Fault attacks and software security

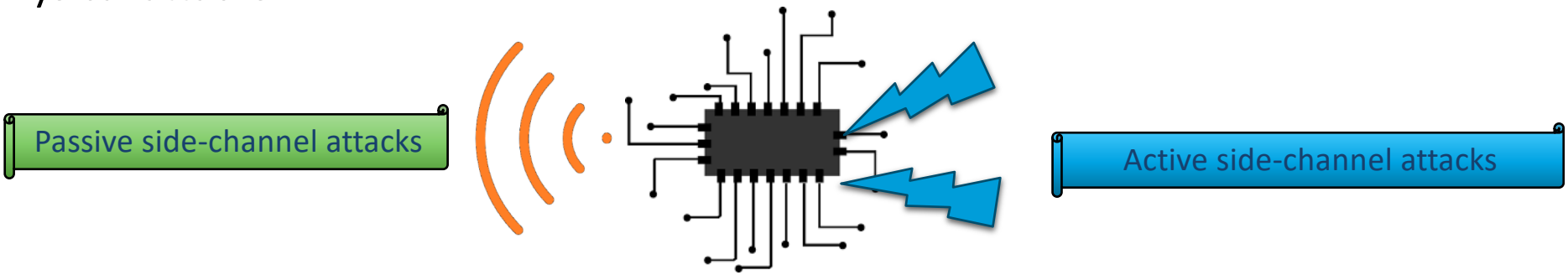
Karine Heydemann

Several classes of attacks

- Software vulnerability exploitation

- Micro-architectural attacks

- Physical attacks

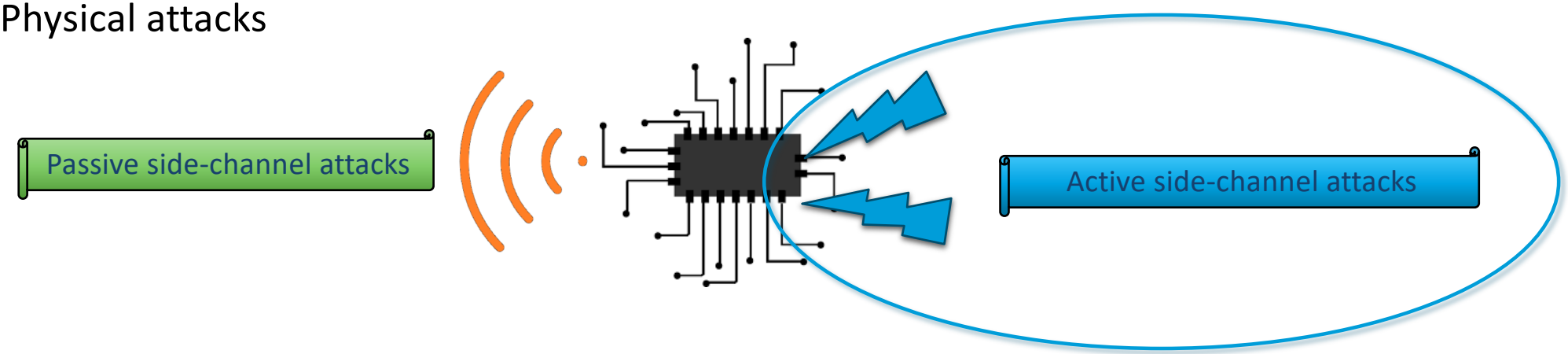


Several classes of attacks

- Software vulnerability exploitation

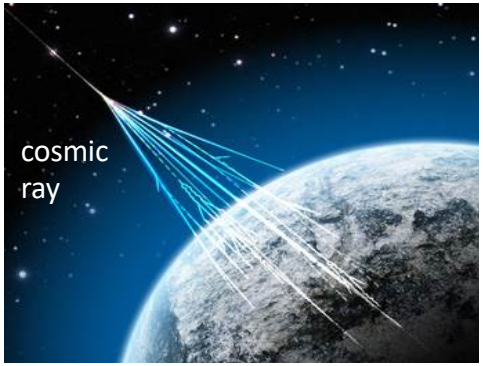
- Micro-architectural attacks

- Physical attacks

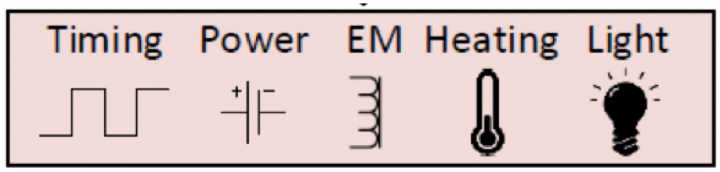


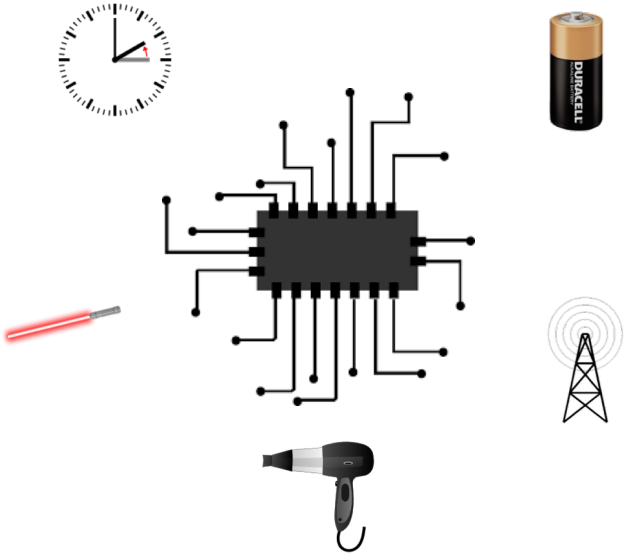
Fault injection attacks

- « Natural » faults

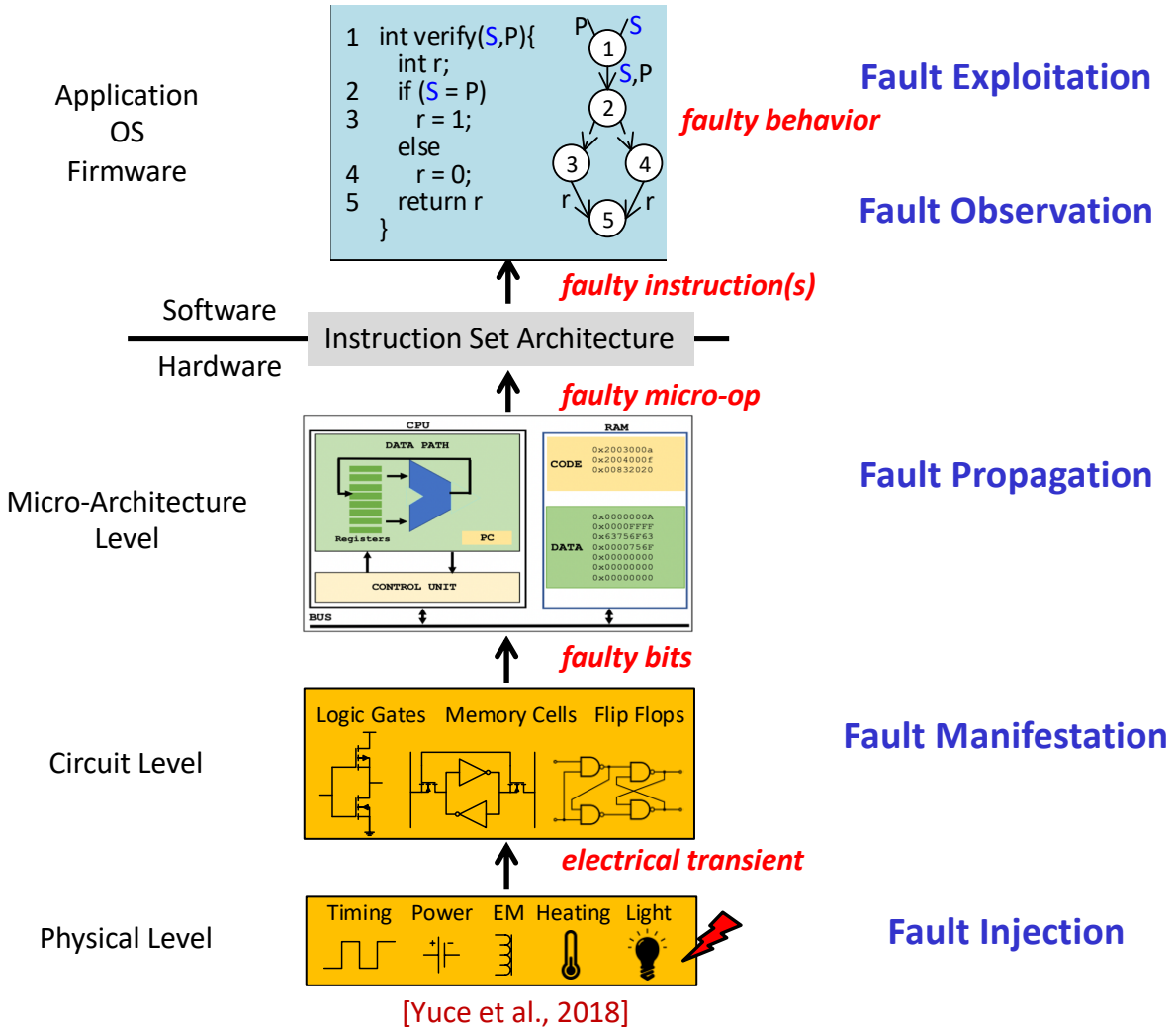


- Intentional faults





Anatomy of a fault attack



[Yuce et al., 2018]

