PEW PEW PEW:

DESIGNING SECURE BOOT SECURELY

Niek Timmers

niek@riscure.com

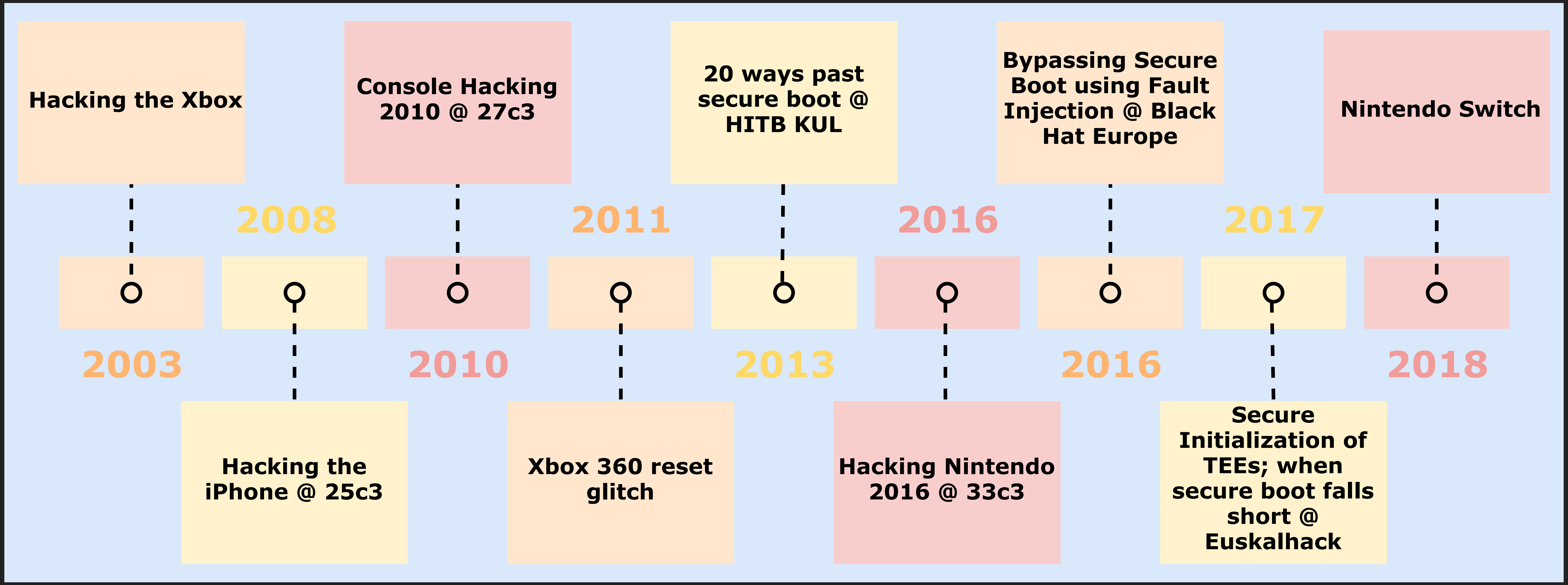
@tieknimmers

Albert Spruyt

albert.spruyt@gmail.com

WHY THIS TALK?

SOME HISTORY...



SECURE BOOT IS STILL OFTEN VULNERABLE...

OUR GOAL

Create a Secure Boot guidance for designers and implementers.

