SILM Security of software / hardware interfaces

Summer school, 8-12/07/2019, Inria Rennes

Session #2: Fault Injection Attacks

Niek Timmers (independent)

info@niektimmers.com / @tieknimmers / http://www.niektimmers.com/

Where did Albert leave us?

Where did Albert leave us?

- Glitches can be injected in chips using various techniques

Where did Albert leave us?

- Glitches can be injected in chips using various techniques
- Hardware vulnerabilities are triggered in order to cause faults

Where did Albert leave us?

- Glitches can be injected in chips using various techniques
- Hardware vulnerabilities are triggered in order to cause faults
- We can exploit devices without relying on software vulnerabilities

Where did Albert leave us?

- Glitches can be injected in chips using various techniques
- Hardware vulnerabilities are triggered in order to cause faults
- We can exploit devices without relying on software vulnerabilities

What are typical targets?

Fault Injection Targets

Fault Injection Targets

- Most standard chips are vulnerable
  - Incl. basic MCUs and (very) advanced SoCs

Fault Injection Targets

- Most standard chips are vulnerable
  - Incl. basic MCUs and (very) advanced SoCs

- Most standard architectures are affected
  - i.e. ARM, MIPS, Intel, etc.

Fault Injection Targets

- Most standard chips are vulnerable
  - Incl. basic MCUs and (very) advanced SoCs
- Most standard architectures are affected
  - i.e. ARM, MIPS, Intel, etc.
- Fast processing (> 1GHz) is not a show stopper

Fault Injection Goals

Fault Injection Goals

- Bypassing security features
  - e.g. debug interface protection, secure boot, etc.

Fault Injection Goals

- Bypassing security features
  - e.g. debug interface protection, secure boot, etc.

- Hijacking control flow
  - i.e. achieving arbitrary code execution

Fault Injection Goals

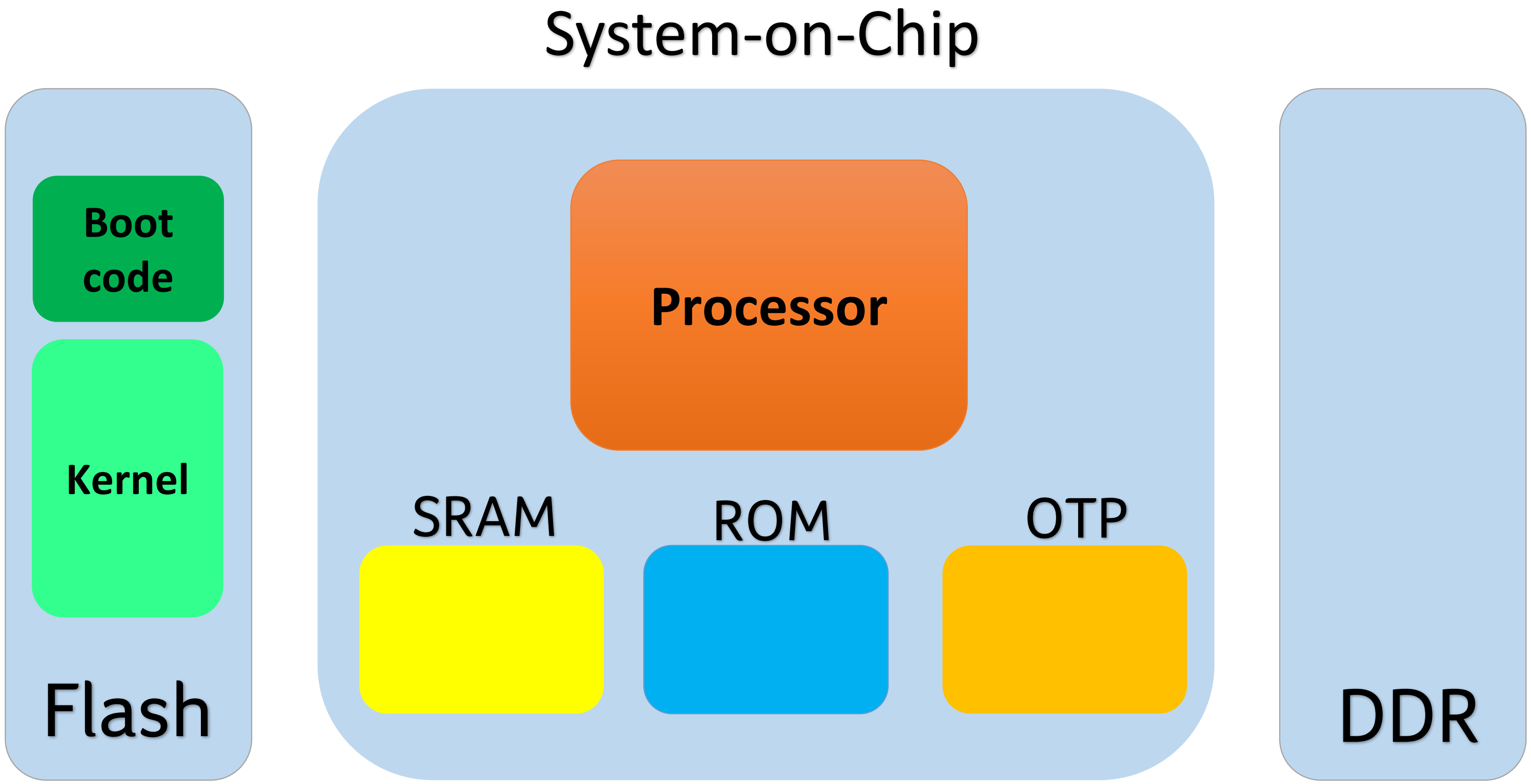
- Bypassing security features
  - e.g. debug interface protection, secure boot, etc.
- Hijacking control flow
  - i.e. achieving arbitrary code execution
- Breaking cryptographic algorithms
  - i.e. differential fault analysis (DFA) attacks

Let’s attack something…

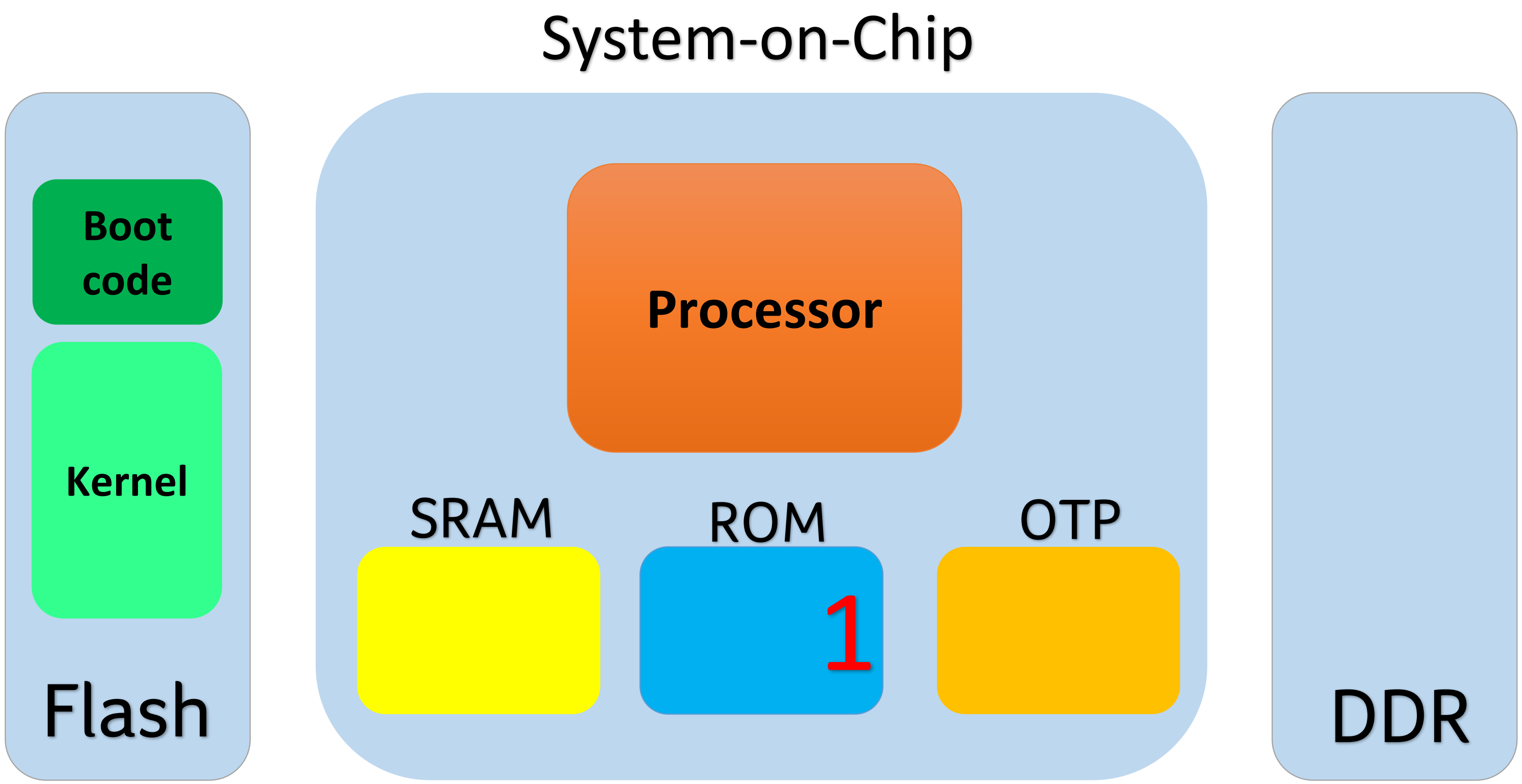
Let’s attack something…

Let’s attack Secure Boot!

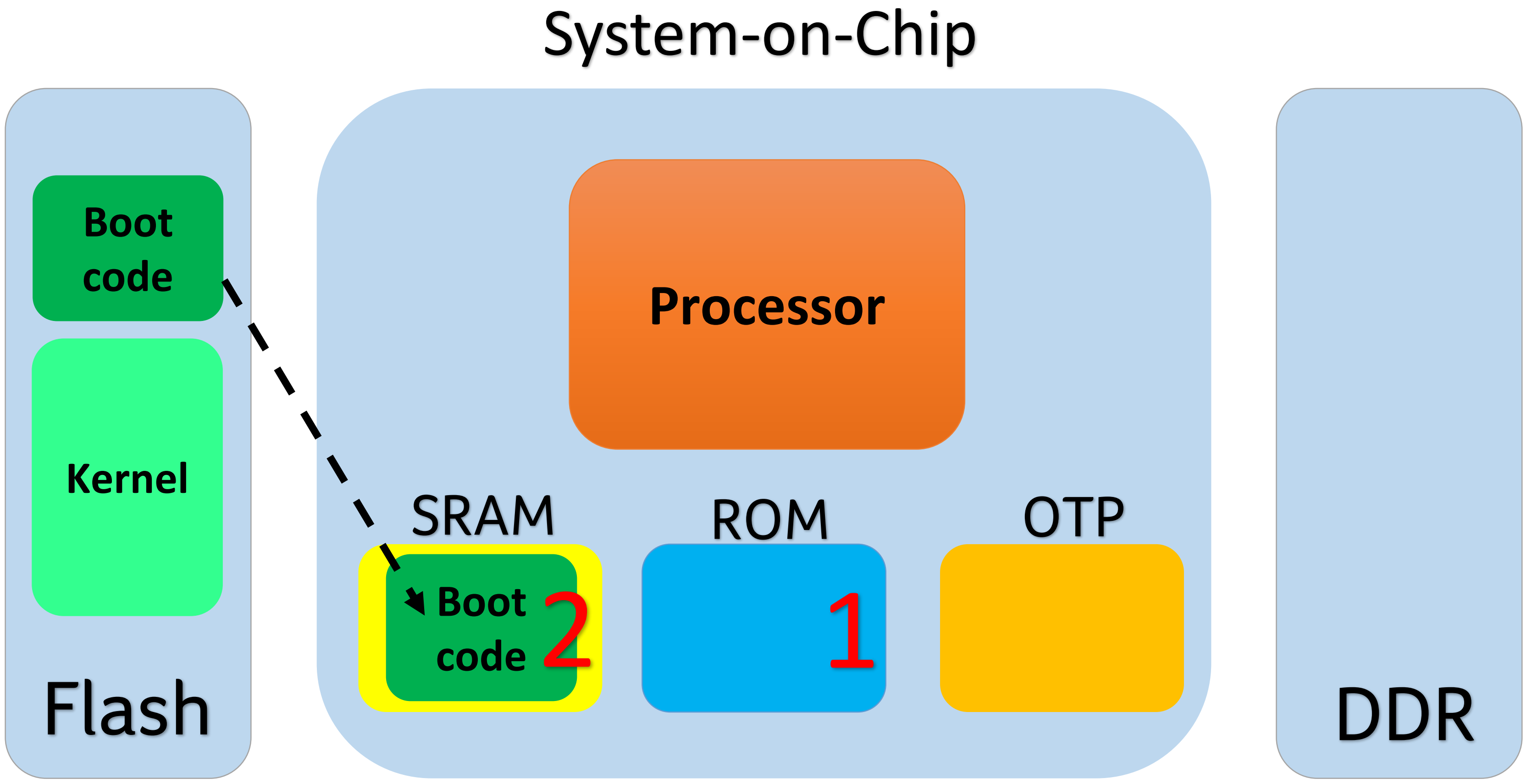
Why do we need secure boot?



