot stage validation*

Instruction skipping

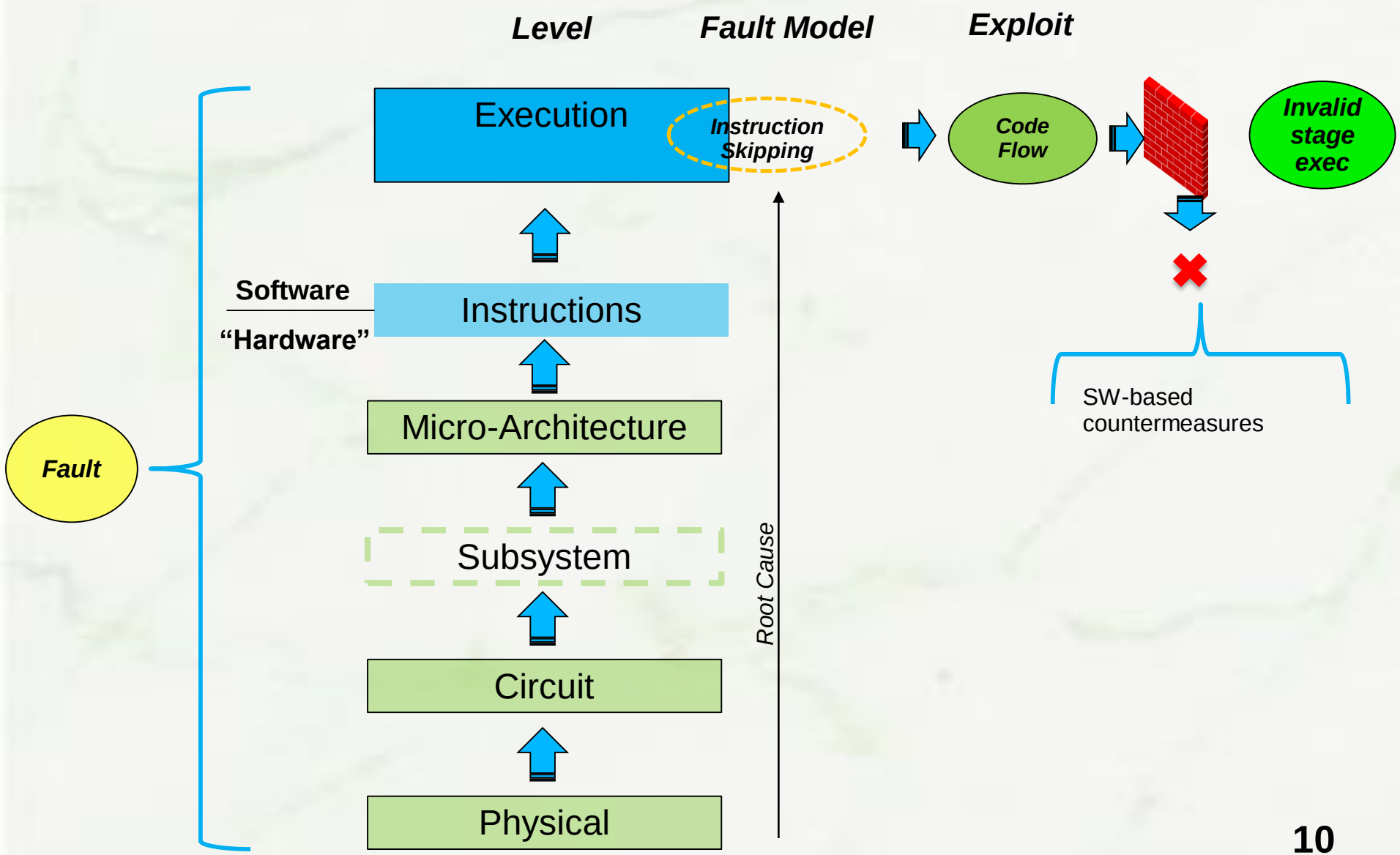


Relevant countermeasures

- SW-based countermeasures:
  - Duplicate checks on “targeted conditionals”
  - Introduce random delays for more difficult targeting
  - Code flow checks
  - …
- Notes:
  - Applied **after exploit phase:** Detection and mitigation
  - Fully exploit dependent:
    - Assume *which* piece of code is targeted
  - Local:
    - Every targeted piece of code needs to be protected

Fully applicable

Instruction skipping: Countermeasures



Question:

What if boot stages are encrypted?

Encrypted Secure Boot

```
memcpy(I_SRAM, I_FLASH, I_SIZE);                // 1. Copy image
decrypt(SYM_KEY, I_SRAM, I_SIZE);               // NEW: Decrypt image
memcpy(S_SRAM, S_FLASH, S_SIZE);                // 2. Copy signature

if (*(OTP_SHADOW) >> 17 & 0x1) {                // 3. Check if enabled
    if(SHA256(I_SRAM, I_SIZE, I_HASH)) {        // 4. Calculate hash
        while(1);
    }

    if(verify(PUBKEY, S_SRAM, I_HASH)) {        // 5. Glitch here!
        while(1);
    }
}

jump();                                         // 6. Jump to next image
```

The image is decrypted after it is copied and before it is verified!

The missing key…

- Encryption key needed for creating a malicious image
- Cannot be obtained via FI…
  - ***With the instruction skipping fault model***

- It is ***commonly believed*** that:
  - FI attacks alone cannot bypass an encrypted Secure Boot
  - Encrypting boot stages is a valid FI countermeasure

That's wrong

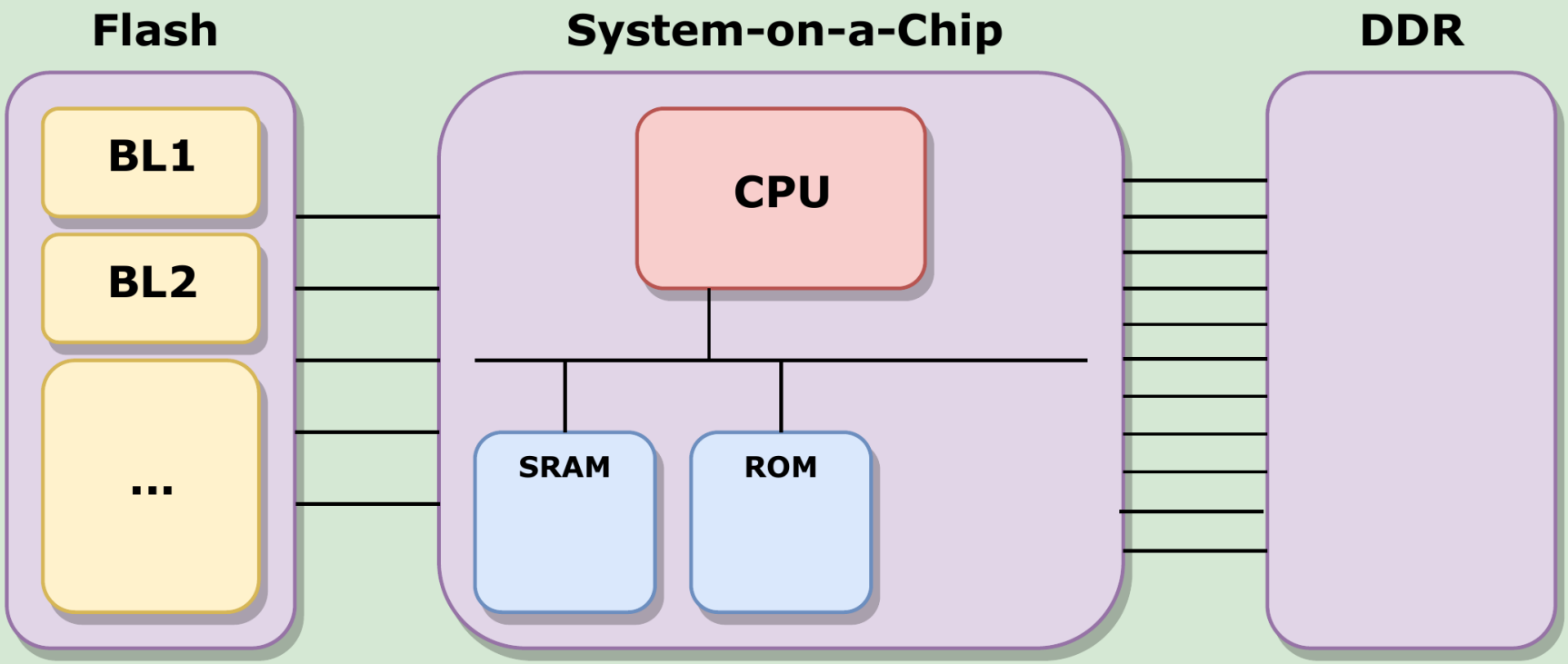
Secure Boot:

“Instruction Corruption”

Attack

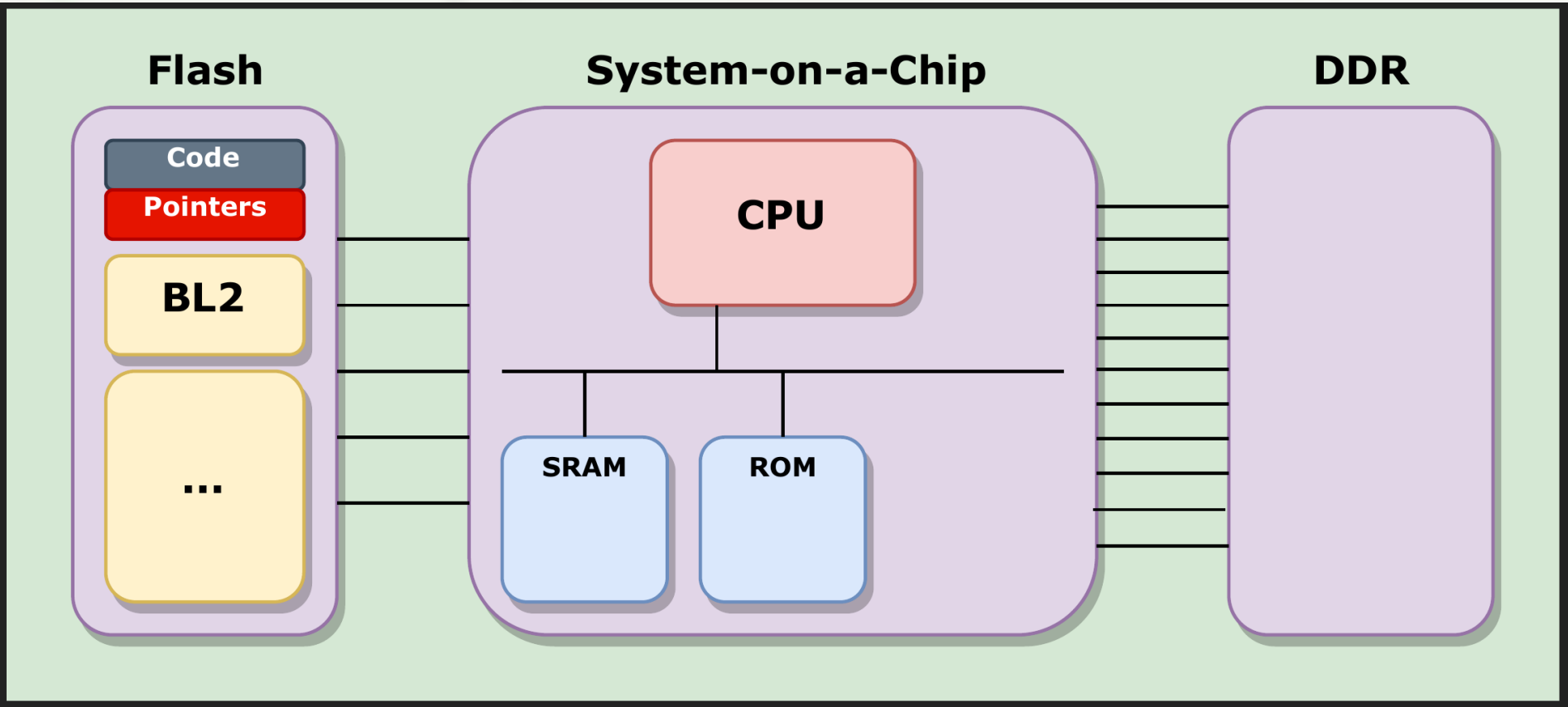
- Fault Model: *“Faults can modify instructions”*
  - (Not only prevent their execution)
  - Instruction Level Fault Model
- Instructions can be mutated:
  - The “where” may be relevant (in some cases)
    - E.g. instructions are fetched encrypted/integrity checked
- Exploit (ARM32 for simplicity):
  1. *“Destination register can be changed during memory transfer”*
  2. *PC can be populated with arbitrary data*
  3. *PC jumps immediately to payload*

Bypassing Encrypted Secure Boot 1/4



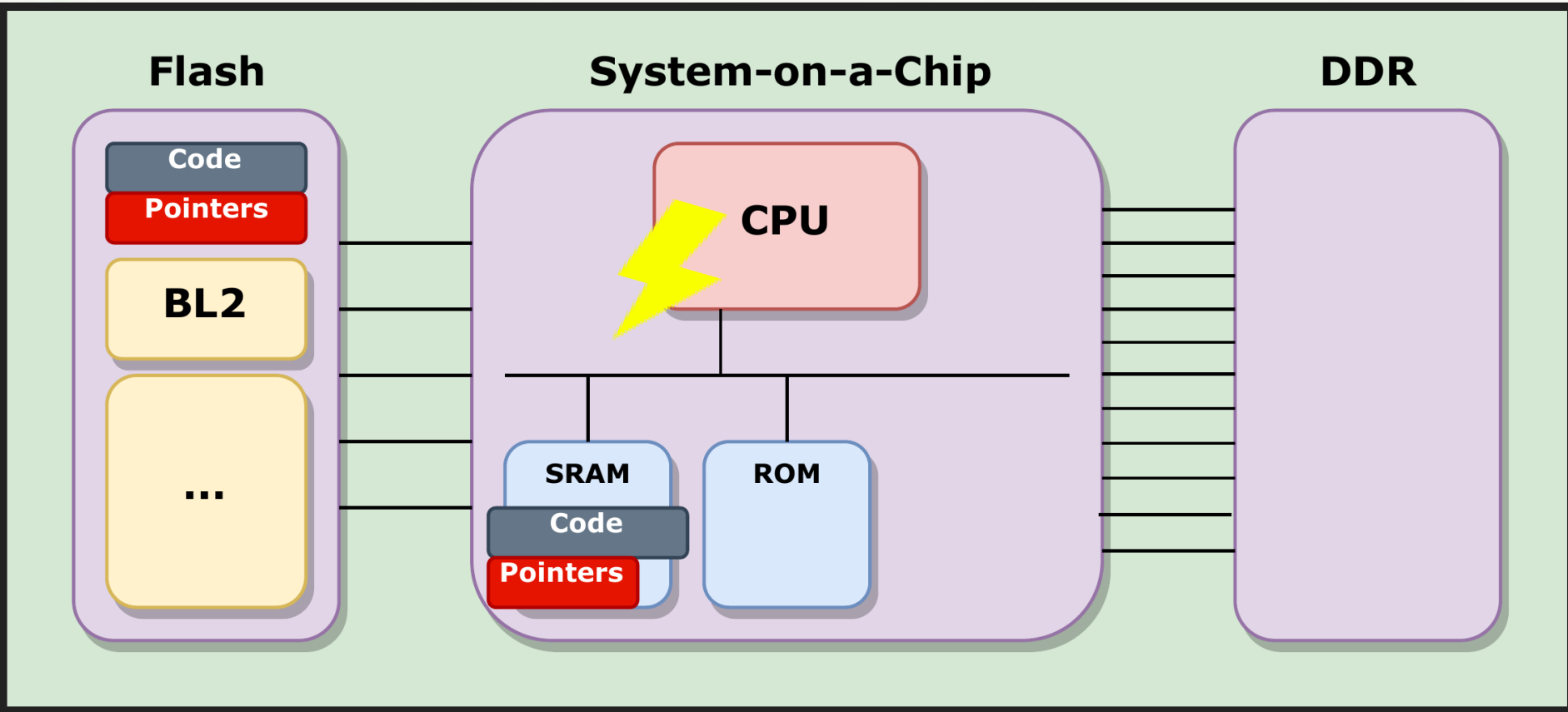
Device is turned off.

Bypassing Encrypted Secure Boot 2/4



Replace encrypted BL1 with plain text code and pointers to SRAM.

Bypassing Encrypted Secure Boot 3/4



Glitch is injected after code copy and while pointers are being copied.

Bypassing Encrypted Secure Boot 4/4

```
memcpy(I_SRAM, I_FLASH, I_SIZE);        // Glitch here!
decrypt(SYM_KEY, I_SRAM, I_SIZE);       // Before decryption
memcpy(S_SRAM, S_FLASH, S_SIZE);        // and

if(SHA256(I_SRAM, I_SIZE, I_HASH)) {    // before
    while(1);
}

if(verify(PUB_KEY, S_SRAM, I_HASH)) {   // verification!
    while(1);
}

jump();                                 // CPU will never reach here
```

Glitch during pointers copy to assign a pointer to the program counter (PC).