THIS PRESENTATION

Defensive focus

Offensive for context

AGENDA

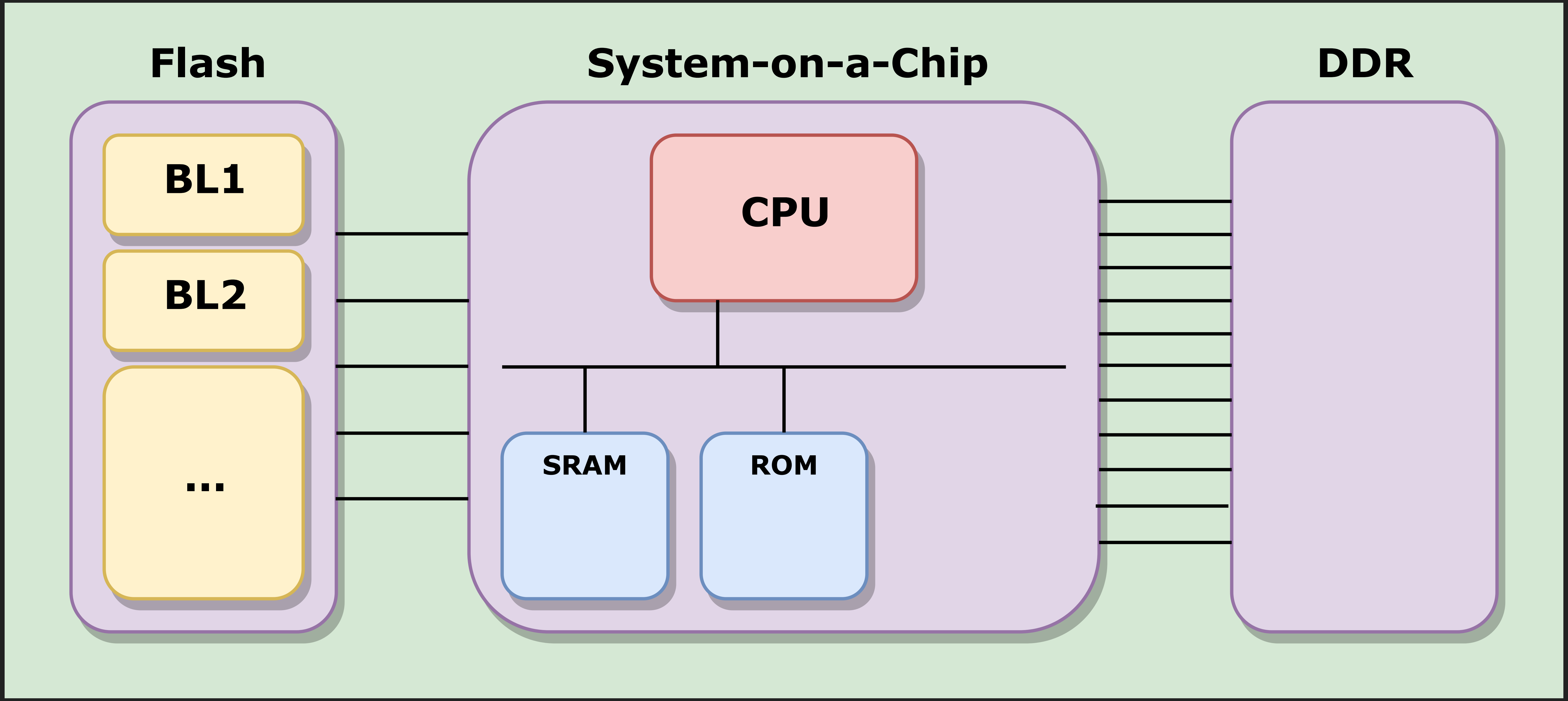
Secure Boot

Fault Injection demo

Designing Secure Boot securely

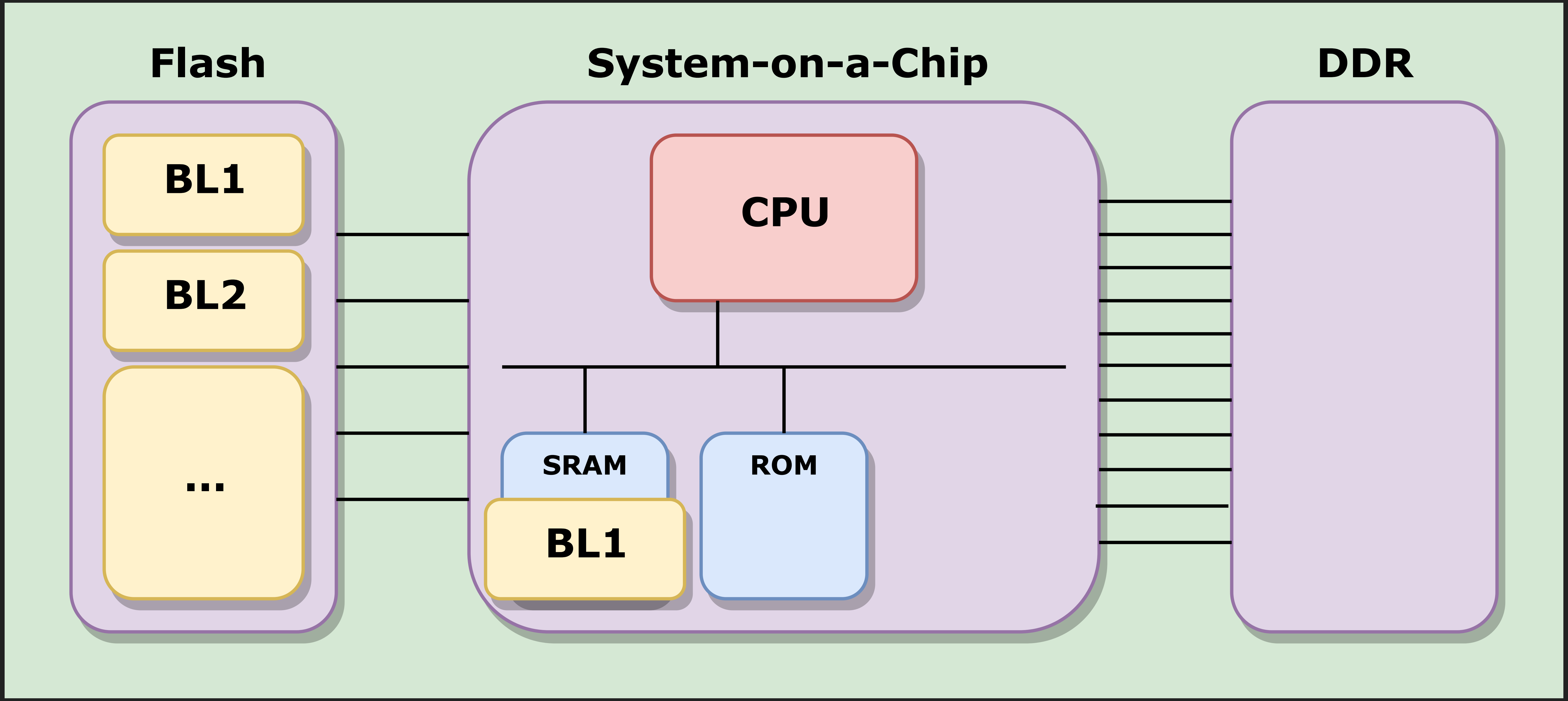
Takeaways

GENERIC DEVICE



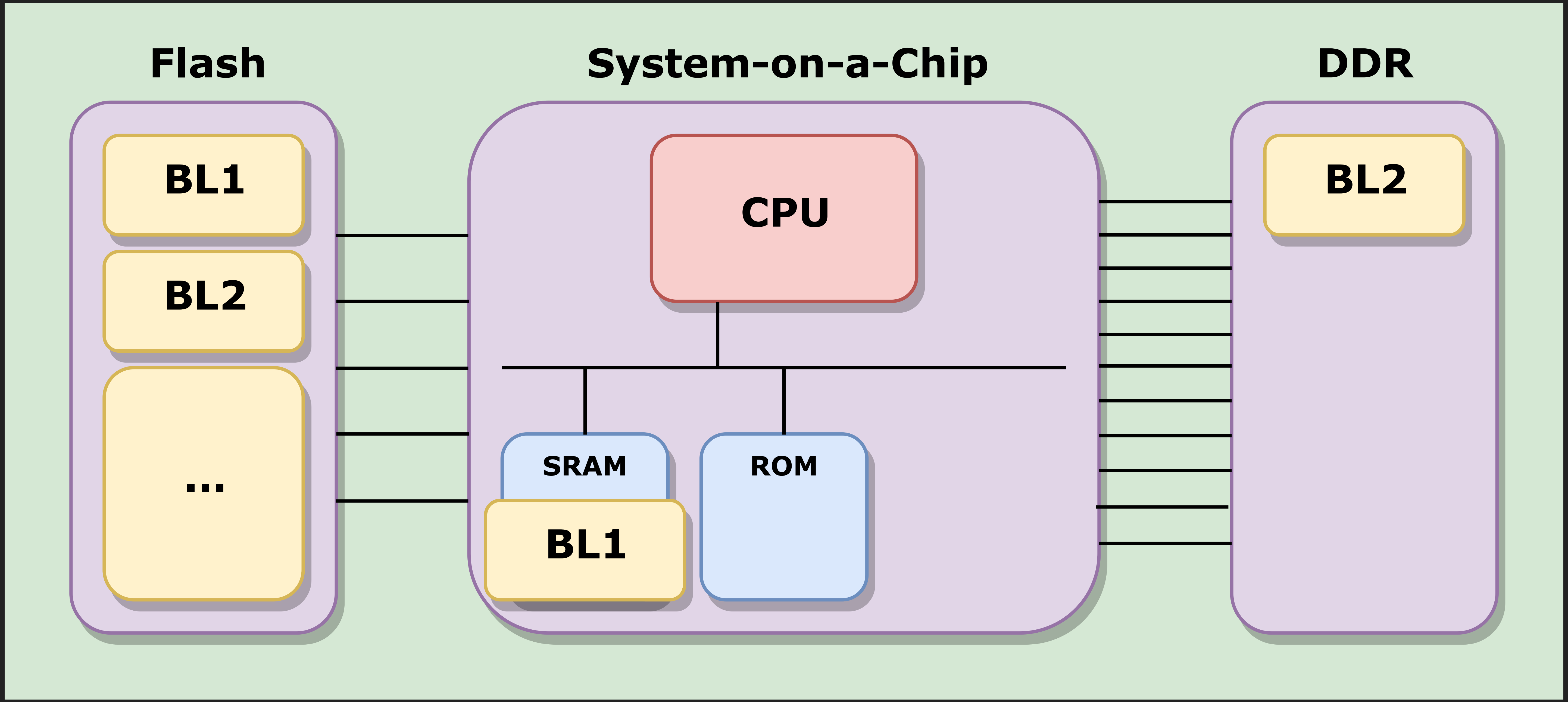
Device is turned off

GENERIC DEVICE



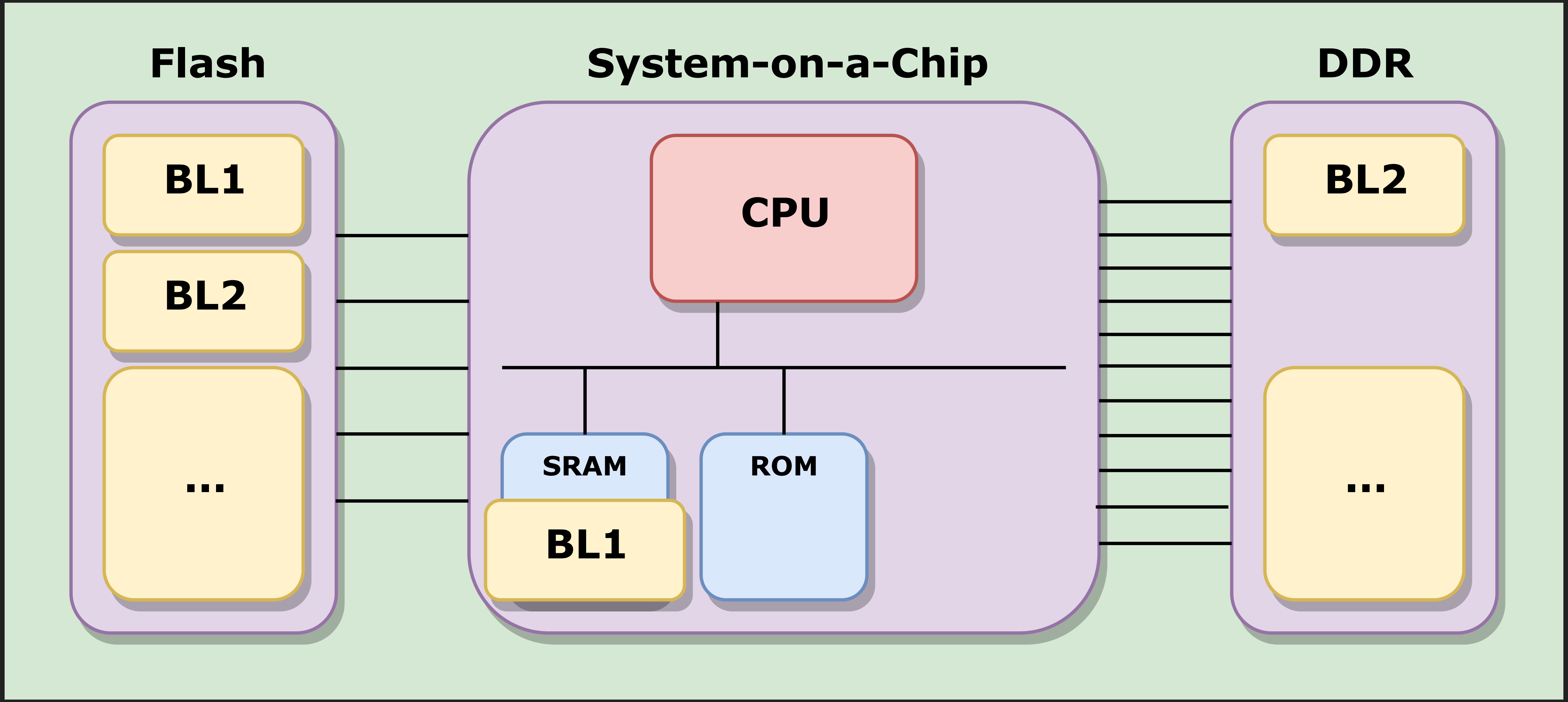
ROM code loads BL1 into internal SRAM

GENERIC DEVICE



BL1 initializes DDR and loads BL2 into DDR

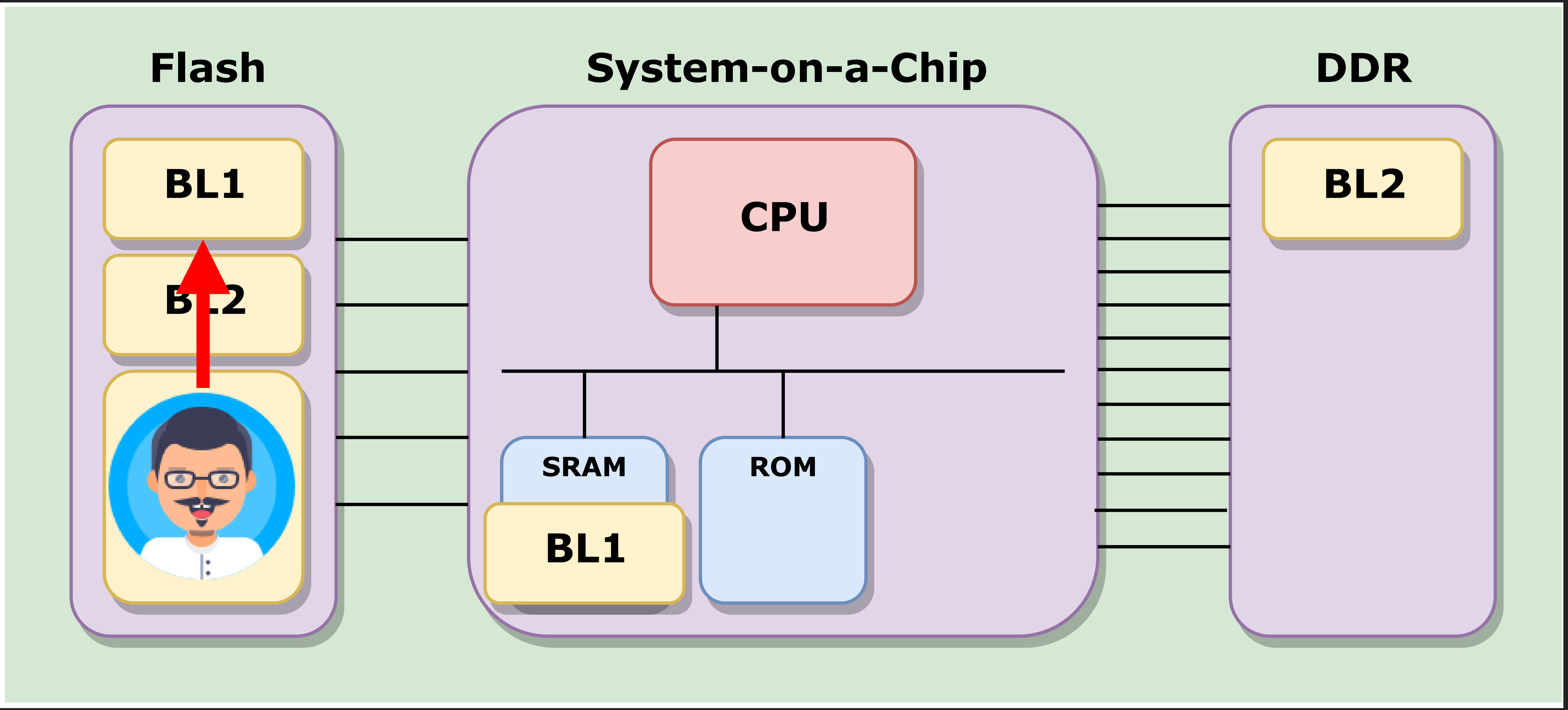
GENERIC DEVICE



Afterwards more is loaded and executed...

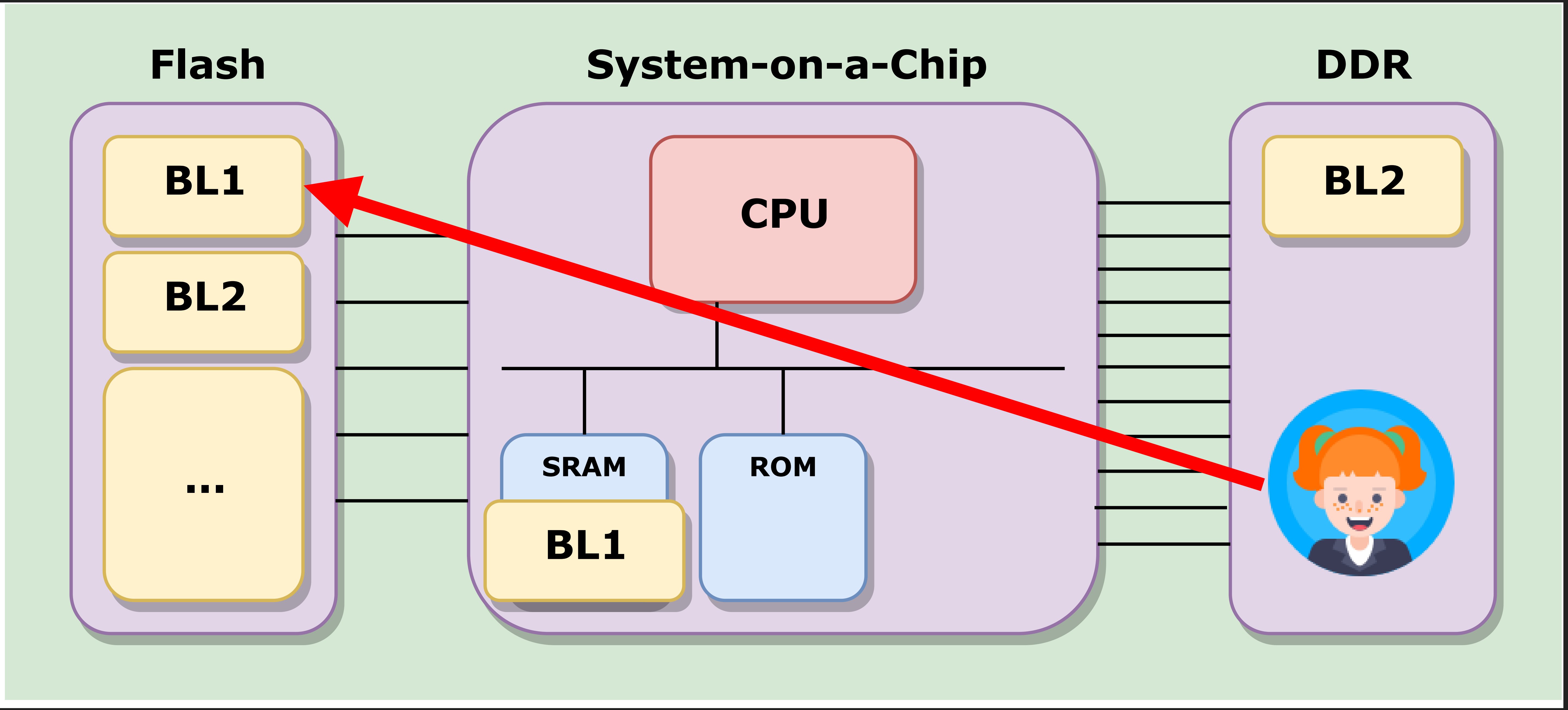
TWO MAJOR THREATS...

ATTACKERS



Attacker 1: hardware hacker modifies flash

ATTACKERS



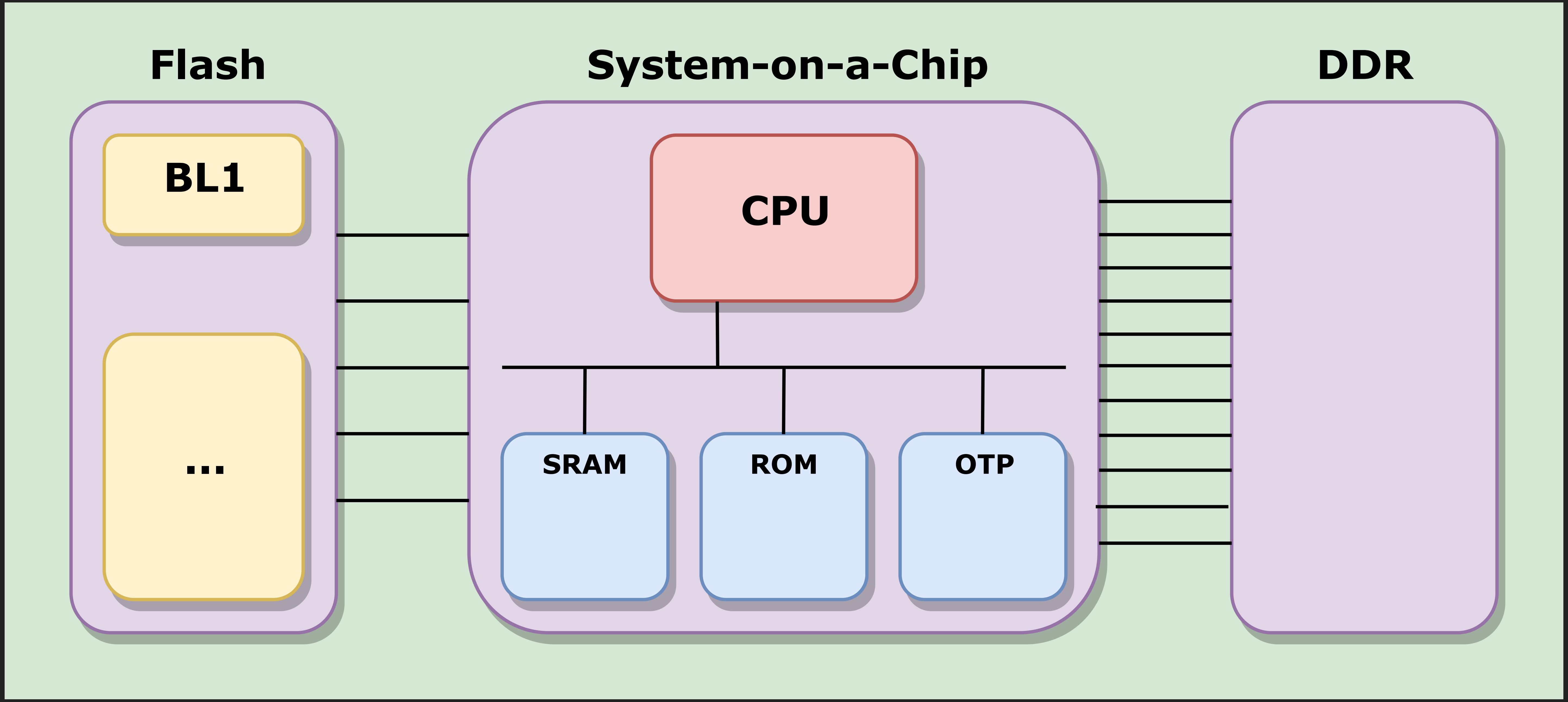
Attacker 2: (remote) software hacker modifies flash

THEREFORE WE NEED SECURE BOOT

SECURE BOOT

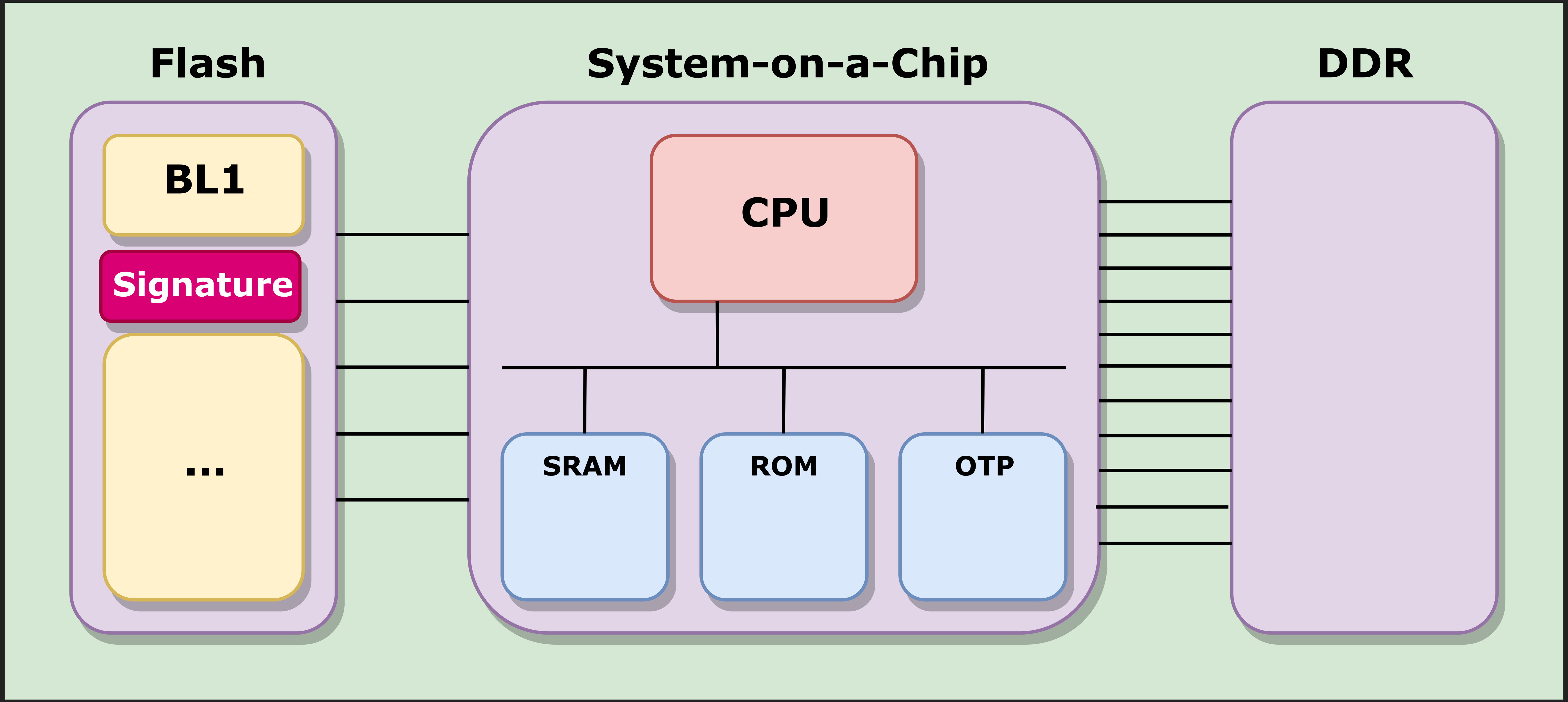
- Root of trust embedded in hardware
  - i.e. immutable code and data (e.g. ROM, OTP)
- Authentication of all code/data
- (Optional): Decryption of all images