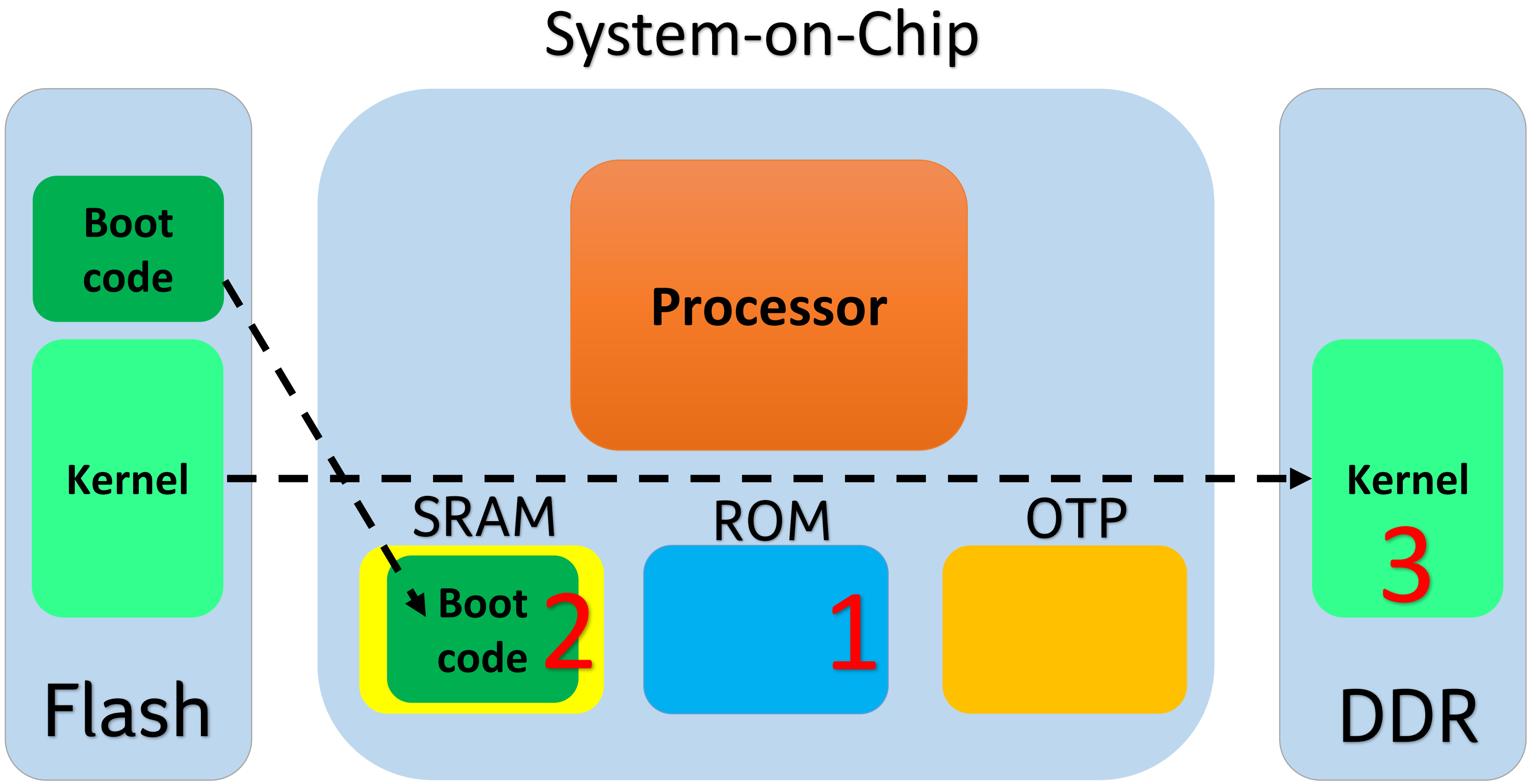
Why do we need secure boot?



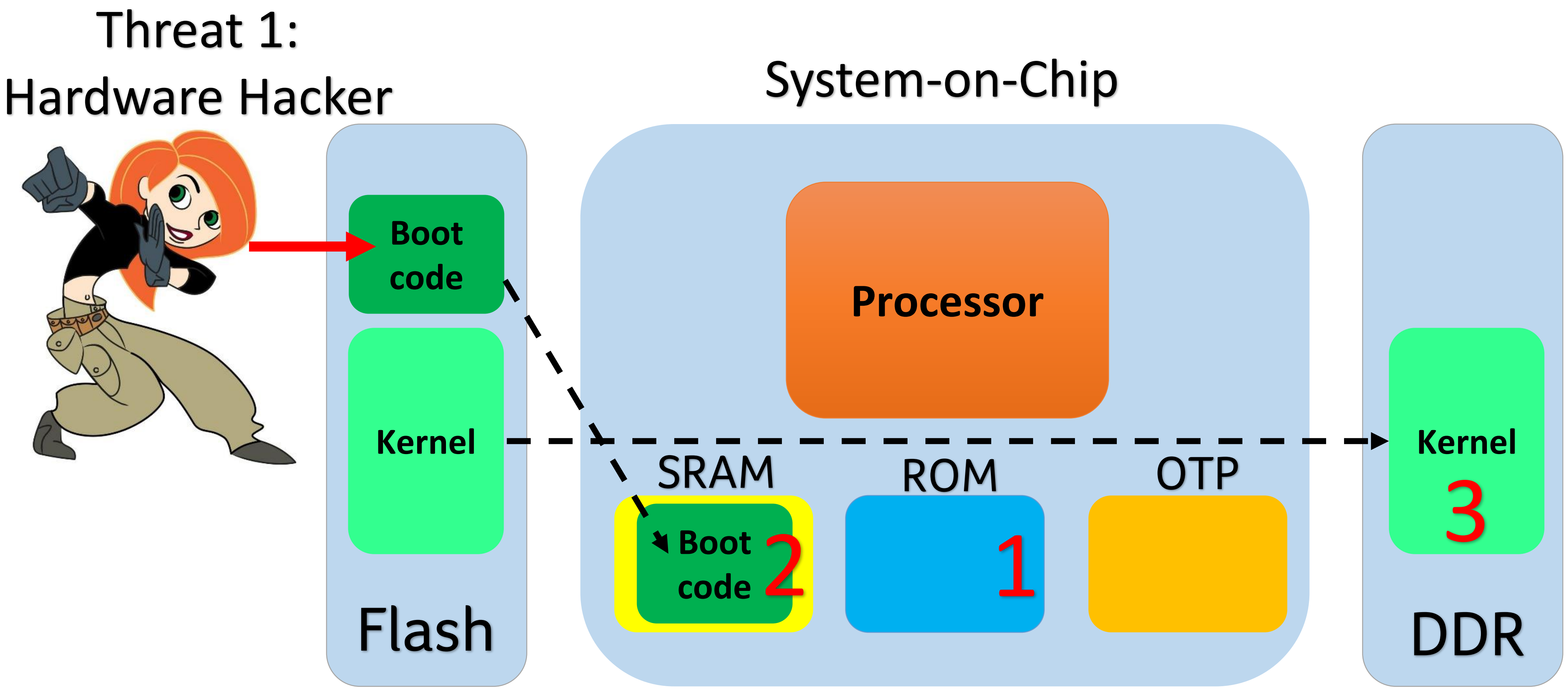
Why do we need secure boot?



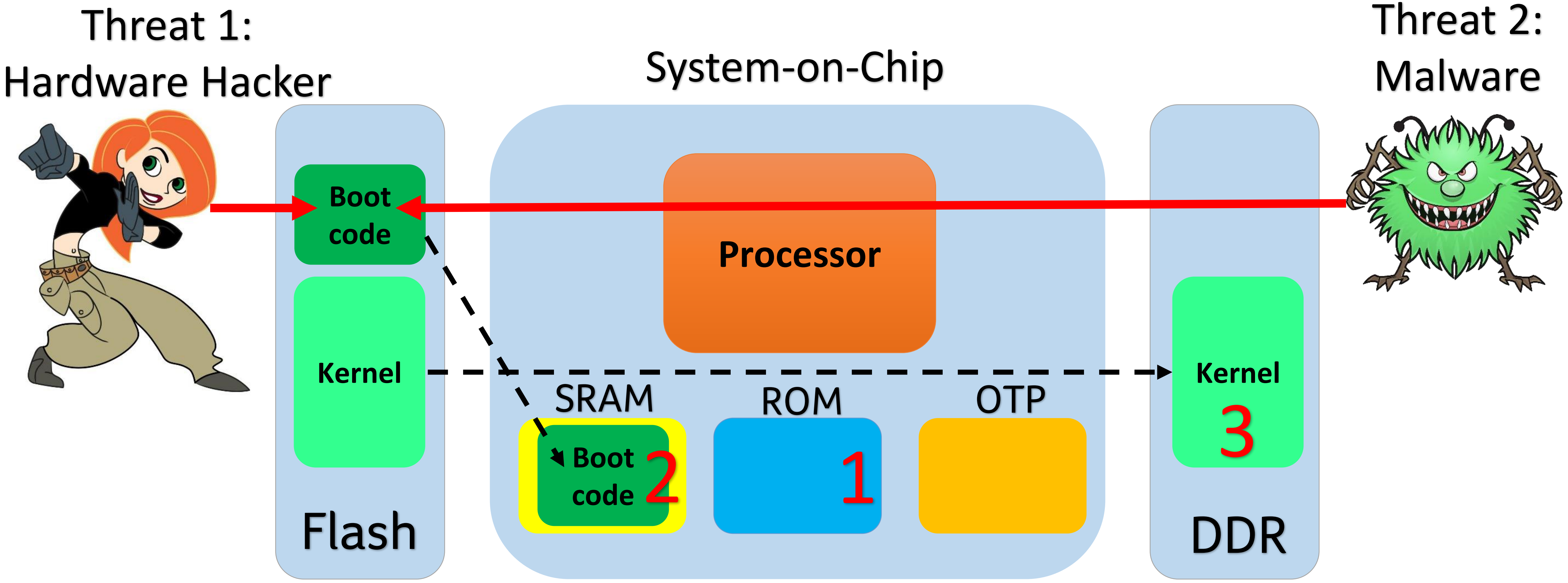
Why do we need secure boot?



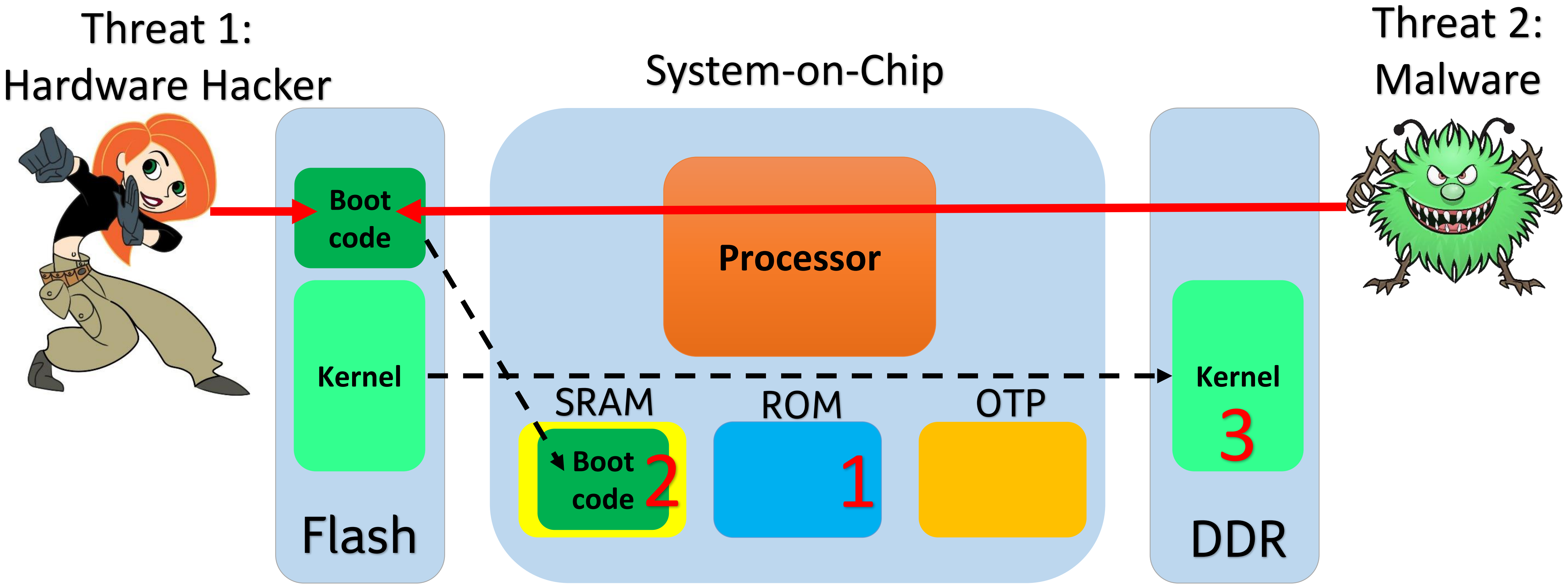
Why do we need secure boot?