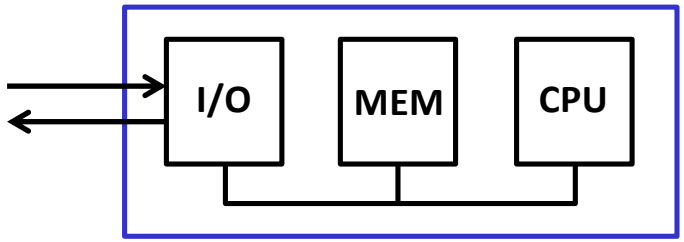
Agenda

- Fault exploitation
- Fault effects and modelling
- Countermeasures
- Robustness analysis
- Conclusion and open research questions

Agenda

- **Fault exploitation**
- Fault effects and modelling
- Countermeasures
- Robustness analysis
- Conclusion and open research questions

Attacks on embedded software



- Embedded Software assumes execution is correct
- Incorrect execution as starting point for attack
  - Privilege Escalation
  - Sensitive Information leakage / key recovery

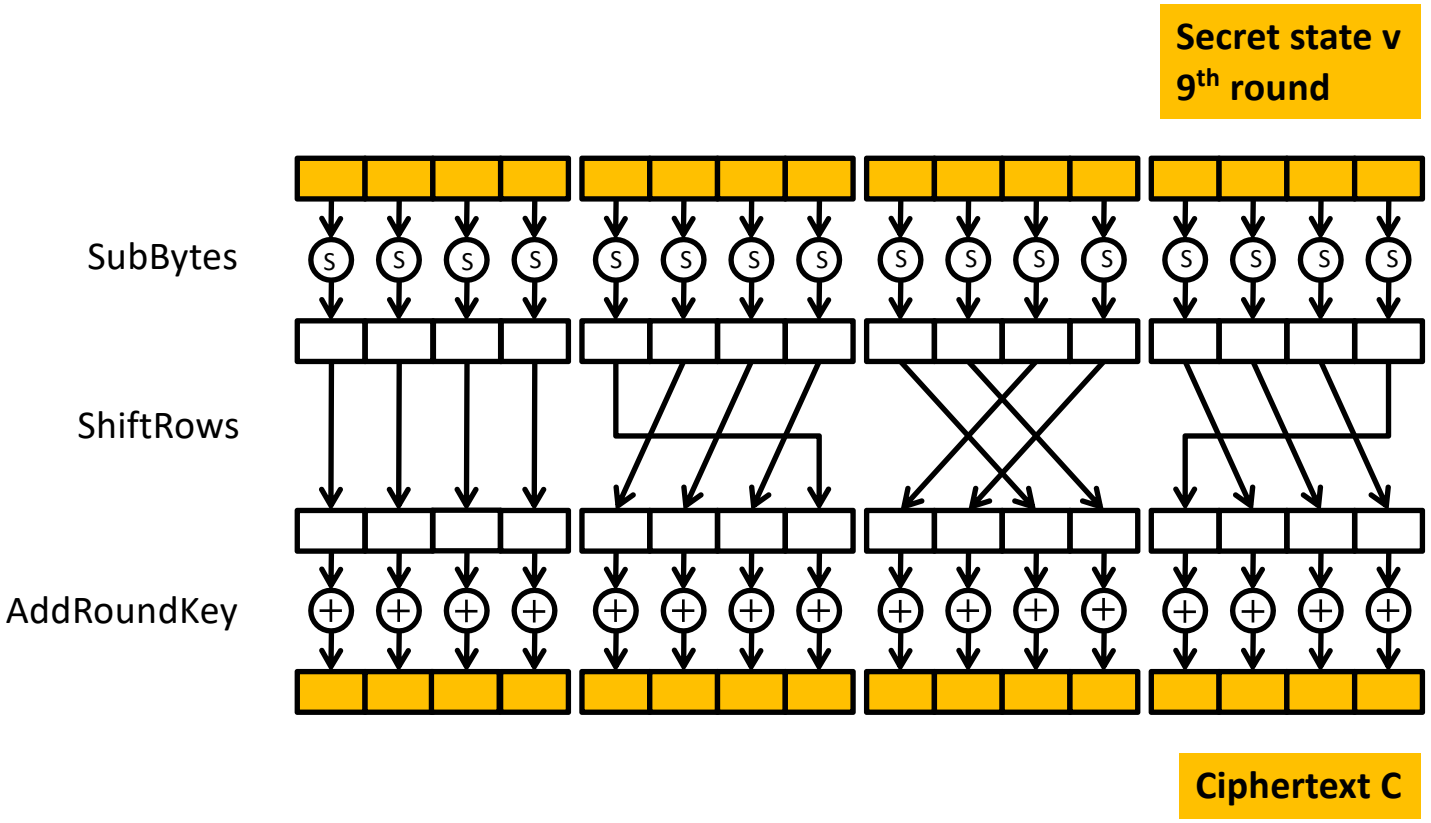
Common fault exploitation

- Cryptanalysis using fault injection
  - Differential Fault Analysis
  - Biased Fault Analysis
  - Safe Error Analysis
  - Algorithm-specific Fault Analysis
- Fault-aided Side-channel Analysis
- Fault-enabled Logical Attacks
- Fault-aided Reverse Engineering