SECURE BOOT



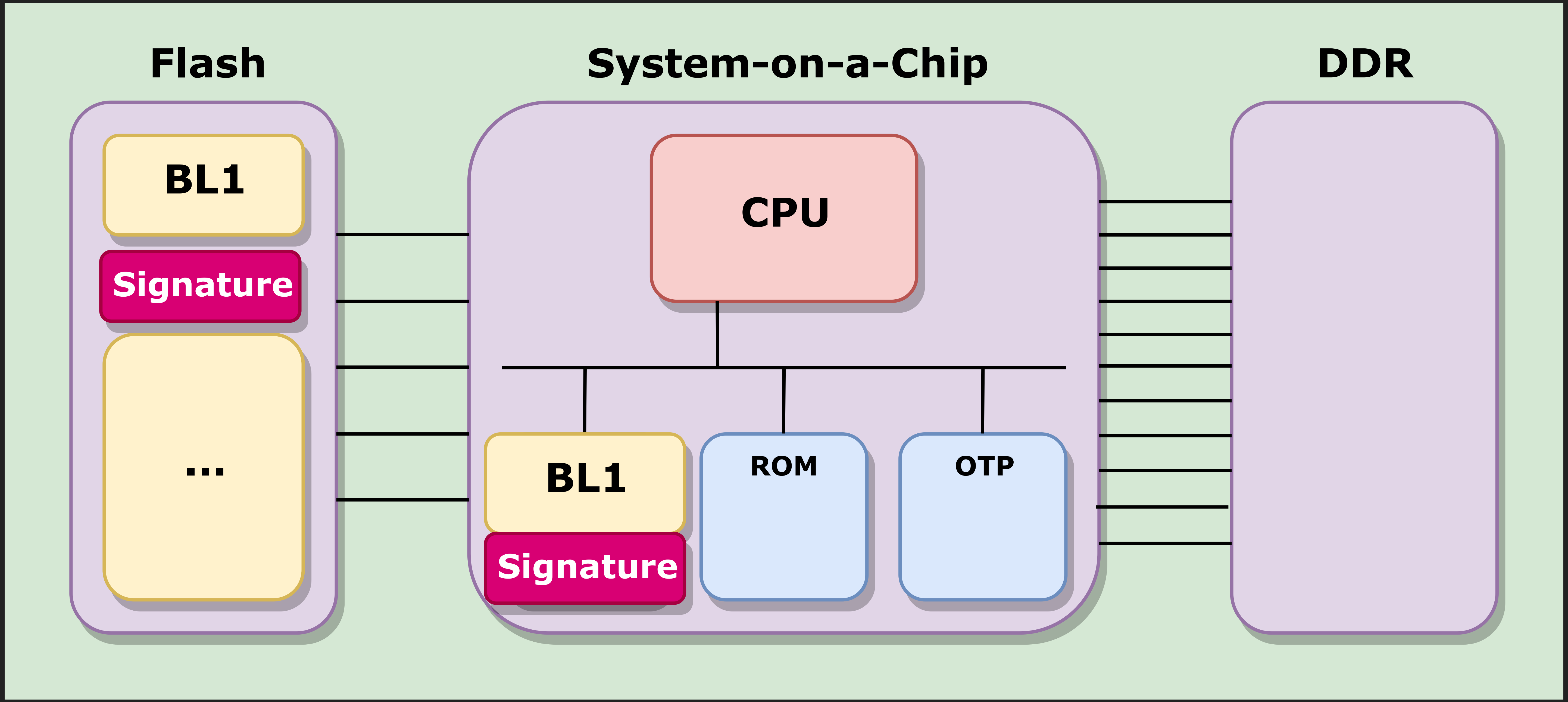
Device is turned off

SECURE BOOT



Next to BL1 a signature is stored

SECURE BOOT



ROM verifies integrity of BL1

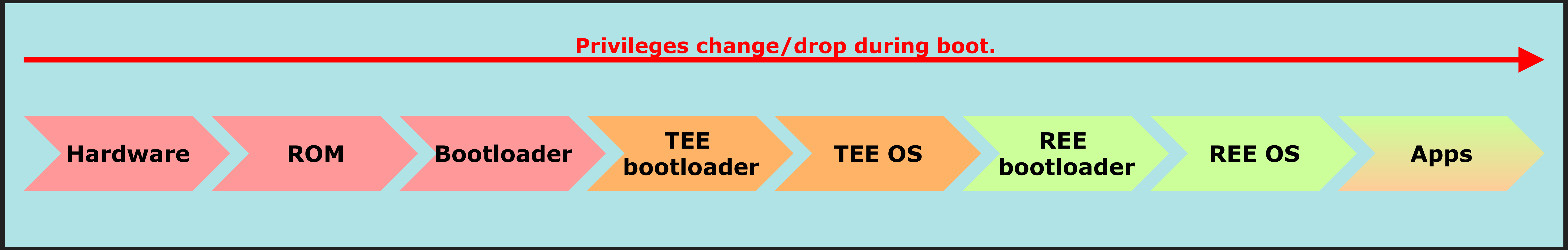
MITIGATING THREATS

- Modifying code/data in flash
- Creating a persistent foothold
- Escalating privileges (e.g. REE to TEE)
  - Access to keys, code and crypto engines
- Bypassing secure update mechanisms

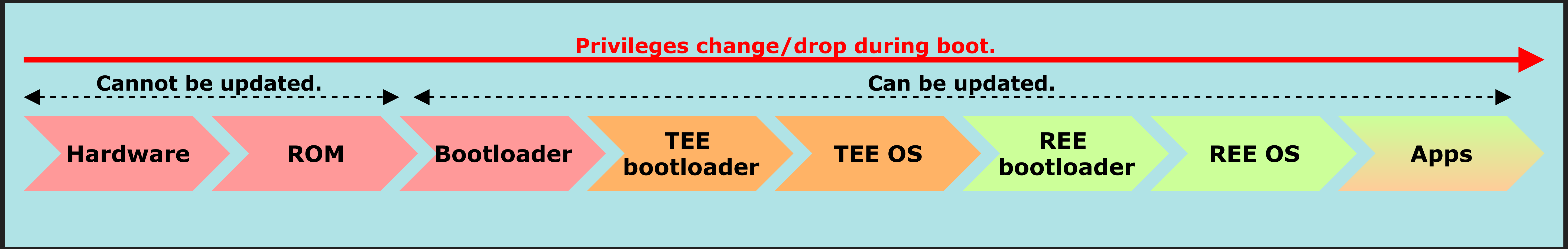
THE REAL WORLD IS A LITTLE MORE COMPLEX...

SECURE BOOT FLOW

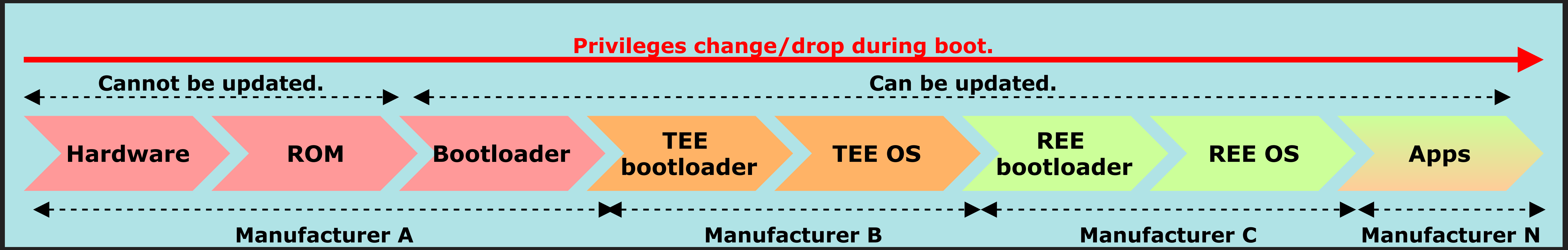
SECURE BOOT FLOW



SECURE BOOT FLOW



SECURE BOOT FLOW



Securing the entire chain is complex...

CONSTRAINTS...

Initializing, and interfacing with, hardware

Performance and code size

Customer needs

Recoverability

Keeping engineering cost low

IT'S IMPORTANT TO GET IT RIGHT

BAD SECURITY IS EXPENSIVE!

Tape out

Crisis management

PR damage

Recall of devices/unsold inventory

Time to market

Additional engineering time

SO… WHERE DO YOU START?

Designing Secure Boot in a Nutshell

Niek Timmers
Riscure
niek@riscure.com

Albert Spruyt
Freelance
albert.spruyt@gmail.com

Cristofaro Mune
Pulse Security
c.mune@pulse-sec.com

- [SBG-01]: Keep it simple
- [SBG-02]: Hardware root of trust
- [SBG-03]: Authenticate everything
- [SBG-04]: Decrypt everything
- [SBG-05]: No weak crypto
- [SBG-06]: No "wrong" crypto
- [SBG-07]: Limit options
- [SBG-08]: Lock hardware down
- [SBG-09]: Drop privileges asap
- [SBG-10]: Make software exploitation hard
- [SBG-11]: Make hardware attacks hard
- [SBG-12]: Stack your defenses
- [SBG-13]: Continuous review and testing
- [SBG-14]: Anti-rollback

LET'S DESIGN SECURE BOOT SECURELY!

BUT... BEFORE WE DO...

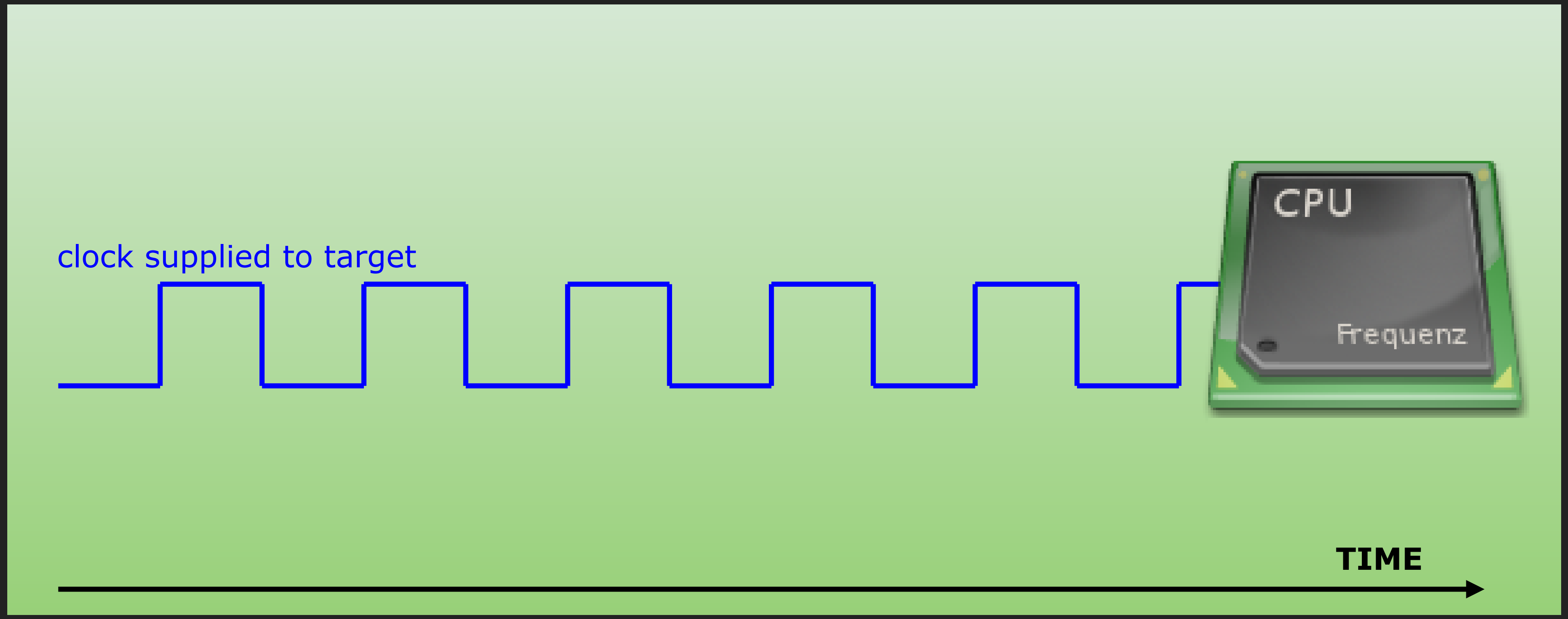
LET'S HAVE SOME FUN FIRST!

FAULT INJECTION

"Introducing faults into a target in order to change its intended behavior."

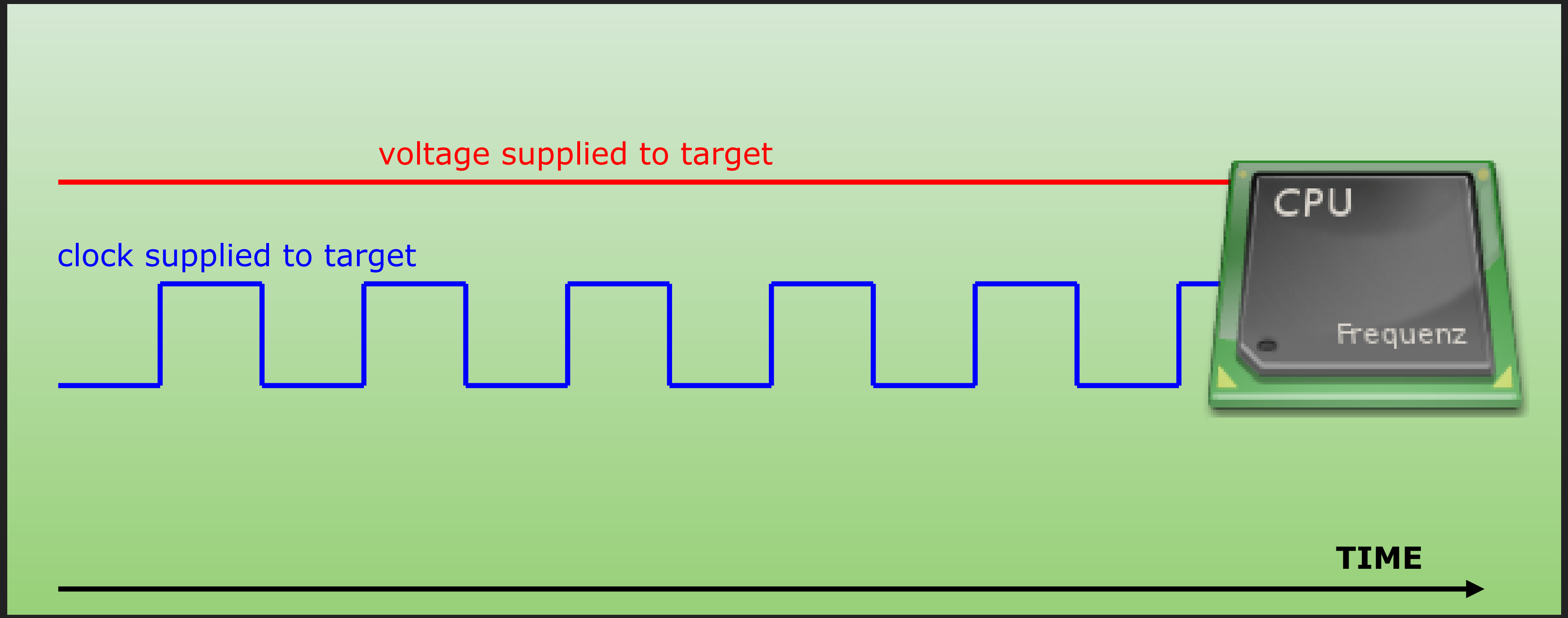
FAULT INJECTION

"Introducing faults into a target in order to change its intended behavior."



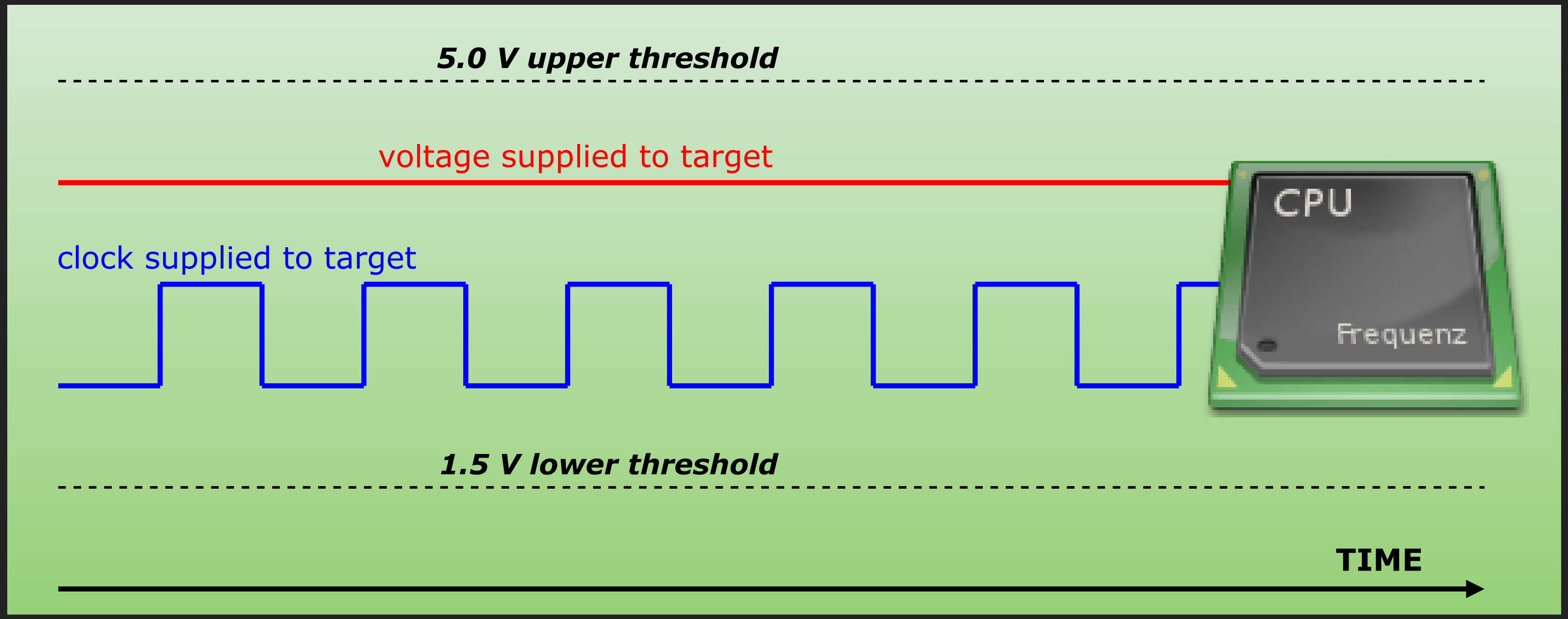
FAULT INJECTION

"Introducing faults into a target in order to change its intended behavior."