Resulting Code execution

```
memcpy(I_SRAM, I_FLASH, I_SIZE);      // Glitch here!
.
.
.
.
.
.
.
.
.
.
.
((void *)())(pointer)();
```

Control flow is hijacked. The decryption and verification of the image is bypassed!

Concretely said…

We turn

ENCRYPTED SECURE BOOT

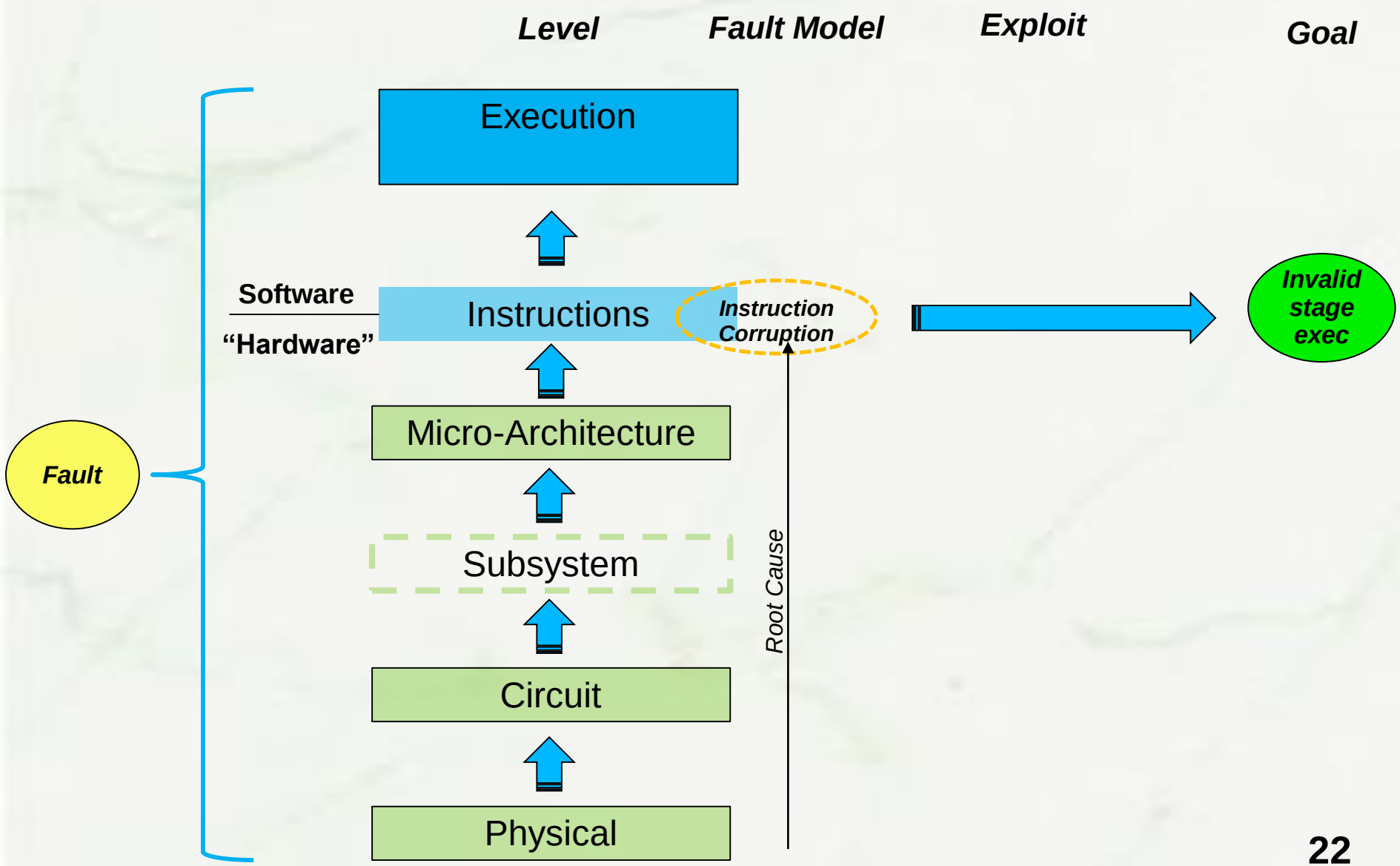
into

PLAINTEXT UNPROTECTED BOOT

using

A SINGLE GLITCH AND NO KEY!

Instruction corruption



New impacts

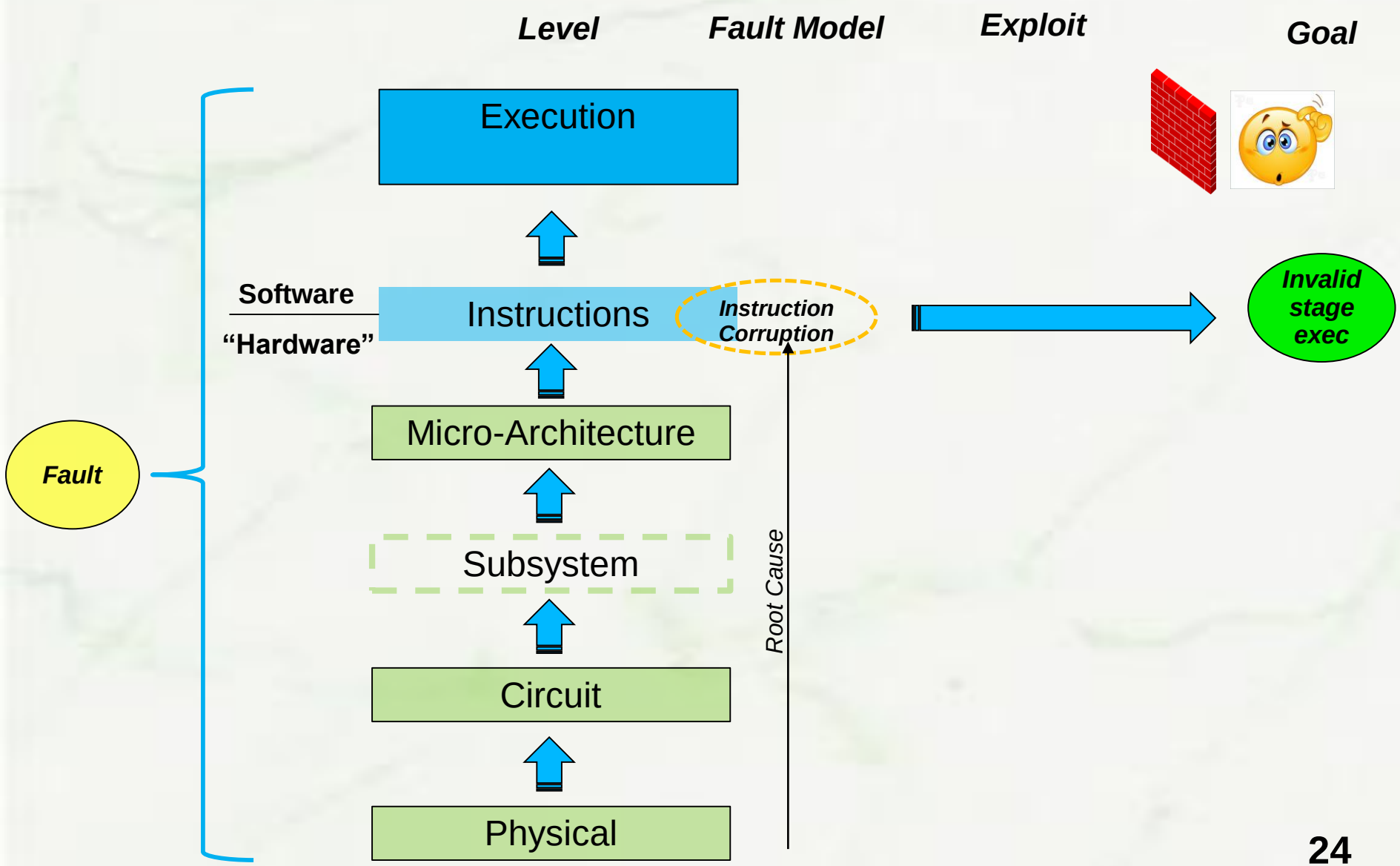
- ***Signature verification not performed***
  - Secure Boot defeated
- ***Decryption not performed***
  - Plaintext code execution