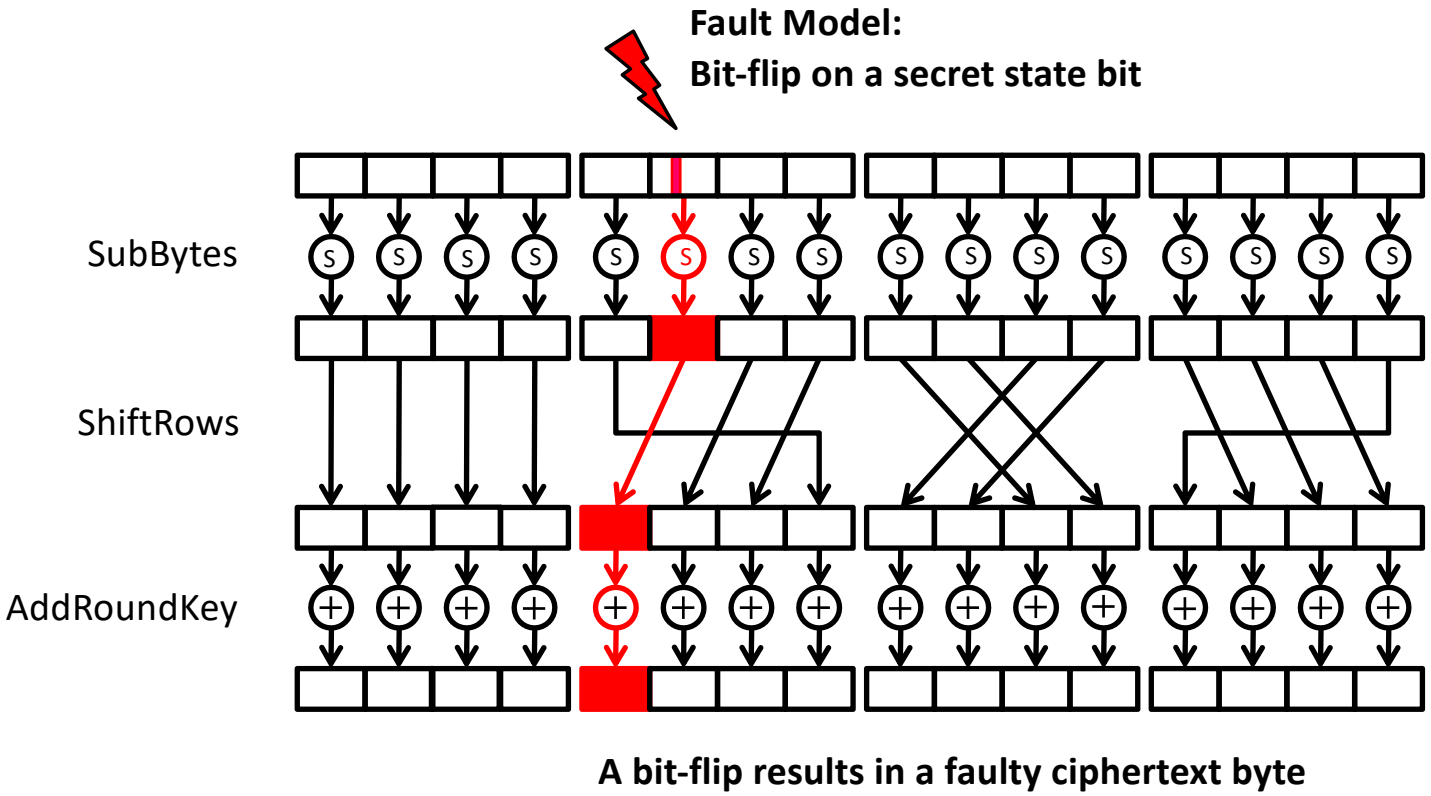
Common fault exploitation

- **Cryptanalysis using fault injection**
  - **Differential Fault Analysis**
  - Biased Fault Analysis
  - **Safe Error Analysis**
  - Algorithm-specific Fault Analysis
- Fault-aided Side-channel Analysis
- **Fault-enabled Logical Attacks**
- Fault-aided Reverse Engineering

Differential Fault Analysis



Bit-flip attack on AES



Bit-flip attack on AES

- **Fault Differential**

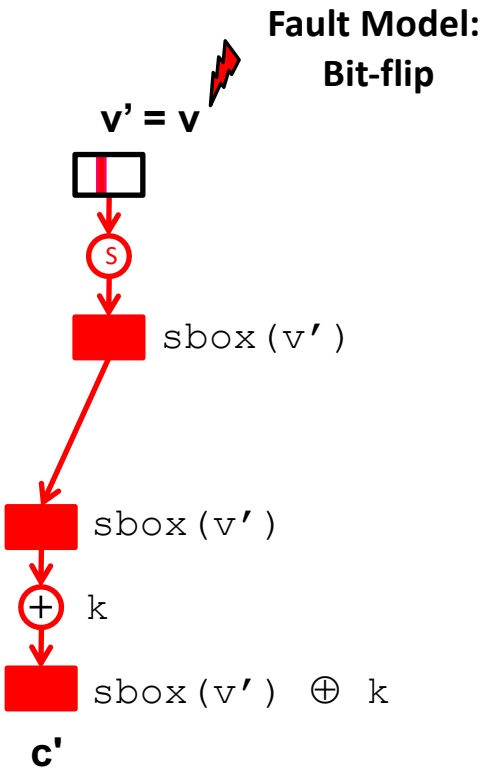
  $c = sbox(v) \oplus k$
  $c' = sbox(v') \oplus k$
  Hence $\Delta = c \oplus c' = sbox(v) \oplus sbox(v')$

- **Fault Analysis**
  - Search v, v' such that

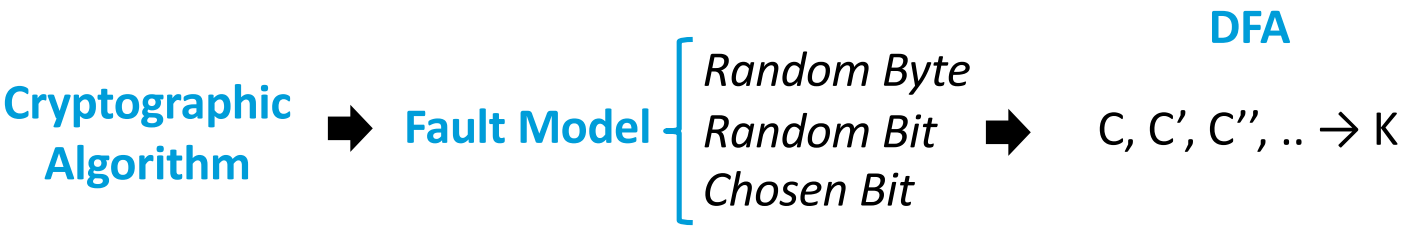
    $HD(v, v') = 1$ AND
    $\Delta = sbox(v) \oplus sbox(v')$
  - Using a second bit flip, determine v
  - Determine the last round-key as:

    $k = sbox(v) \oplus c$

- 32 bit-flip faults in round 10 disclose entire key



Classic DFA



[Tunstall 2010] Single random byte fault at 8th round of AES-128: Key 2^{128} → 2^{12}

[Ali 2012] Two seq. byte fault at 9th, 10th round of AES-192: Key 2^{192} → 1

Current DFA methods are *optimal*

IF

the fault model can be realized

Safe-error analysis

```
Input: Elliptic Curve Point P
       secret integer k = {k_{n-1}k_{n-2}…k_1k_0}
Output: k.P
  R[0] = 0
  for i = n - 1 down to 0 do
     R[0] = 2.R[0]
     R[1] = R[0] + P
     R[0] = R[k_i]
  end for
  return R[0]
```

Double-Add Always
(SPA Countermeasure)

Safe-error analysis

```
Input: Elliptic Curve Point P
       secret integer k = {k_{n-1}k_{n-2}…k_1k_0}
Output: k.P
  R[0] = 0
  for i = n - 1 down to 0 do
     R[0] = 2.R[0]
     R[1] = R[0] + P
     R[0] = R[k_i]
  end for
  return R[0]
```

Dummy operation when k_i is equal to 0

Safe-error analysis

```
Input: Elliptic Curve Point P
       secret integer k = {k_{n-1}k_{n-2}…k_1k_0}
Output: k.P
  R[0] = 0
  for i = n - 1 down to 0 do
     R[0] = 2.R[0]
     R[1] = R[0] + P
     R[0] = R[k_i]
  end for
  return R[0]
```

C-safe error

Injecting a fault in a dummy operation will not affect the output

Fault enabled logical attacks

- General-purpose computing
  - Memory dump / extraction
  - Control-flow hijacking
  - Privilege escalation
  - Secure Boot bypass

Memory dump attack

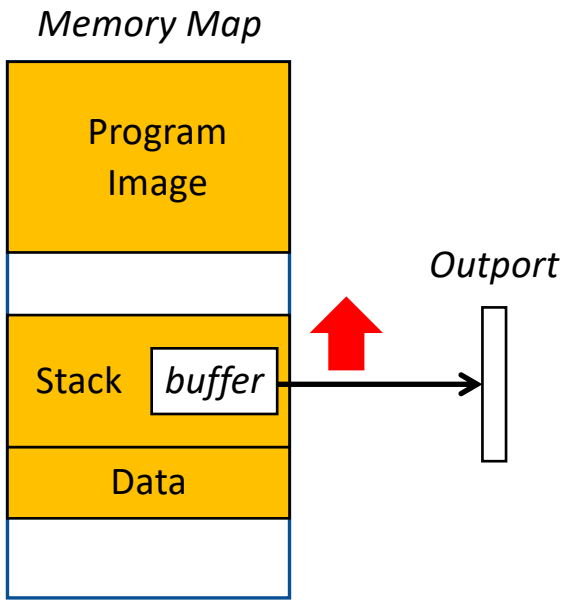
A typical subroutine found in security processors is a loop that writes the contents of a limited memory range to the serial port:

```
1  b = answer_address
2  a = answer_length
3  if (a == 0) goto 8
4  transmit(*b)
5  b = b + 1
6  a = a - 1
7  goto 3
8  ...
```

Instruction-skip

Instruction-skip or other instr replacement

We can look for a glitch that increases the program counter as usual but transforms either the conditional jump in line 3 or the loop variable decrement in line 6 into something else.