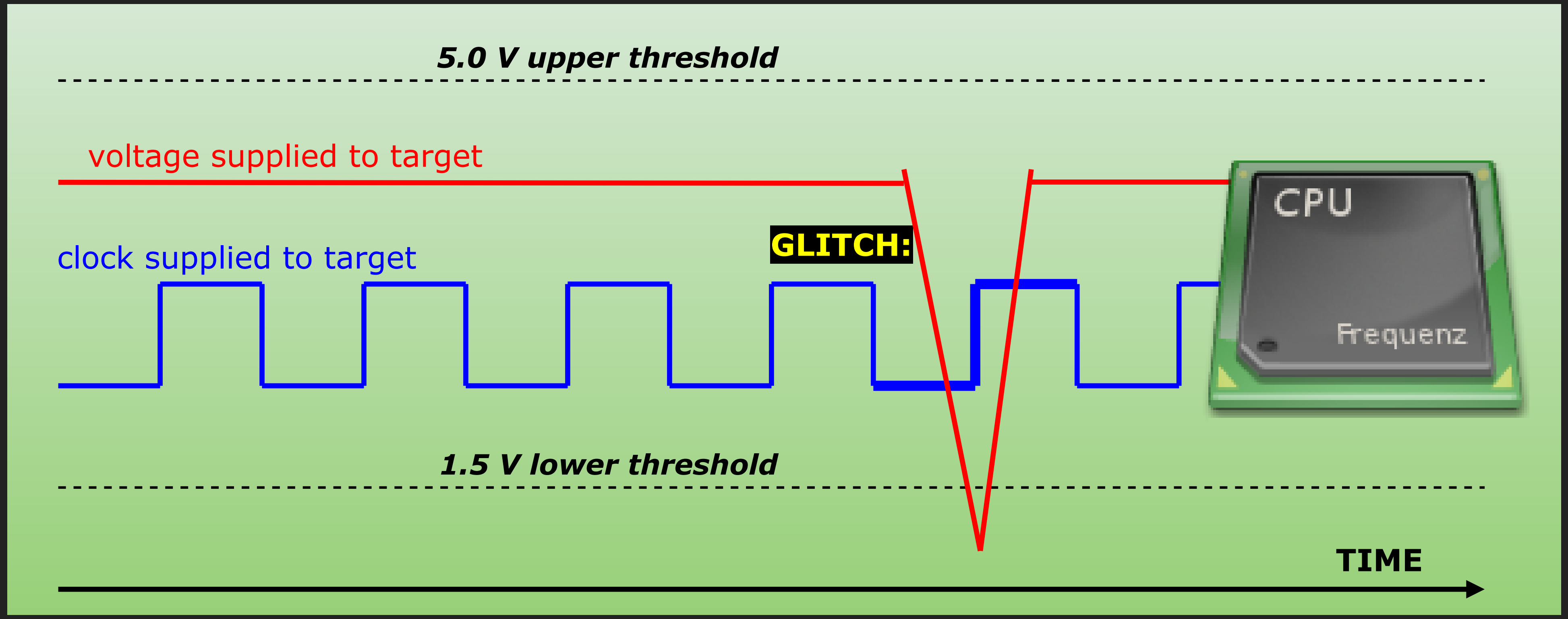
FAULT INJECTION

"Introducing faults into a target in order to change its intended behavior."



FAULT INJECTION

"Introducing faults into a target in order to change its intended behavior."

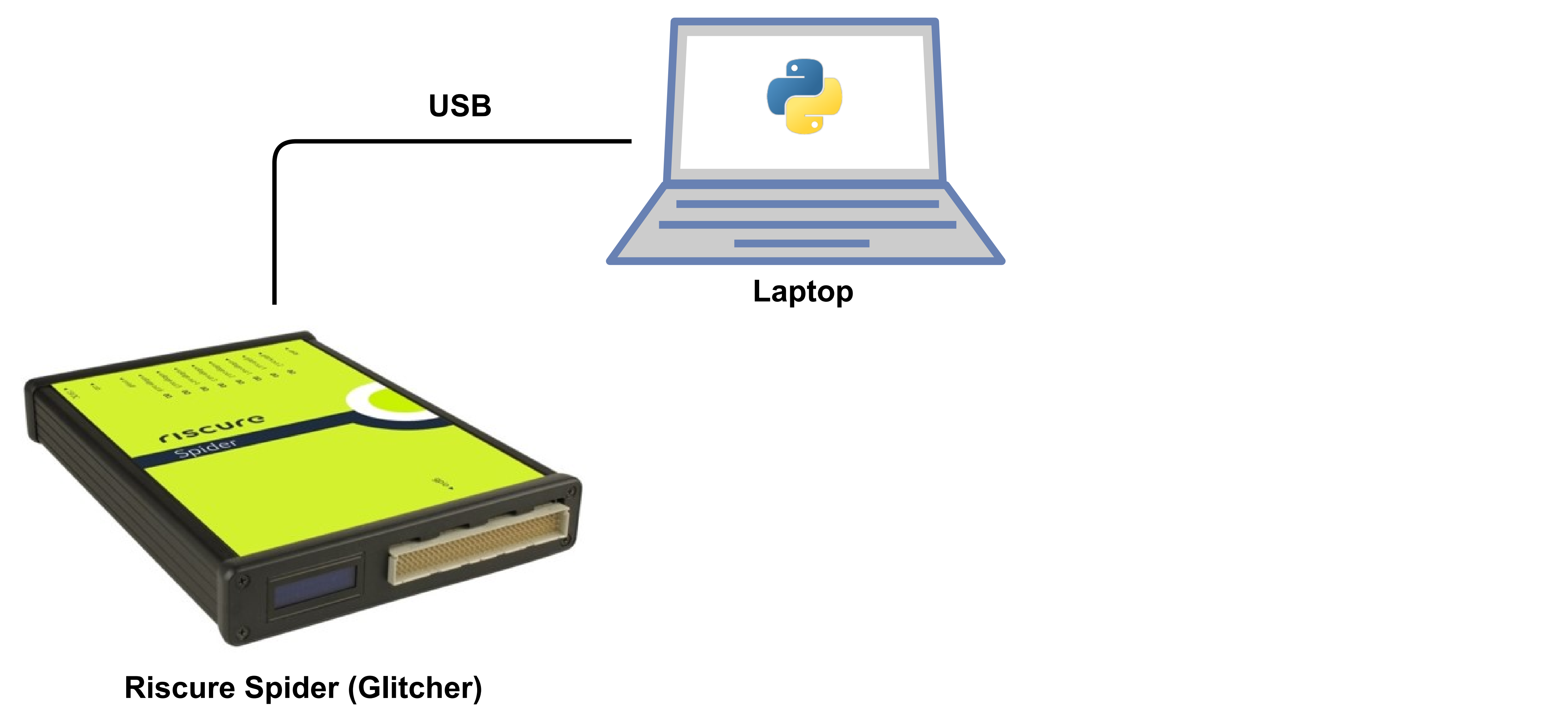


FAULT INJECTION SETUP

Riscure Spider (Glitcher)

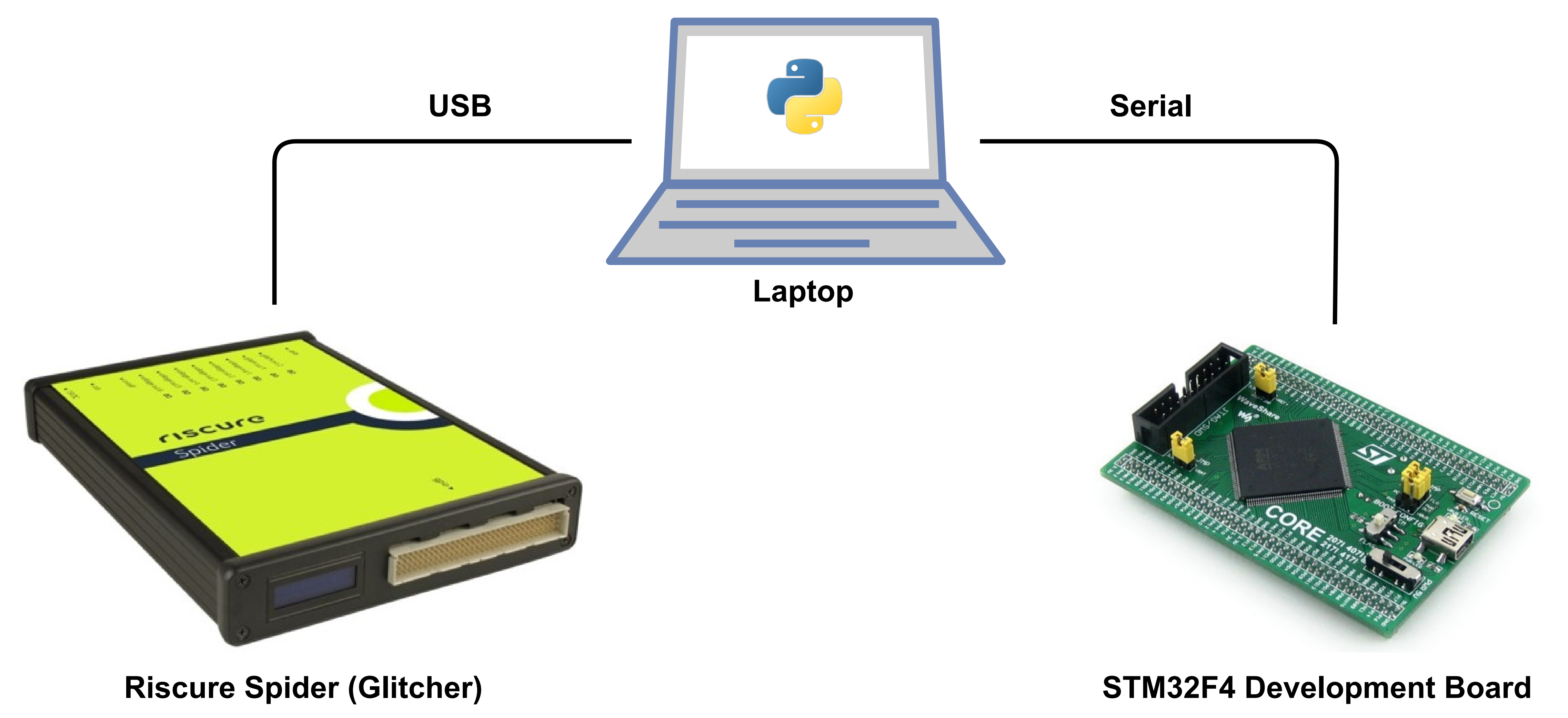
You can use NewAE's ChipWhisperer too!

FAULT INJECTION SETUP



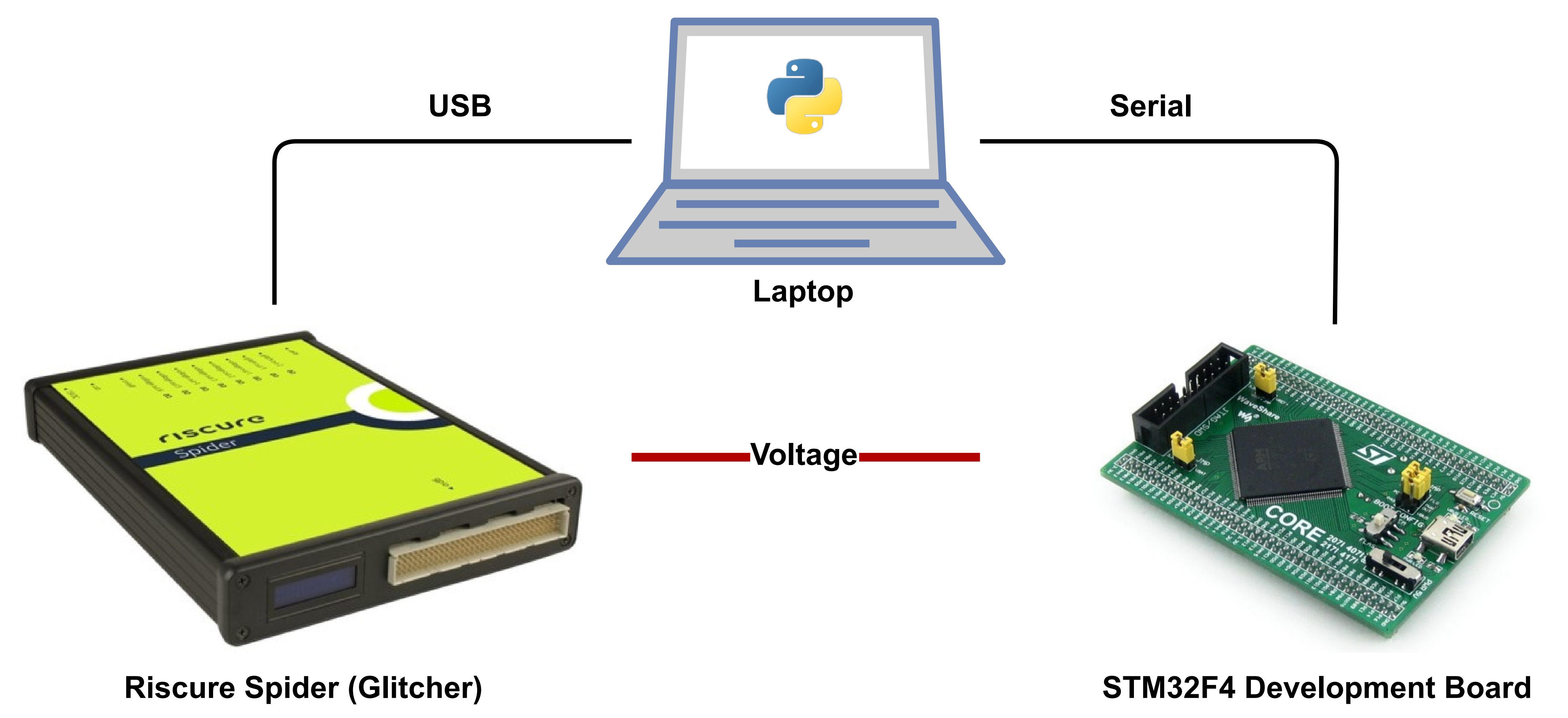
You can use NewAE's ChipWhisperer too!

FAULT INJECTION SETUP



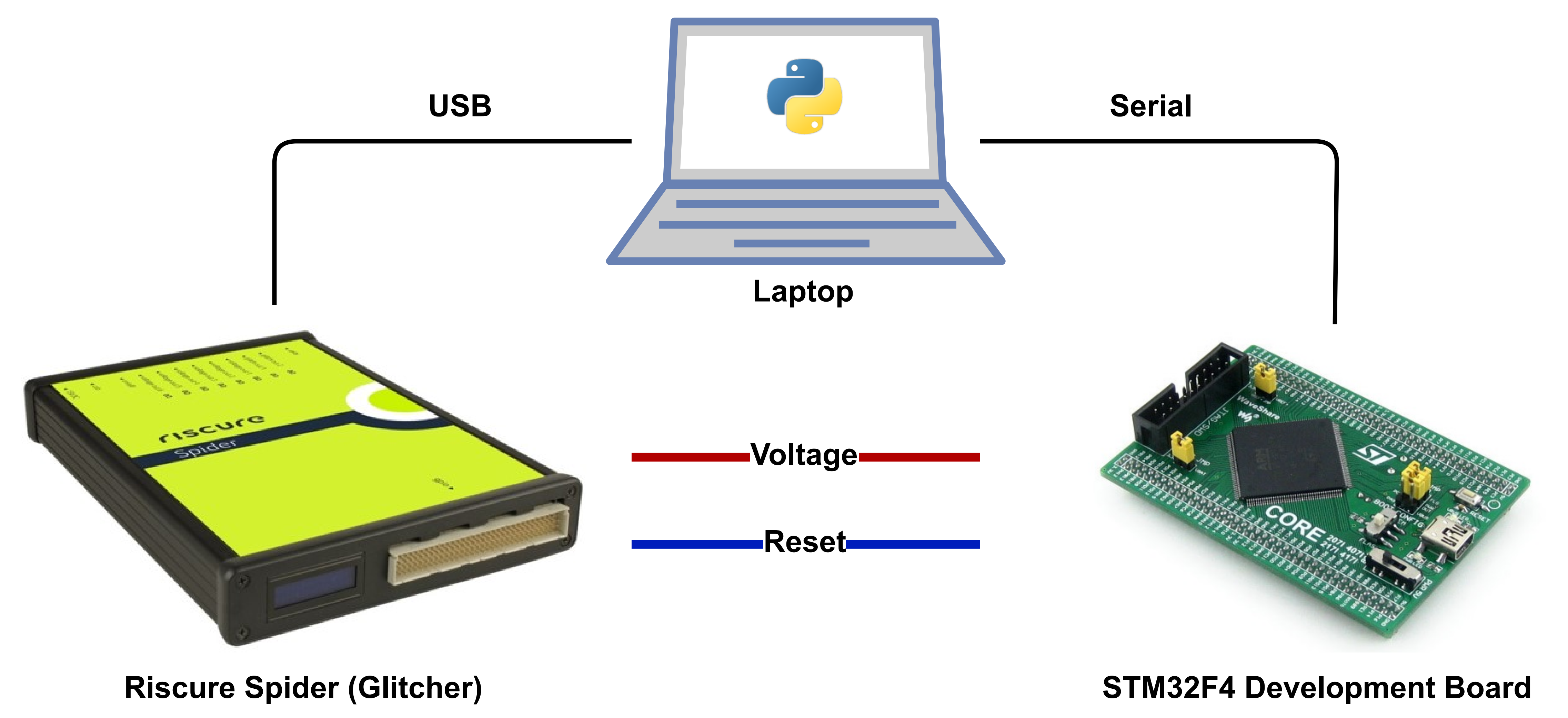
You can use NewAE's ChipWhisperer too!