- Code execution achieved in ***verifying context***

- ***ROM-level code execution***

NOT possible with “Instruction Skipping” fault model

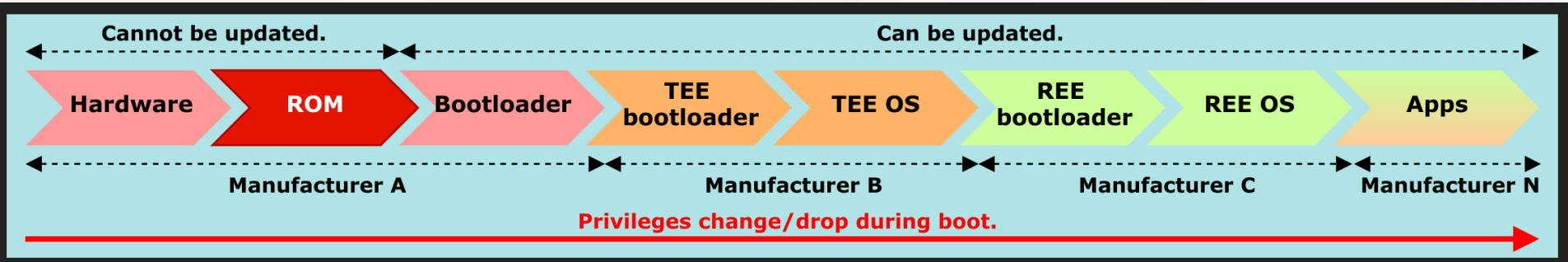
Instruction corruption: SW Countermeasures



Secure Boot:

“OTP Transfer”

OTP and Secure Boot



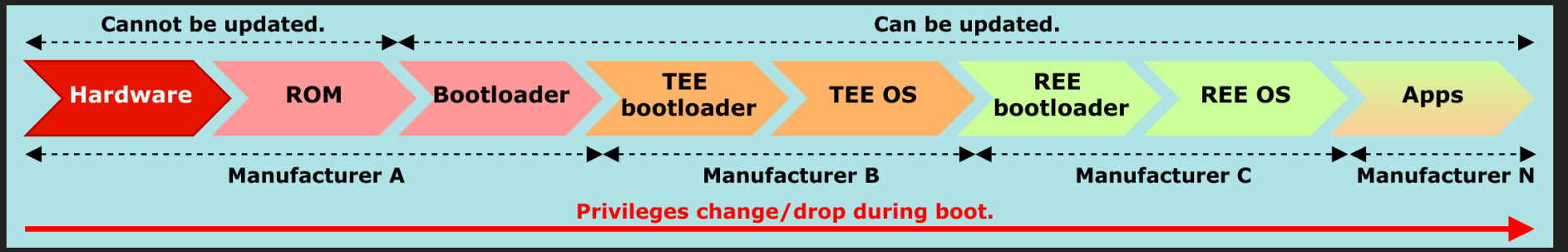
ROM code uses values from OTP for enabling/disabling security features.

Example

```
memcpy(I_SRAM, I_FLASH, I_SIZE);                    // 1. Copy image
memcpy(S_SRAM, S_FLASH, S_SIZE);                    // 2. Copy signature

if (*(OTP_SHADOW) >> 17 & 0x1) {                    // 3. Check if enabled
    if(SHA256(I_SRAM, I_SIZE, I_HASH)) {            // 4. Calculate hash
        while(1);
    }

    if(verify(PUBKEY, S_SRAM, I_HASH)) {            // 5. Verify image
        while(1);
    }
}

jump();                                             // 6. Jump to next image
```

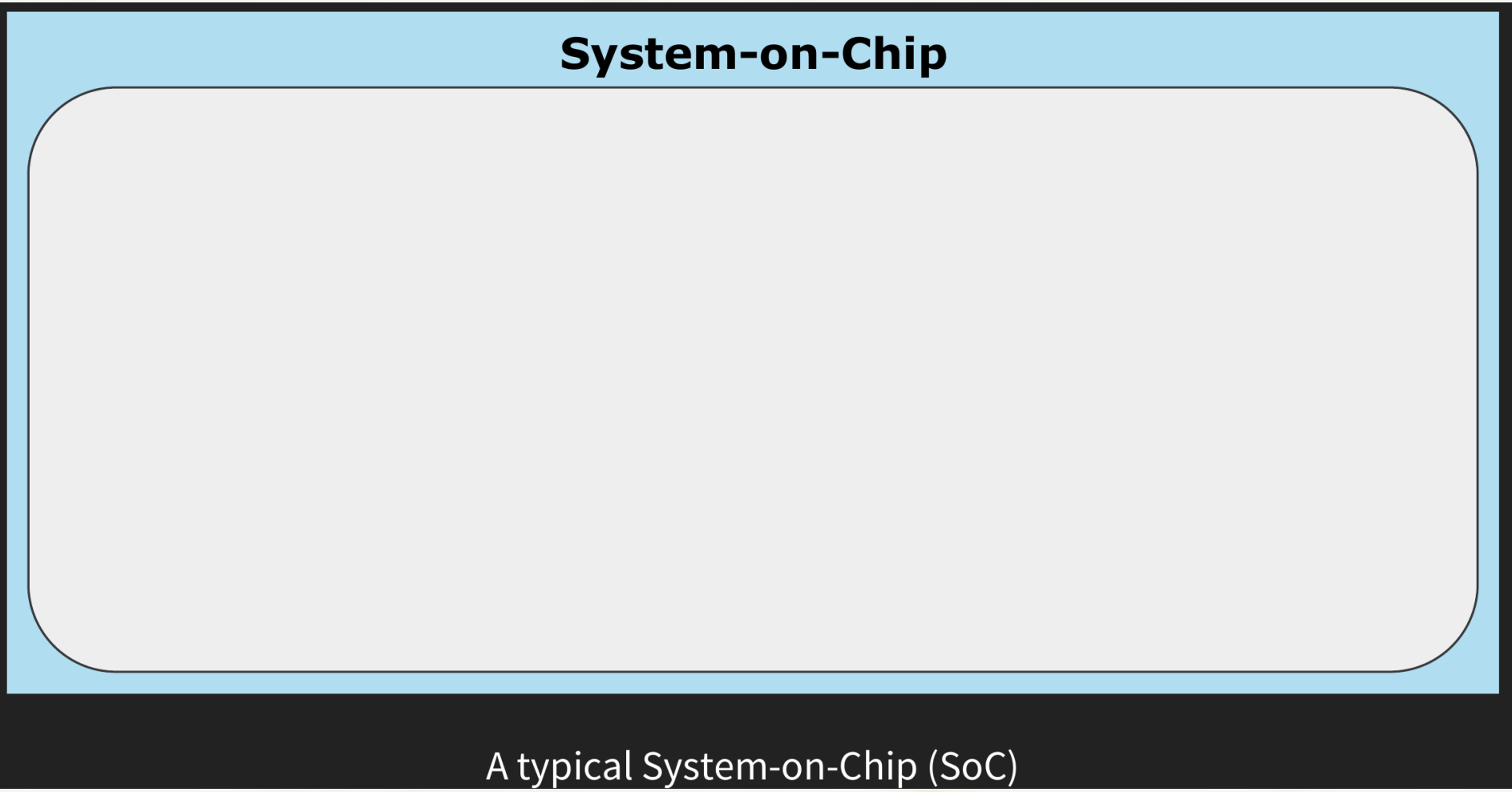
Value stored in shadow registers. Populated by OTP Transfer.

