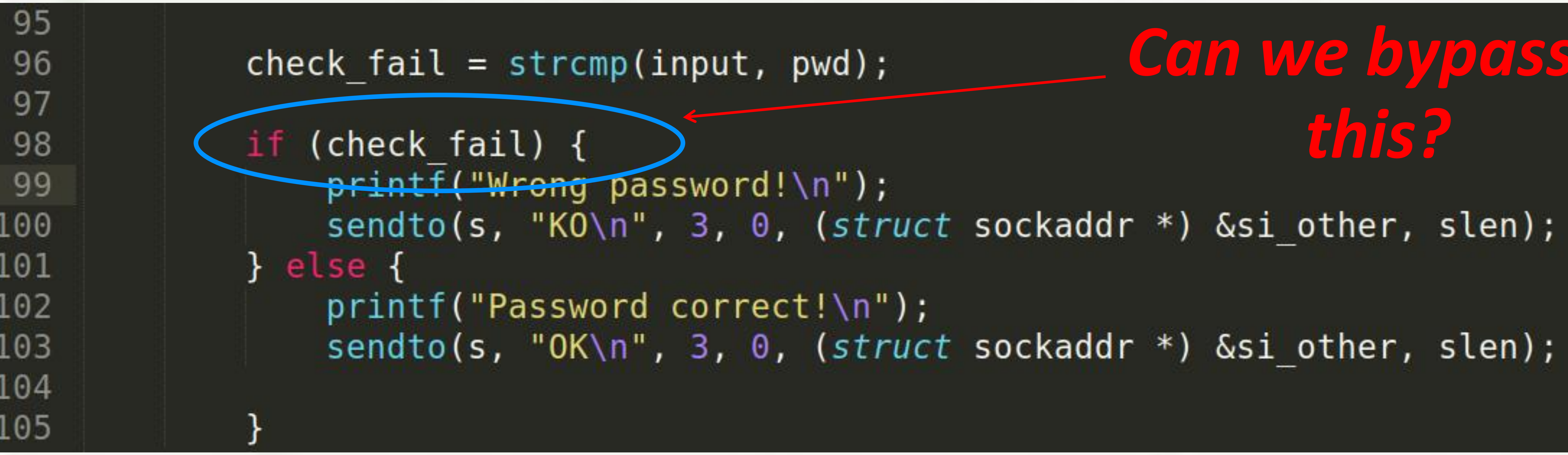
Advancing FI attacks: Fault Models opportunities

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

Password check

A simple password check

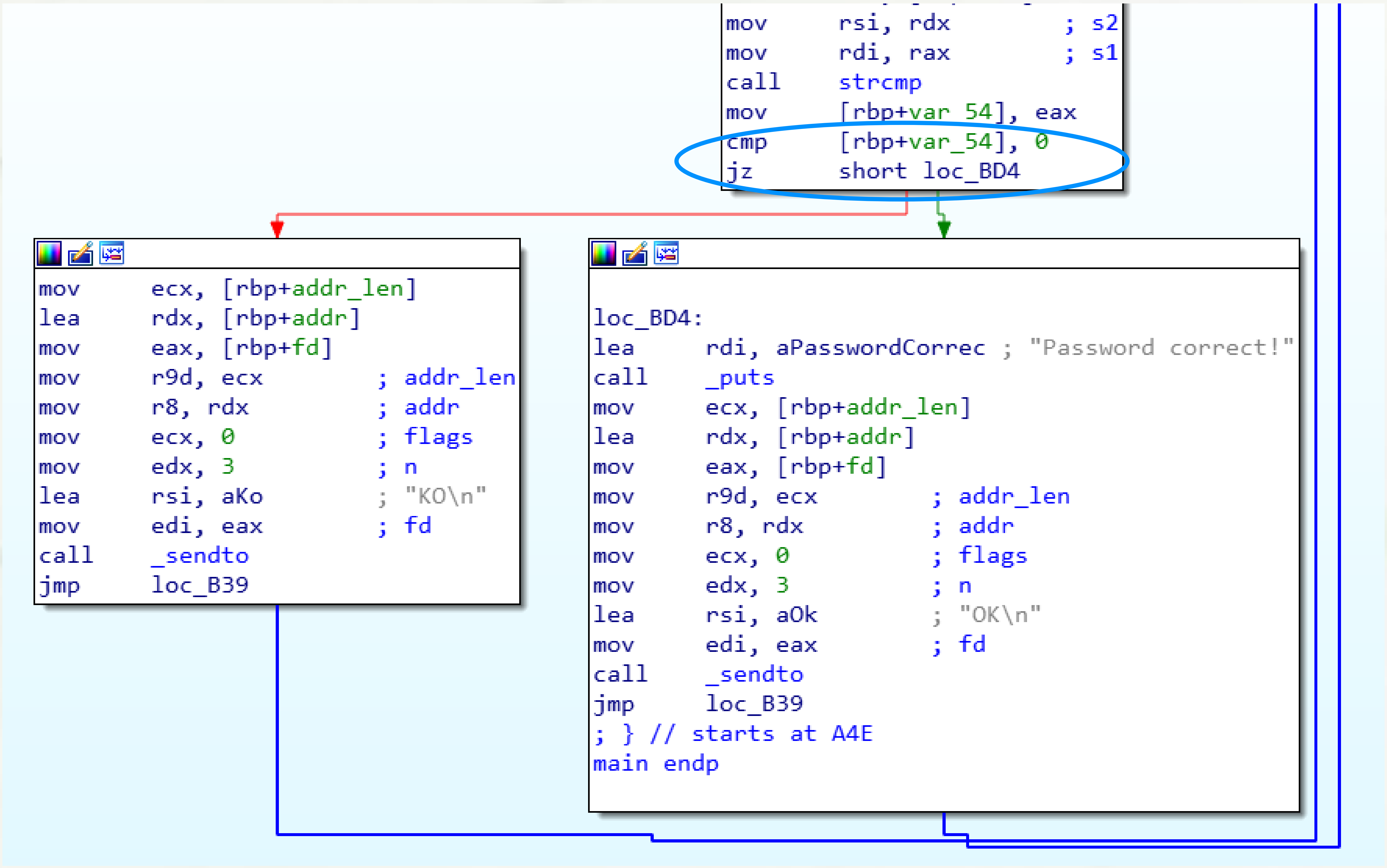


```
    check_fail = strcmp(input, pwd);

    if (check_fail) {
        printf("Wrong password!\n");
        sendto(s, "KO\n", 3, 0, (struct sockaddr *) &si_other, slen);
    } else {
        printf("Password correct!\n");
        sendto(s, "OK\n", 3, 0, (struct sockaddr *) &si_other, slen);

    }
```

Can we bypass this?

What do you think?