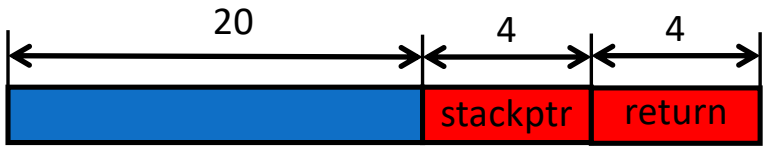


R. Anderson and M. Kuhn, "Tamper resistance: a cautionary note," *2nd USENIX Workshop on Electronic Commerce*. 1996.

Buffer overflow attack

```
void myfunc(char *buf) {
    char msg[20] = {0};
    memcpy(msg, buf, sizeof(msg)-1);
    ..
}

 void *memcpy (void *dest,
               const void *src,
               size_t len) {
    char *d = dest;
    const char *s = src;
    while (len--)
      *d++ = *s++;
    return;
 }
```

Instruction-skip



malicious buf

ARM Cortex M0

S. Nashimoto et al. *Buffer overflow attack with multiple fault injection and a proven countermeasure*. J. Cryptographic Engineering 7(1): 35-46 (2017)

Privilege escalation

```
r0 = 0 .. r11 = 0
r7 = 0xd0
svc #0          // setuid system call
if (r0 == 0) // success
  system("/bin/sh");
…
```

Privilege Escalation

= Adversarial Control of Critical Decisions

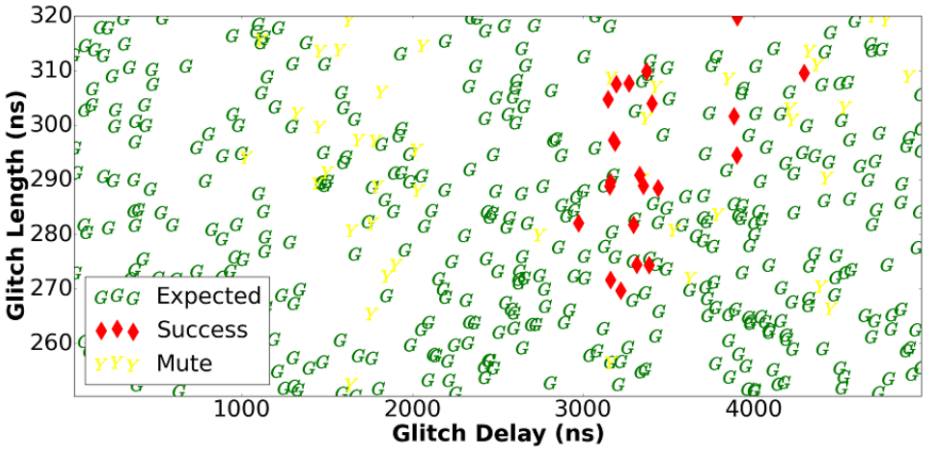
```
if (access_allowed == 0)
    sensitive_op( );
…
```

Niek Timmers, Cristofaro Mune: Escalating Privileges in Linux Using Voltage Fault Injection. FDTC 2017

Privilege escalation

Instruction skip

```
r0 = 0 .. r11 = 0
r7 = 0xd0
svc #0          // setuid system call
if (r0 == 0) // success
  system("/bin/sh");
…
```



Privilege Escalation

= Adversarial Control of Critical Decisions

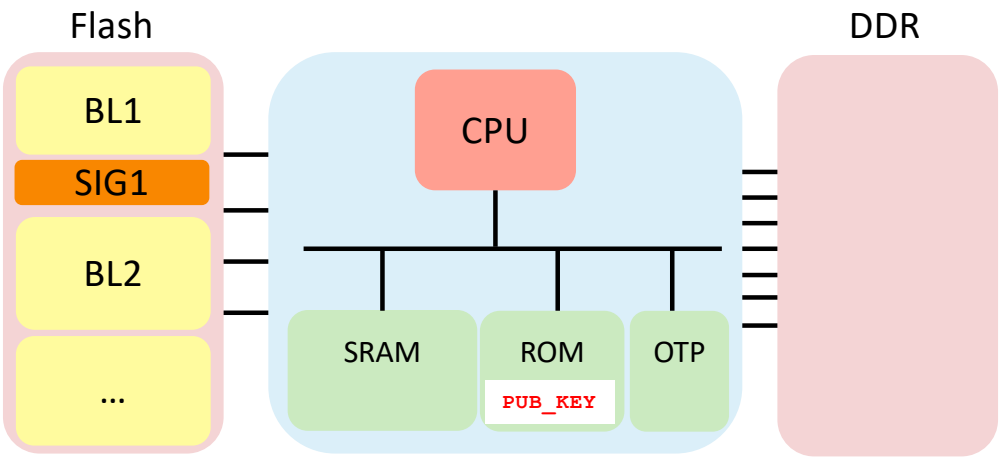
```
if (access_allowed == 0)
    sensitive_op( );
…
```

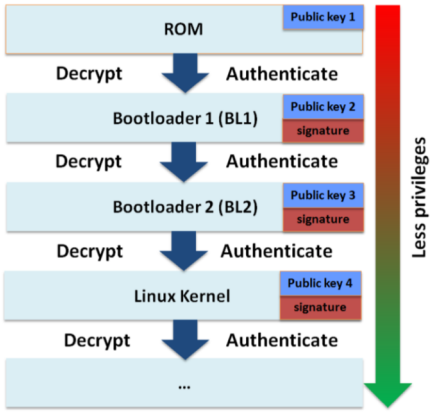
- 18968 experiments
- 21 hours
- 1.3% success rate
- Root Shell spawned every 5 minutes

ARM Cortex A-9

Niek Timmers, Cristofaro Mune: Escalating Privileges in Linux Using Voltage Fault Injection. FDTC 2017

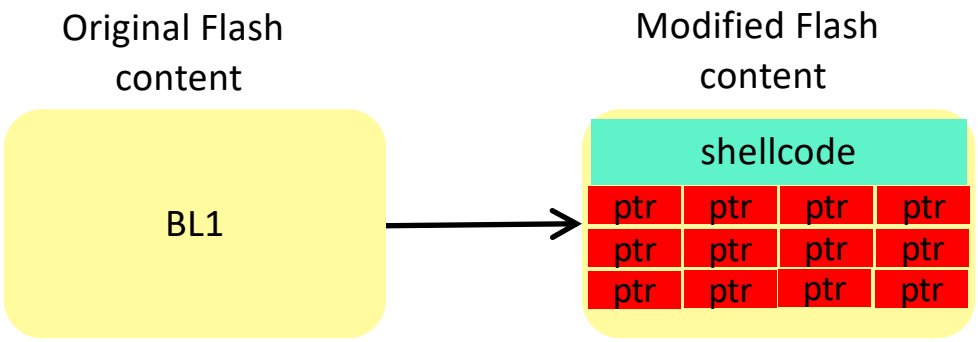
Bypassing secure boot





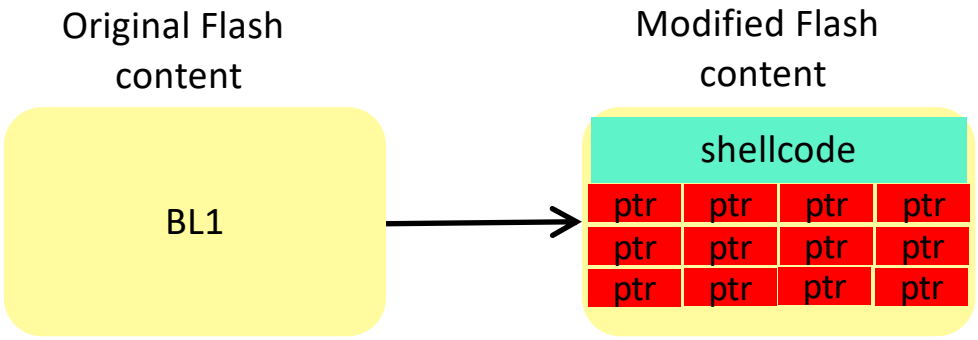

```
/* copy image from flash to sram */
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE)

/* decryption ? */
if (secure_boot_dec){
    /* decrypt image in place */
    decrypt(IMG_RAM, IMG_SIZE, KEY)
}
/* authentication ? */
if (secure_boot_en){
    /* copy signature from flash to sram */
    memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);

    /* compute hash over SRAM image */
    sha(IMG_RAM, IMG_SIZE, IMG_HASH);

    /* compute hash from signature */
    rsa(PUB_KEY, SIG_RAM, SIG_HASH)

    /* compare hashes */
    if (compare(IMG_HASH, SIG_HASH) != 0){
            while(1);
    }
jump_to_next_stage();
}
```

Bypassing secure boot




```
/* copy image from flash to sram */
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE)

/* decryption ? */
if (secure_boot_dec){
    /* decrypt image in place */
    decrypt(IMG_RAM, IMG_SIZE, PUB_KEY)
}
/* authentication ? */
if (secure_boot_en){
    /* copy signature from flash to sram */
    memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);

    /* compute hash over SRAM image */
    sha(IMG_RAM, IMG_SIZE, IMG_HASH);

    /* compute hash from signature */
    rsa(PUB_KEY, SIG_RAM, SIG_HASH)

    /* compare hashes */
    if (compare(IMG_HASH, SIG_HASH) != 0){
            while(1);
    }
jump_to_next_stage();
}
```