Why do we need secure boot?



Why do we need secure boot?



Secure boot assures integrity of code/data in flash!

Secure boot

Secure boot

- Authentication of loading images

Secure boot

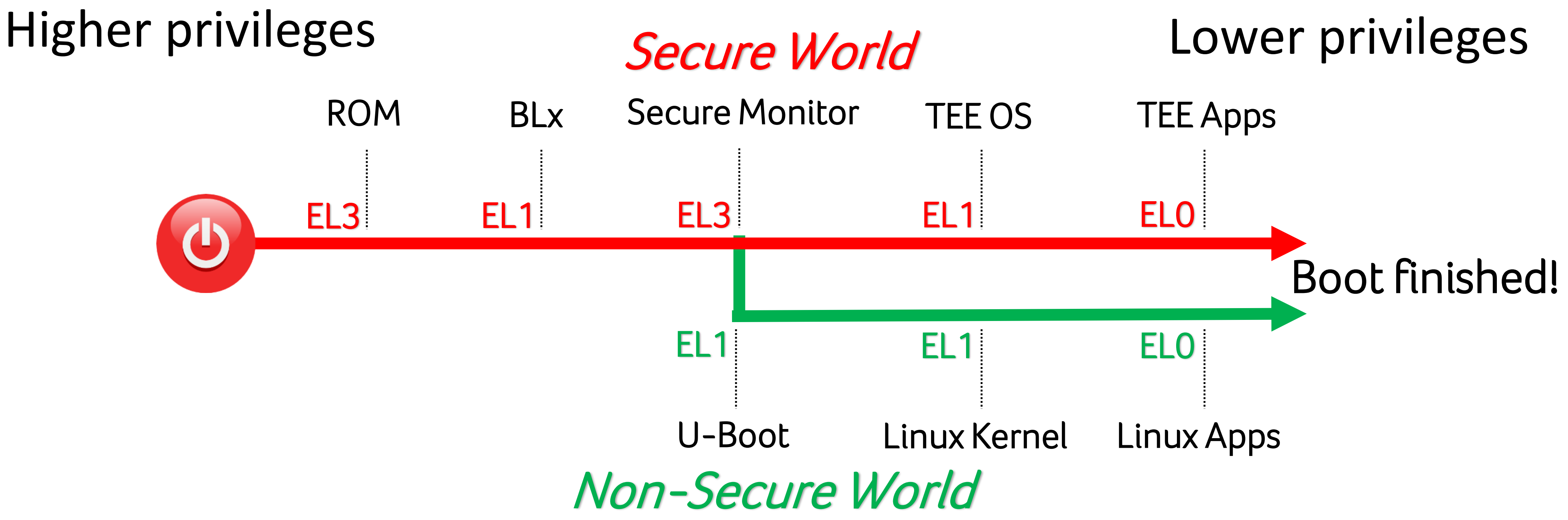
- Authentication of loading images
- Root of trust embedded in hardware
  - i.e. immutable code/data using read-only-memory (ROM)

Secure boot

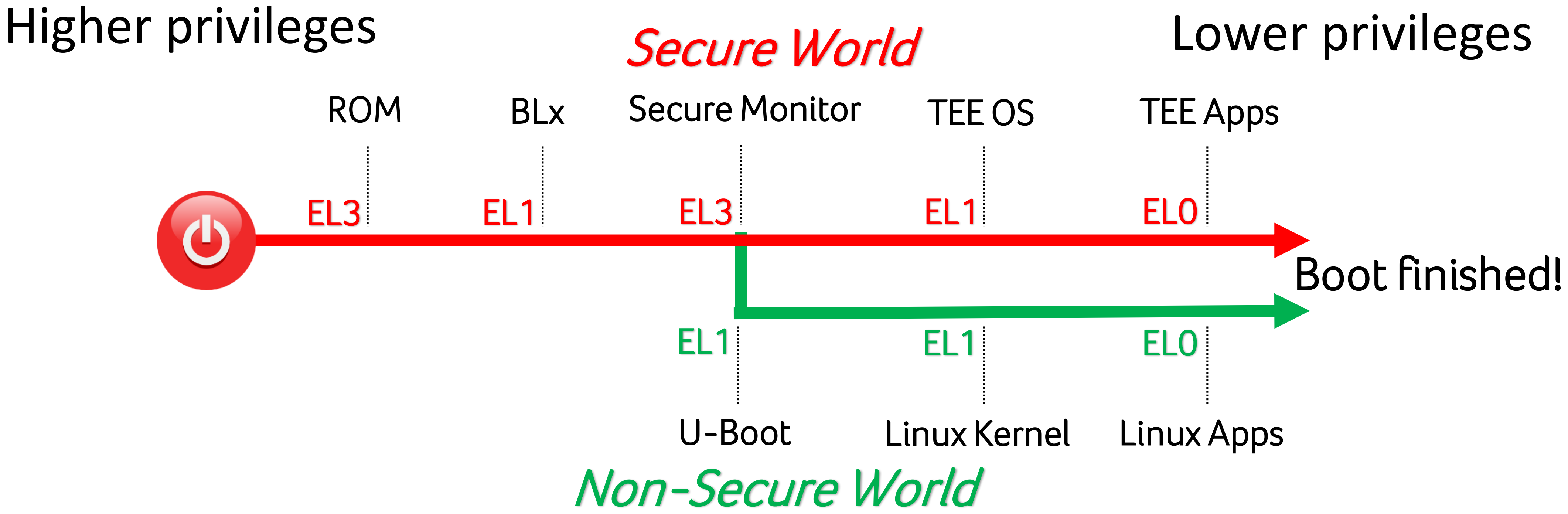
- Authentication of loading images
- Root of trust embedded in hardware
  - i.e. immutable code/data using read-only-memory (ROM)
- (optional): assure confidentiality by encrypting flash

The real world is more complex!

The real world is more complex!



The real world is more complex!



The chain can break at any stage. Earlier is better!

Breaking Secure Boot early

Breaking Secure Boot early

- Early boot stage run at the highest privilege
  - e.g. unrestricted access

Breaking Secure Boot early

- Early boot stage run at the highest privilege
  - e.g. unrestricted access
- Security features often not initialized yet
  - e.g. access control

Breaking Secure Boot early

- Early boot stage run at the highest privilege
  - e.g. unrestricted access
- Security features often not initialized yet
  - e.g. access control
- Access assets that are not accessible after a certain stage
  - e.g. ROM code and keys

Why use Fault Injection on Secure Boot?

Why use Fault Injection on Secure Boot?

- Usually a small code base

Why use Fault Injection on Secure Boot?

- Usually a small code base
- Limited attack surface

Why use Fault Injection on Secure Boot?

- Usually a small code base
- Limited attack surface
- Should be extensively reviewed

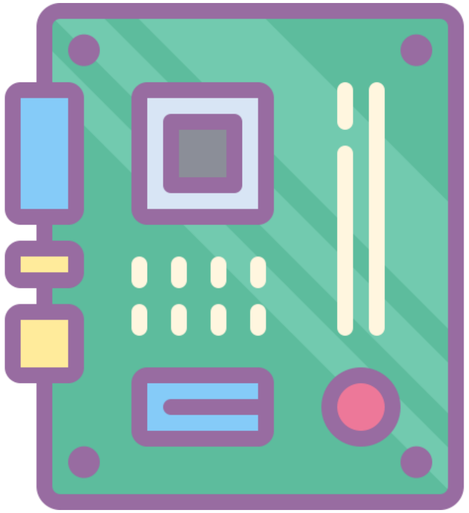
Why use Fault Injection on Secure Boot?

- Usually a small code base
- Limited attack surface
- Should be extensively reviewed

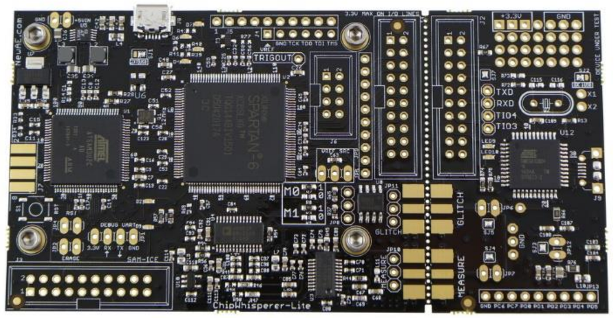
Software vulnerabilities not guaranteed to be present!