A better view



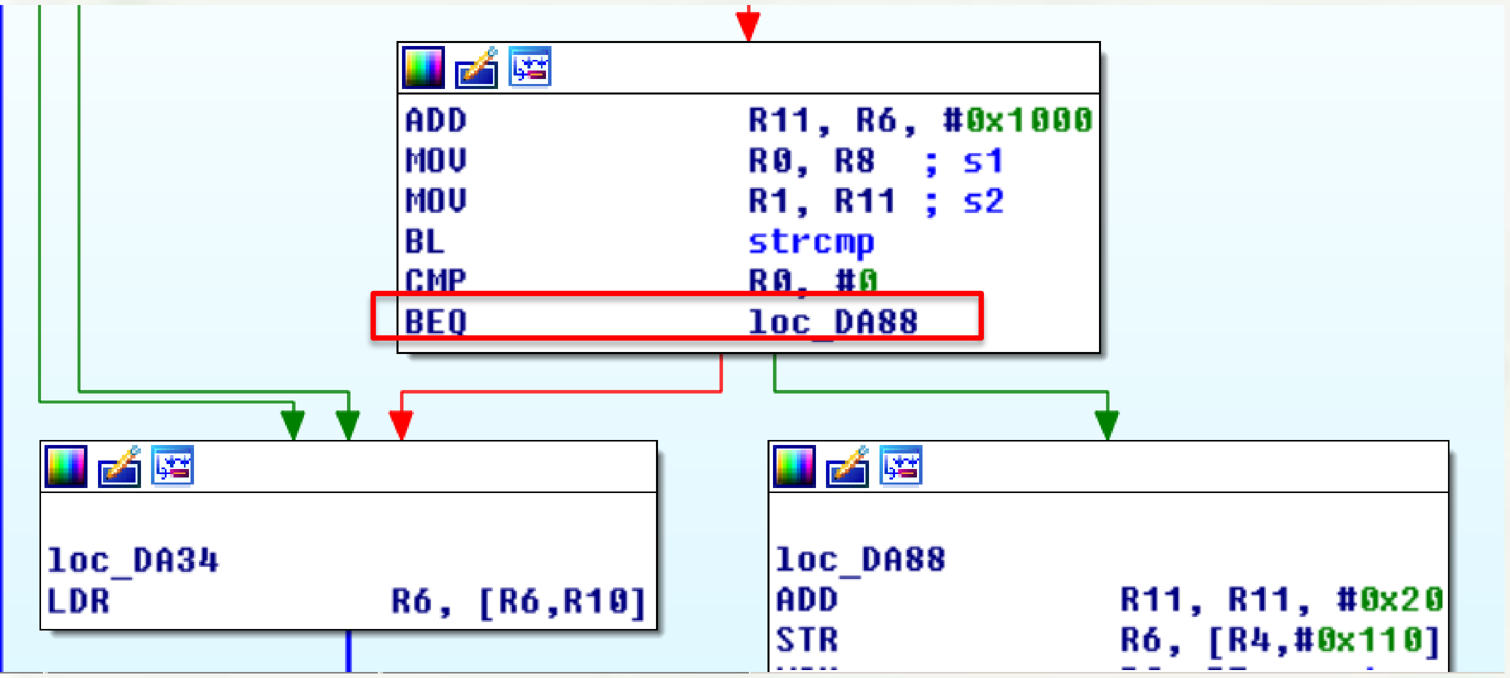
```
mov     rsi, rdx            ; s2
mov     rdi, rax            ; s1
call    strcmp
mov     [rbp+var_54], eax
cmp     [rbp+var_54], 0
jz      short loc_BD4
```

```
mov     ecx, [rbp+addr_len]
lea     rdx, [rbp+addr]
mov     eax, [rbp+fd]
mov     r9d, ecx            ; addr_len
mov     r8, rdx             ; addr
mov     ecx, 0              ; flags
mov     edx, 3              ; n
lea     rsi, aKo            ; "KO\n"
mov     edi, eax            ; fd
call    _sendto
jmp     loc_B39
```

```
loc_BD4:
lea     rdi, aPasswordCorrec ; "Password correct!"
call    _puts
mov     ecx, [rbp+addr_len]
lea     rdx, [rbp+addr]
mov     eax, [rbp+fd]
mov     r9d, ecx            ; addr_len
mov     r8, rdx             ; addr
mov     ecx, 0              ; flags
mov     edx, 3              ; n
lea     rsi, aOk            ; "OK\n"
mov     edi, eax            ; fd
call    _sendto
jmp     loc_B39
; } // starts at A4E
main endp
```

Traditional fault models

Control flow corruption
by skipping instructions

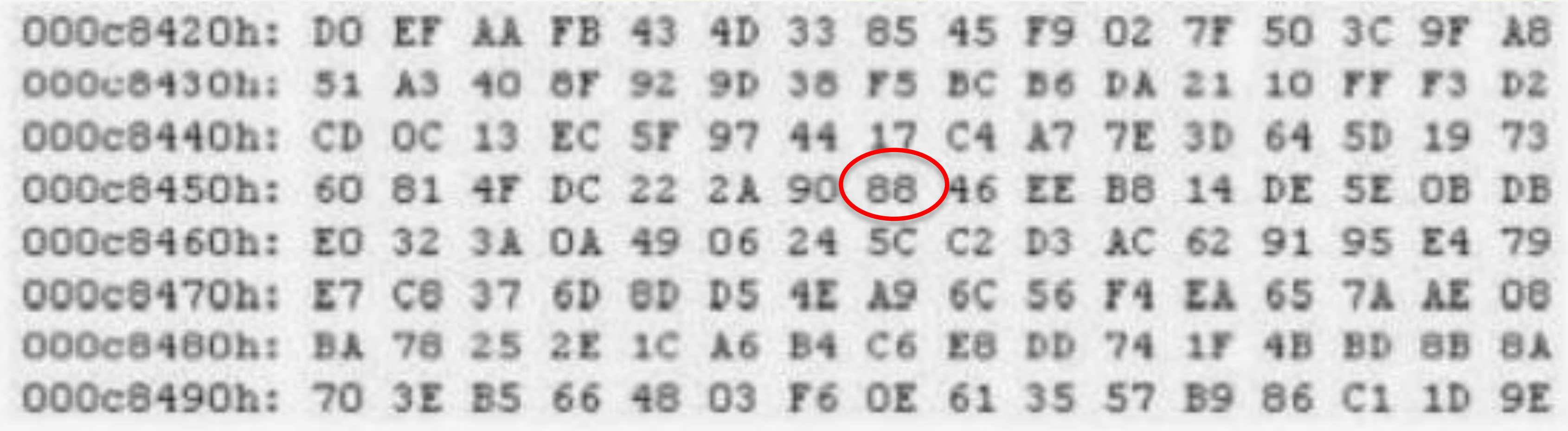


```
ADD     R11, R6, #0x1000
MOV     R0, R8   ; s1
MOV     R1, R11  ; s2
BL      strcmp
CMP     R0, #0
BEQ     loc_DA88
```

```
loc_DA34
LDR     R6, [R6,R10]
```

```
loc_DA88
ADD     R11, R11, #0x20
STR     R6, [R4,#0x110]
```

Data corruption
by flipping bits



```
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
```

Textbook attack

- ***If the check can be skipped...***

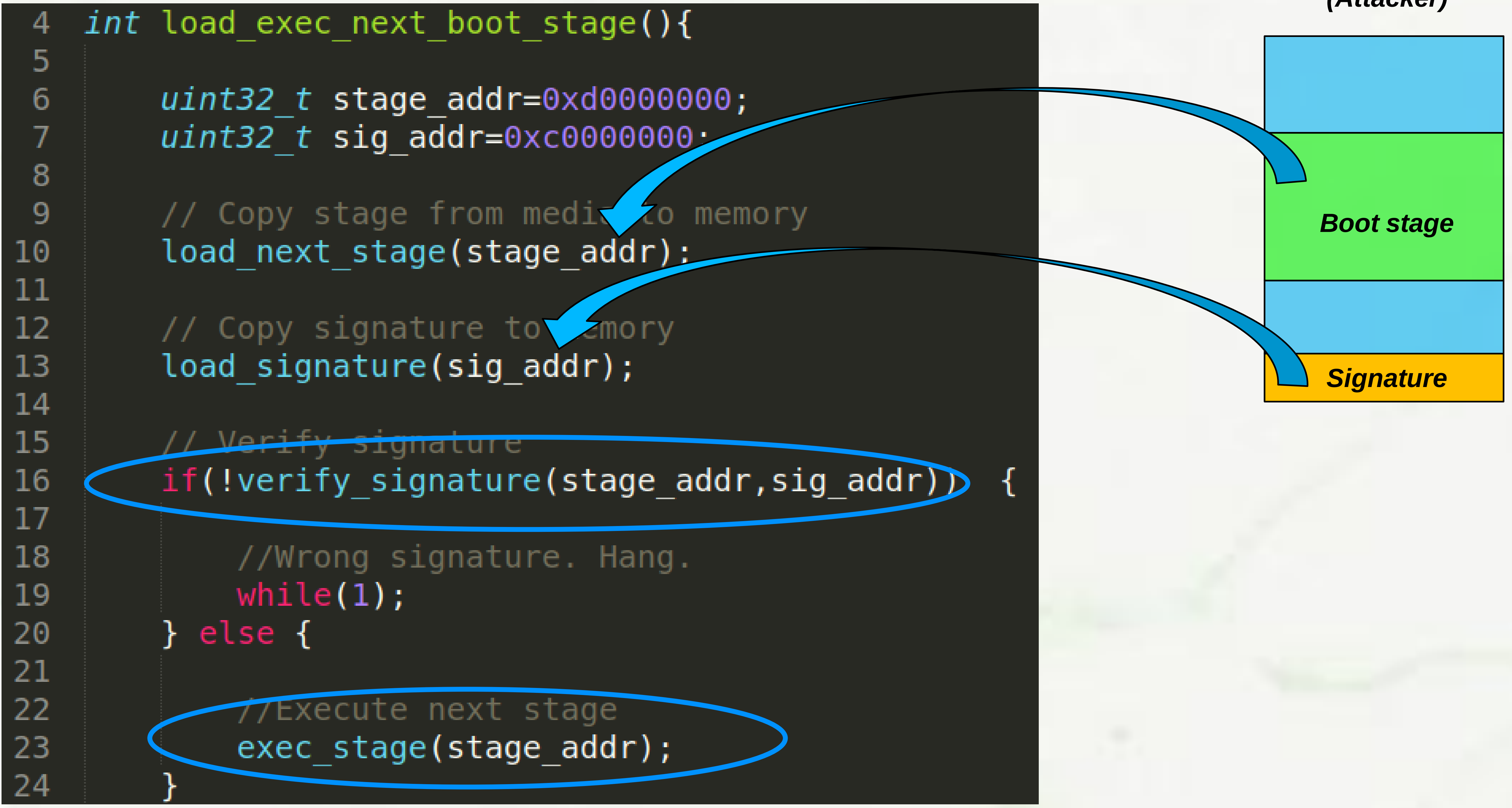
- Execution falls through the right case.
  - ***Regardless of the password.***