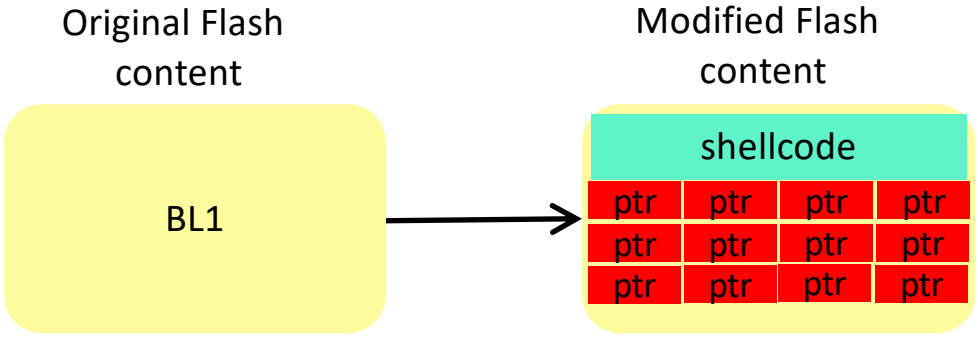
Bypassing secure boot




```
MultiWorldCopy:
  LDMIA r1!, {r3 - r10}
  STMIA r0!, {r3 - r10}
  SUBS r2, r2, #32
  BGE MultiWorldCopy
```

```
/* copy image from flash to sram */
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE)

/* decryption ? */
if (secure_boot_dec){
    /* decrypt image in place */
    decrypt(IMG_RAM, IMG_SIZE, PUB_KEY)
}
/* authentication ? */
if (secure_boot_en){
    /* copy signature from flash to sram */
    memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);

    /* compute hash over SRAM image */
    sha(IMG_RAM, IMG_SIZE, IMG_HASH);

    /* compute hash from signature */
    rsa(PUB_KEY, SIG_RAM, SIG_HASH)

    /* compare hashes */
    if (compare(IMG_HASH, SIG_HASH) != 0){
            while(1);
    }
jump_to_next_stage();
}
```

Bypassing secure boot




```
MultiWorldCopy:
  LDMIA r1!, {r3 - r10}
  STMIA r0!, {r3 - r10}
  SUBS r2, r2, #32
  BGE MultiWorldCopy
```

LDMIA r1!,{…,…,pc}

N. Timmers, A. Spruyt and M. Witteman.
Controlling PC on ARM Using Fault Injection. FDTC 2016

```
/* copy image from flash to sram */
memcpy(IMG_RAM, IMG_FLASH, IMG_SIZE)

/* decryption ? */
if (secure_boot_dec){
    /* decrypt image in place */
    decrypt(IMG_RAM, IMG_SIZE, PUB_KEY)
}
/* authentication ? */
if (secure_boot_en){
    /* copy signature from flash to sram */
    memcpy(SIG_RAM, SIG_FLASH, SIG_SIZE);

    /* compute hash over SRAM image */
    sha(IMG_RAM, IMG_SIZE, IMG_HASH);

    /* compute hash from signature */
    rsa(PUB_KEY, SIG_RAM, SIG_HASH)

    /* compare hashes */
    if (compare(IMG_HASH, SIG_HASH) != 0){
            while(1);
    }
jump_to_next_stage();
}
```

Real world fault attacks

- Traditionally, a smart-card concern
  - Security requirements include fault attack resistance
- Xbox reset glitch hack[1] -- 2011
  - Launch your own code (Linux kernel)
- Glitching the (Nintendo) Switch[2] -- 2018
- Firmware extraction of different widespread microcontrollers – 2019[3] & 2020[4]

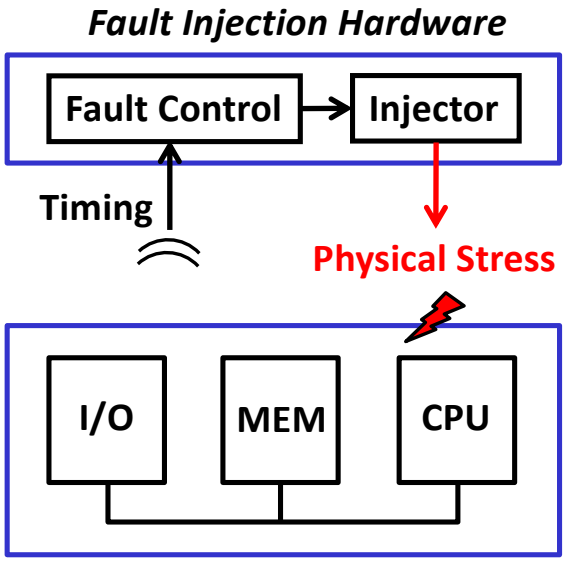
Often make use of vulnerabilities or lack of secure coding

[1] http://www.logic-sunrise.com/news-341321-the-reset-glitch-hack-a-new-exploit-on-xbox-360-en.html
[2] https://media.ccc.de/v/c4.openchaos.2018.06.glitching-the-switch
[3] https://tches.iacr.org/index.php/TCHES/article/view/7390
[4] https://tches.iacr.org/index.php/TCHES/article/view/8727