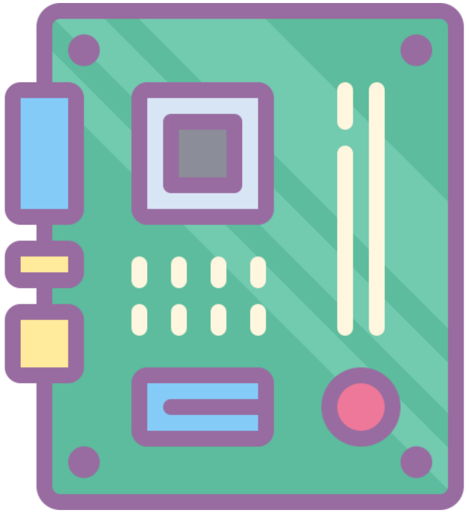
Fault injection setup

Fault injection setup

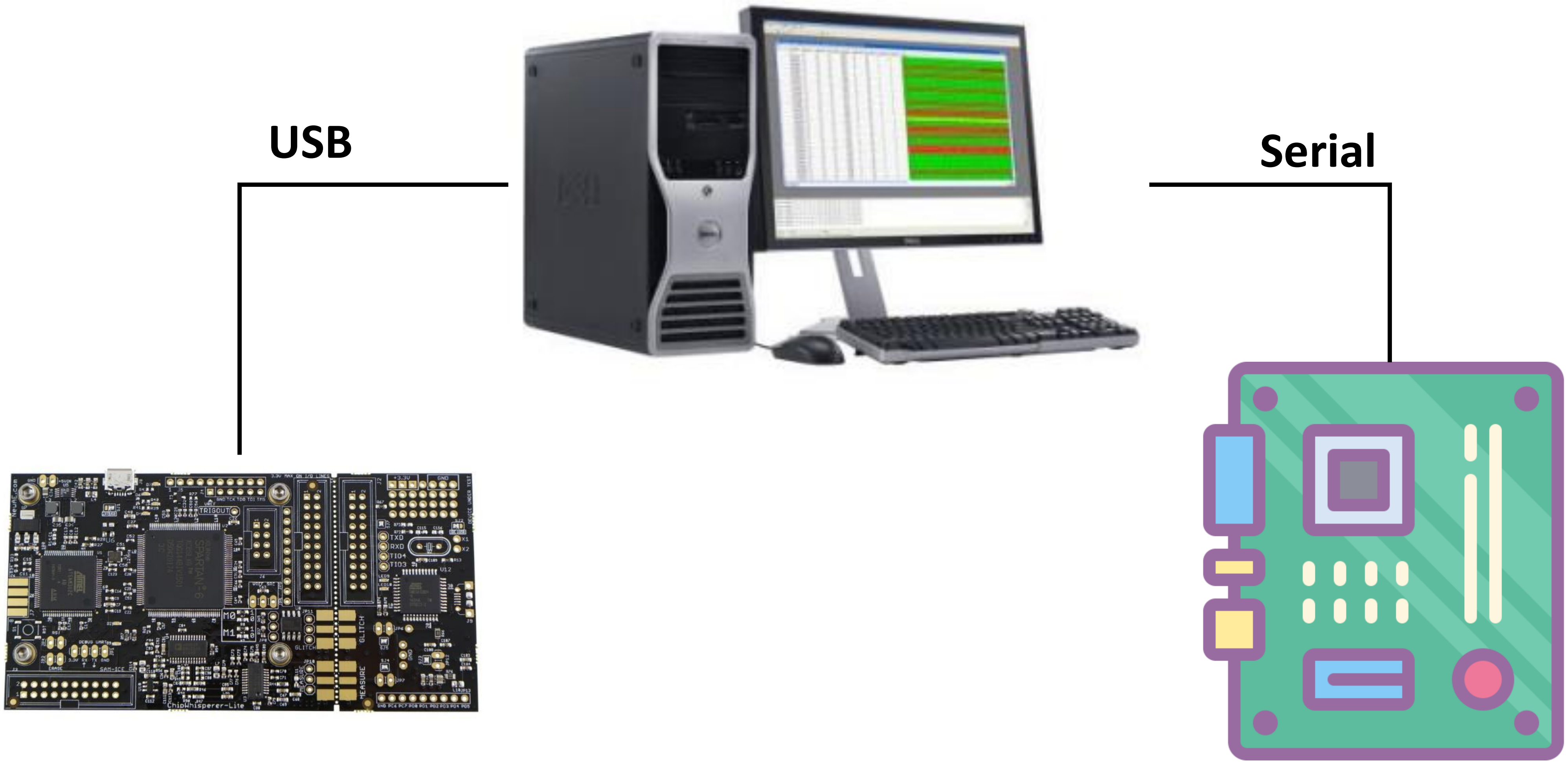


Fault injection setup

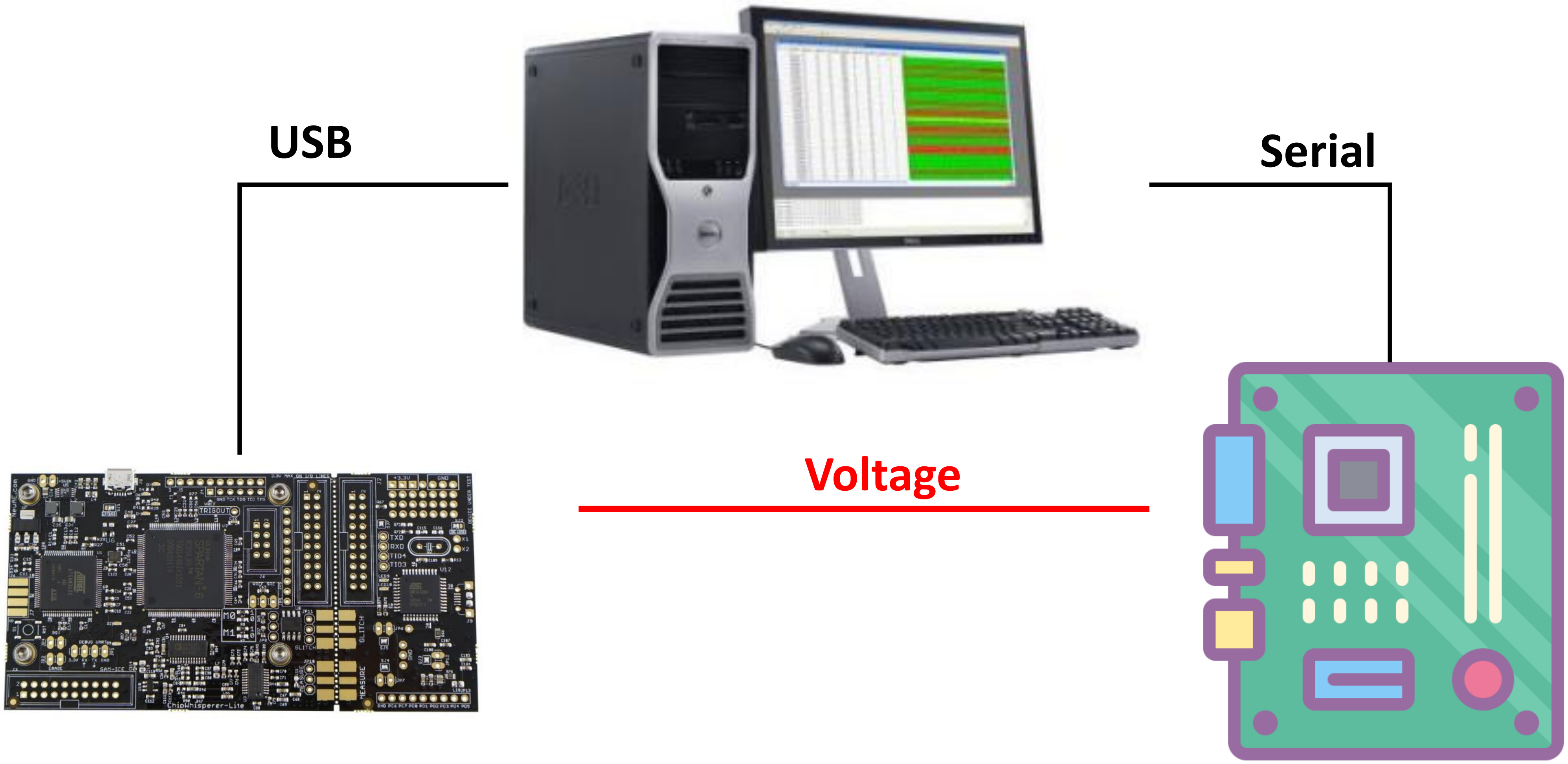




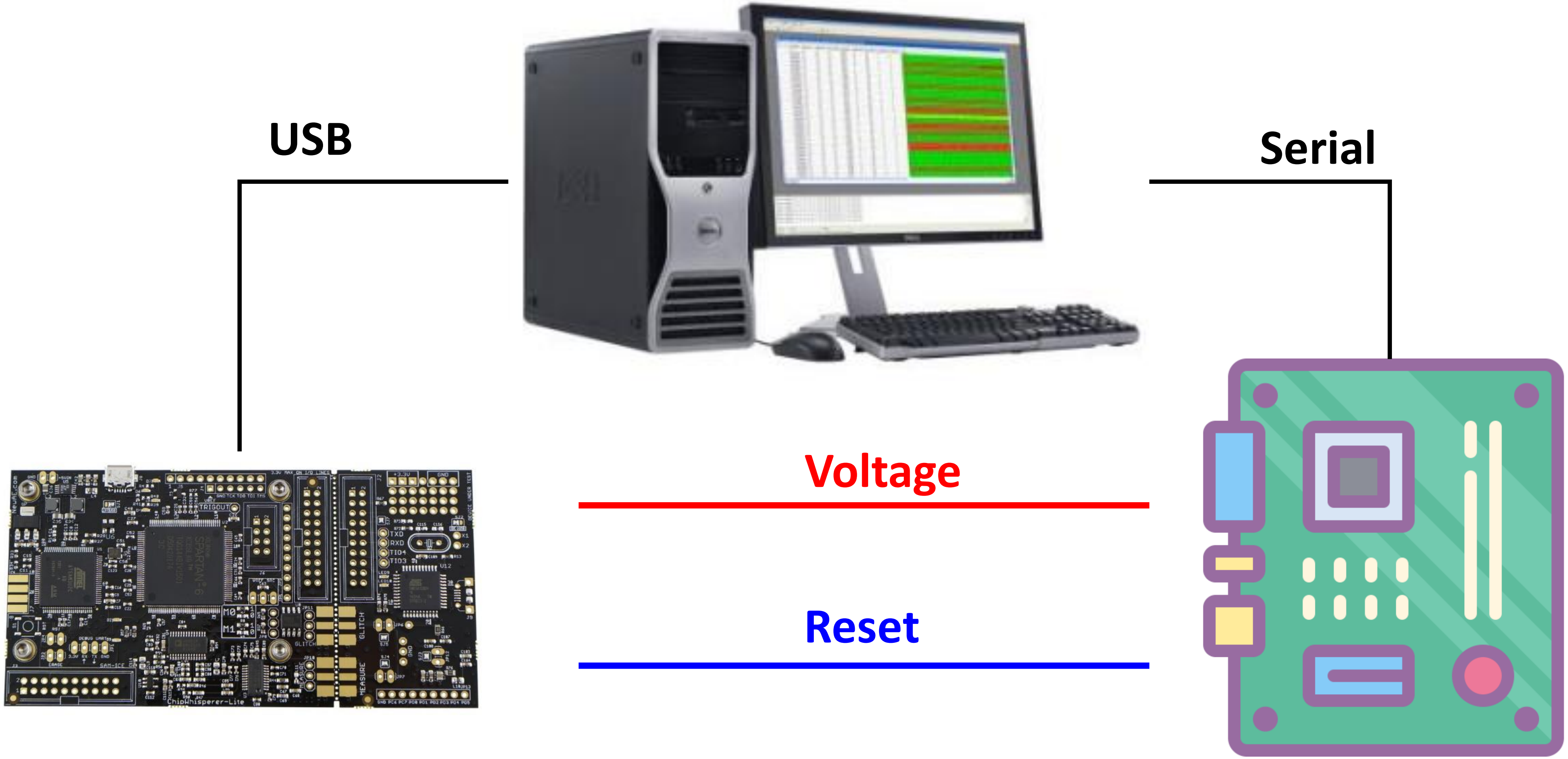
Fault injection setup



Fault injection setup



Fault injection setup



Signature verification

```
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
```

How do we attack?

Fault Injection Fault Model: *“Instruction skipping.”*

Fault Injection Fault Model: *"Instruction skipping."*

- Faults can cause "instructions not to be executed"

Fault Injection Fault Model: *“Instruction skipping.”*

- Faults can cause “instructions not to be executed”
- Inaccurate but sufficient to think about attacks (and defenses)

Fault Injection Fault Model: *“Instruction skipping.”*

- Faults can cause “instructions not to be executed”
- Inaccurate but sufficient to think about attacks (and defenses)
- Widely adopted by academia and industry

Fault Injection Fault Model: *“Instruction skipping.”*

- Faults can cause “instructions not to be executed”
- Inaccurate but sufficient to think about attacks (and defenses)
- Widely adopted by academia and industry
- Useful for affecting the code flow

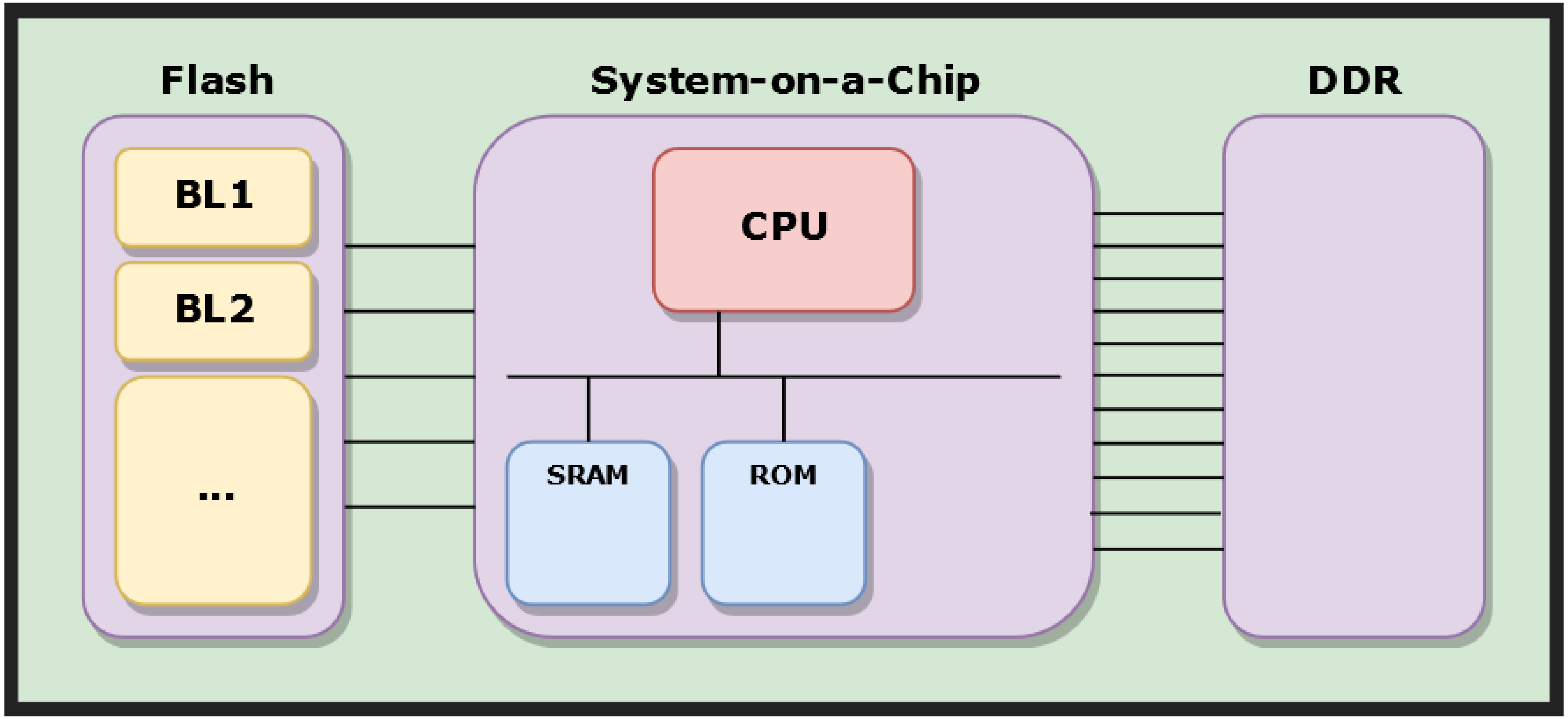
Fault Injection Fault Model: *“Instruction skipping.”*

- Faults can cause “instructions not to be executed”
- Inaccurate but sufficient to think about attacks (and defenses)
- Widely adopted by academia and industry
- Useful for affecting the code flow