- ***SUCCESS!***

“Instruction skipping”

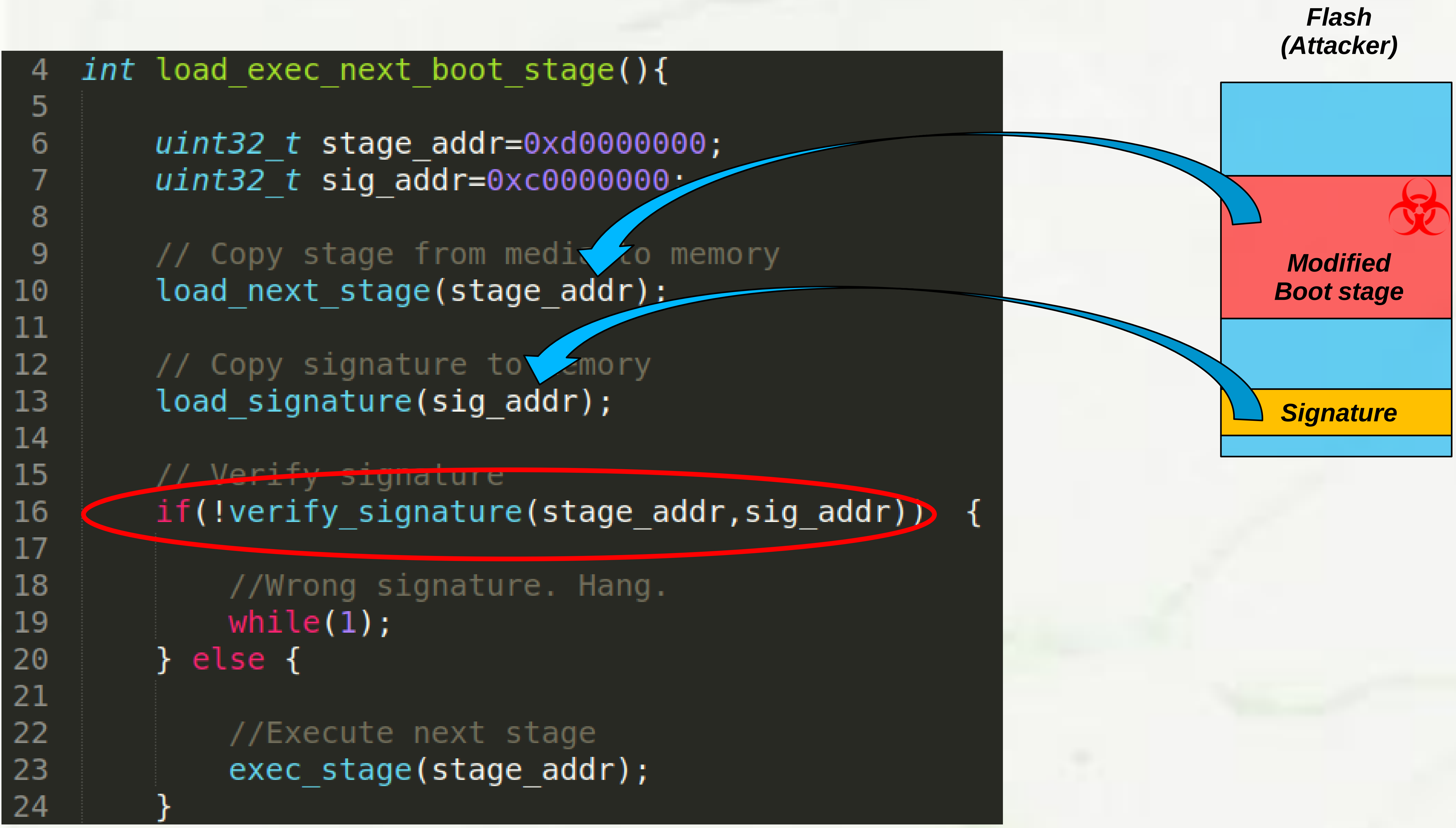
Case study:
Secure Boot

Secure Boot



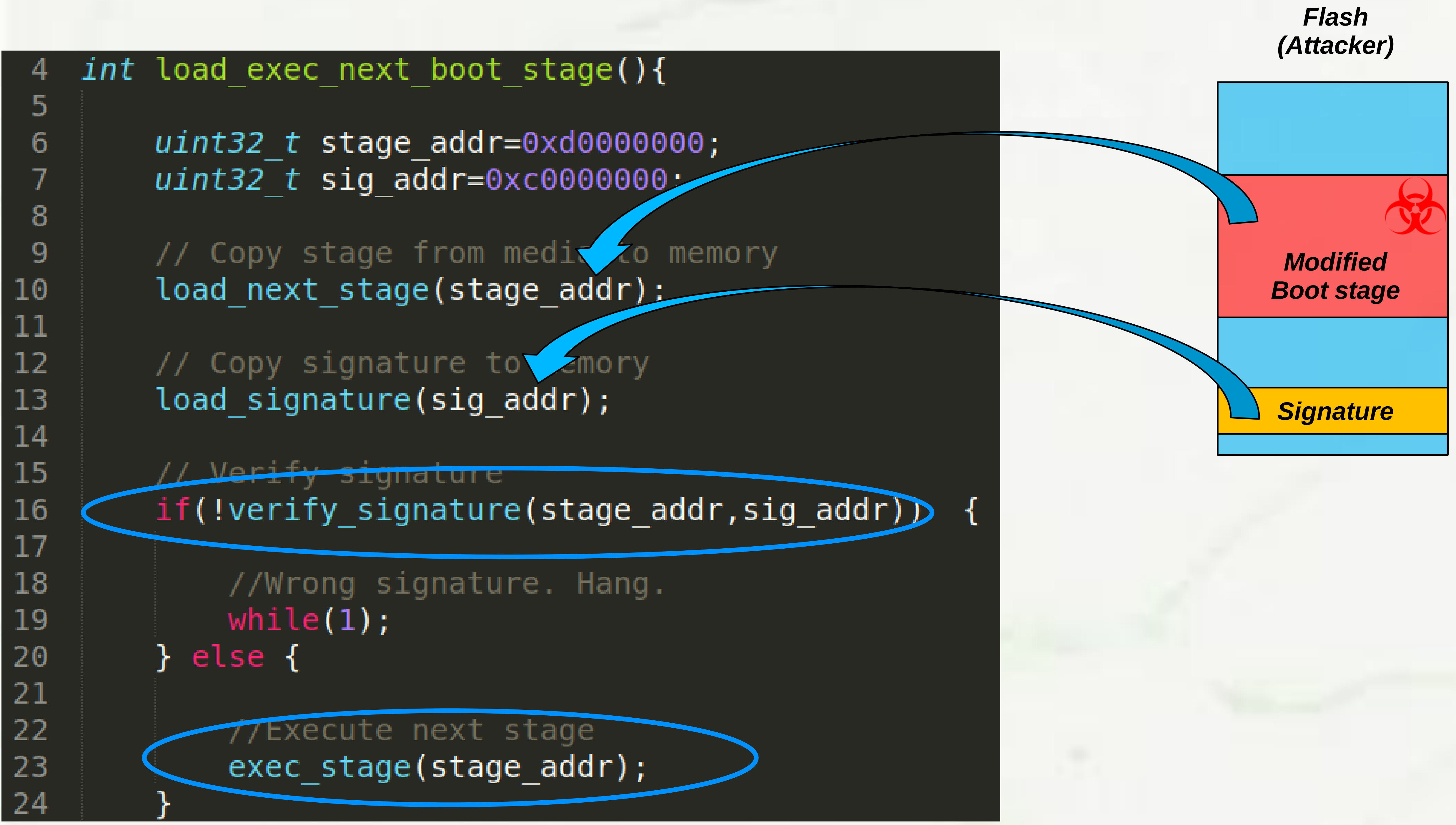
```
int load_exec_next_boot_stage(){

    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    // Copy stage from media to memory
    load_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Verify signature
    if(!verify_signature(stage_addr,sig_addr))  {

        //Wrong signature. Hang.
        while(1);
    } else {

        //Execute next stage
        exec_stage(stage_addr);
    }
```