The present and the future

Hardware-controlled Fault Injection

1997 (Bellcore) - now



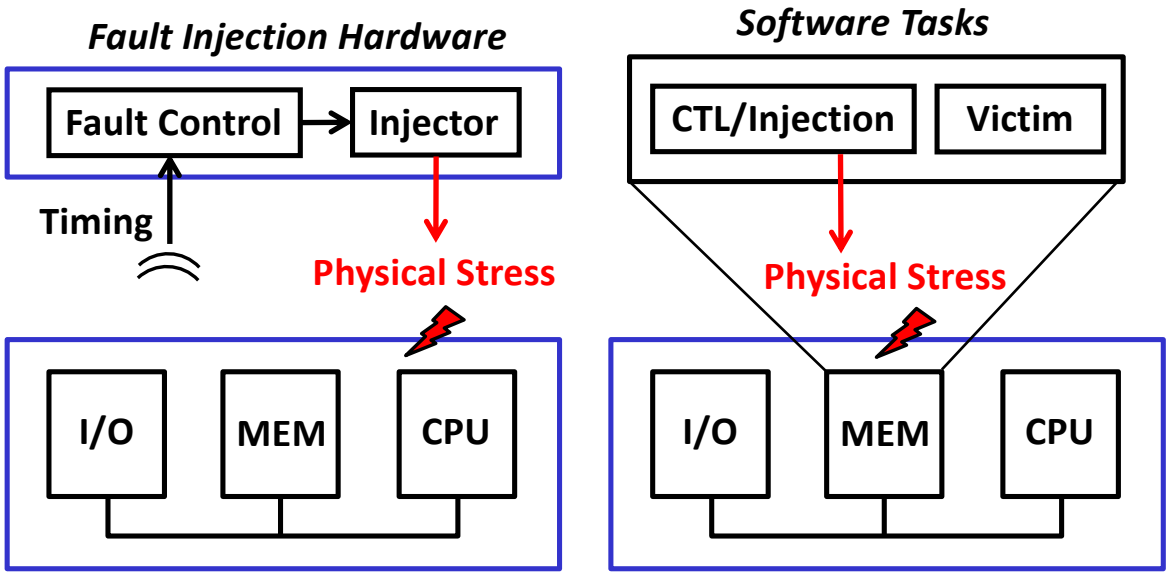
The present and the future

Hardware-controlled Fault Injection

1997 (Bellcore) - now

Software-controlled Fault Injection

2014 (Rowhammer) - now

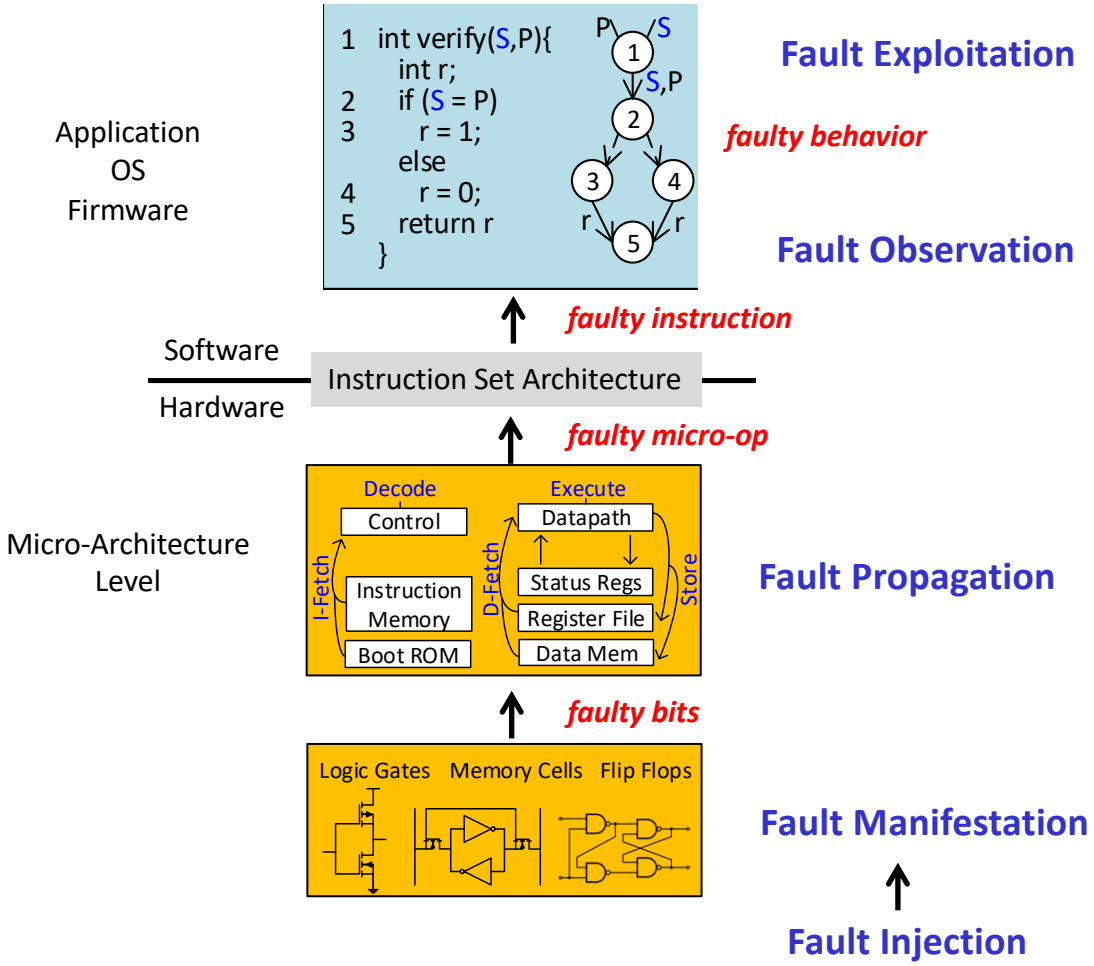


2014 : Rowhammer
2017 : CLKSCREW
2019 : VOLTJOCKEY
2020 : PlunderVolt
…

Agenda

- Fault exploitation
- **Fault effects and modelling**
- Countermeasures
- Robustness analysis
- Research lines in this area

Fault effects and fault modeling



Fault exploitation

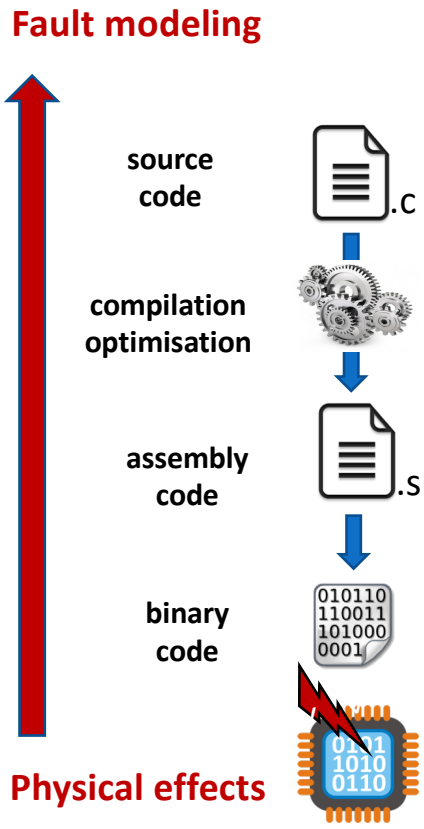
- **Macro view of fault attacks**
  - Cryptographic key retrieving [Dehbaoui 2013] [Kumar 2017]
  - Bypassing secure boot [Timmers 2016]
  - Taking over a device [Timmers 2017]
  - Privilege escalation [Vasselle 2017]
  - Firmware extraction [Bozzato 2019]
- **Useful from an attacker point of view**

Fault effects characterization

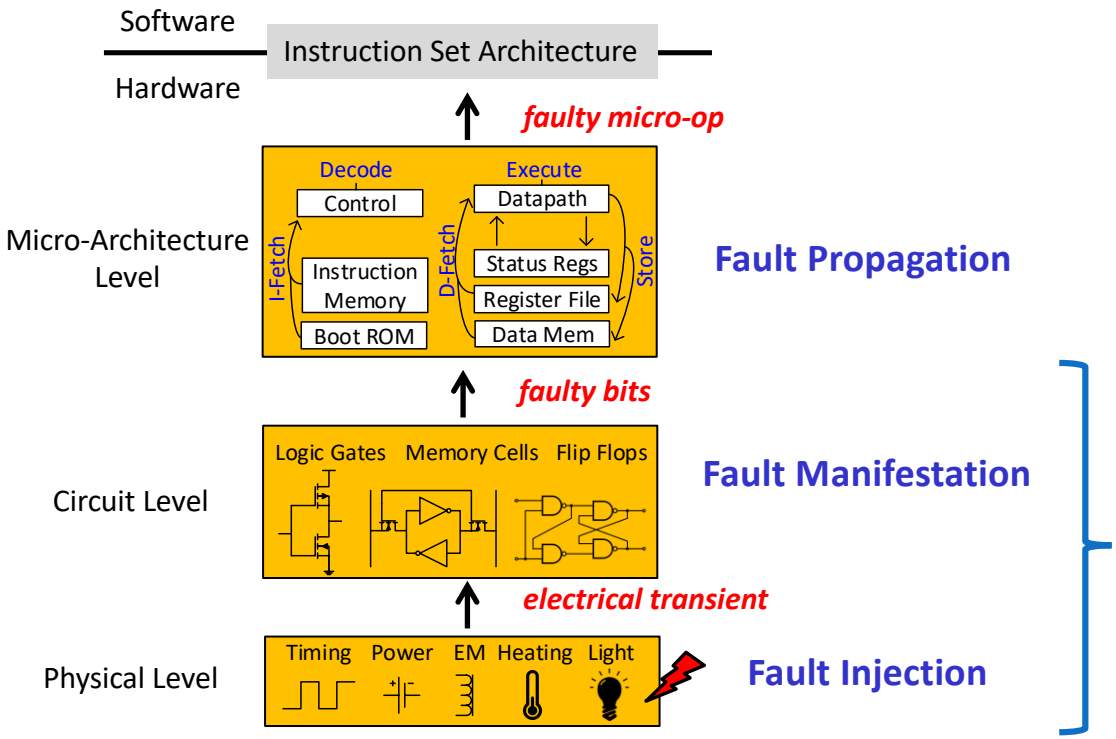
- **Necessary to design** countermeasures

Fault models

- **Simplified or abstracted representation** of a physical fault effects
- **At a given code level**
  - Hardware : logical, micro-architectural
  - Software : binary, assembly code, IR, source code