Populating shadow registers



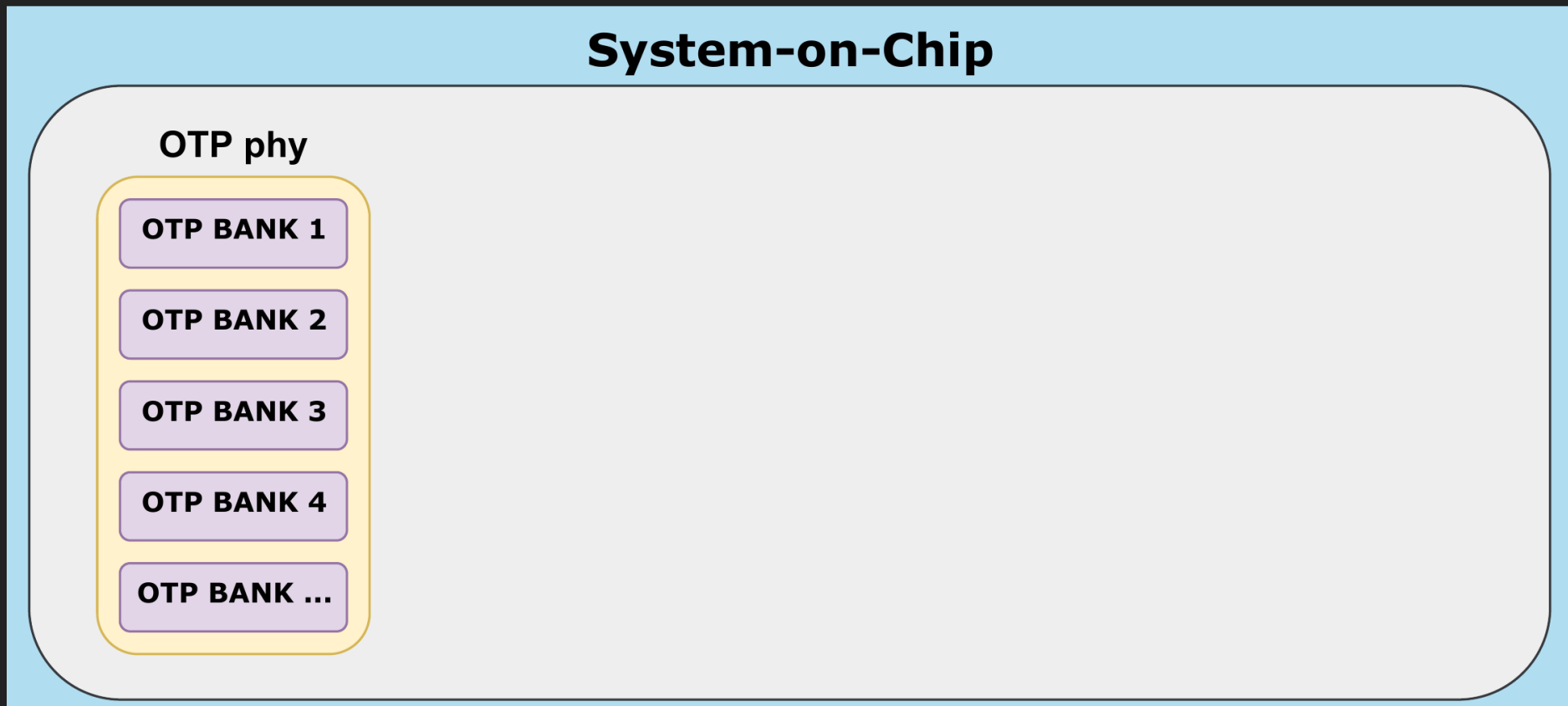
OTP Transfer performed in hardware. BEFORE any ROM code is executed.

OTP Transfer 1/5



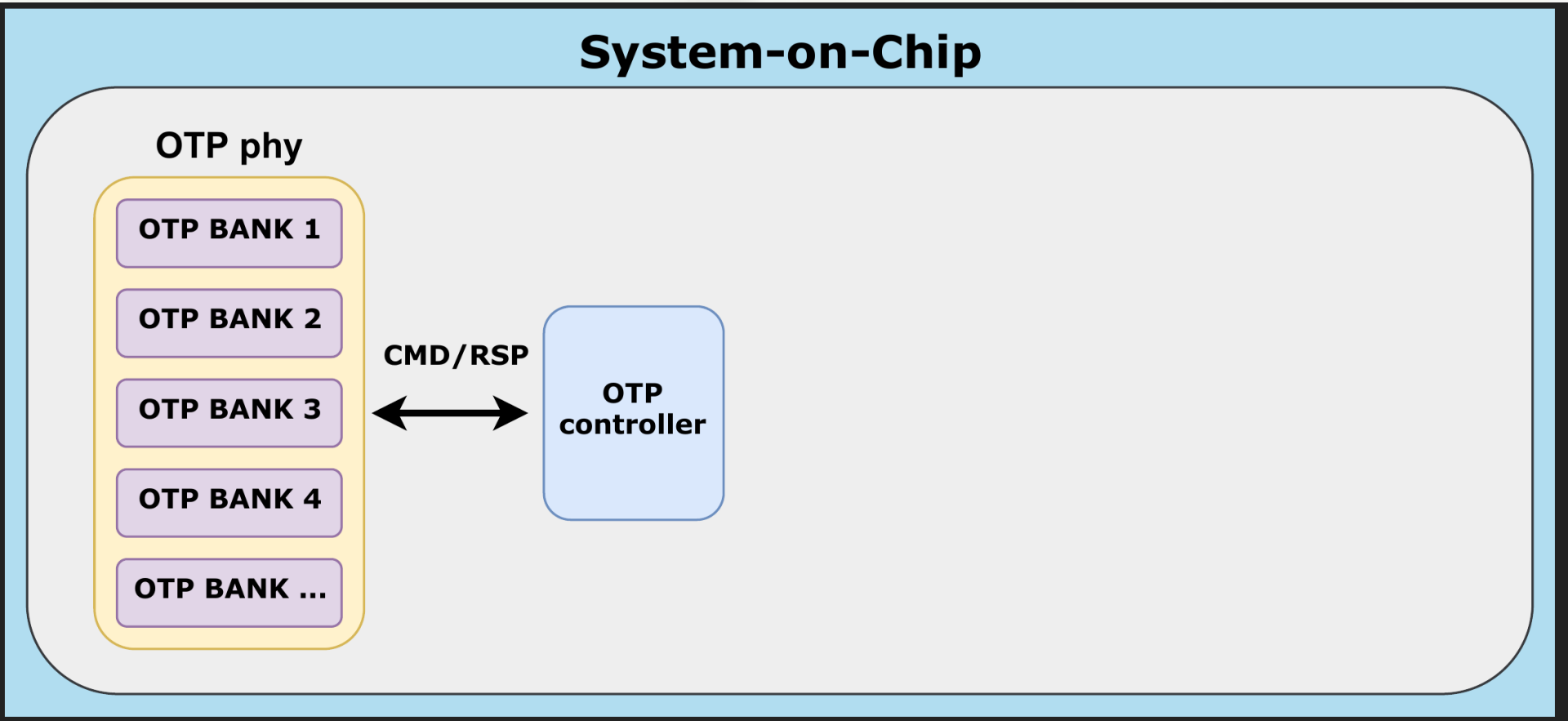
A typical System-on-Chip (SoC)