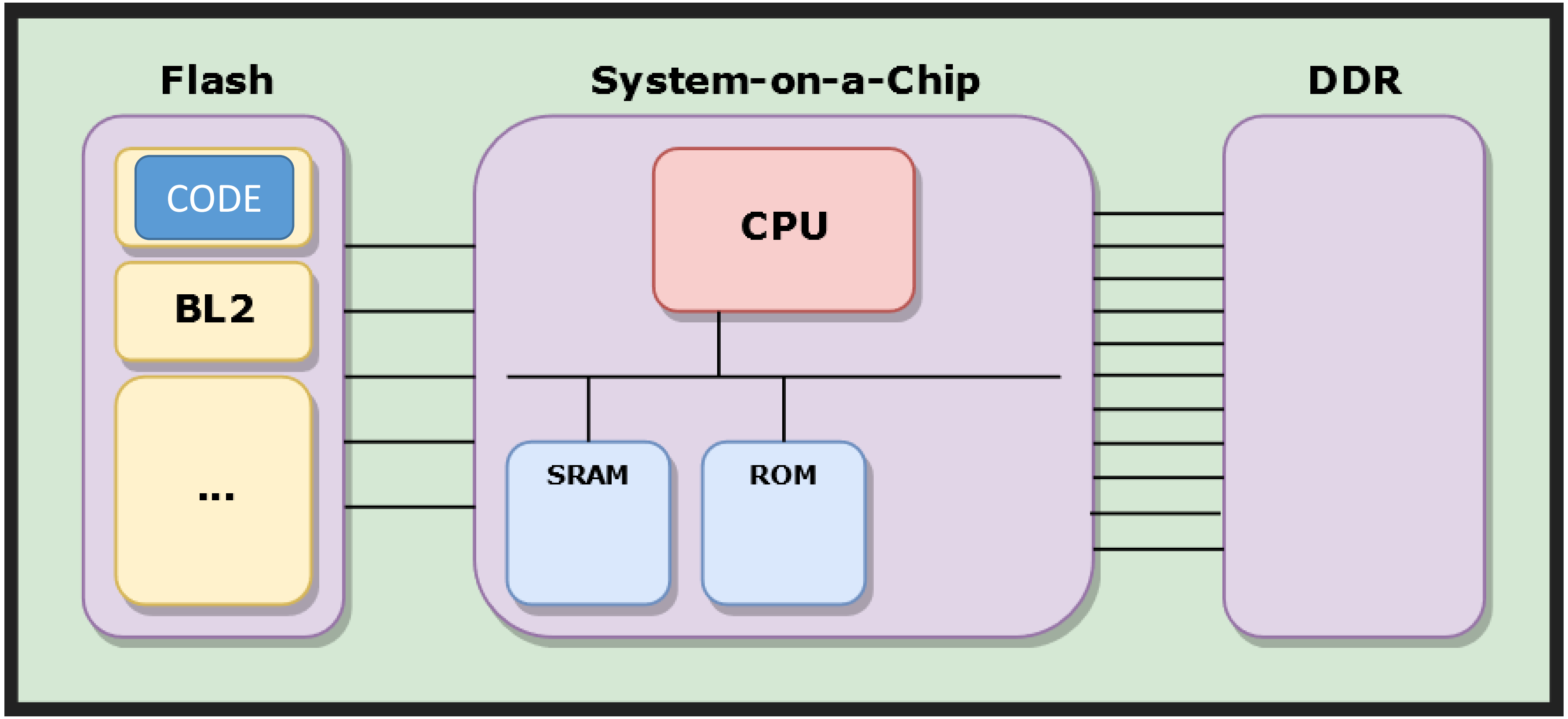
Cristofaro will dive much deeper into fault models!

Let’s use it to bypass Secure Boot!

Textbook Fault Injection Attack 1/4



Textbook Fault Injection Attack 2/4



Textbook Fault Injection Attack 3/4

```
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)) {
        while(1);                                   // 5. Glitch here!
    }
}

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

Textbook Fault Injection Attack 4/4

```
103ec:  e59f3020    ldr r3, [sp, #32]
103f0:  e5933000    ldr r3, [r3]
103f4:  e59f201c    ldr r2, [sp, #28]
103f8:  e1530002    cmp r3, r2                  // if conditional
103fc:  1a000000    bne 10404 <func+0x20>
10400:  eafffffe    b   10400 <func+0x1c>   // endless loop
10404:  ebffffef    bl  103c8 <jump>            // jump to next image
```

Textbook Fault Injection Attack 4/4

```
103ec:   e59f3020    ldr r3, [sp, #32]
103f0:   e5933000    ldr r3, [r3]
103f4:   e59f201c    ldr r2, [sp, #28]
103f8:   e1530002    cmp r3, r2                  // if conditional
103fc:   1a000000    bne 10404 <func+0x20>

10404:   ebffffef    bl  103c8 <jump>            // jump to next image
```

Textbook Fault Injection Attack 4/4

```
103ec:   e59f3020    ldr r3, [sp, #32]
103f0:   e5933000    ldr r3, [r3]
103f4:   e59f201c    ldr r2, [sp, #28]
103f8:   e1530002    cmp r3, r2                  // if conditional
103fc:   1a000000    bne 10404 <func+0x20>

10404:   ebffffef    bl  103c8 <jump>            // jump to next image
```

Complete bypass of Secure Boot!

Let’s attack something else…

Escalating Privileges in Linux using Voltage Fault Injection

Niek Timmers
Riscure – Security Lab
timmers@riscure.com / @tieknimmers

Cristofaro Mune
Embedded Security Consultant
pulsoid@icysilence.org / @pulsoid

Paper / Presentation / Video (2017)

Target

Target

- Fast and feature rich System-on-Chip (SoC)

Target

- Fast and feature rich System-on-Chip (SoC)
- ARM Cortex-A9 (AArch32) @ ~ 1 GHz

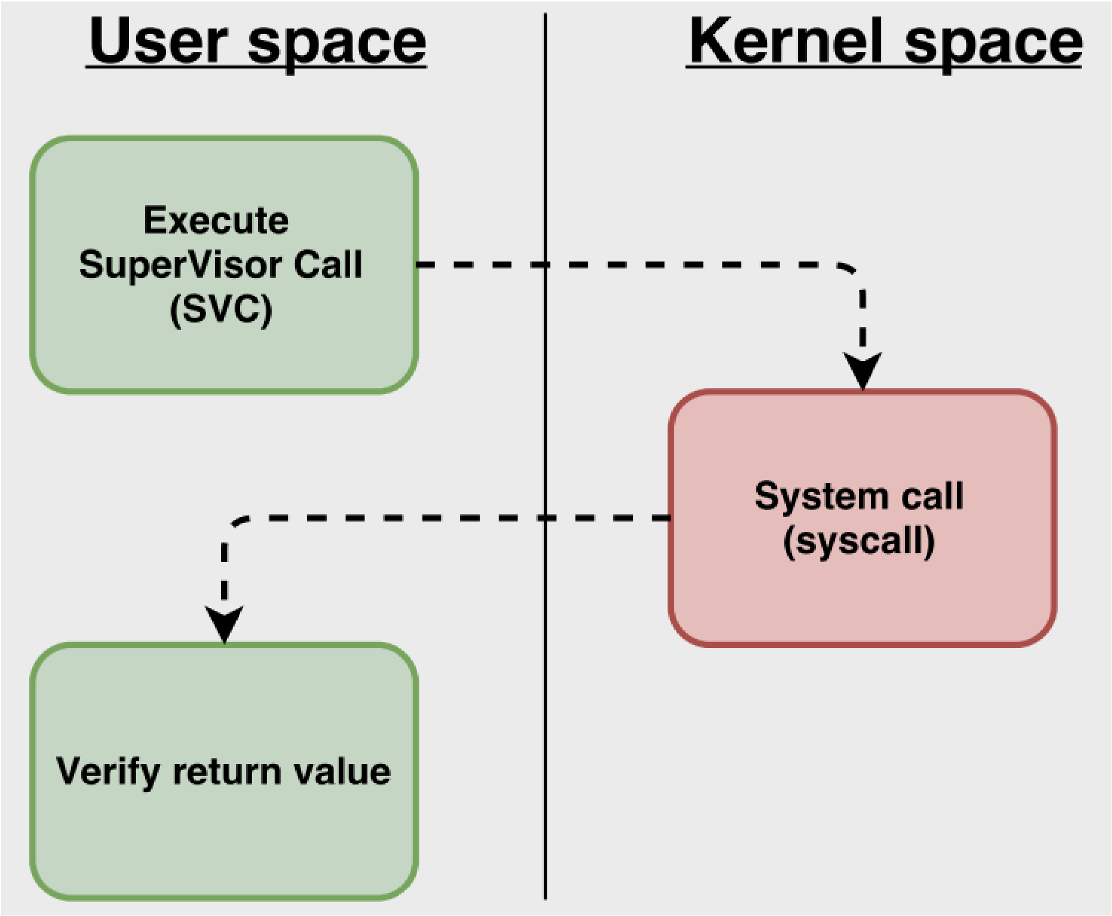
Target

- Fast and feature rich System-on-Chip (SoC)
- ARM Cortex-A9 (AArch32) @ ~ 1 GHz
- Ubuntu 14.04 LTS (fully patched)

Application vs Kernel

Application vs Kernel

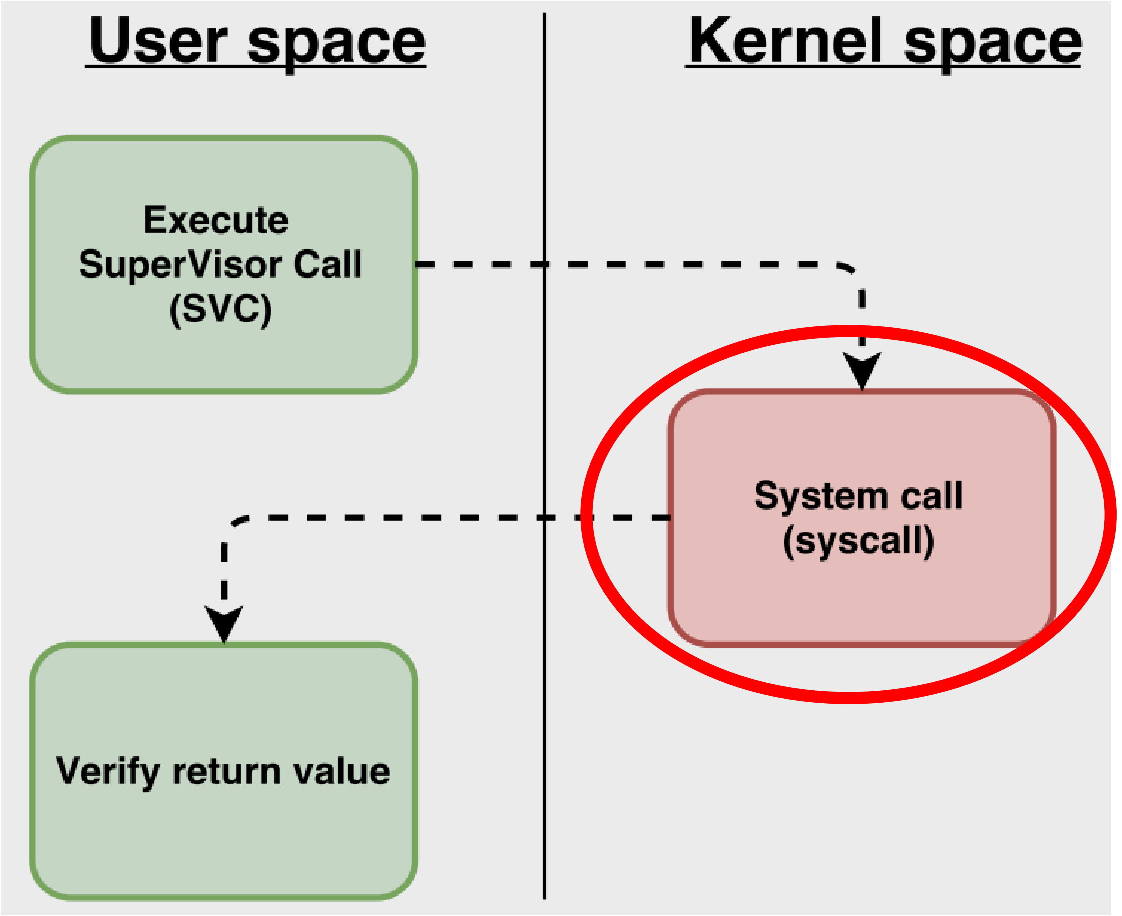
We assume the attacker can execute code as user without privileges



Application vs Kernel

We assume the attacker can execute code as user without privileges

The Kernel perform checks for security critical syscalls which will be the target for our attacks



Attack #1: Mapping of arbitrary memory

Attack #1: Mapping of arbitrary memory

1. Open */dev/mem* using ***open*** syscall from userspace process

Attack #1: Mapping of arbitrary memory

1. Open */dev/mem* using ***open*** syscall from userspace process

2. *Bypass checks performed by Linux kernel using a glitch*

Attack #1: Mapping of arbitrary memory

1. Open */dev/mem* using ***open*** syscall from userspace process

2. *Bypass checks performed by Linux kernel using a glitch*

3. Map arbitrary physical address in userspace

Attack #1: Mapping of arbitrary memory

1. Open */dev/mem* using ***open*** syscall from userspace process

2. *Bypass checks performed by Linux kernel using a glitch*

3. Map arbitrary physical address in userspace

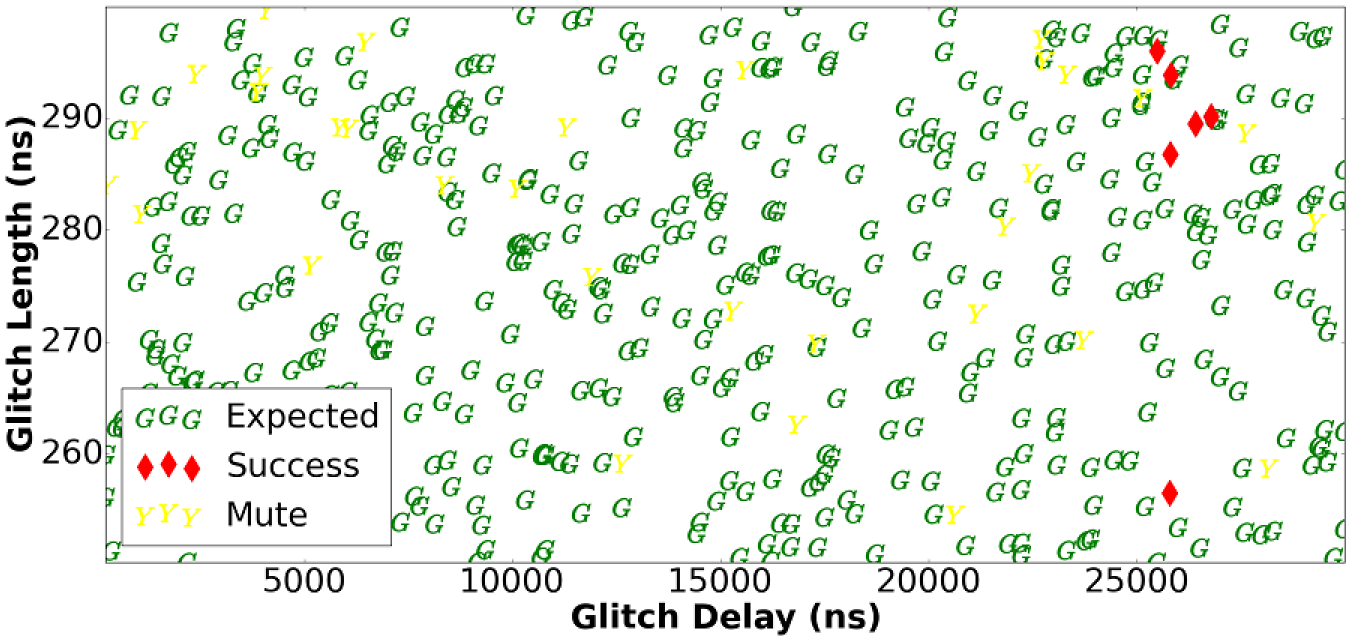
A successful glitch gives **(unrestricted) access to Kernel memory**!

Attack code for mapping arbitrary memory

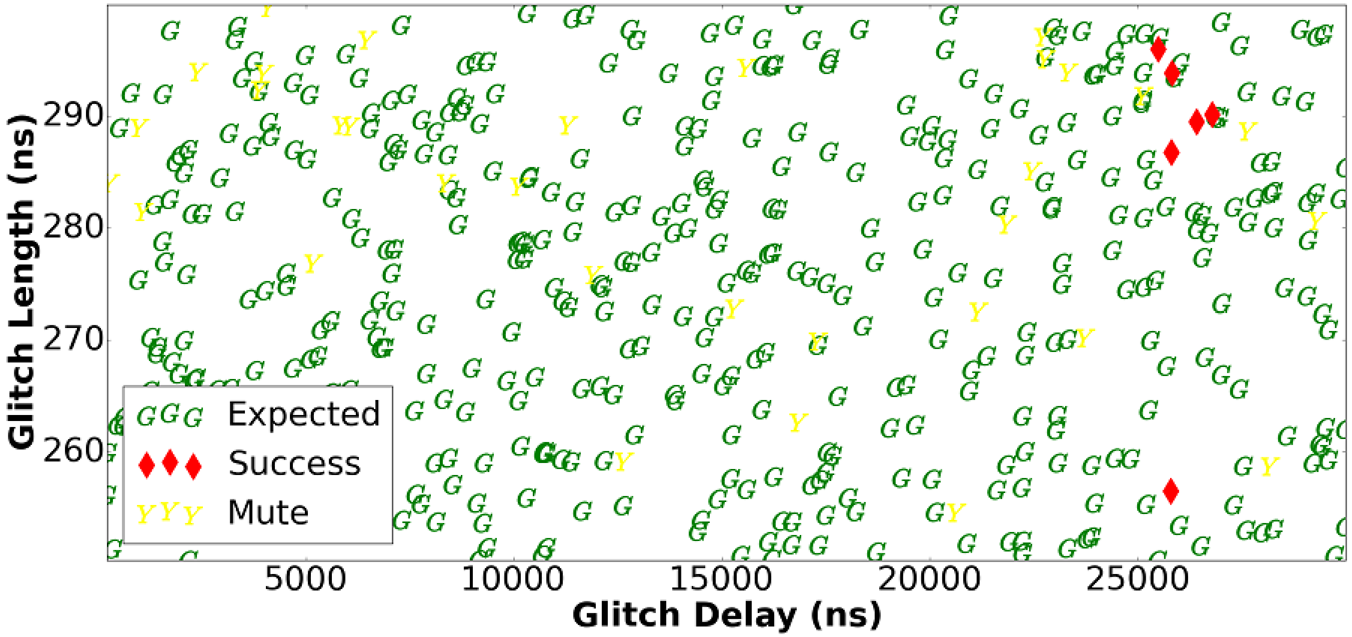
```
*(volatile unsigned int *)(trigger) = HIGH;