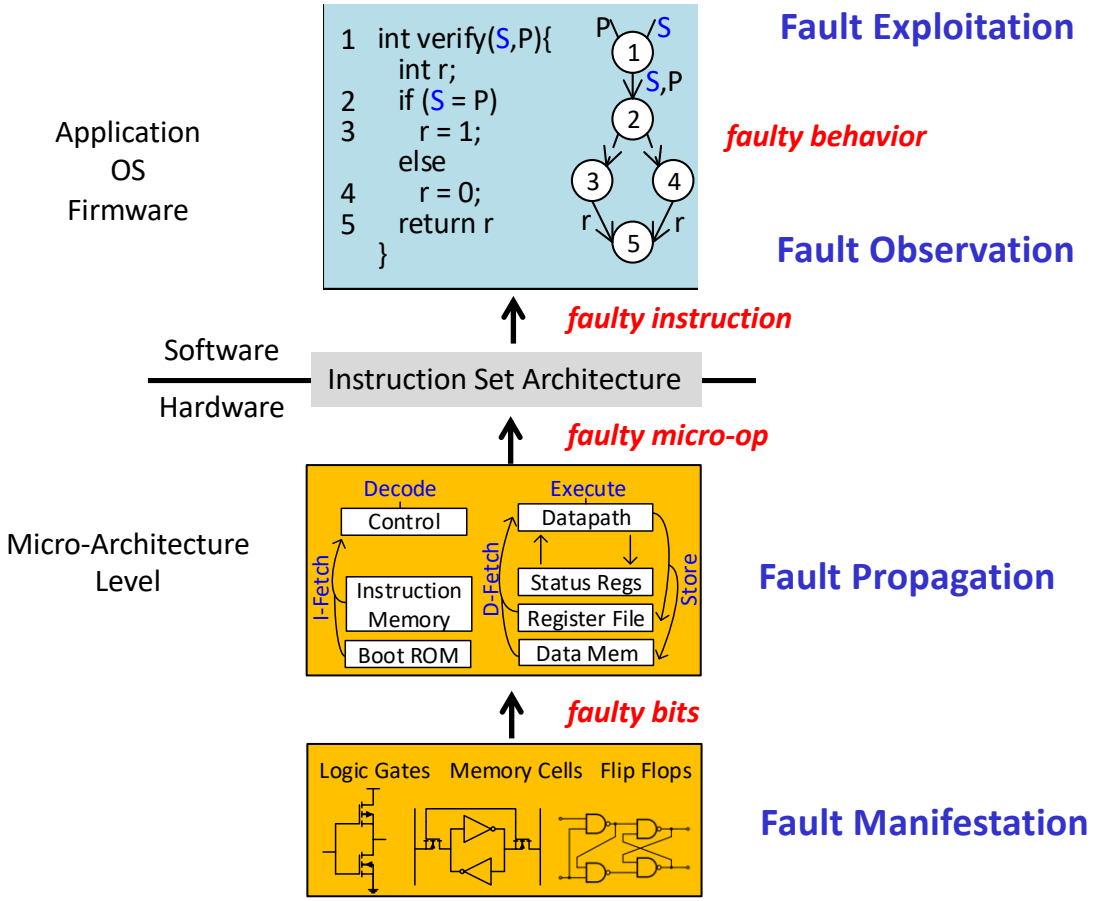
Fault attacks at hardware level



HW fault model

1. Granularity
   - Single bit, few bits, word
2. Fault type
   - Bitflip, set/reset, random
3. Location and timing control
   - Precise, loose, none
4. Fault duration
   - Transient, permanent, destructive

Fault attacks at software level



Observation depends on

- HW target
- Fault injection means
  - Clock/Power glitch
  - EM pulse
    - Targeted part of the HW
  - Laser
    - Targeted part of the HW
- **Running code**

Characterization of faults effect

- **No methodology** or easy way to characterize achievable faults (grey-box model)
- **Huge parameter space**: running code, parameters of the fault injection mean, target HW
- Common steps for SW fault modeling / characterization:
  1. **Scan the parameter space** to find out configurations where faulty outputs are observed
  2. **Select one configuration** with a high probability to observe a faulty output
  3. **Fault model elaboration** on this selected area

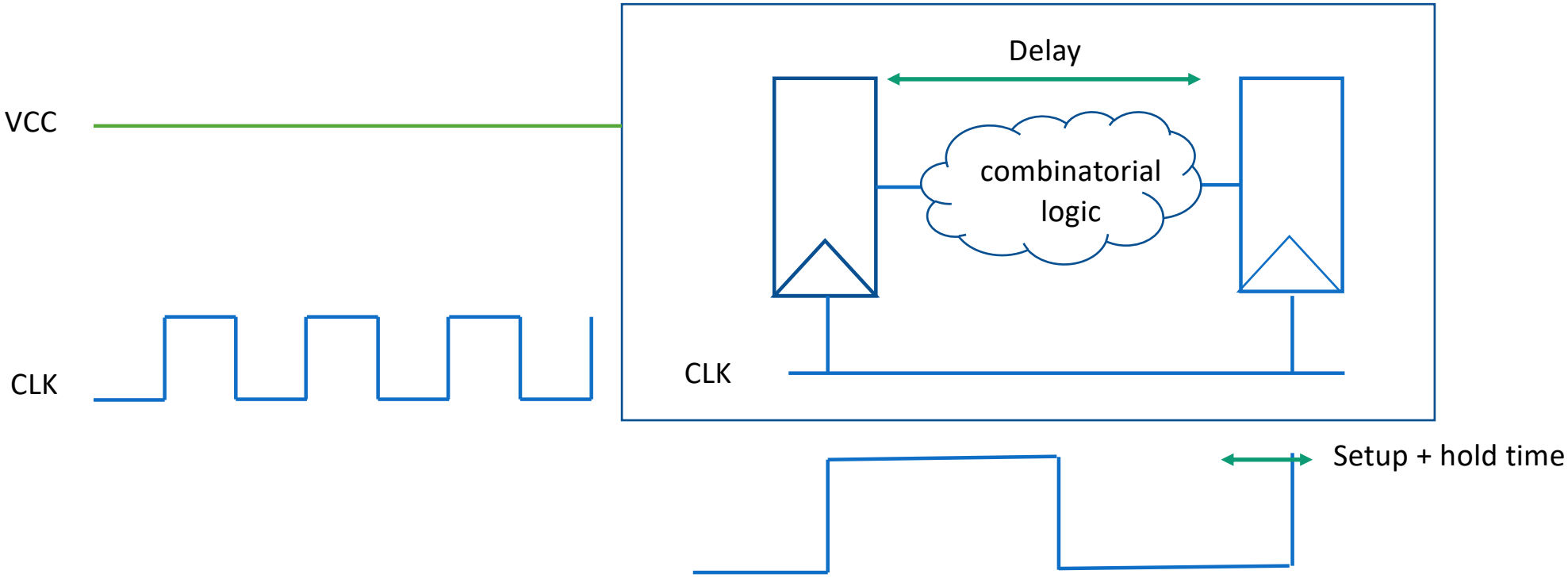
Modeling of fault effects up to software level

1. **Inject faults while running specific and carefully selected test codes :**
   - Put the processor in a known state A (contents of registers and memory)
   - Run a carefully chosen code (that normally leads to the final state S)
   - Inject a fault
   - Output the content of registers / memory – final state S'
2. **Analyze all the output S' by comparing it to the expected one S**
3. **Infer possible explanations / fault models at different level (e.g. micro-architectural level, ISA level)**
4. **Validate the fault models**
   - By simulation: comparison of observed results with the simulation outputs
   - By refinement: use specifically designed test codes and go back to step 1

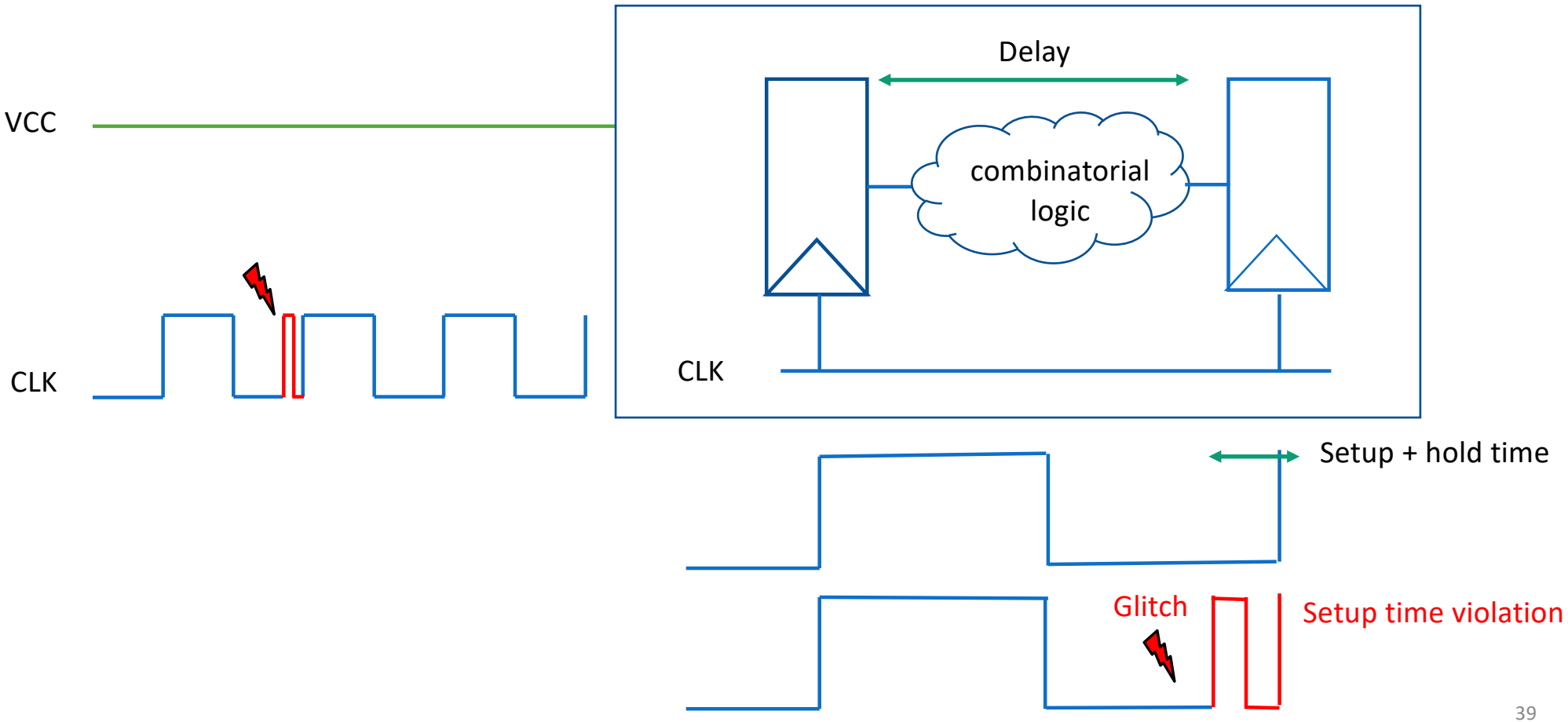
Modeling of fault effects up to software level