FAULT INJECTION SETUP



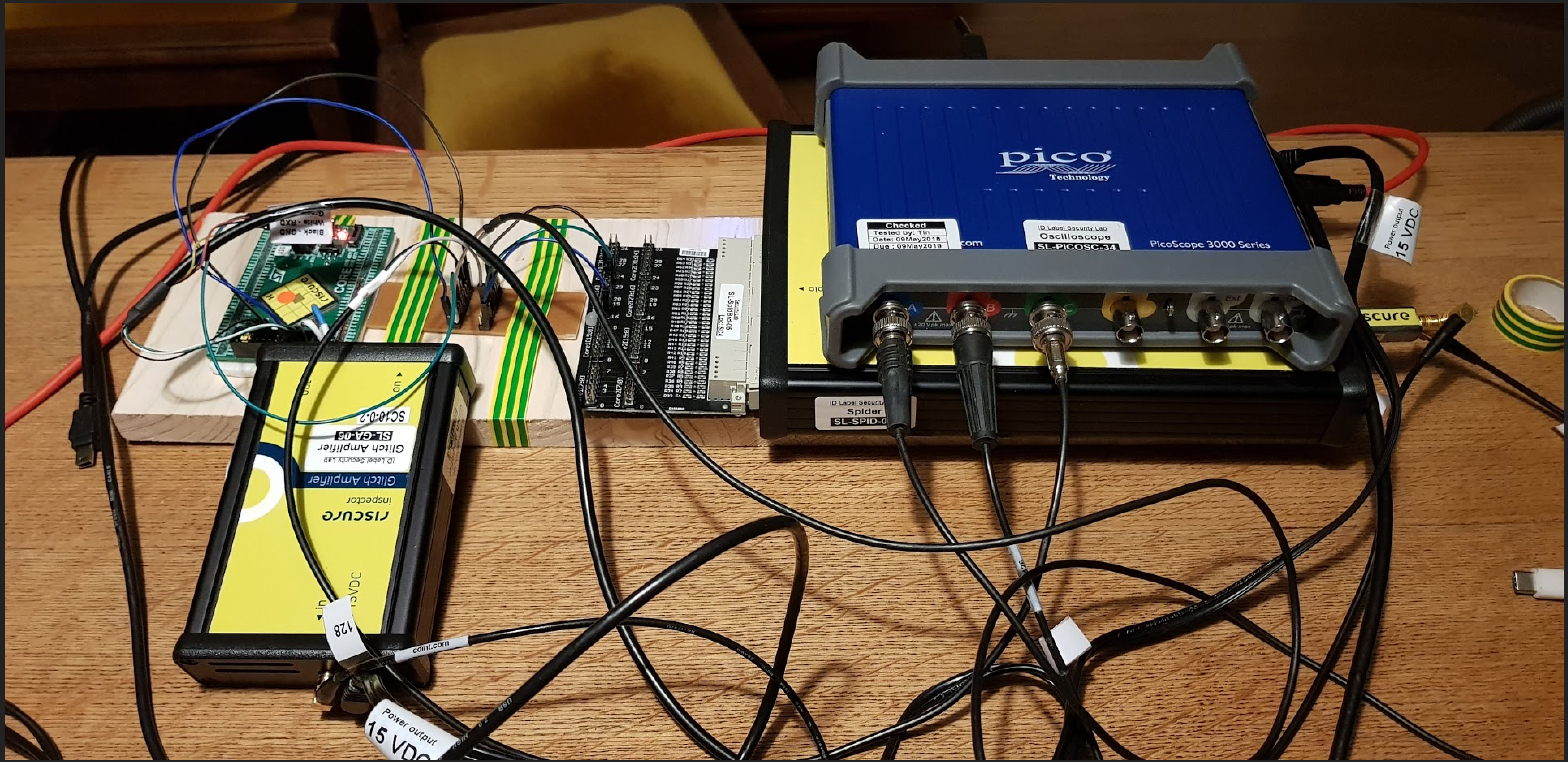
You can use NewAE's ChipWhisperer too!

FAULT INJECTION SETUP



You can use NewAE's ChipWhisperer too!

REAL WORLD SETUP



Even for simple setups there are cables everywhere...

FAULT INJECTION FAULT MODEL

Instruction corruption.

- Glitches can modify instructions
- Great for modifying code and getting control
- Breaks any software security model

Original instruction:

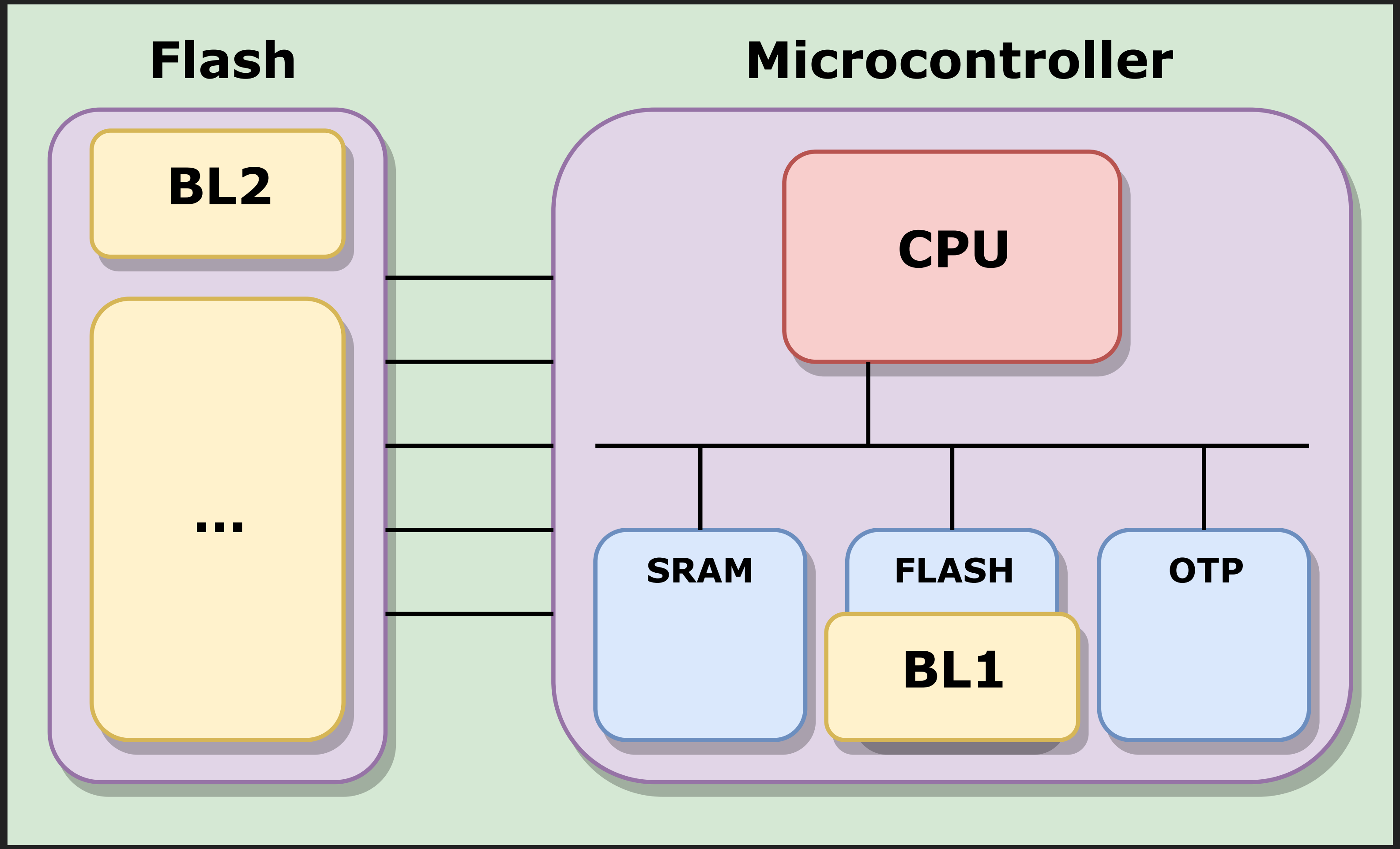
```
add r0, r1, r3 1110 1011 0000 0001
               0000 0000 0000 0011
```

Glitched instruction:

```
add r0, r1, r2 1110 1011 0000 0001
               0000 0000 0000 0010
```

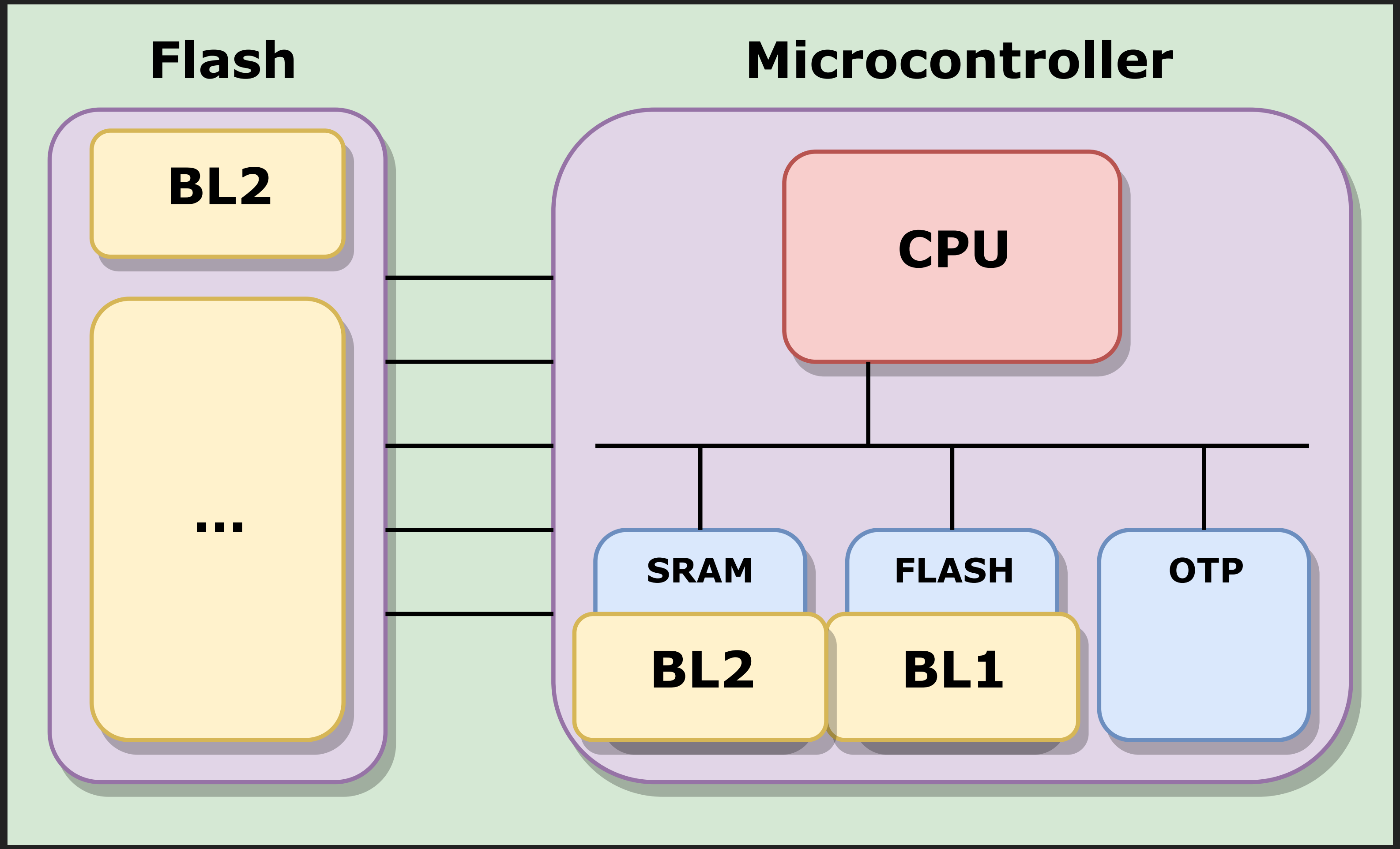
LET'S USE IT TO BYPASS ENCRYPTED SECURE BOOT!

ENCRYPTED SECURE BOOT DESIGN



BL1 is executed from internal flash

ENCRYPTED SECURE BOOT DESIGN



BL1 loads, decrypts and verifies BL2

ENCRYPTED SECURE BOOT IMPLEMENTATION

```
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);  // 1. Copy image
decrypt(IMG_RAM, IMG_SIZE, KEY);
memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);

sha(IMG_RAM, IMG_SIZE, IMG_HASH);
rsa(PUB_KEY, SIG_RAM,  SIG_HASH);

if(compare(IMG_HASH, SIG_HASH) != 0) {
    while(1);
}

((void *)())(IMG_RAM)();
```

ENCRYPTED SECURE BOOT IMPLEMENTATION

```
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);  // 1. Copy image
decrypt(IMG_RAM, IMG_SIZE, KEY);       // 2. Decrypt image
memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);

sha(IMG_RAM, IMG_SIZE, IMG_HASH);
rsa(PUB_KEY, SIG_RAM,  SIG_HASH);

if(compare(IMG_HASH, SIG_HASH) != 0) {
    while(1);
}

((void *)())(IMG_RAM)();
```

ENCRYPTED SECURE BOOT IMPLEMENTATION

```
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);  // 1. Copy image
decrypt(IMG_RAM, IMG_SIZE, KEY);       // 2. Decrypt image
memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);  // 3. Copy signature

sha(IMG_RAM, IMG_SIZE, IMG_HASH);
rsa(PUB_KEY, SIG_RAM,  SIG_HASH);

if(compare(IMG_HASH, SIG_HASH) != 0) {
    while(1);
}

((void *)())(IMG_RAM)();
```

ENCRYPTED SECURE BOOT IMPLEMENTATION

```
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);  // 1. Copy image
decrypt(IMG_RAM, IMG_SIZE, KEY);       // 2. Decrypt image
memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);  // 3. Copy signature

sha(IMG_RAM, IMG_SIZE, IMG_HASH);      // 4. Calculate hash from image
rsa(PUB_KEY, SIG_RAM,  SIG_HASH);

if(compare(IMG_HASH, SIG_HASH) != 0) {
    while(1);
}

((void *)())(IMG_RAM)();
```

ENCRYPTED SECURE BOOT IMPLEMENTATION

```
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);   // 1. Copy image
decrypt(IMG_RAM, IMG_SIZE, KEY);        // 2. Decrypt image
memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);   // 3. Copy signature

sha(IMG_RAM, IMG_SIZE, IMG_HASH);       // 4. Calculate hash from image
rsa(PUB_KEY, SIG_RAM,  SIG_HASH);       // 5. Obtain hash from signature

if(compare(IMG_HASH, SIG_HASH) != 0) {
    while(1);
}

((void *)())(IMG_RAM)();
```