Code modified? → Hang



```c
int load_exec_next_boot_stage(){

    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    // Copy stage from media to memory
    load_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Verify signature
    if(!verify_signature(stage_addr,sig_addr))  {

        //Wrong signature. Hang.
        while(1);
    } else {

        //Execute next stage
        exec_stage(stage_addr);
    }
```

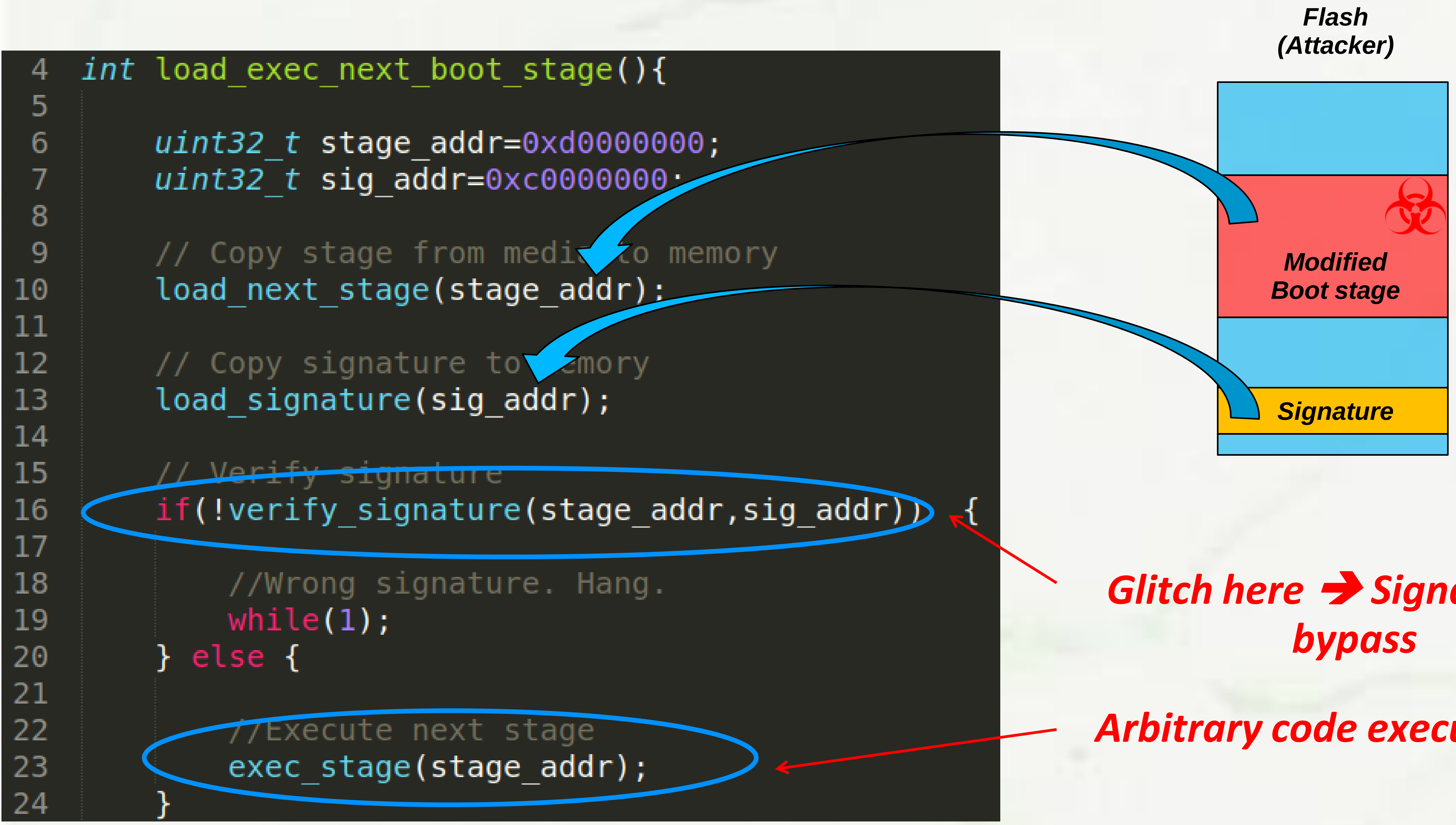
Executing our code?



```c
int load_exec_next_boot_stage(){

    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    // Copy stage from media to memory
    load_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Verify signature
    if(!verify_signature(stage_addr,sig_addr))  {

        //Wrong signature. Hang.
        while(1);
    } else {

        //Execute next stage
        exec_stage(stage_addr);
    }
```

How?

Textbook attack



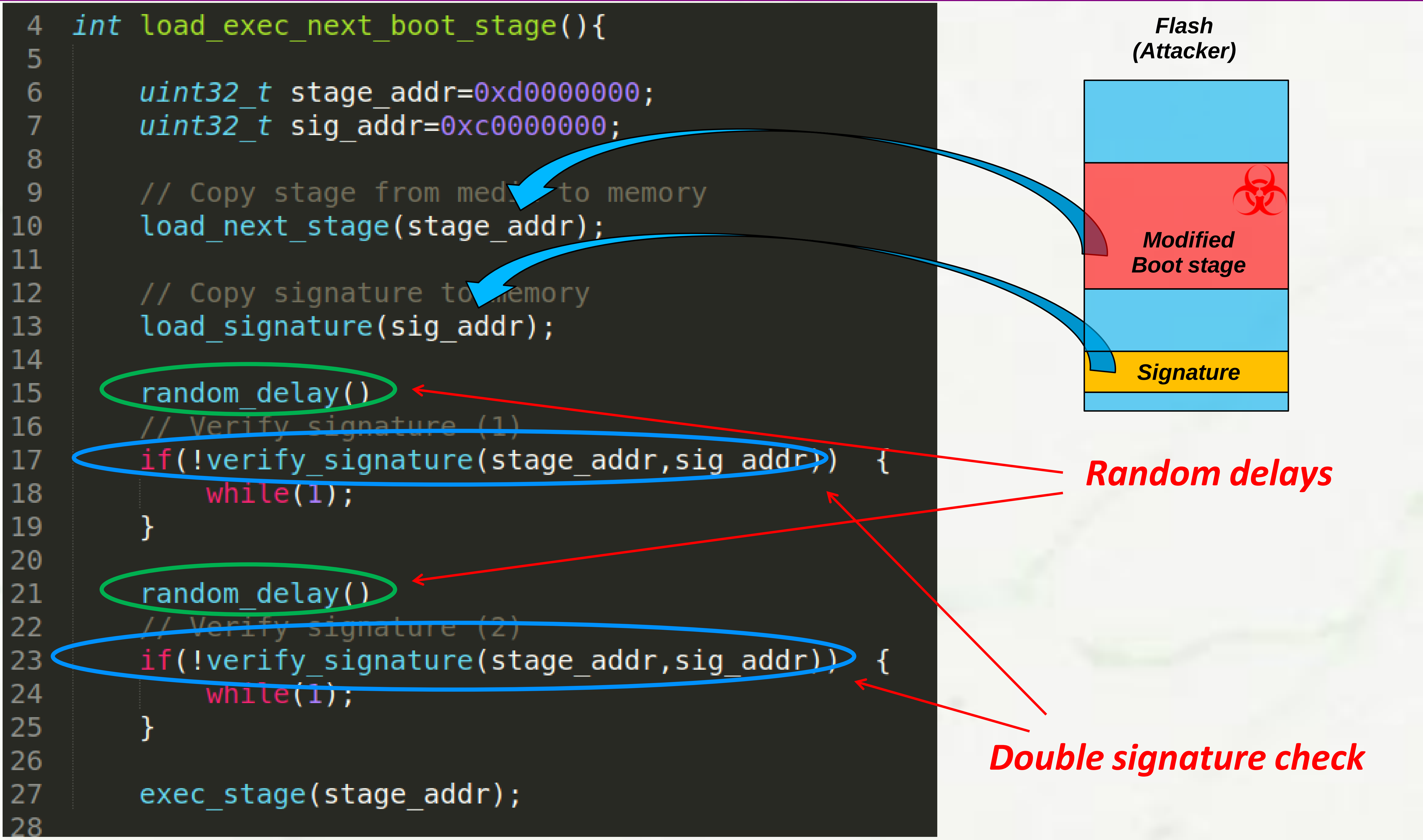
```
int load_exec_next_boot_stage(){

    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    // Copy stage from media to memory
    load_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Verify signature
    if(!verify_signature(stage_addr,sig_addr)) {

        //Wrong signature. Hang.
        while(1);
    } else {

        //Execute next stage
        exec_stage(stage_addr);
    }
```