OTP Transfer 2/5



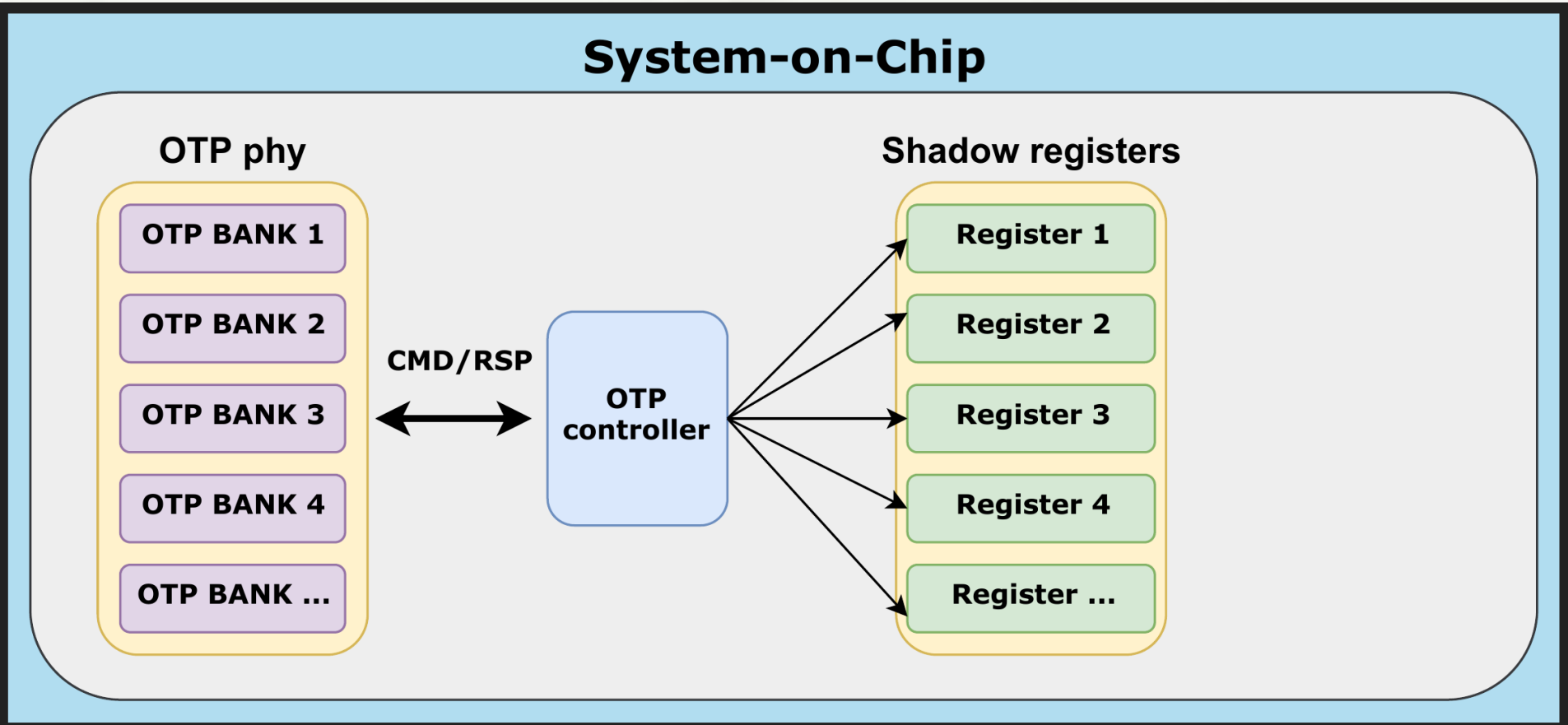
Contains a special OTP hardware block

OTP Transfer 3/5



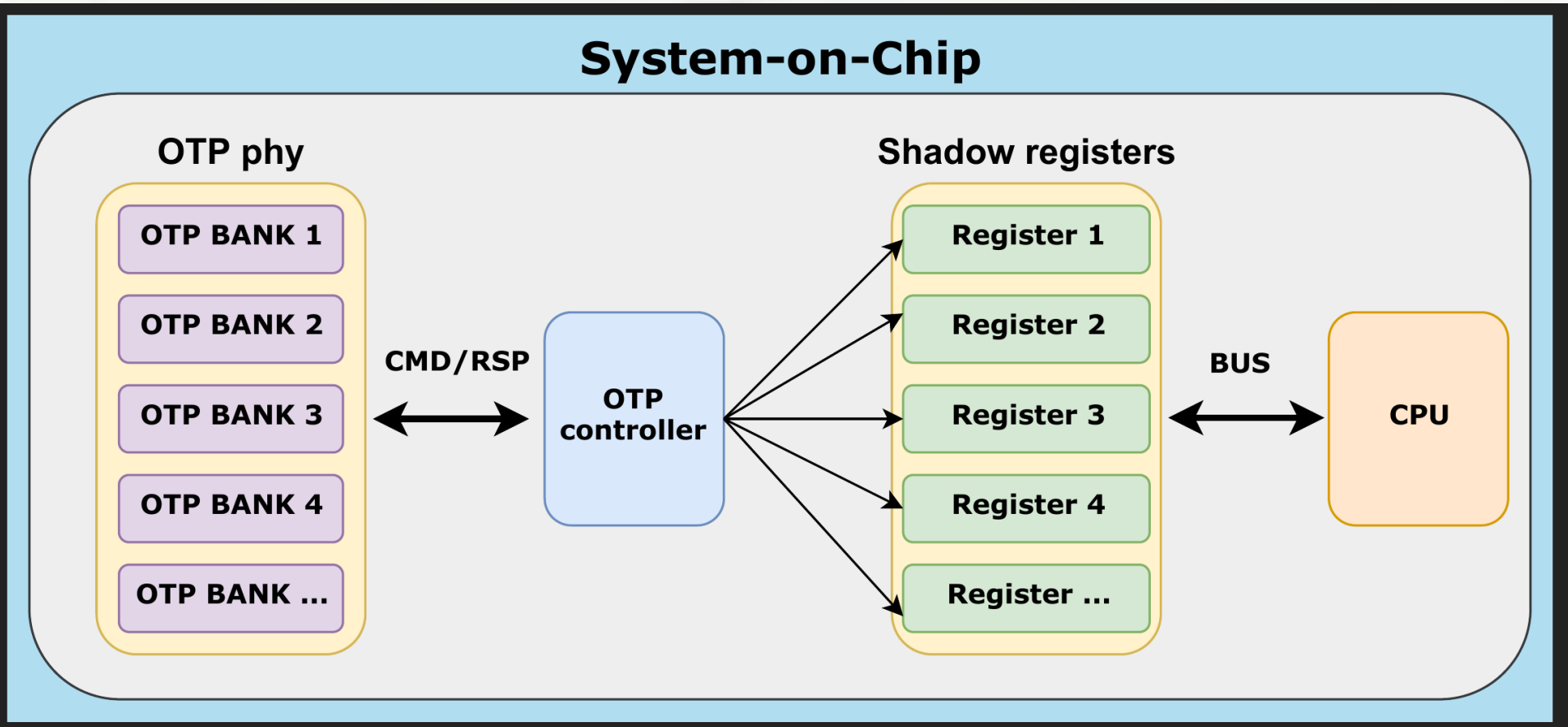
Which is wrapped by a hardware controller

OTP Transfer 4/5



This controller copies the OTP values to dedicated registers after SoC reset

OTP Transfer 5/5

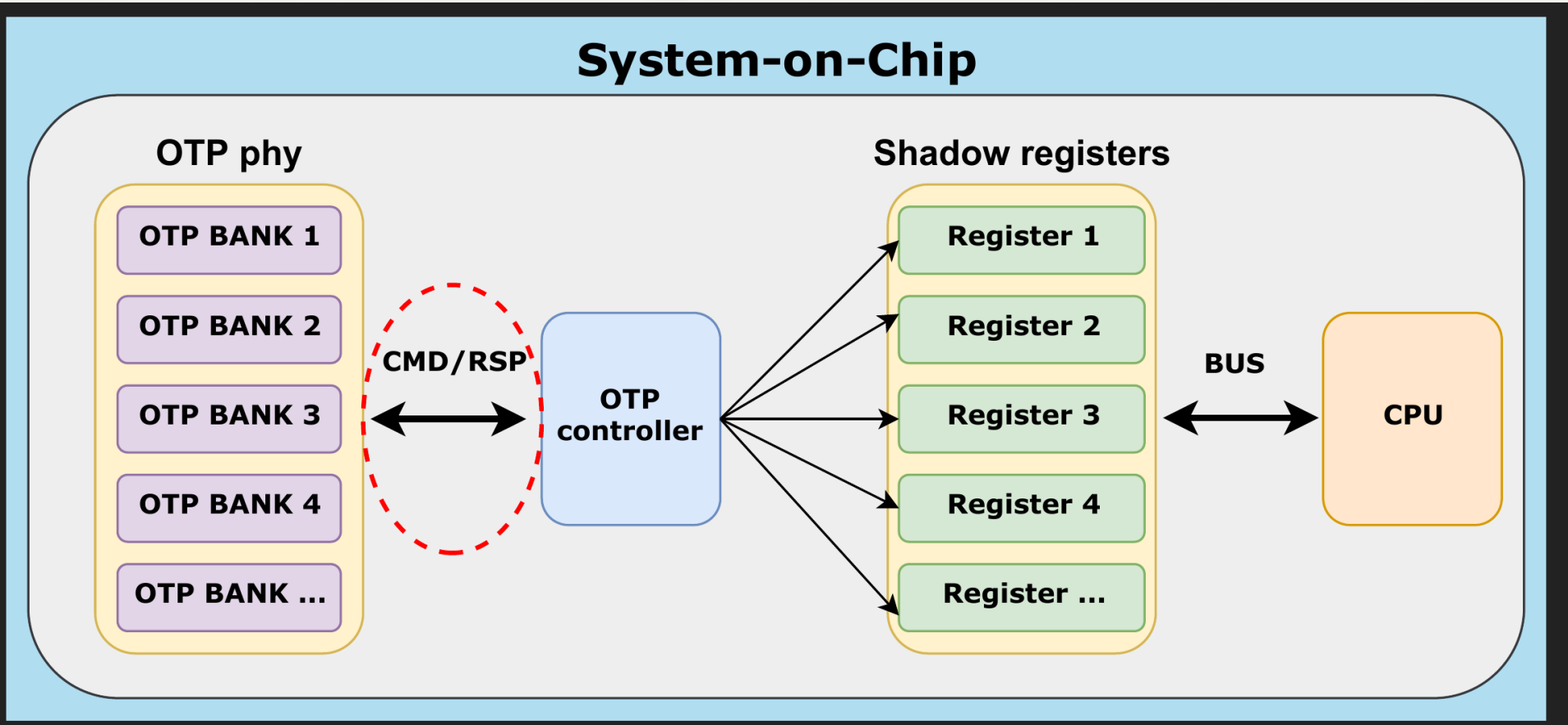


CPU is released from reset. Shadow registers can be read using system bus.