ENCRYPTED SECURE BOOT IMPLEMENTATION

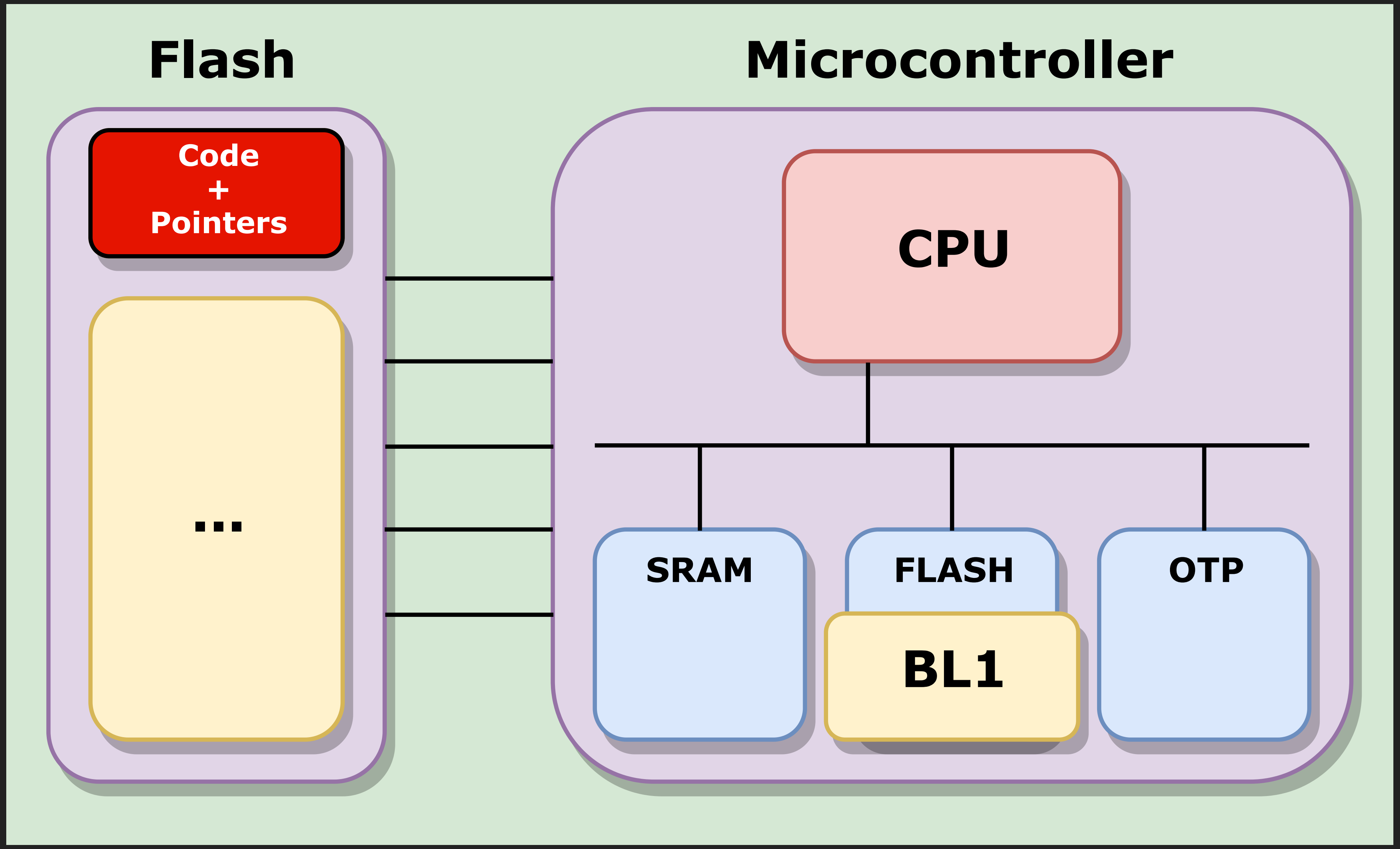
```
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);  // 1. Copy image
decrypt(IMG_RAM, IMG_SIZE, KEY);       // 2. Decrypt image
memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);  // 3. Copy signature

sha(IMG_RAM, IMG_SIZE, IMG_HASH);      // 4. Calculate hash from image
rsa(PUB_KEY, SIG_RAM,  SIG_HASH);      // 5. Obtain hash from signature

if(compare(IMG_HASH, SIG_HASH) != 0) { // 6. Compare hashes
    while(1);
}

((void *)())(IMG_RAM)();
```

ENCRYPTED SECURE BOOT IMPLEMENTATION

```
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);   // 1. Copy image
decrypt(IMG_RAM, IMG_SIZE, KEY);        // 2. Decrypt image
memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);   // 3. Copy signature

sha(IMG_RAM, IMG_SIZE, IMG_HASH);       // 4. Calculate hash from image
rsa(PUB_KEY, SIG_RAM,  SIG_HASH);       // 5. Obtain hash from signature

if(compare(IMG_HASH, SIG_HASH) != 0) { // 6. Compare hashes
    while(1);
}

((void *)())(IMG_RAM)();                // 7. Jump to next image
```

HOW DO WE ATTACK?

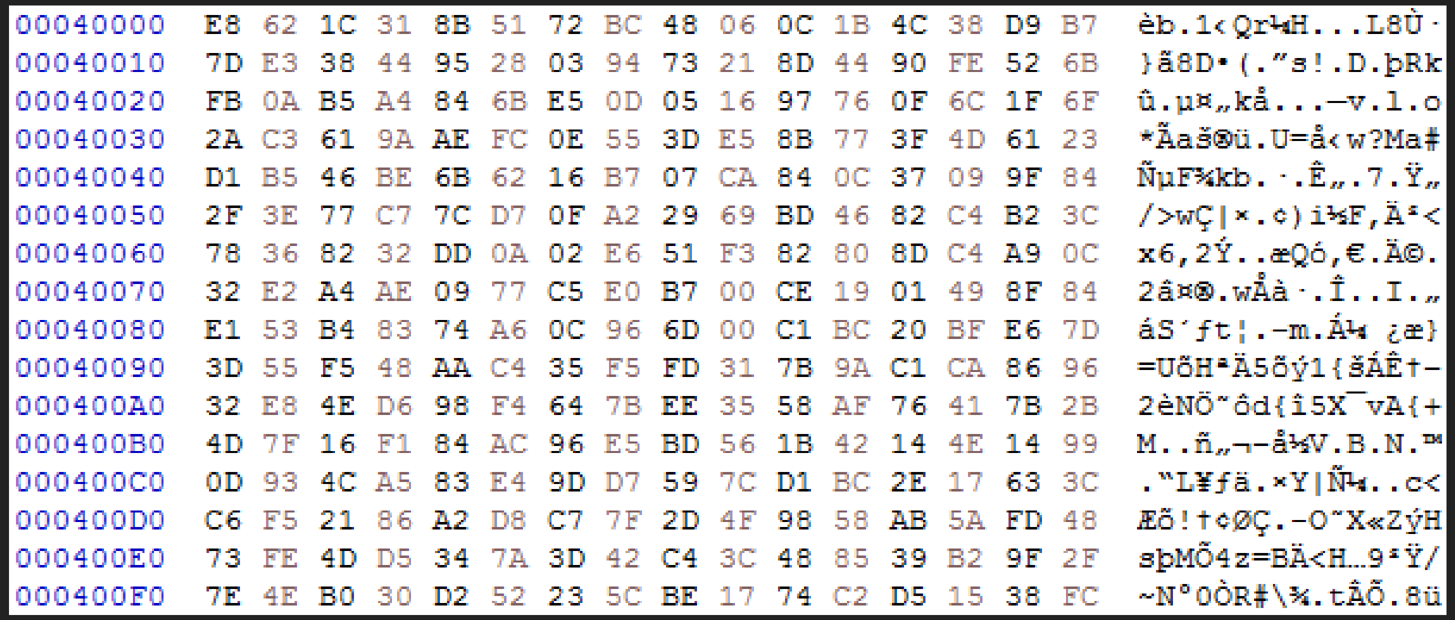
BYPASSING ENCRYPTED SECURE BOOT



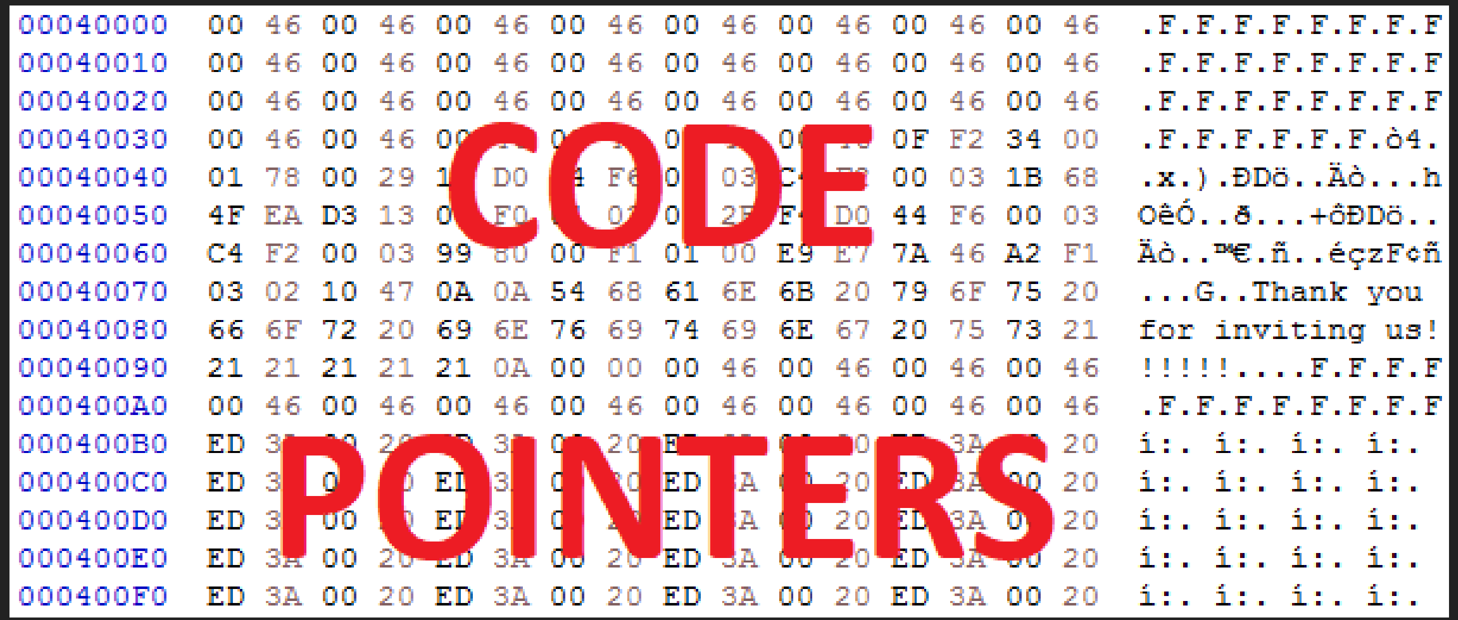
BL2 is replaced with code and pointers to SRAM

FLASH IMAGE MODIFICATION AND BEHAVIOR

Valid BL2 image



Malicious BL2 image



Valid BL2 image UART output

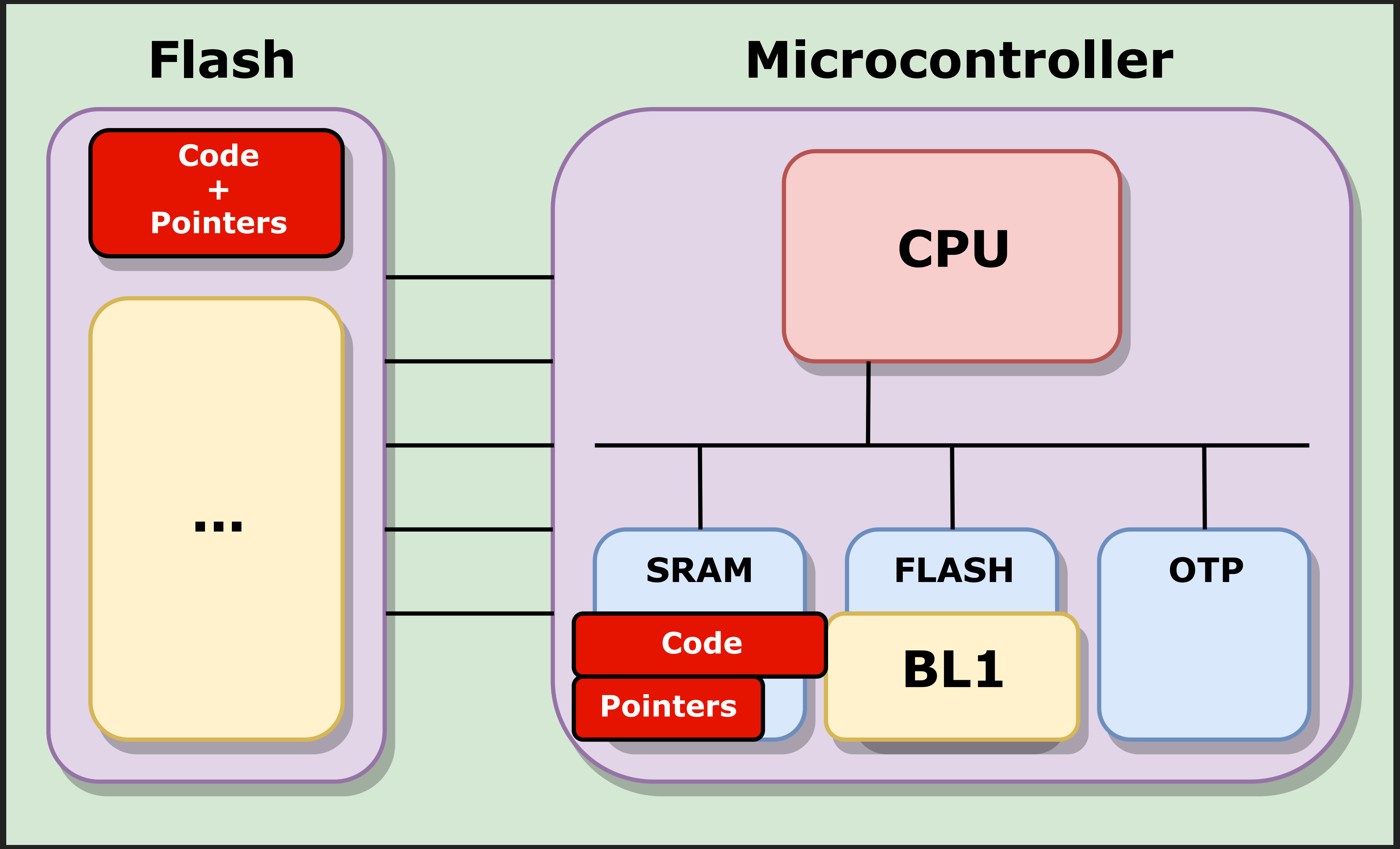
```
[BL1]: Successfully started.
[BL1]: Loading BL2 successful.
[BL1]: Decrypting BL2 successful.
[BL1]: Authenticating BL2
       successful.
[BL1]: Jumping to BL2...
[BL2]: Successfully started.
```

Malicious BL2 image UART output

```
[BL1]: Successfully started.
[BL1]: Loading BL2 successful.
[BL1]: Decrypting BL2 successful.
[BL1]: Authenticating BL2
       unsuccessful. Stopping!
```

WHEN DO WE INJECT THE GLITCH?

BYPASSING ENCRYPTED SECURE BOOT



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

BYPASSING ENCRYPTED SECURE BOOT

```
...
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);  // GLITCH HERE
.
.
.
.
.
.
.
.
.
.
((void *)())(pointer)();
...
```

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

LET'S DO THIS!

On another laptop...

CONCRETELY SAID...

WE TURN

ENCRYPTED SECURE BOOT

INTO

PLAIN TEXT UNPROTECTED BOOT

USING

A SINGLE GLITCH AND NO KEY!

WHY DOES THIS WORK?

CONTROLLING PC

- Glitch controlled value into PC directly (see: paper)
- LDM/STM instructions used for copying memory

Original:

```
08008102
08008102 loop_0
08008102 LDMIA.W R1!, {R4-R11}
08008106 STMIA.W R0!, {R4-R11}
0800810A SUBS    R2, #0x20
0800810C BNE     loop_0
```

Glitched:

```
08008102
08008102 loop_0
08008102 LDMIA.W R1!, {R4- PC }
08008106 STMIA.W R0!, {R4-R11}
0800810A SUBS    R2, #0x20
0800810C BNE     loop_0
```

- Demonstrated attack is 32-bit ARM specific
- Variants of this attack applicable to all architectures

IS THIS THE ONLY FI ATTACK ON SECURE BOOT?

ENUMERATION OF FI ATTACKS ON SECURE BOOT

Bypassing Secure Boot using Fault Injection

Niek Timmers
Riscure
timmers@riscure.com

Albert Spruyt
Freelance
albert.spruyt@gmail.com

Cristofaro Mune
Pulse Security
c.mune@pulse-sec.com

Please see our offensive paper!