Glitch here → Signature bypass

Arbitrary code execution

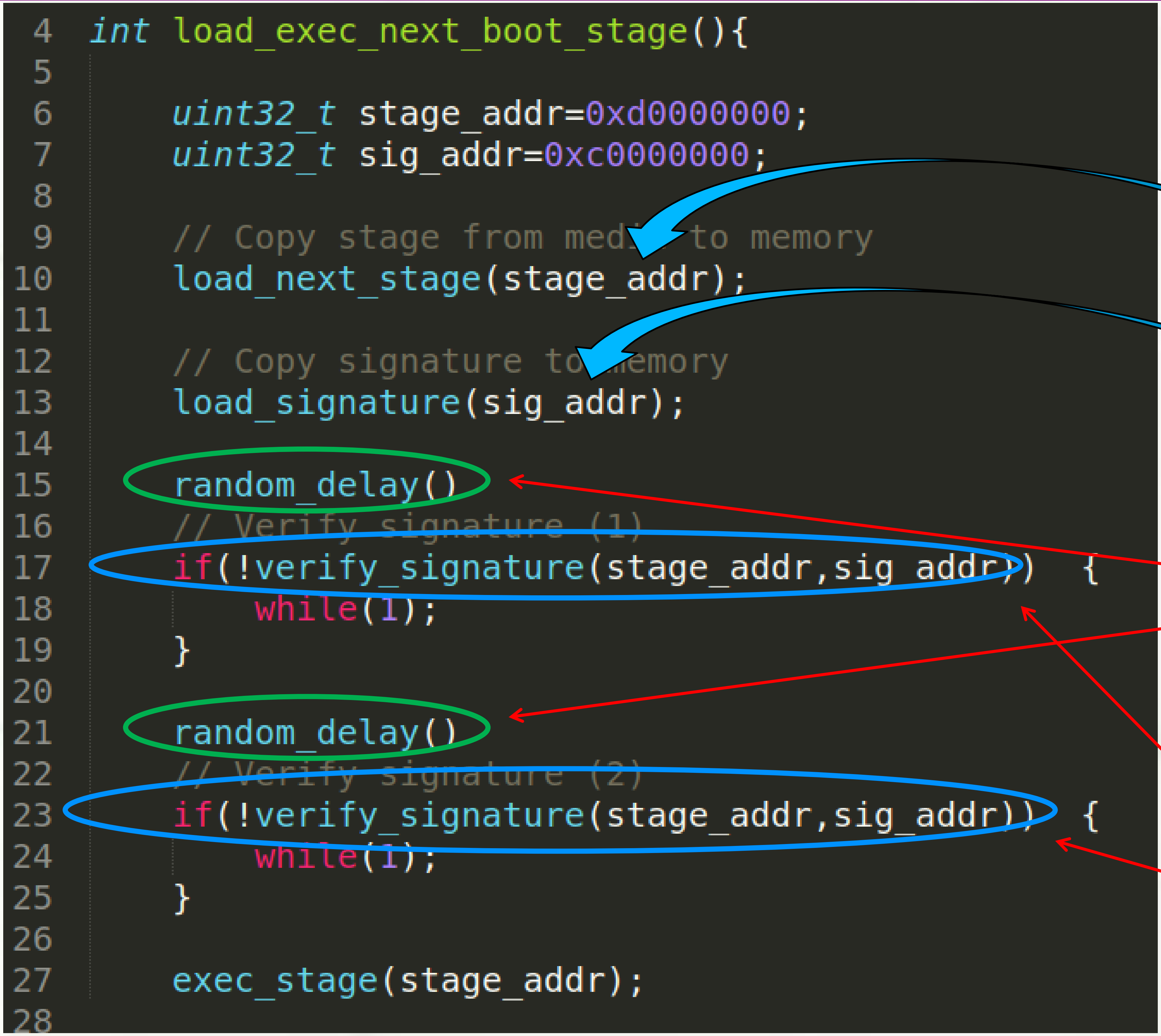
"Instruction skipping"

Secure Boot with countermeasures

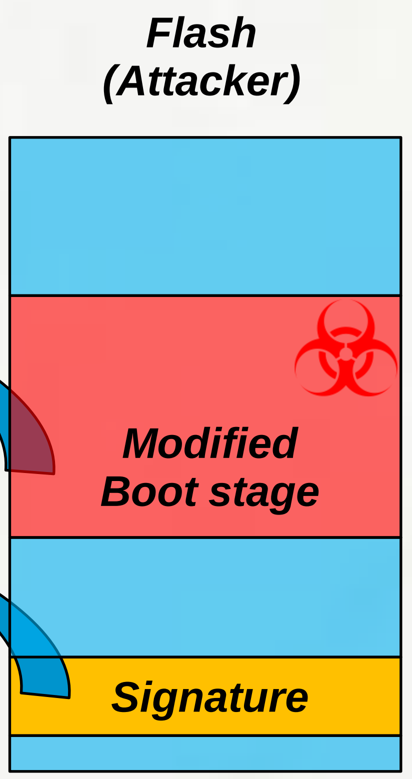


```c
int load_exec_next_boot_stage(){

    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    // Copy stage from media to memory
    load_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    random_delay()
    // Verify signature (1)
    if(!verify_signature(stage_addr,sig_addr))  {
        while(1);
    }

    random_delay()
    // Verify signature (2)
    if(!verify_signature(stage_addr,sig_addr))  {
        while(1);
    }

    exec_stage(stage_addr);

```

But still doable!



```c
int load_exec_next_boot_stage(){

    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    // Copy stage from media to memory
    load_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    random_delay()
    // Verify signature (1)
    if(!verify_signature(stage_addr,sig_addr))  {
        while(1);
    }

    random_delay()
    // Verify signature (2)
    if(!verify_signature(stage_addr,sig_addr))  {
        while(1);
    }

    exec_stage(stage_addr);

```



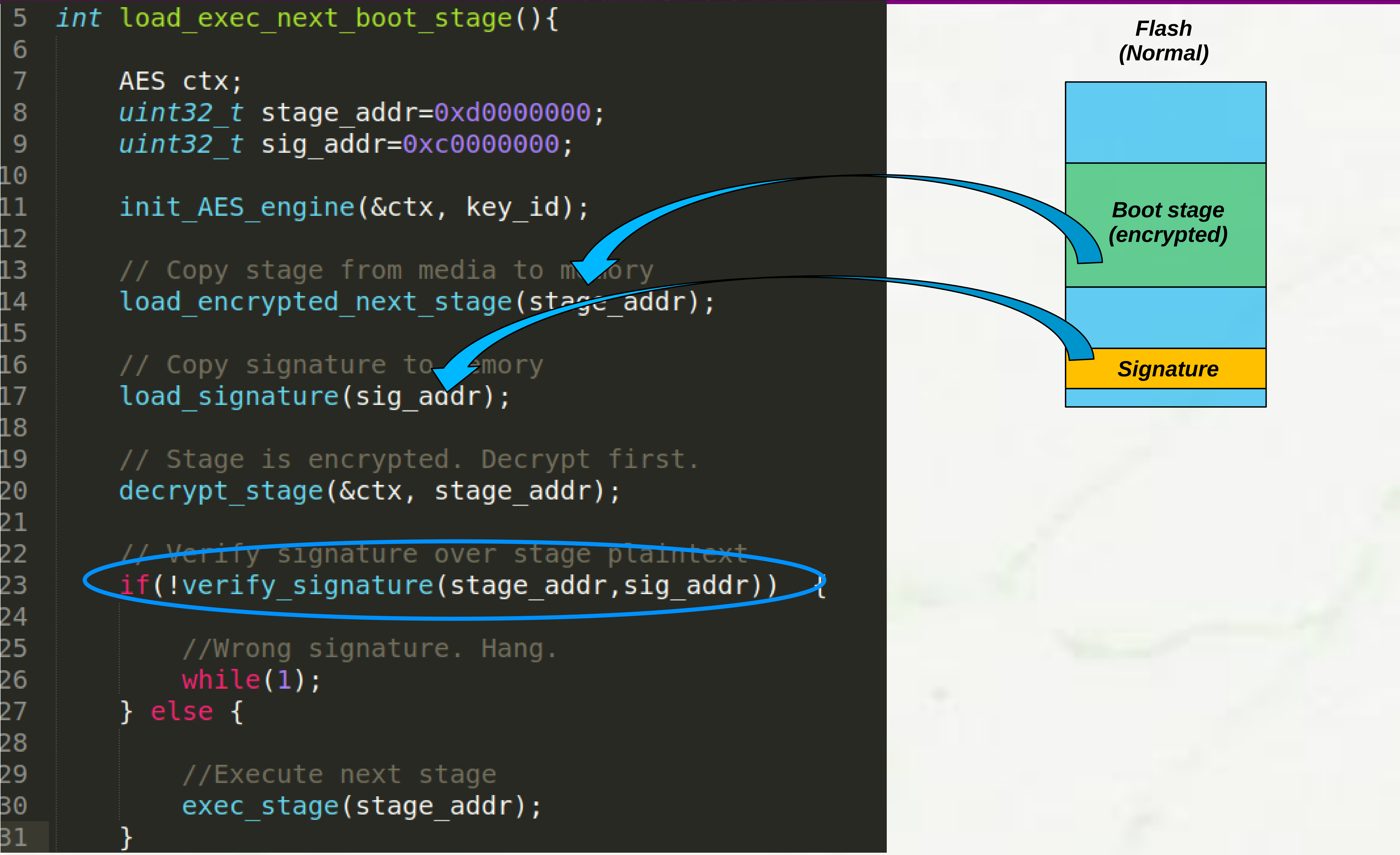
Synchronization more difficult

Two faults (maybe) required

Case study:

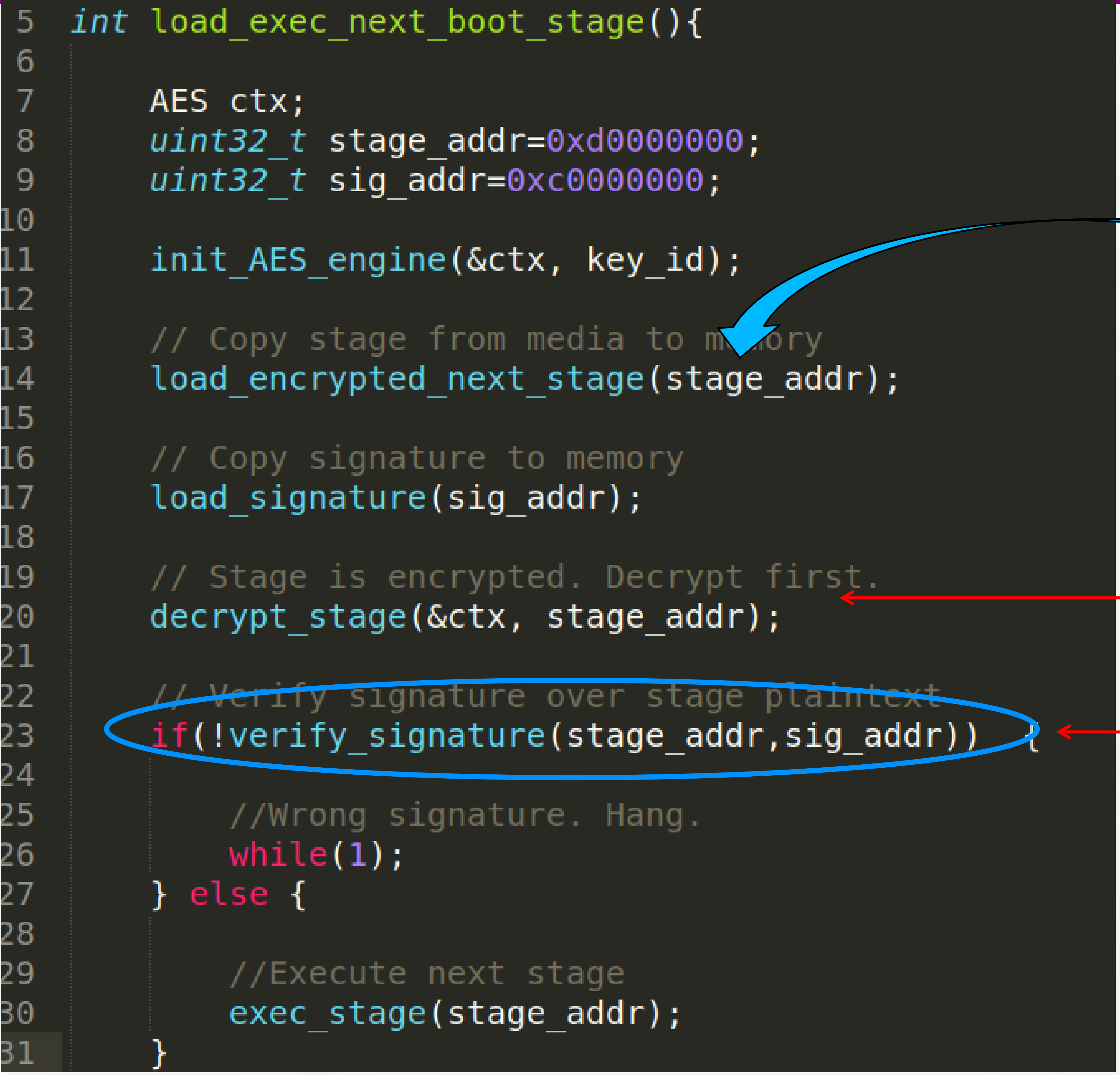
Encrypted Secure Boot

Encrypted Secure Boot

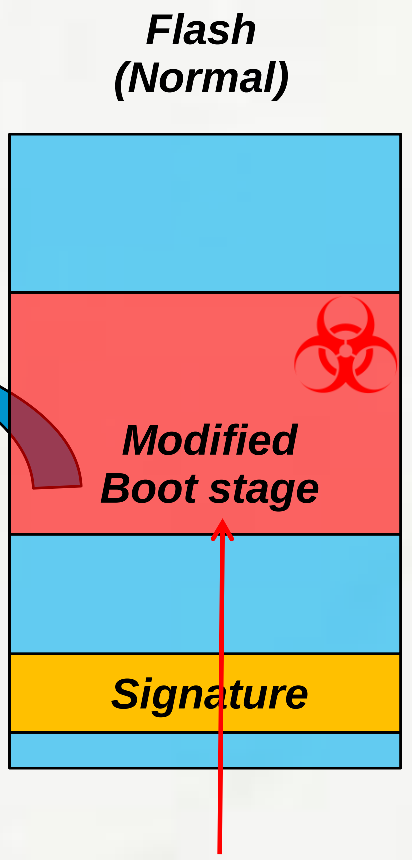


```c
int load_exec_next_boot_stage(){

    AES ctx;
    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    init_AES_engine(&ctx, key_id);

    // Copy stage from media to memory
    load_encrypted_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Stage is encrypted. Decrypt first.
    decrypt_stage(&ctx, stage_addr);

    // Verify signature over stage plaintext
    if(!verify_signature(stage_addr,sig_addr)) {

        //Wrong signature. Hang.
        while(1);
    } else {

        //Execute next stage
        exec_stage(stage_addr);
    }
```

FI textbook attack insufficient



```
int load_exec_next_boot_stage(){

    AES ctx;
    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    init_AES_engine(&ctx, key_id);

    // Copy stage from media to memory
    load_encrypted_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Stage is encrypted. Decrypt first.
    decrypt_stage(&ctx, stage_addr);

    // Verify signature over stage plaintext
    if(!verify_signature(stage_addr,sig_addr)) {

        //Wrong signature. Hang.
        while(1);
    } else {

        //Execute next stage
        exec_stage(stage_addr);
    }
```



Unknown encryption key

Signature bypass useless

Security Certification

- Security Lab report:
  - *"We have not been able to bypass Secure Boot"*
  - *"SCA attack needed for extracting encryption key*

↓

- CC Attack rating *goes stellar*
  - *If attack possible at all*

↓

- **Product is SECURE**

Reflections

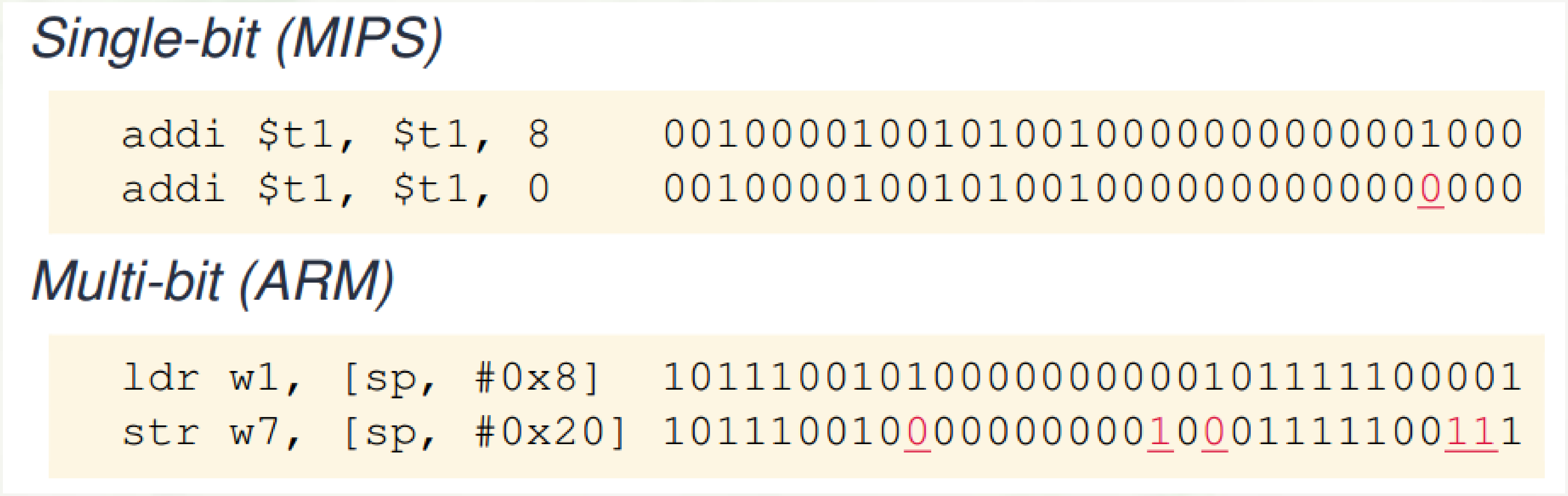
Instruction...skipping???