Question:

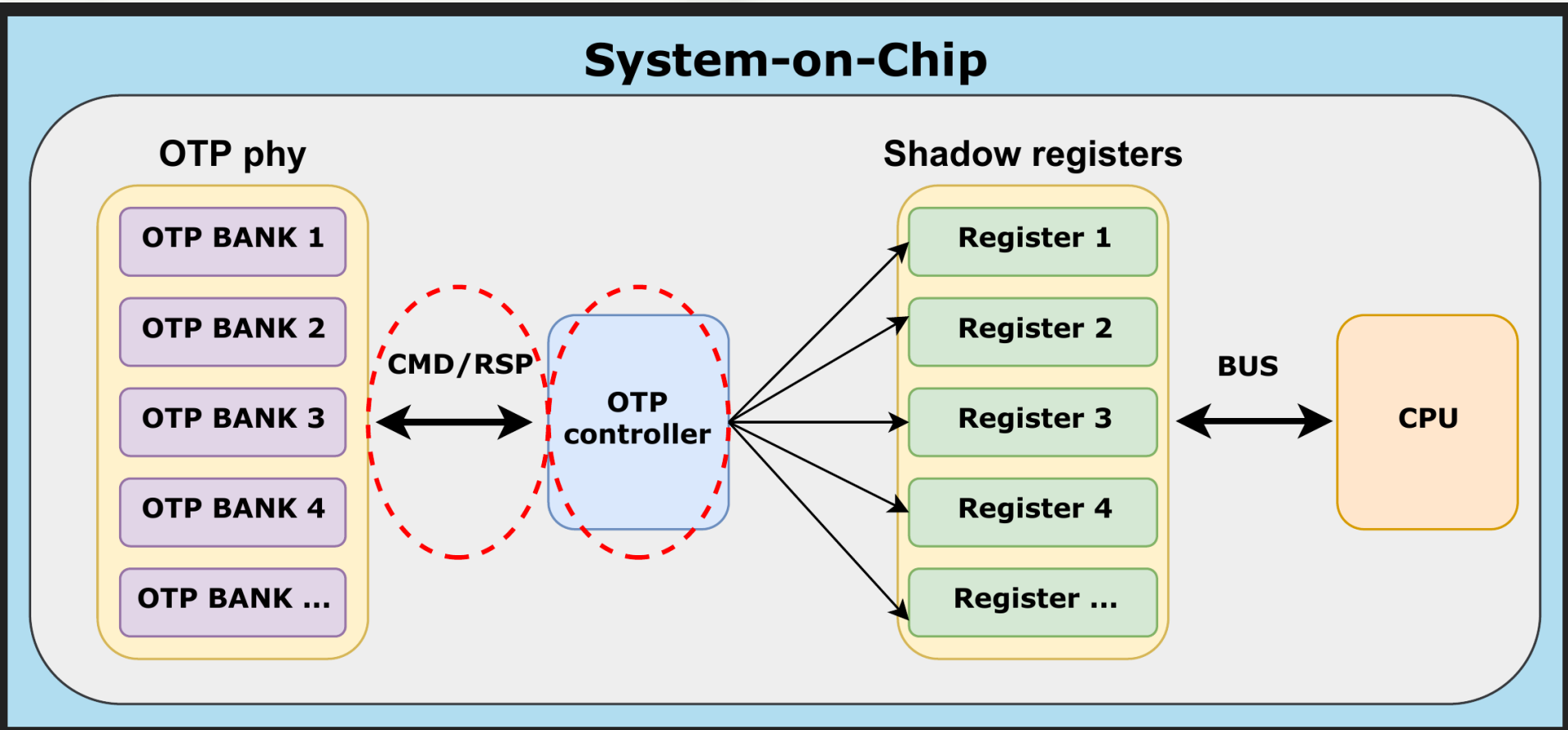
Where can we attack?

ANYWHERE!



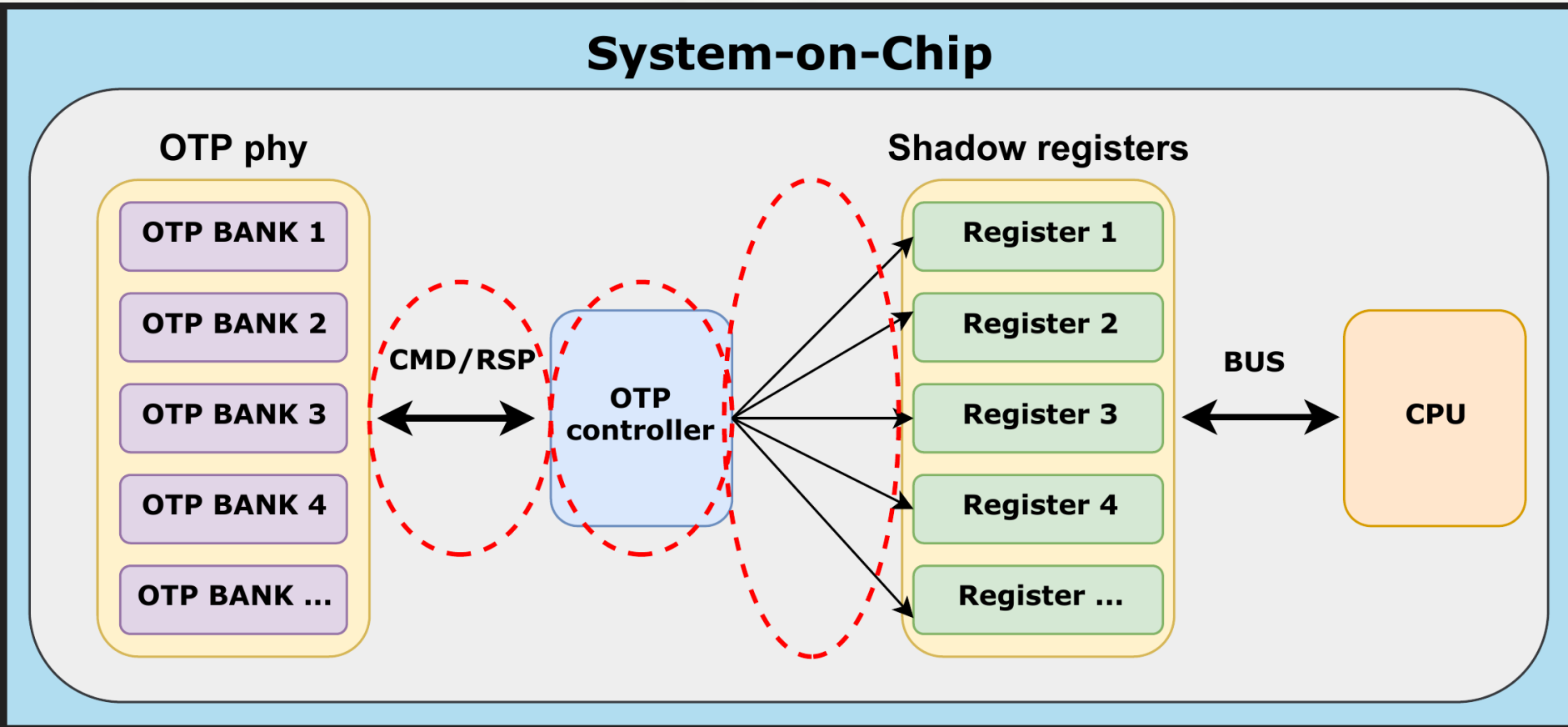
Attack the bus between the OTP PHY and the OTP controller.

ANYWHERE!



Attack the OTP controller directly.

ANYWHERE!