IT'S TIME TO DESIGN SECURE BOOT SECURELY...

LET'S GET THE FUNDAMENTALS RIGHT!

SECURE BOOT FUNDAMENTALS

- Hardware root of trust
- Authenticate everything
- Encrypt everything
- Use strong crypto
- Use crypto correctly

We assume you all agree. But... it goes often wrong!

HARDWARE ROOT OF TRUST

How many devices do you know without ROM/OTP?

Real world Secure Boot bypass:
Intel's Root of Trust using SPI flash.

AUTHENTICATE EVERYTHING

How does the ROM know how large the image is?

AUTHENTICATE EVERYTHING


```c
struct header {
    uint32_t  BL1_length;
    uint32_t  BL1_destination;
    uint32_t  BL1_entry_point;
} _header;
```

Length and destination are used before verification

AUTHENTICATE EVERYTHING


```c
struct header {
    uint32_t  BL1_length;
    uint32_t  BL1_destination;
    uint32_t  BL1_entry_point;
    uint8_t   BL1_header_sig[0x100];
} _header;
```

Header needs its own signature

AUTHENTICATE EVERYTHING

- Authenticate all security relevant code and data
- Try to prevent mistakes:
    - Design should enforce authenticating everything

Real world Secure Boot bypass:
AMD Secure Boot by CTSLabs

ENCRYPT EVERYTHING

Are u proposing security by obscurity?

- There will be software vulnerabilities
- Make analyzing the firmware hard
- Attacks may be more difficult to perform

FUNDAMENTALS MAKE SENSE… WHAT ELSE?

KEEP IT SIMPLE

- Nobody wants complex parsing during boot
- Do not support the world (especially in ROM)
- Make auditing the code easier

Real world Secure Boot bypass:
U-Boot vulnerability in file system parser

DROP PRIVILEGES ASAP

- Not just operating modes:
  - Monitor, Hypervisor, Kernel, User
- But also access to:
  - Keys, ROM, crypto engines

LET'S ASSUME THE DESIGN IS GREAT!

BUT CONTAINS SOFTWARE VULNERABILITIES...

EXPLOITATION MITIGATIONS AT RUNTIME

STACK CANARY	NX	PIE	RPATH	RUNPATH	Symbols	FORTIFY
Canary found	NX enabled	PIE enabled	No RPATH	No RUNPATH	No Symbols	Yes

- Binaries are hardened by the compiler
- Operating system makes exploitation difficult too
- Stack cookies, W^X, ASLR, CFI, etc.

DO YOU THINK THAT'S DONE AT EARLY BOOT?

MOST EARLY BOOT STAGES DO:

- not have stack cookies
- not have ASLR
- not have CFI
- not have the MPU/MMU enabled/configured
- not have IOMMU/SMMU enabled/configured

COME ON! IT'S 2019...

YOU MAY GET THESE ALMOST FOR FREE:

- Stack cookies
- Control flow integrity (CFI)

MEMORY PROTECTION MAY BE MORE CHALLENGING:

- MPU/MMU
  - W^X
- IOMMU/SMMU
  - Prevent DMA from overwriting code/data

BUT WAIT...

WHAT ABOUT HARDWARE HACKERS?

EVERYTHING APPLIES!

PLUS SOME MORE...

PCB LEVEL ATTACKS

- An attacker can tamper with signals on the PCB
- Copy data from external memory once
  - Operate only on the internal copy
  - Prevent TOCTOU / Double Fetch vulnerabilities
- Flash emulator

LOCK DOWN YOUR HARDWARE

- Disable peripherals that are not used
  - e.g. external memories, USB, etc.
  - No access to external flash; no TOCTOUs
- Disable or protect JTAG/DEBUG ports
- Disable debug messages on serial ports

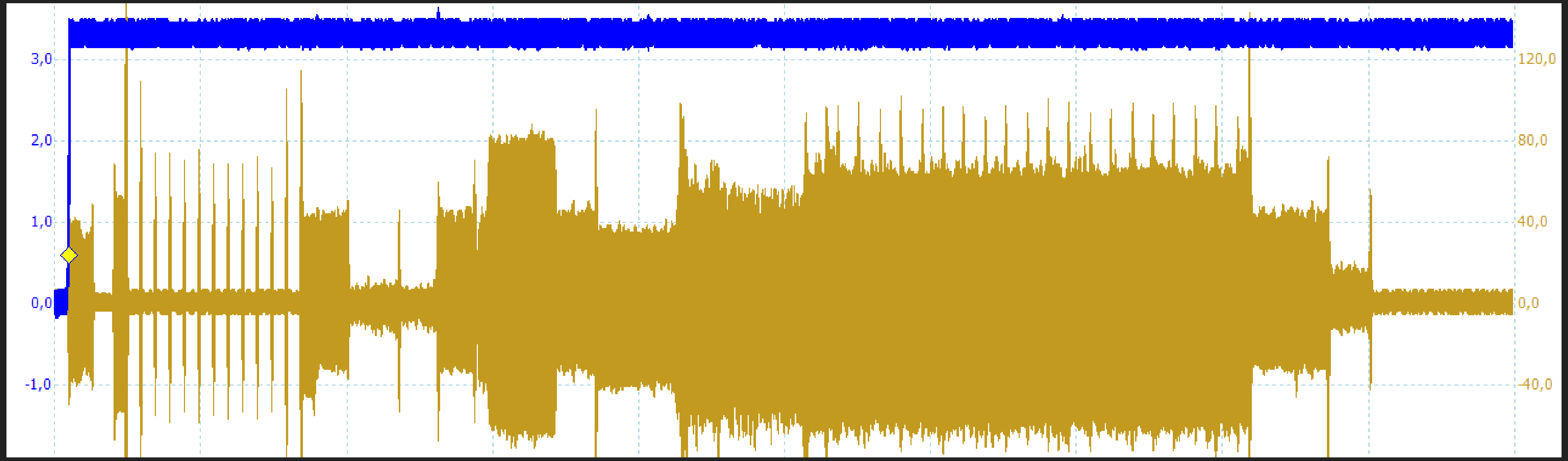
WHAT ABOUT ATTACKERS WITH MORE THAN A:

ONLY PEW PEW PEW LIKE IN THE DEMO?

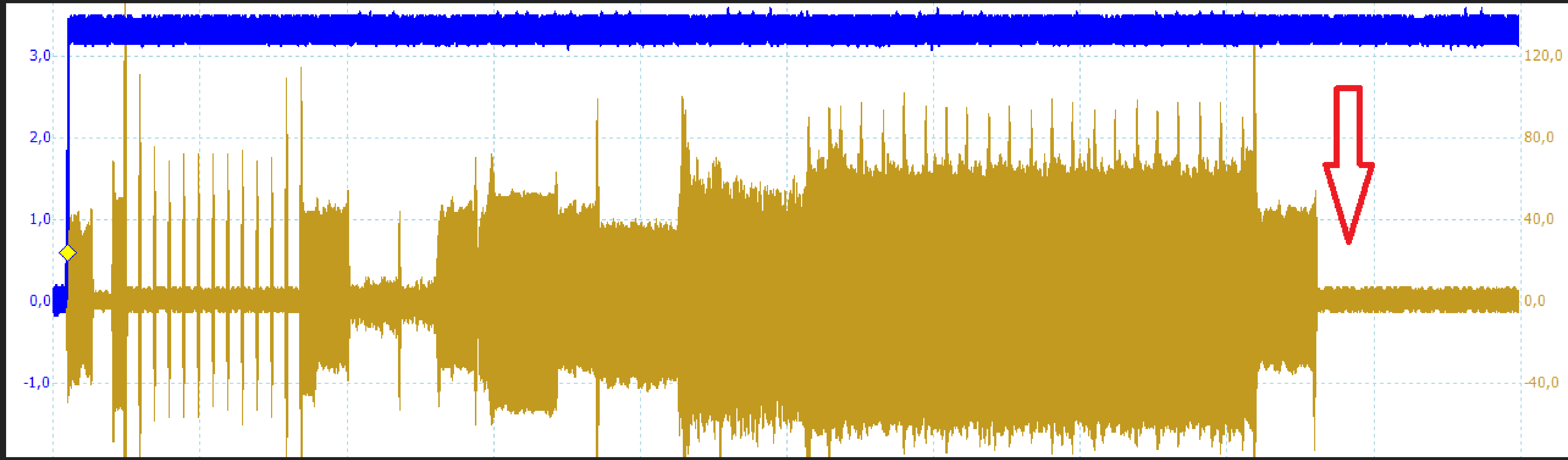
SIDE CHANNEL ATTACKS

WHAT ARE SIDE CHANNEL ATTACKS? 1/2

Power consumption of a valid image



Power consumption of an invalid image



IS THIS THE ONLY SIDE CHANNEL?

WHAT ARE SIDE CHANNEL ATTACKS? 2/2

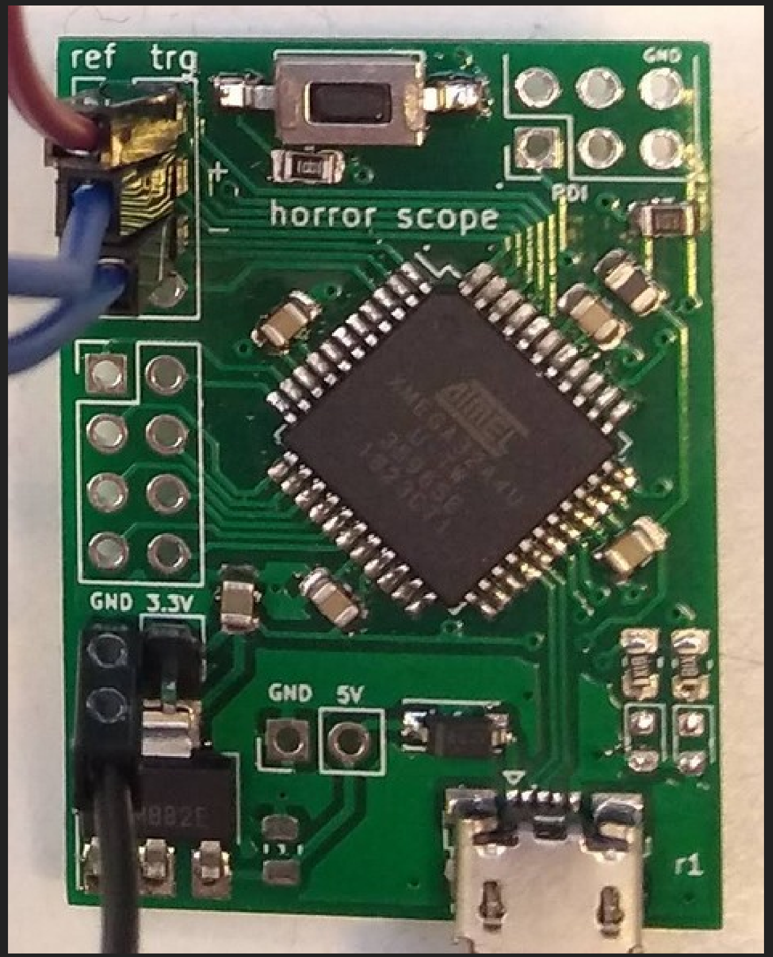
- Timing attacks to recover HMAC/CMAC
  - Real world example: Xbox 360
- DPA attack to recover encryption keys

Do not expect secrets (i.e. keys) will be secret forever!

ARE FI AND SCA ATTACKS EXPENSIVE?

FI AND SCA ARE NOT (ALWAYS) EXPENSIVE!

- The HorrorScope ($5)
  - By Albert and Alyssa (@noopwafel)
  - FI and SCA
- Alternatives:
  - Any board with fast ADC/GPIO (free?)
  - ChipWhisperer Nano (~$50)



Please see our presentation at Troopers 2019!

SCA AND ESPECIALLY FI ARE REAL THREATS!

LET'S MAKE FAULT INJECTION HARDER!

GOALS WHEN MITIGATING FI

- Lower the probability of success
- Low enough probability equals infeasible
  - infeasbible equals *takes too much time*

HARDENING HARDWARE (ICS) AGAINST FI

- Redundancy
- Checksums
- Clock jitter
- Glitch/Fault detectors

Lots of academic research e.g.:

The Sorcerer’s Apprentice Guide to Fault Attacks

CHALLENGES FOR HARDWARE MITIGATION

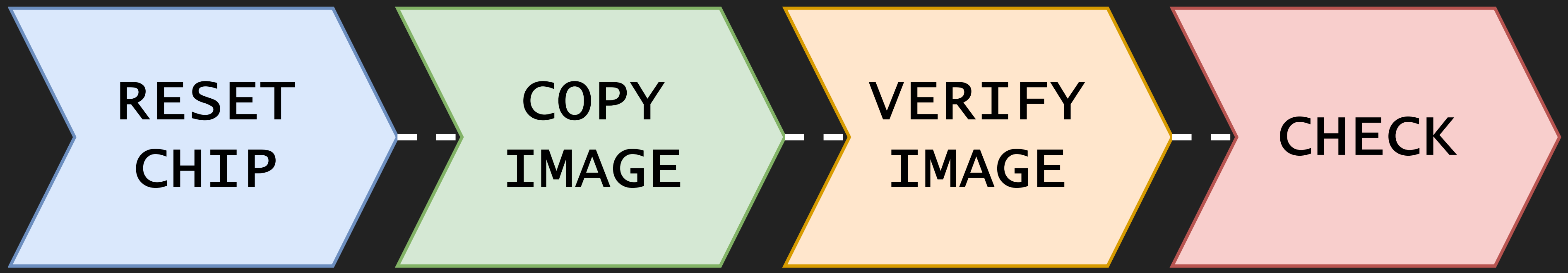
- Hardware is fixed
- Adding hardware is costly
- Detectors need calibration

FI resistent hardware is not yet realistic for most devices!

WHAT CAN BE DONE

WITHOUT MODIFYING HARDWARE?

LET'S MAKE BYPASSING A CHECK HARD



USING STANDARD HARDWARE AND SOFTWARE!

MAKING BYPASSING A CHECK HARD

Multiple checks

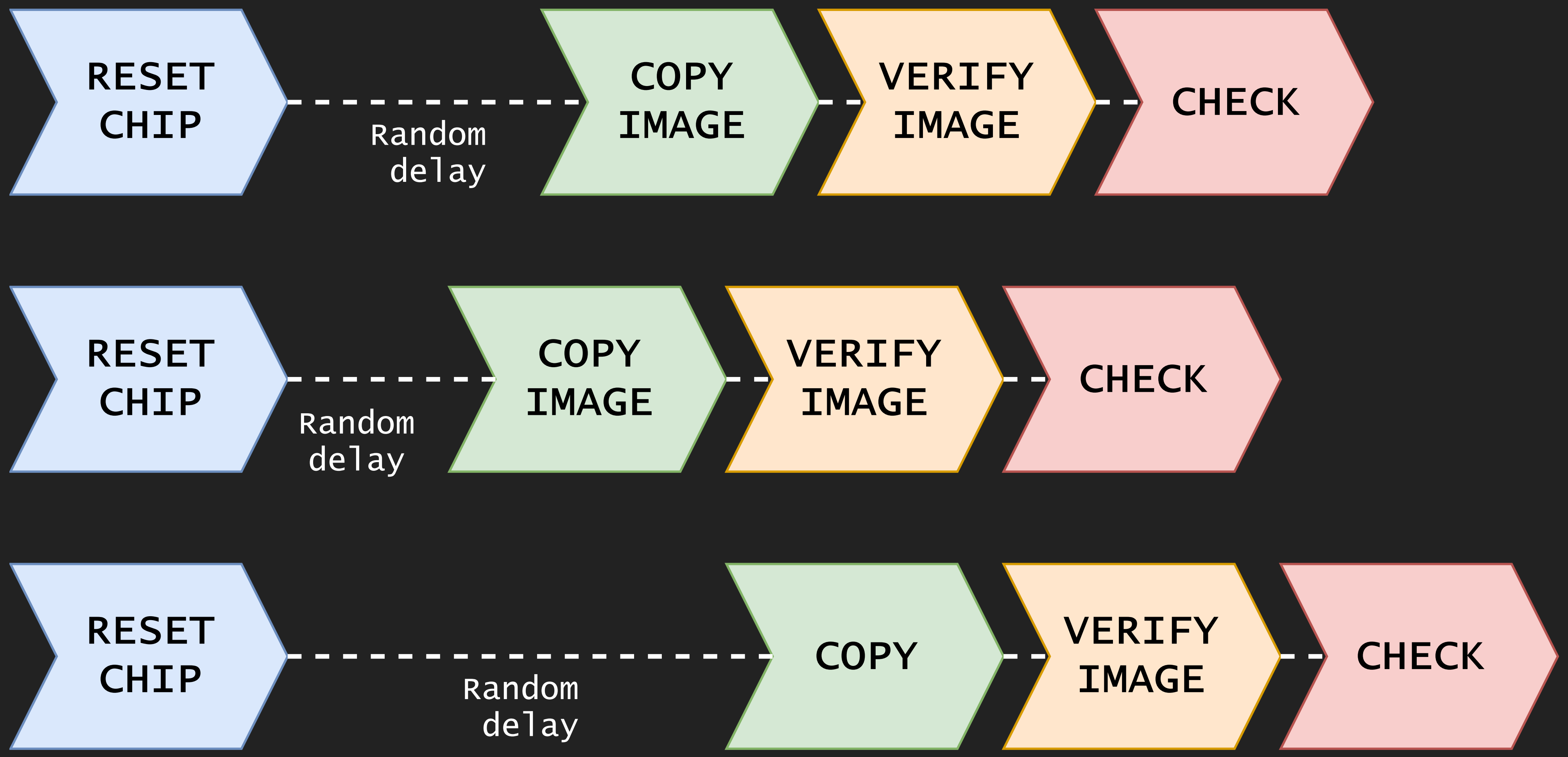
- Identify all critical checks in your code
- Perform these checks multiple times