- ***Convenience fault model***
  - Widely used for characterizing effects on SW execution
- Has *limitations*:
  - Simplistic:
    - focused on conditionals, single-point decisions.
  - Not realistic:
    - Research shows *instructions are most likely "corrupted". Not skipped.*
- ***Insufficient for precise modeling of attacks aimed at SW execution.***

IN 2018!? SERIOUSLY?

A different fault model

A generic one: ***"instruction corruption"****



Single-bit (MIPS)

```
addi $t1, $t1, 8    00100001001010010000000000001000
addi $t1, $t1, 0    00100001001010010000000000000000
```

Multi-bit (ARM)

```
ldr w1, [sp, #0x8]  10111001010000000000010111100001
str w7, [sp, #0x20] 10111001000000000001000111100111
```

Remarks

- Limited control over which bit(s) will be corrupted
- Also *includes other fault models as sub-cases* (e.g. instruction skipping)

**[FTDC 2016]: Spruyt, Timmers, Witteman*

Controlling PC (or SP)

- ARM32 has an interesting ISA
- *Program Counter (PC) is directly accessible*

Valid ARM instructions

```
MOV r7,r1            00000001 01110000 10100000 11100001
EOR r0,r1            00000001 00000000 00100000 11100000
LDR r0,[r1]          00000000 00000000 10010001 11100101
LDMIA r0,{r1}        00000010 00000000 10010000 11101000
```

Corrupted ARM instructions may *directly set PC*

```
MOV pc,r1            00000001 11110000 10100000 11100001
EOR pc,r1            00000001 11110000 00101111 11100000
LDR pc,[r1]          00000000 11110000 10010001 11100101
LDMIA r0,{r1, pc}    00000010 10000000 10010000 11101000
```

Attack variations (SP-control) also affect other architectures

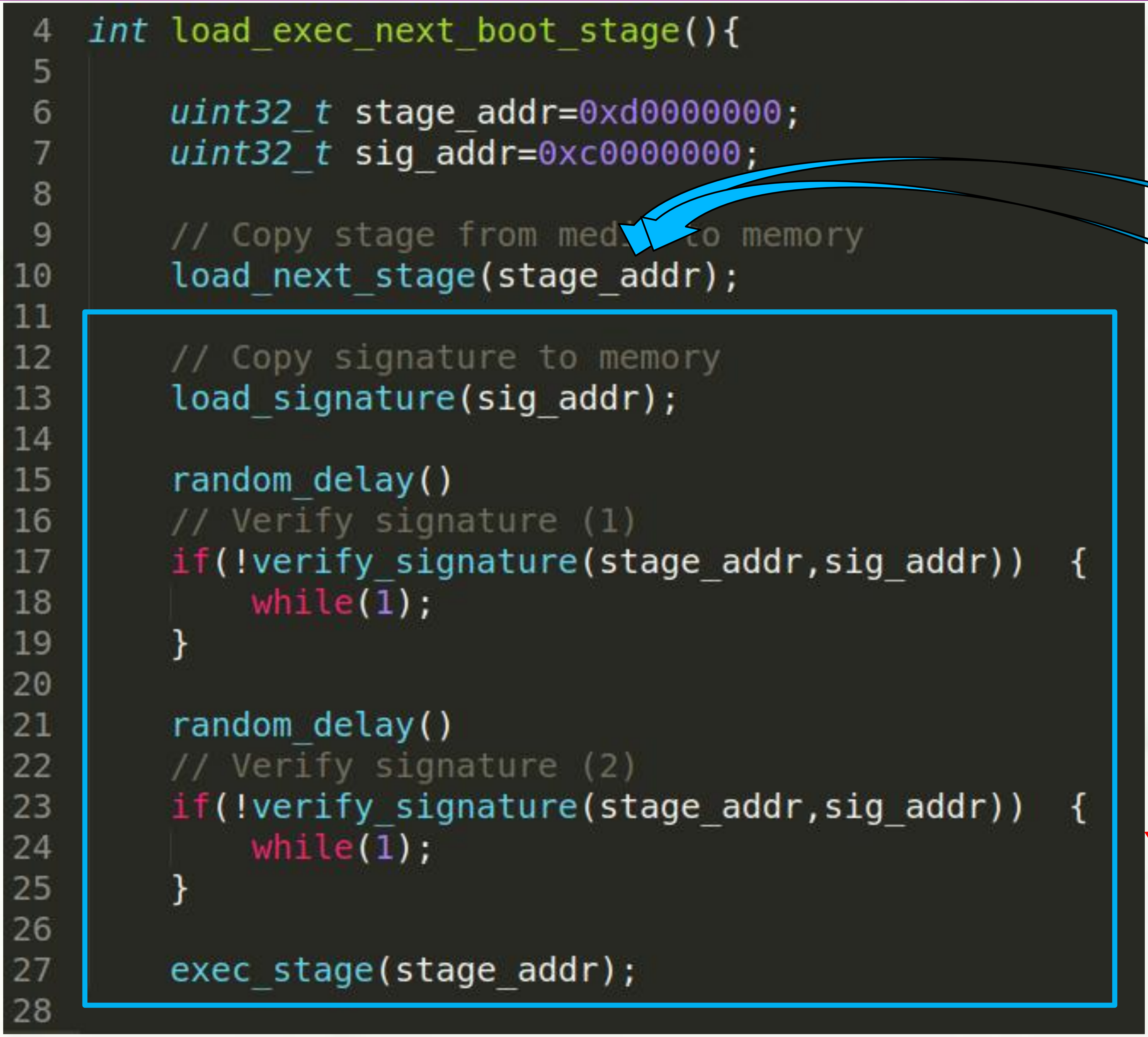
Execution primitives...out of thin air

- ***ANY memory read*** can be *redirected to PC (or SP)*
  - Hence, *ANY memcpy()*
- ***PC*** *(or SP)* ***immediately assigned*** *with content from memory*
  - *Following SW checks may not be executed*

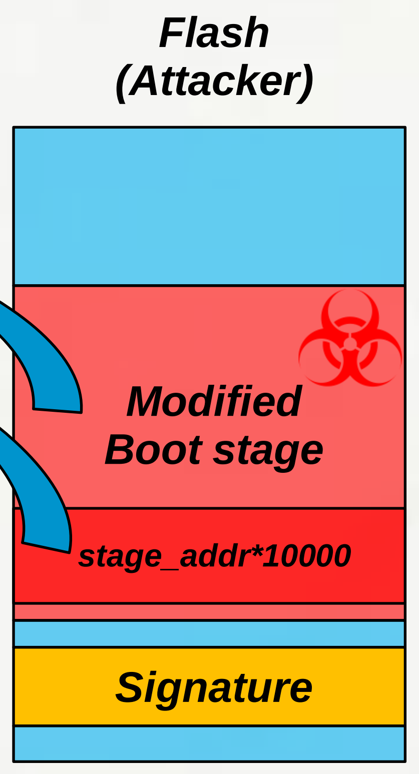
- ***A new target for FI:***
  - Security checks
  - Crypto algorithms
  - ...

Code Execution

Example: Secure Boot + countermeasures



```c
int load_exec_next_boot_stage(){

    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    // Copy stage from media to memory
    load_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    random_delay()
    // Verify signature (1)
    if(!verify_signature(stage_addr,sig_addr))  {
        while(1);
    }

    random_delay()
    // Verify signature (2)
    if(!verify_signature(stage_addr,sig_addr))  {
        while(1);
    }

    exec_stage(stage_addr);

```



Glitch here → Code execution at stage_addr*

Never executed

*also see [FDTC 2016]: Timmers, Spruyt, Witteman

Analysis

- Assumptions:
  - PC directly addressable (e.g. ARM32)
  - *Attackers knows code location*
  - *Destination memory writable and executable:*

- ***All FI SW countermeasures ineffective***
  - *Fault* ⇨ *Code execution transition happens in HW*
    - *More nuanced for code execution achieved via SP-control*

A SW attacker's dream...