int mem = open("/dev/mem", O_RDWR | O_SYNC);

*(volatile unsigned int *)(trigger) = LOW;

if( mem == 4 ) {
  void * addr = mmap ( 0, ..., ..., mem, 0);
  printf("%08x\n", *(unsigned int *)(addr));
}
. . .
```

- Code is running in user space
- Linux syscall: sys_open (0x5)

Results for mapping arbitrary memory

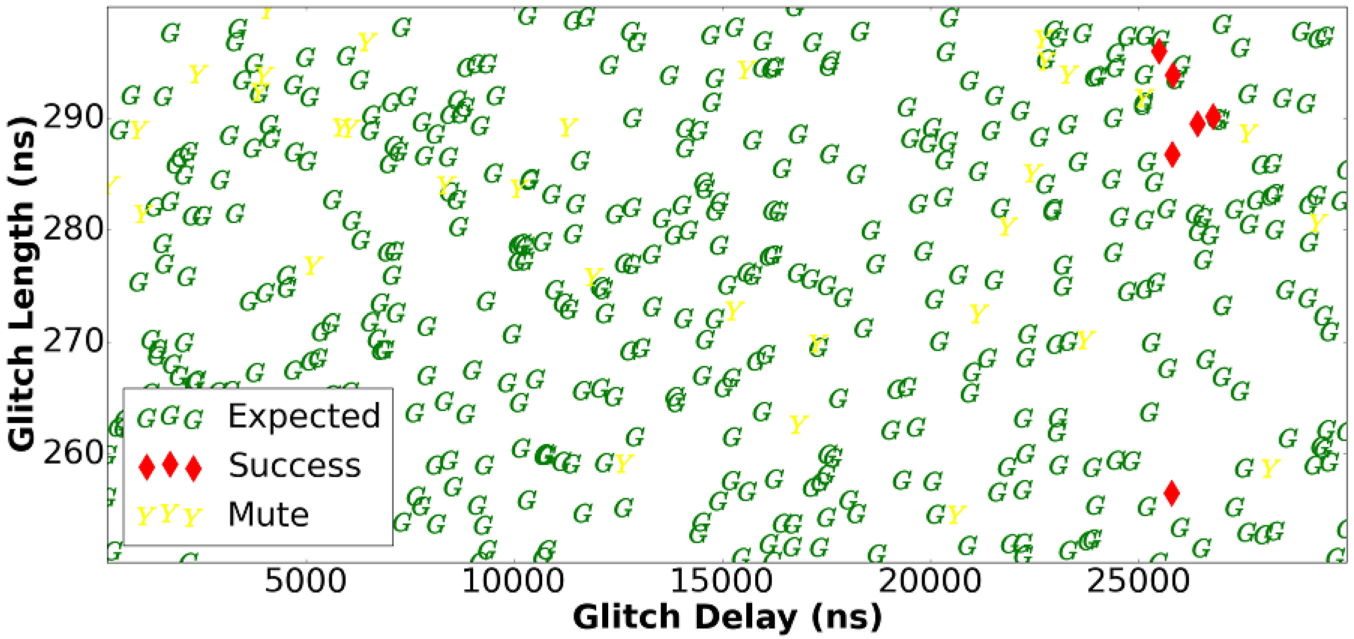


Results for mapping arbitrary memory



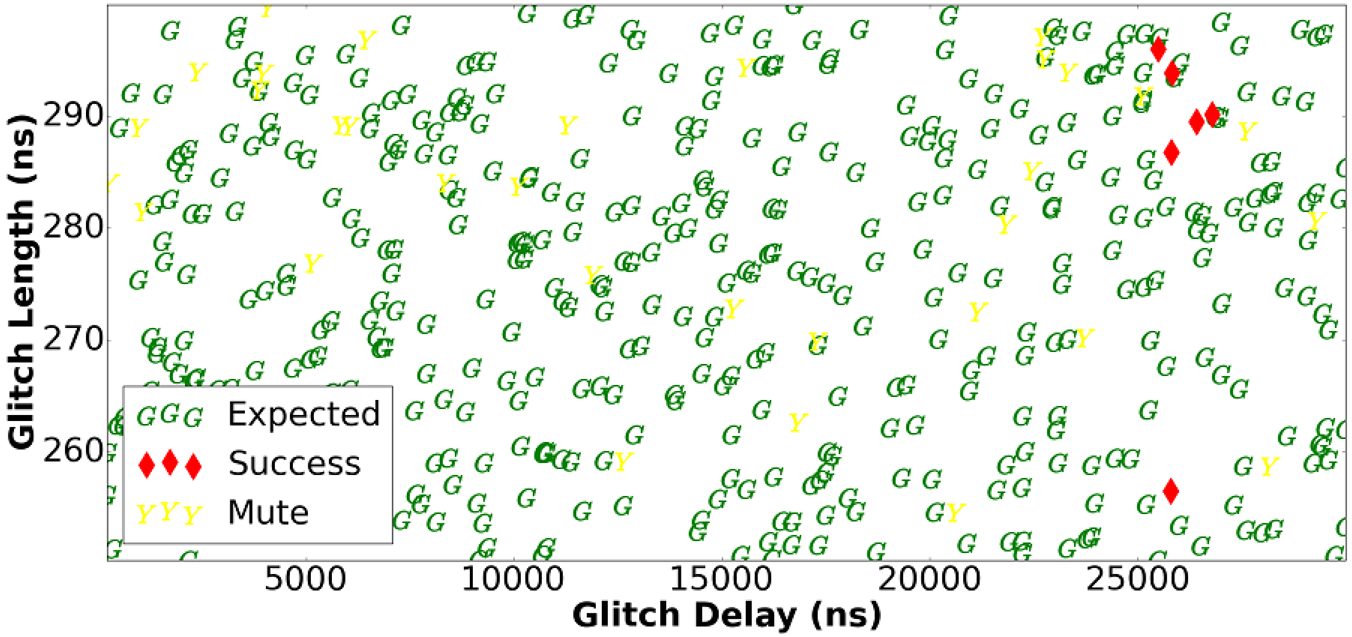
- Performed 22118 experiments in 17 hours

Results for mapping arbitrary memory



- Performed 22118 experiments in 17 hours
- Success rate between 25.5 µs and 26.8 µs: 0.53%

Results for mapping arbitrary memory



- Performed 22118 experiments in 17 hours
- Success rate between 25.5 µs and 26.8 µs: 0.53%
- ***Kernel "pwned" every 10 minutes***

What about popping a root shell directly?

Attack #2: Popping a root shell directly

Attack #2: Popping a root shell directly

1. Set all CPU registers to 0 to increase success probability

Attack #2: Popping a root shell directly

1. Set all CPU registers to 0 to increase success probability

2. Perform ***setresuid*** syscall to set process IDs to root

Attack #2: Popping a root shell directly

1. Set all CPU registers to 0 to increase success probability

2. Perform ***setresuid*** syscall to set process IDs to root

3. Bypass checks performed by Linux kernel using a glitch

Attack #2: Popping a root shell directly

1. Set all CPU registers to 0 to increase success probability
2. Perform ***setresuid*** syscall to set process IDs to root
3. Bypass checks performed by Linux kernel using a glitch
4. Execute shell using ***system*** function

Attack #2: Popping a root shell directly

1. Set all CPU registers to 0 to increase success probability
2. Perform ***setresuid*** syscall to set process IDs to root
3. Bypass checks performed by Linux kernel using a glitch
4. Execute shell using ***system*** function

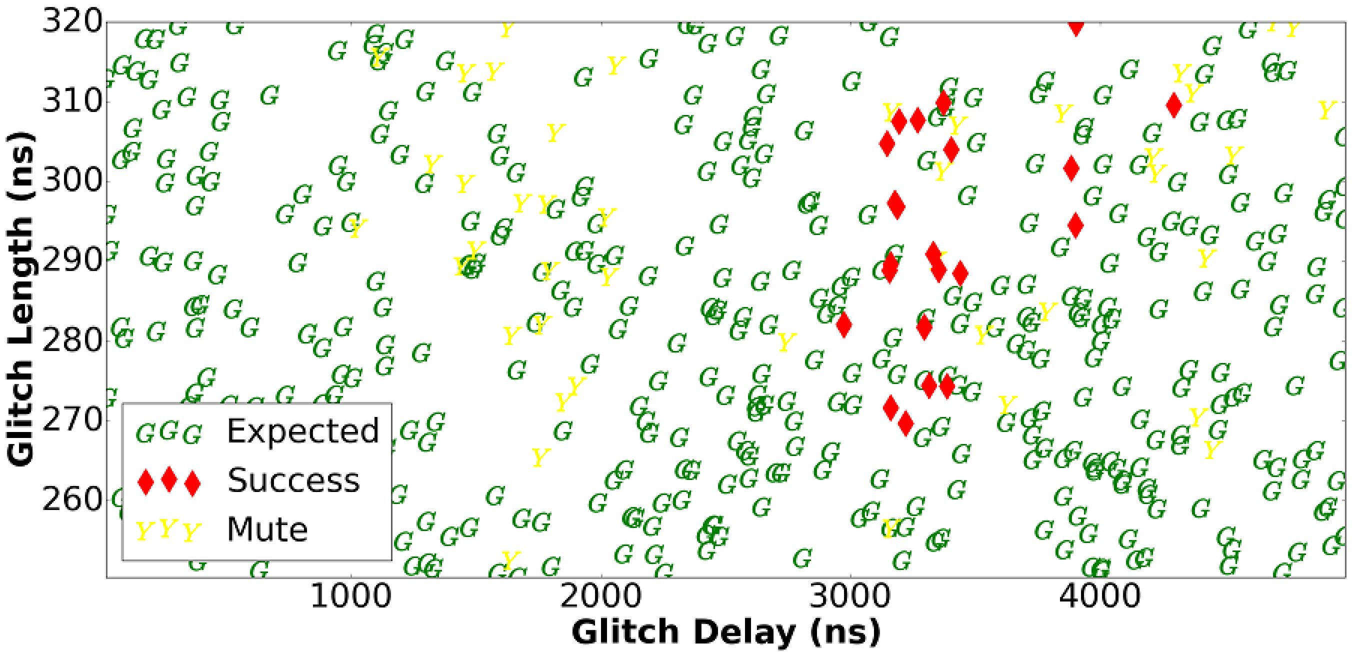
A successful glitch gives a **shell with root privileges**!

Attack code for popping a root shell directly

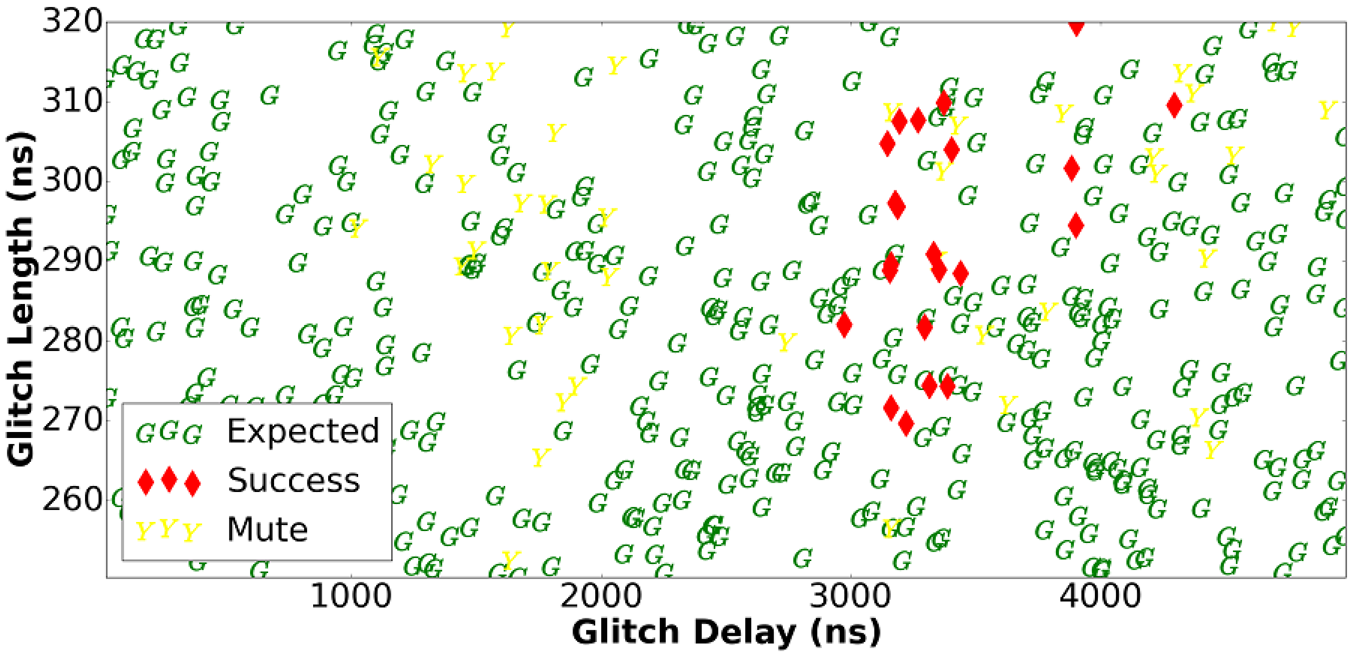
```
*(volatile unsigned int *)(trigger) = HIGH;