- Probability for success will likely drop

MAKING BYPASSING A CHECK HARD

Random delays

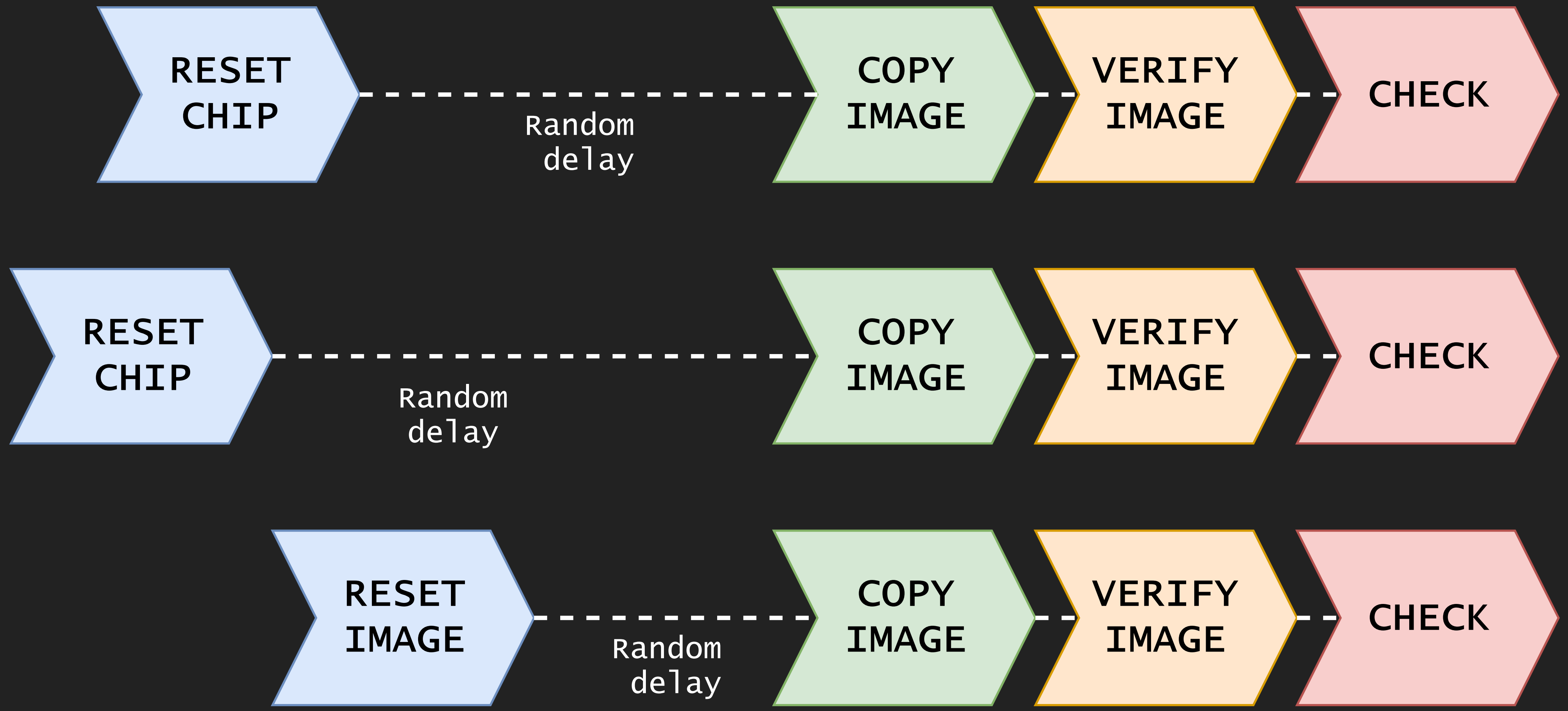
- Randomize critical checks in time



- Probability for success will likely drop more

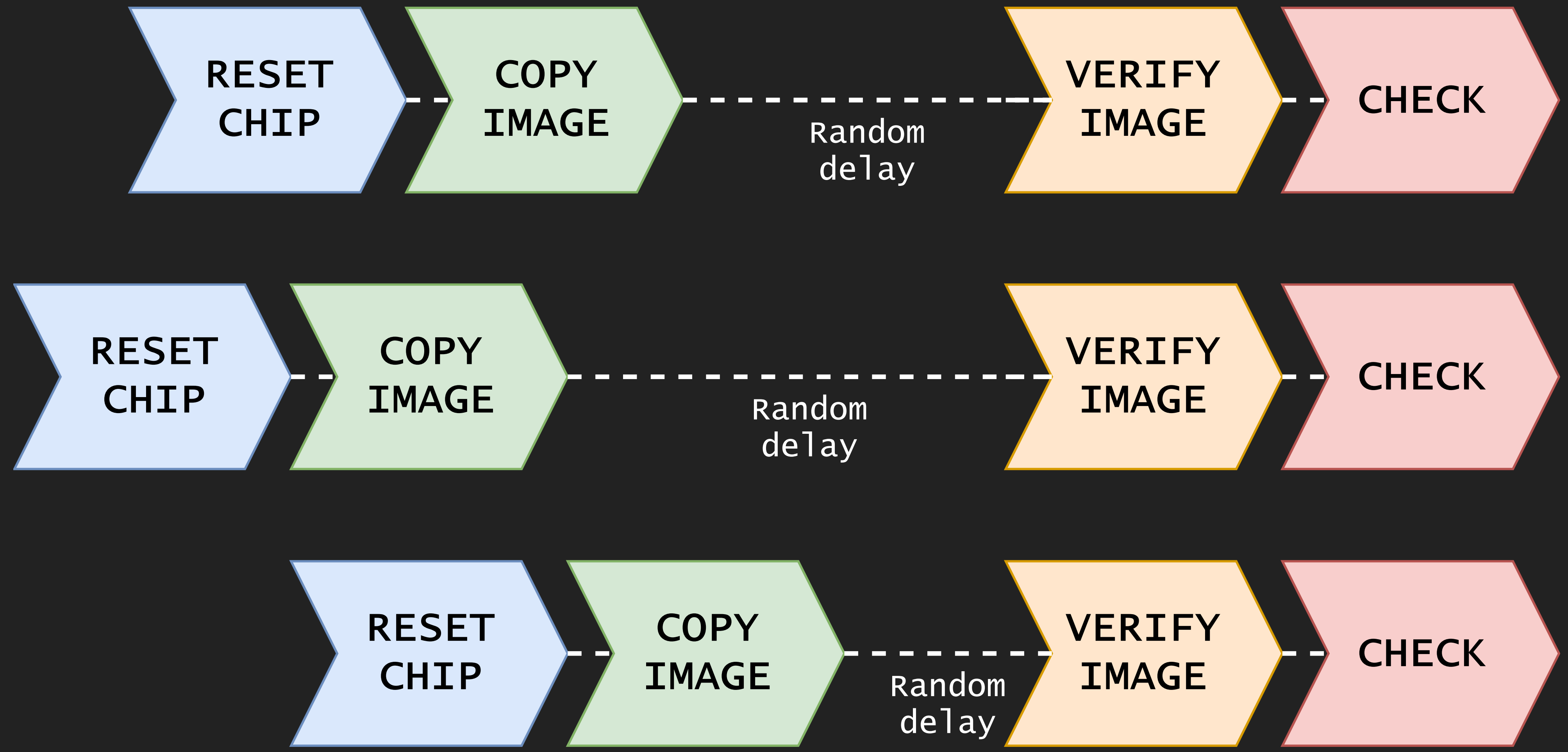
WHAT GOES WRONG?

BAD RANDOM DELAY #1



External SPI communication can be used for timing!

BAD RANDOM DELAY #2



Power consumption can be also used for timing!

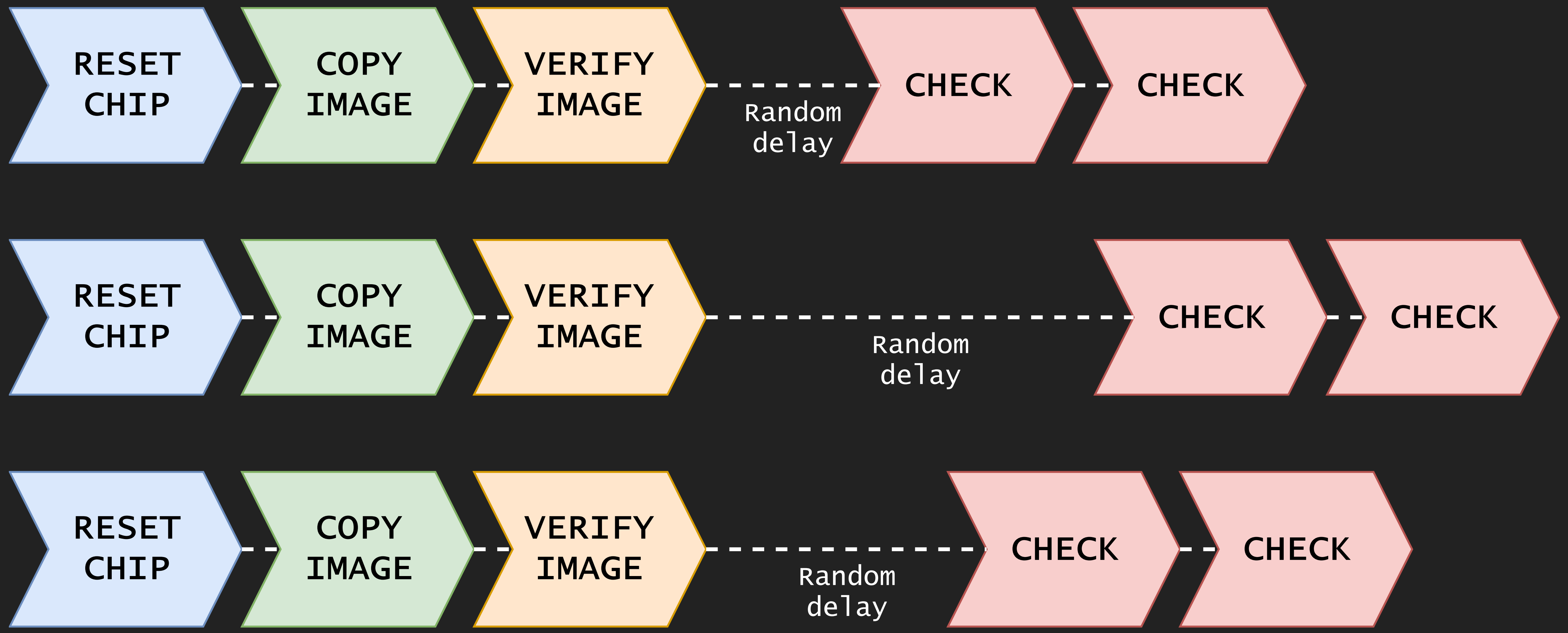
GOOD RANDOM DELAY!

Little time after random delay to inject glitch

WHAT ABOUT COMBINING <u>MULTIPLE CHECKS</u> AND <u>RANDOM DELAYS</u>?

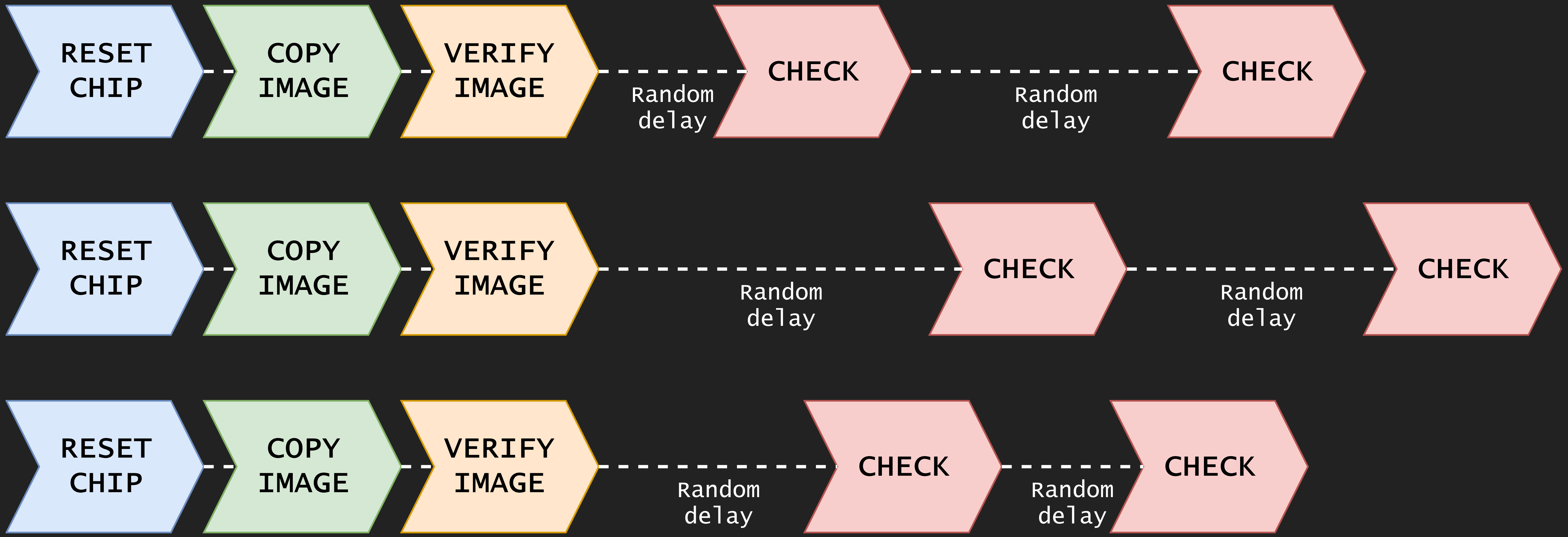
COMBINED MITIGATION #1

Random delays + Multiple checks



What could be improved?

BETTER COMBINATION!



Probability for success drops signifcantly!

Let's combine some more...

COMBINED MITIGATION #2

W^X + Multiple checks

Let's use it to mitigate the attack from the demo!

COMBINED MITIGATION #2: MULTIPLE CHECKS

```
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);
memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);
sha(IMG_RAM, IMG_SIZE, IMG_HASH);
rsa(PUB_KEY, SIG_RAM,  SIG_HASH);

if(compare(IMG_HASH, SIG_HASH) != 0) { // Compare hashes
    while(1);
}
if(compare(IMG_HASH, SIG_HASH) != 0) { // Compare hashes again
    while(1);
}
((void *)())(IMG_RAM)();
```

COMBINED MITIGATION #2: MULTIPLE CHECKS + W^X

```
makeWritable(IMG_RAM,IMG_SIZE);        // Make IMG_RAM read-write

memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE);
memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);

sha(IMG_RAM, IMG_SIZE, IMG_HASH);
rsa(PUB_KEY, SIG_RAM,  SIG_HASH);

if(compare(IMG_HASH, SIG_HASH) != 0) {
    while(1);
}

makeExecutable(IMG_RAM, IMG_SIZE);     // Make IMG_RAM executable

if(compare(IMG_HASH, SIG_HASH) != 0) {
    while(1);
}
((void *)())(IMG_RAM)();
```

COMBINED MITIGATION #2

W^X + Multiple checks

- Control flow cannot be hijacked at the memcpy
- The code needs to be made executable
- Multiple glitches required to bypass secure boot

THESE ARE JUST SOME EXAMPLES... BE CREATIVE!

KEY TAKEAWAYS

1. Secure boot design is hard (even for experts)
2. Smart secure boot design saves money
3. Software mitigations can be cheap
4. Stacking different mitigations can be effective
5. Testing is essential to verify the implementation

THANK YOU. QUESTIONS?

Do you think Secure Boot implementations can be improved significantly without significant costs?

Niek Timmers

niek@riscure.com

@tieknimmers

Albert Spruyt

albert.spruyt@gmail.com

Riscure is hiring and visit our booth!