- Like a SW exploit. *Without a SW vulnerability.*
- *ALL SW exploitation techniques fully applicable*
  - e.g. ROP, JOP, COP,… for SP-control

↓

- *SW exploitation mitigations can be effective*

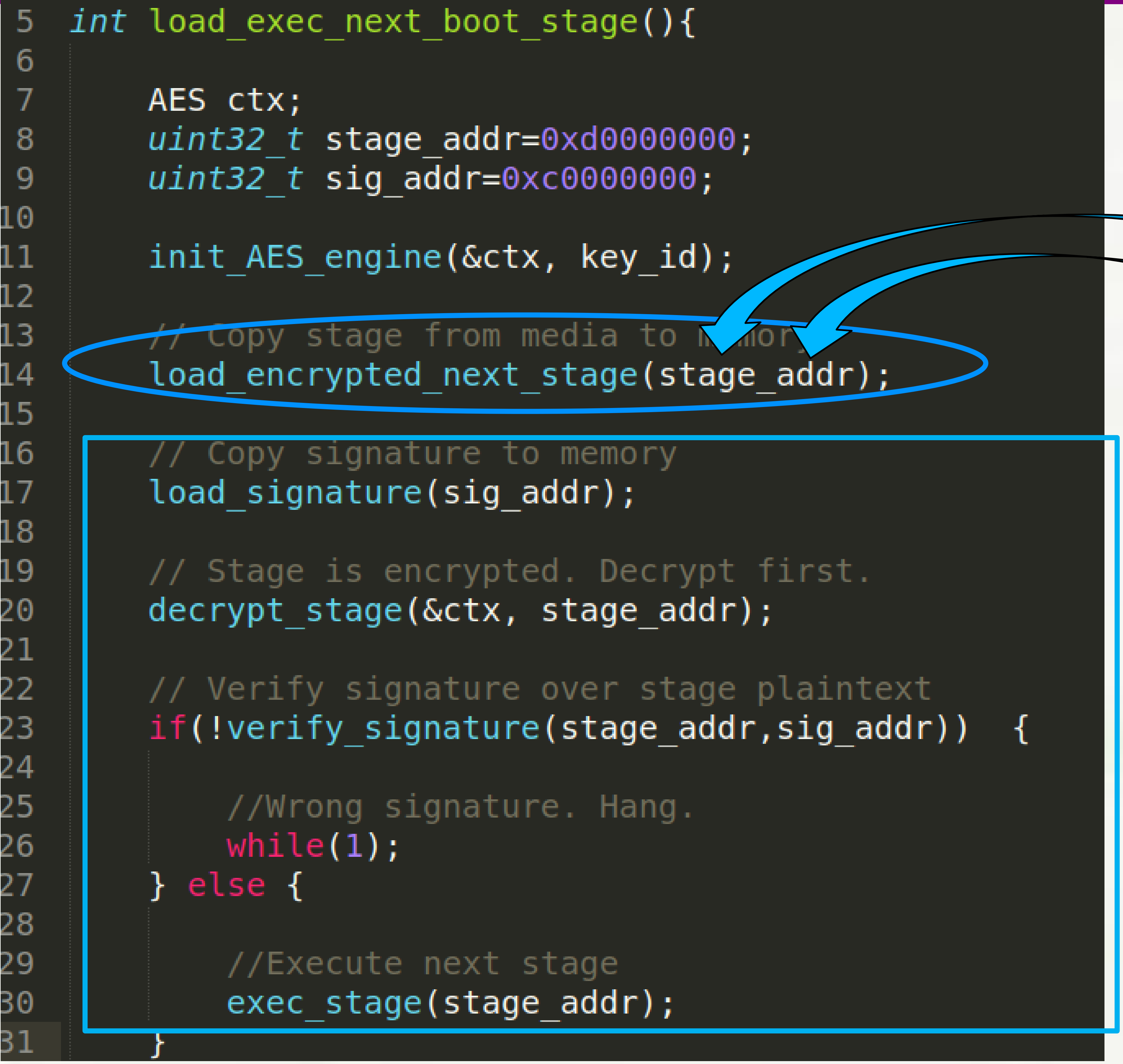
Defeating

Encrypted Secure Boot

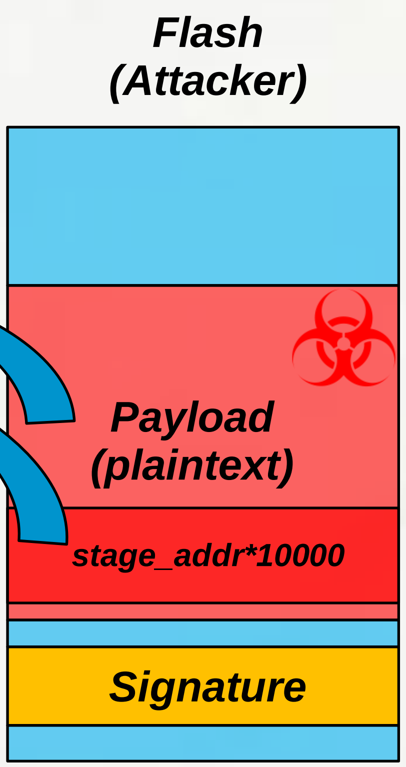
Attack preparation

- We can apply the same fault model:
  - Extends applicability of *Timmers, Spruyt, Witteman* technique (see FTDC 2016 presentation)
  - Also see *Timmers, Spruyt* @BlackHat 2016
- Strategy:
  - Redirect control flow via PC hijacking ➡ *Code execution*
    - *In the running context!*
  - Signature check not executed ➡ *Secure Boot bypass*
  - Decryption not executed ➡ *Plaintext code execution*
- Flash boot stage preparation:
  - Execution payload
  - “Sled” of PC target address

ROM code: secure boot + encryption



```
int load_exec_next_boot_stage(){

    AES ctx;
    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    init_AES_engine(&ctx, key_id);

    // Copy stage from media to memory
    load_encrypted_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Stage is encrypted. Decrypt first.
    decrypt_stage(&ctx, stage_addr);

    // Verify signature over stage plaintext
    if(!verify_signature(stage_addr,sig_addr))  {

        //Wrong signature. Hang.
        while(1);
    } else {

        //Execute next stage
        exec_stage(stage_addr);
    }
```



Glitch while loading pointers → Code exec at stage_addr*

Never executed

Analysis

- ***Signature verification not performed***
  - *Secure boot deafeated*
- ***Decryption not performed***
  - *Plaintext code execution*

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

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

Countermeasures

Analysis

- Hardware FI countermeasures *fully applicable*
    - Detect glitch injection or fault generation
- *FI SW countermeasures likely not executed*
    - A successful attack hijacks control flow immediately
- *Localized software FI countermeasures are insufficient*
  - Any instruction is a potential target for corruption

SW Exploit mitigations

- Fully applicable.
- **Relevant:** Limiting usage of an hijacked control flow
  - DEP/NX
  - ASLR
  - CFI
  - …
- **Irrelevant:** Preventing control flow hijacking:
  - Stack cookies
  - SEHOP
  - …

Recommendations

- Use **FI HW countermeasures** for ***prevention***
  - Applicable regardless of fault model
- FI SW countermeasures *can only mitigate “classical attacks”*
- **Adopt modern SW security paradygms:**
  - *SW exploit mitigations*
  - Defense in depth
  - Secure SDLC
  - ...
- ***Learn from:***
  - *SW attackers*
  - ***SW and Mobile security industry***

Final thoughts

A new attack

- ***Technique for bypassing encrypted secure boot:***
  - *One single fault*
  - *ROM-level code execution*
  - *Arbitrary plaintext payload*
  - *No encryption key needed*

Advancing FI

- Different fault models can yield *dramatically different results*
- Simplistic fault modeling can be *dangerous*:
  - High impact attacks may go undetected for decades
- *SW execution integrity is critical nowadays*:
  - Must be fully in scope for FI
- Must also fall scope for modern HW security industry, in general

HW & SW security

- *HW and SW attacks more intertwined*:
  - Also see *micro-architectural attacks*
  - Unexpected implications
- *HW security industry needs SW security experts*
  - And viceversa
- We need more *“in between*” expertise:
  - Single domain expertise not sufficient anymore
- Attackers’ and engineers’ perspectives need blending
- ***Holystic****, system-level view needed*

Finally...

- *If your HW and SW security engineering teams...*
- *do not talk to each other...*

You are likely doing it wrong.

Contacts

Cristofaro Mune

Product Security Consultant

c.mune@pulse-sec.com