asm volatile (
  "movw r12, #0x0;" // Repeat for other
  "movt r12, #0x0;" // unused registers
  . . .
  "mov r7, #0xd0;"  // setresuid syscall
  "swi #0;"         // Linux kernel takes over

  "mov %[ret], r0;" // Store return value in r0
  : [ret] "=r" (ret) : : "r0", . . ., "r12" )

*(volatile unsigned int *)(trigger) = LOW;

if(ret == 0) { system("/bin/sh"); }
```

- Code is running in user space
- Linux syscall: setresuid (0xd0)

Results for popping a root shell directly

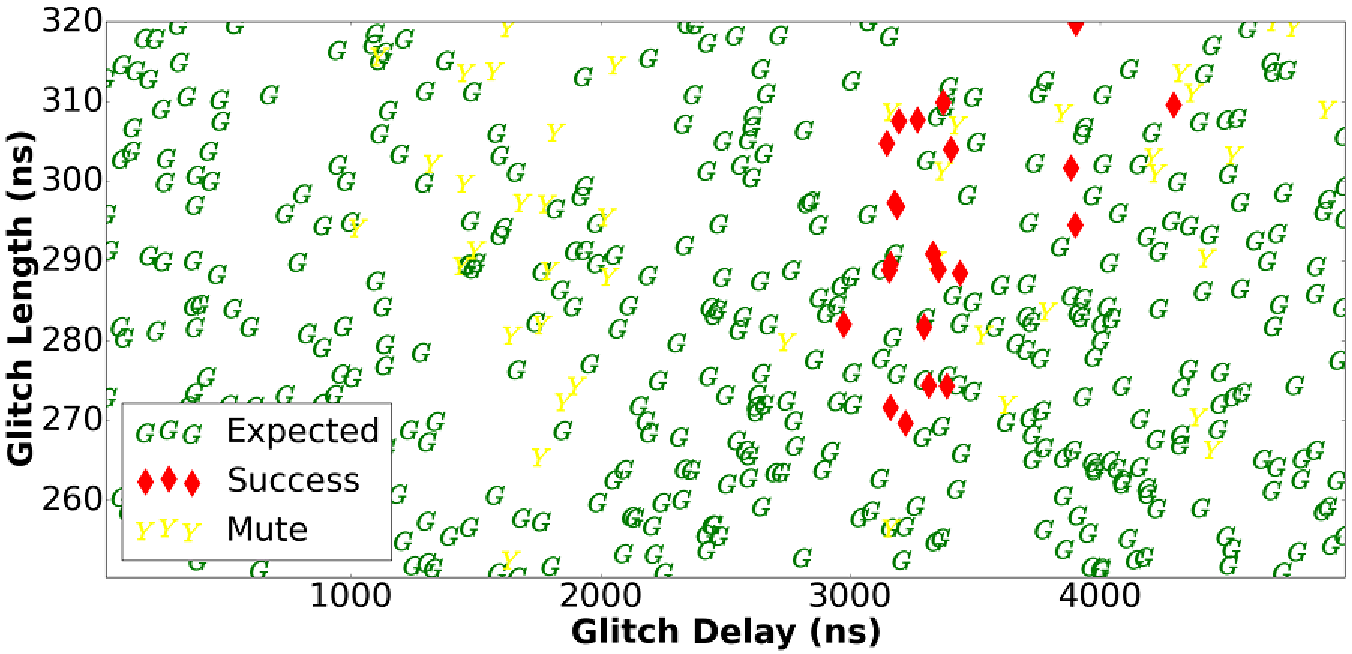


Results for popping a root shell directly



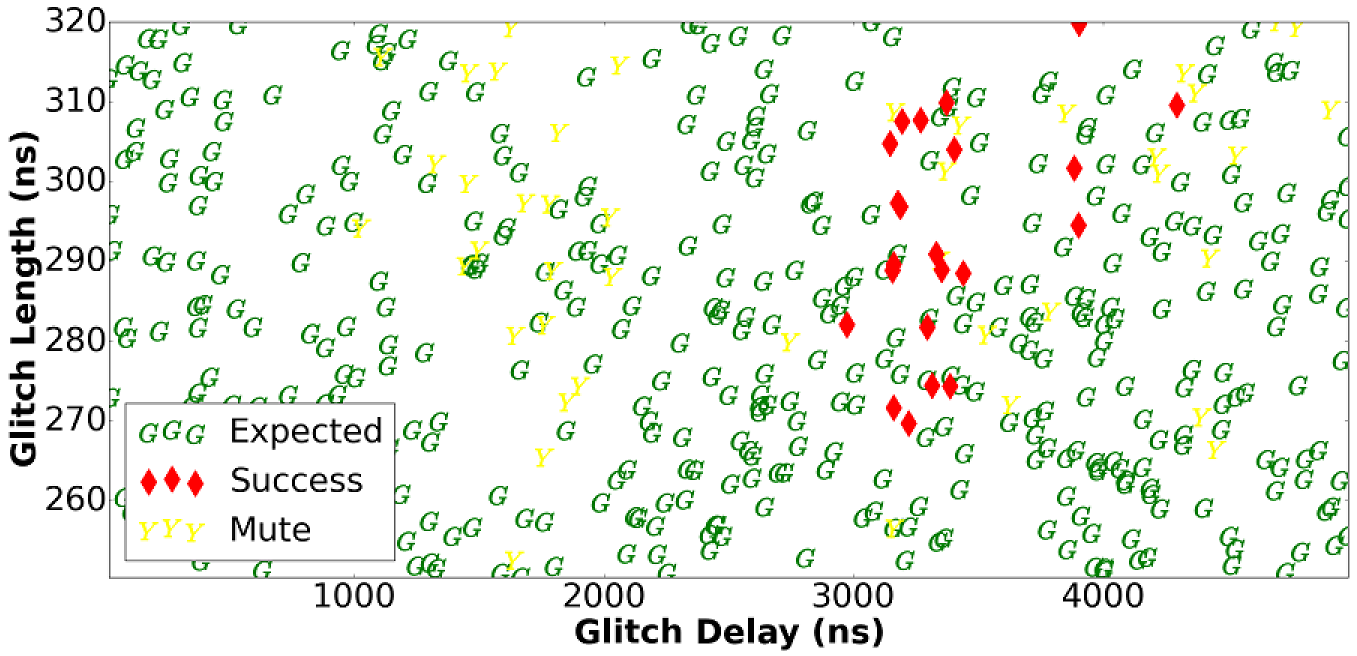
- Performed 18968 experiments in 21 hours

Results for popping a root shell directly



- Performed 18968 experiments in 21 hours
- Success rate between 3.14 µs and 3.44 µs: 1.3%

Results for popping a root shell directly



- Performed 18968 experiments in 21 hours
- Success rate between 3.14 µs and 3.44 µs: 1.3%
- ***Root shell "popped" every 5 minutes***

What about controlling the Program Counter (PC) in Kernel mode directly?!

What about controlling the Program Counter (PC) in Kernel mode directly?!

Controlling PC on ARM using Fault Injection

Niek Timmers
Riscure – Security Lab
Delft, The Netherlands
timmers@riscure.com

Albert Spruyt
Riscure – Security Lab
Delft, The Netherlands
spruyt@riscure.com

Marc Witteman
Riscure – Security Lab
Delft, The Netherlands
witteman@riscure.com

Paper / Presentation (2016)

Attack #3: Controlling PC directly

Attack #3: Controlling PC directly

1. Set all registers to a specific value (e.g. 0x41414141)

Attack #3: Controlling PC directly

1. Set all registers to a specific value (e.g. 0x41414141)

2. Execute random Linux system calls

Attack #3: Controlling PC directly

1. Set all registers to a specific value (e.g. 0x41414141)

2. Execute random Linux system calls

3. Load an controlled value into the PC register using a glitch

Attack #3: Controlling PC directly

1. Set all registers to a specific value (e.g. 0x41414141)

2. Execute random Linux system calls

3. Load an controlled value into the PC register using a glitch

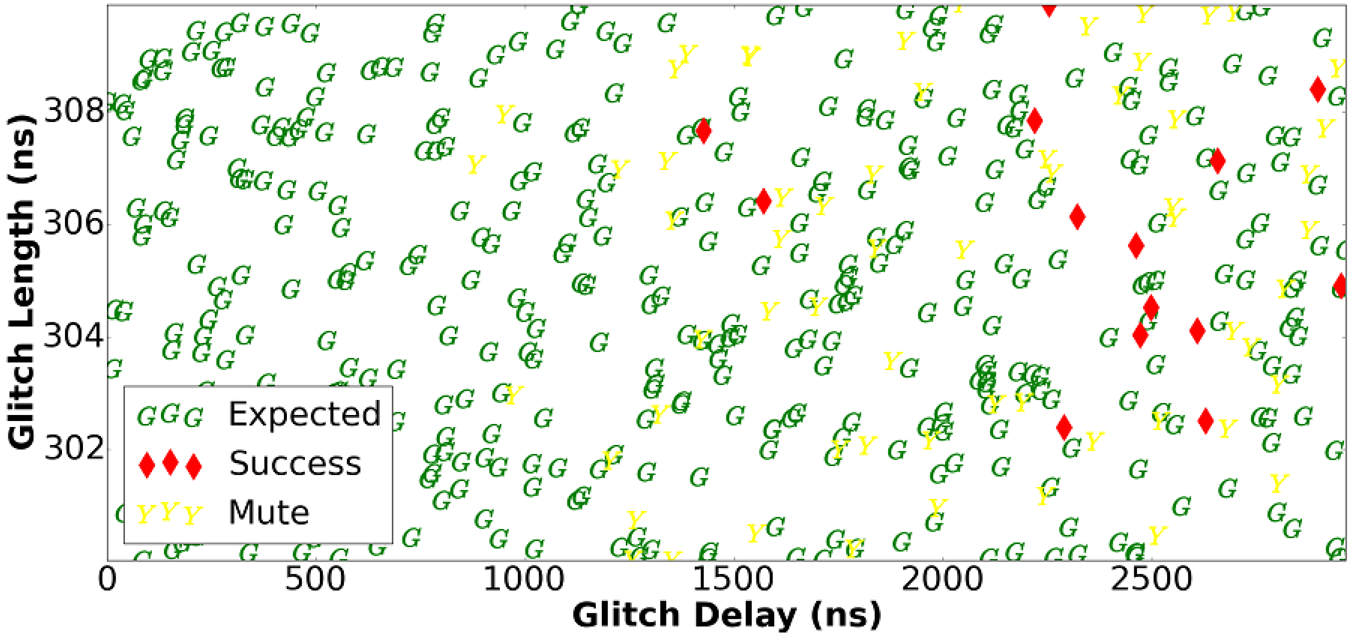
A successful glitch will **hijack the control flow**!

Attack code for controlling PC directly

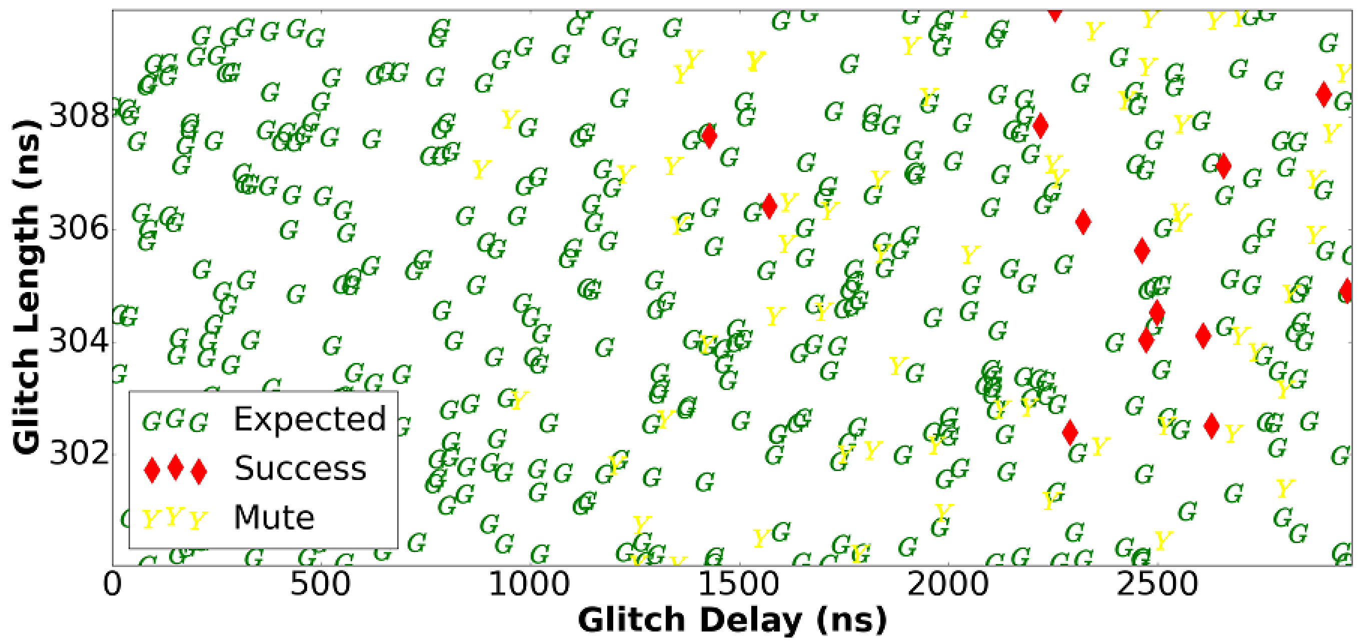
```
. . .