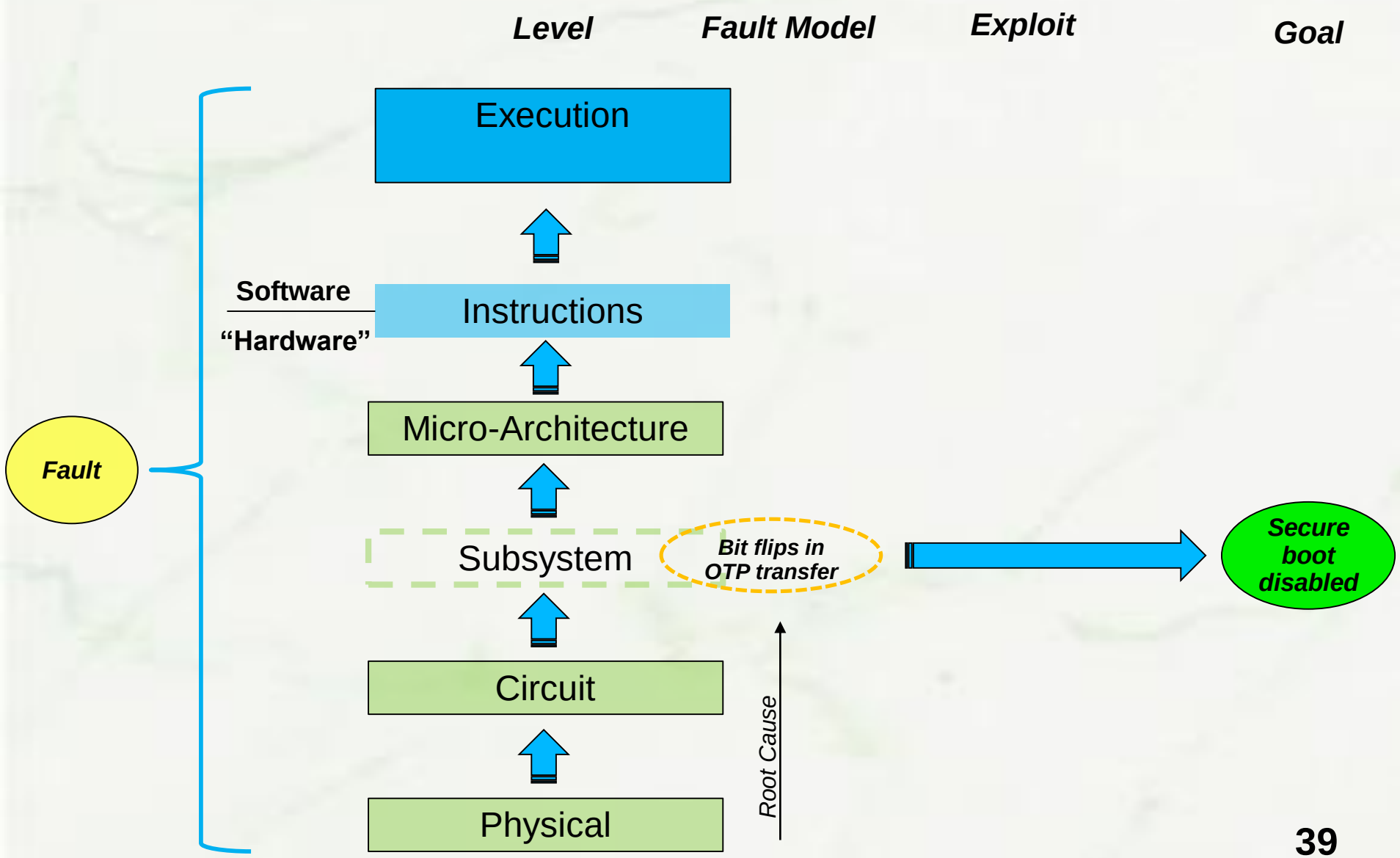


Attack the bus between the OTP controller and the shadow registers.

Attack

- Fault Model: *“Bit flips during HW bus transfers”*
  - Logic Level
- Target:
  - OTP configuration bits
  - While being transferred into shadow registers
    - Before CPU is released from resets
- Exploit:
  1. *OTP configuration bits can be modified*
  2. *Wrong configuration in shadow registers*
  3. *Secure boot code does not execute signature verification*
     - And possibly stage decryption

OTP Transfer

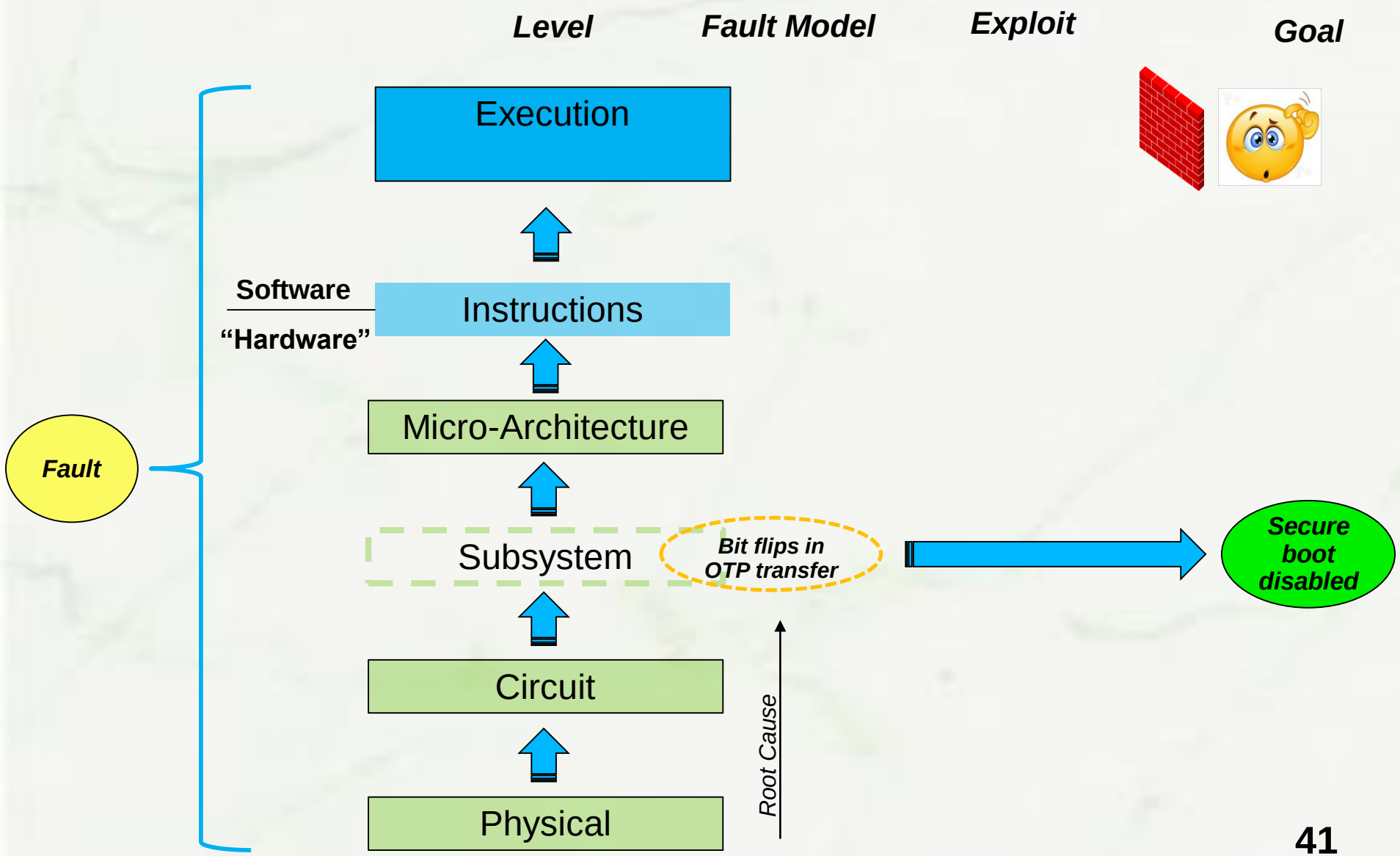


Analysis

- Integrity of SW execution not affected
- CPU subsystem not the target of the attack!
  - *OTP subsystem is targeted*
- ***Secure Boot disabled***
  - *Incorrect configuration in shadow registers → used at boot*
- ***Attack BEFORE any SW execution*** occurs

- *Unlikely with previous Fault Models:*
  - Exploit faults affecting CPU sub-system at runtime
    - Instruction representation/decode, Execution

OTP Transfer: SW Countermeasures



Analysis: Countermeasures

- SW-based countermeasures ineffective:
  - Integrity of SW control flow not affected
  - **No SW is executed at fault injection time**
- CPU-oriented countermeasures ineffective:
  - Exploit leverage faults in a different subsystem (OTP)
- HW-based countermeasures applicable:
  - Only if targeting:
    - The specific injection technique
    - OTP subsystem
      - more general, but localized

Conclusions

Fault Models…

- New fault models enable new attacks with:
  - Different prerequisites
  - Completely new impacts
  - Improved effectiveness
  - Challenging to be defended against

Countermeasures

- **Switching fault model may bypass entire classes of countermeasures**

- *Effective defensive design should not assume:*
  - a specific Fault Model
  - a specific Exploit

Contacts

Cristofaro Mune

Product Security Consultant

c.mune@pulse-sec.com