*(volatile unsigned int *)(trigger) = HIGH;

volatile (
  "movw r12, #0x4141;" // Repeat for other
  "movt r12, #0x4141;" // unused registers
  . . .
  "mov r7, %[rand];" // Random syscall nr
  "swi #0;"          // Linux kernel takes over
  . . .

*(volatile unsigned int *)(trigger) = LOW;
. . .
```

- Code running in userspace
- Linux syscall: initially random
- We found **getgroups** and **prctl** to be more effective

Results for controlling PC directly

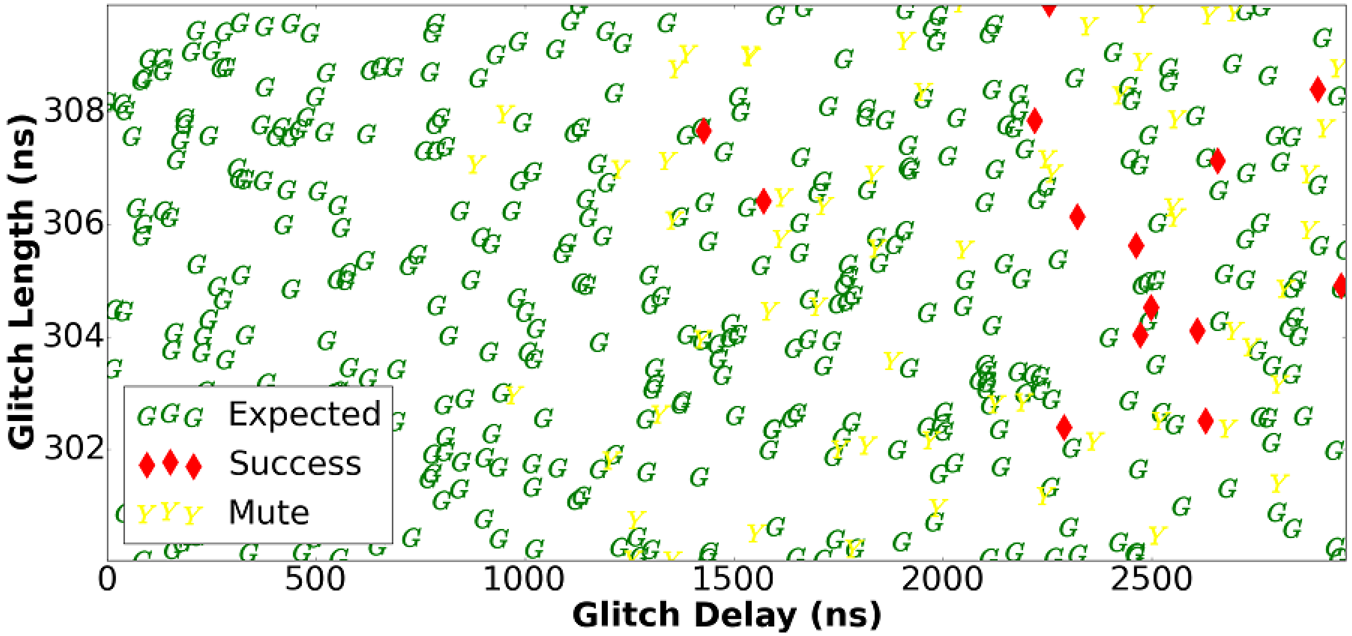


Results for controlling PC directly



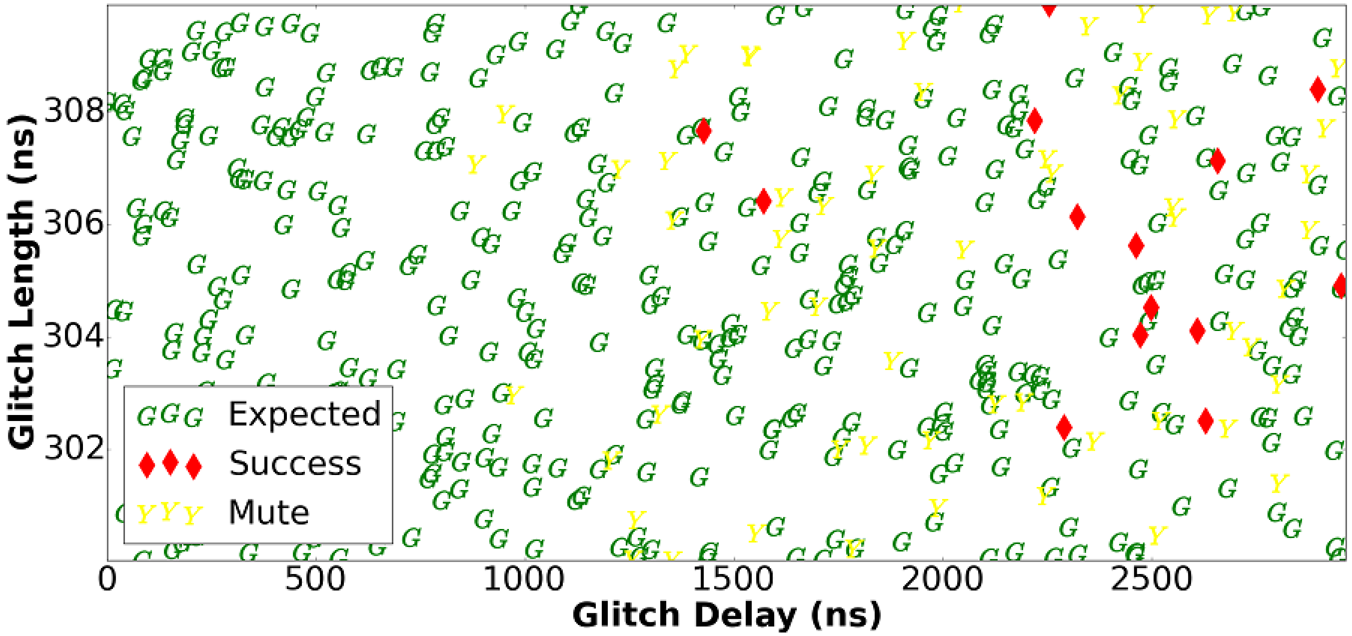
- Performed 12705 experiments in 14 hours

Results for controlling PC directly



- Performed 12705 experiments in 14 hours
- Success rate between 2.2 µs and 2.65 µs: 0.63%

Results for controlling PC directly



- Performed 12705 experiments in 14 hours
- Success rate between 2.2 µs and 2.65 µs: 0.63%
- ***Control of PC in Kernel mode gained every 10 minutes***

This is magic! Why does this work?

This is magic! Why does this work?

You will hear that in the **next** sessions…

To conclude…

To conclude…

- Fault injection practical and available to the masses
  (it will not go away)

To conclude…

- Fault injection practical and available to the masses
  (it will not go away)

- They can easily subvert typical software security models
  (adjust your threat model)

To conclude…

- Fault injection practical and available to the masses
  (it will not go away)

- They can easily subvert typical software security models
  (adjust your threat model)

- Most standard devices are vulnerable
  (factor in countermeasures from the start)

Thank you! Any questions?!