- Balasch et al., *An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MC*Us FDTC 2011.
- Moro et al., *Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller*. FDTC 2013.
- Dureuil et al., *From code review to fault injection attacks: Filling the gap using fault model inference*. CARDIS 2015.
- Kelly et al., *Characterising a CPU fault attack model via run-time data analysis*. HOST 2017
- Kumar et al. *An In-depth and Black-Box Characterization of the Effects of Laser Pulses on ATmega328P.* CARDIS 2018
- Colombier et al., *Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller,* HOST 2019.

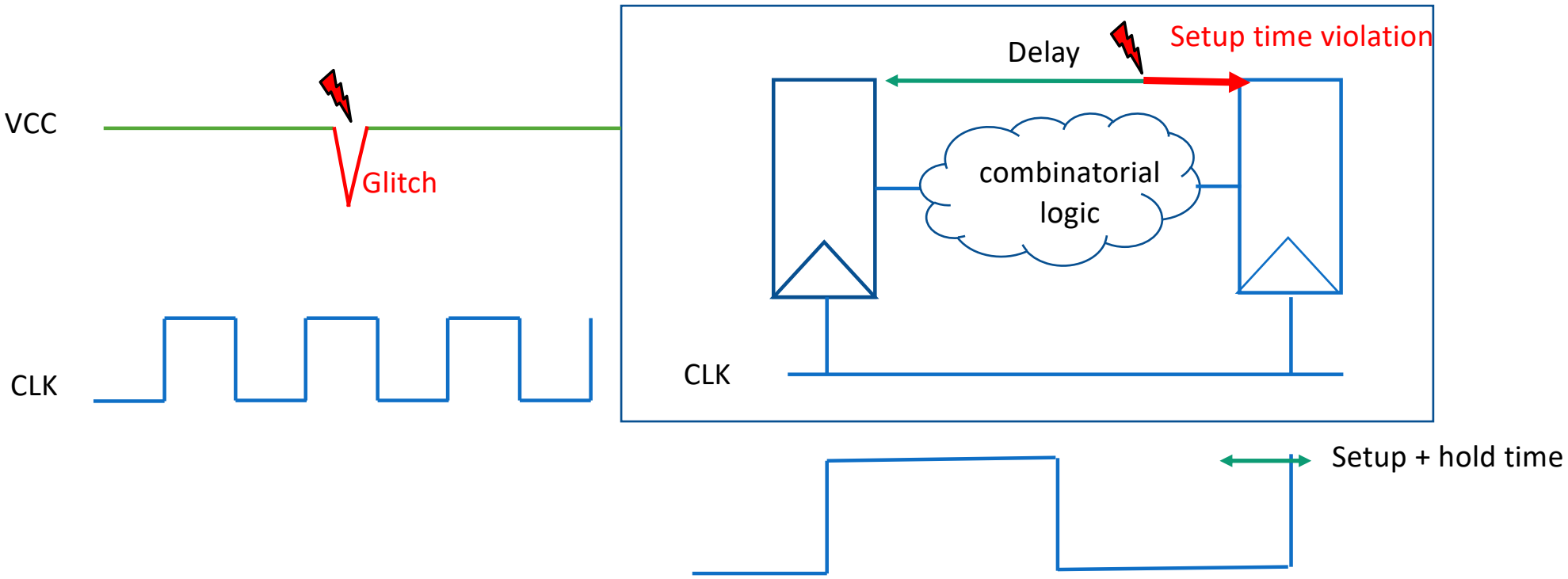
Voltage or clock glitch effects



Voltage or clock glitch effects

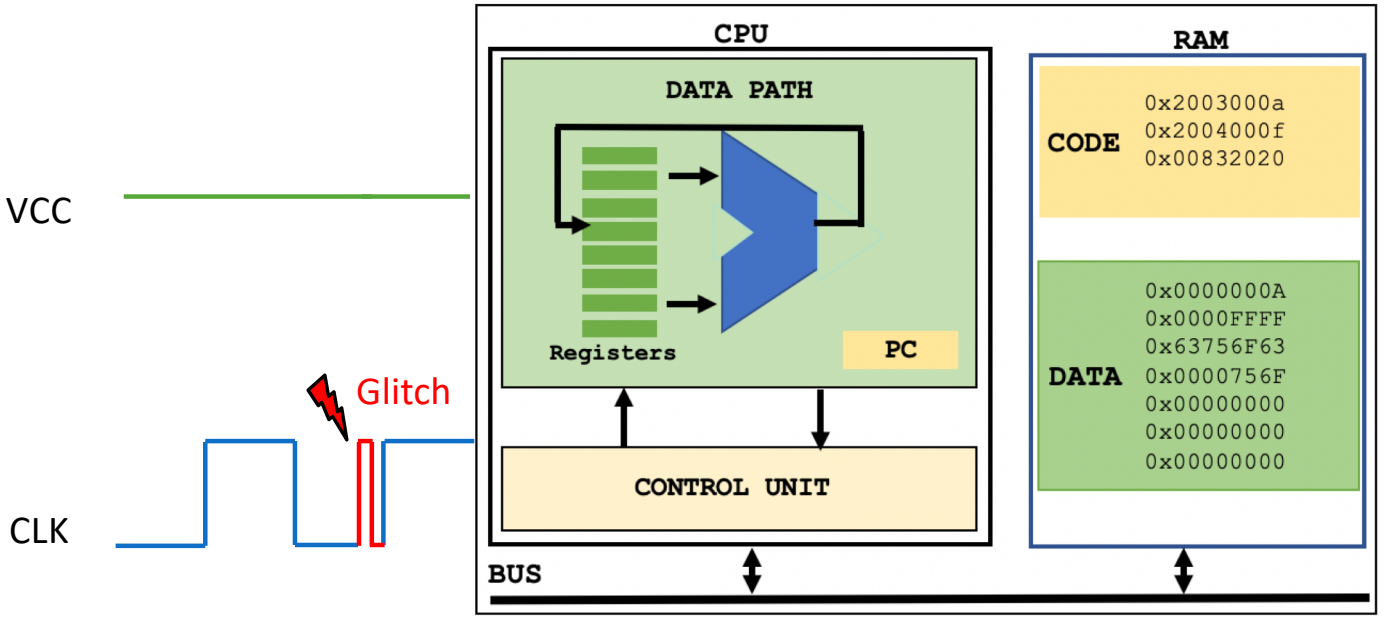


Voltage or clock glitch effects

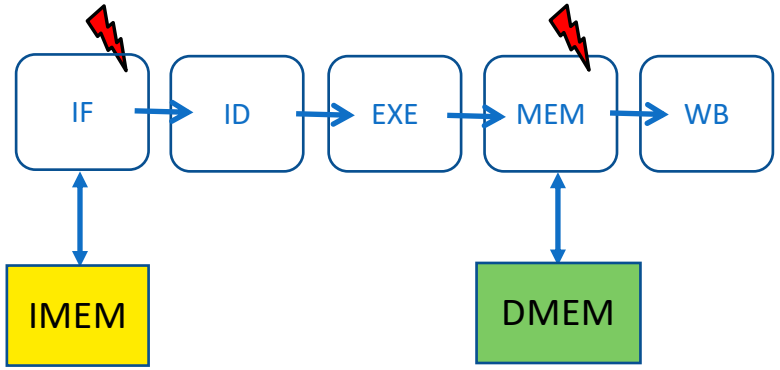


Voltage or clock glitch effects up to software level

- Instruction replacement
- Loaded data corruption
- Glitch parameters must be carefully selected !

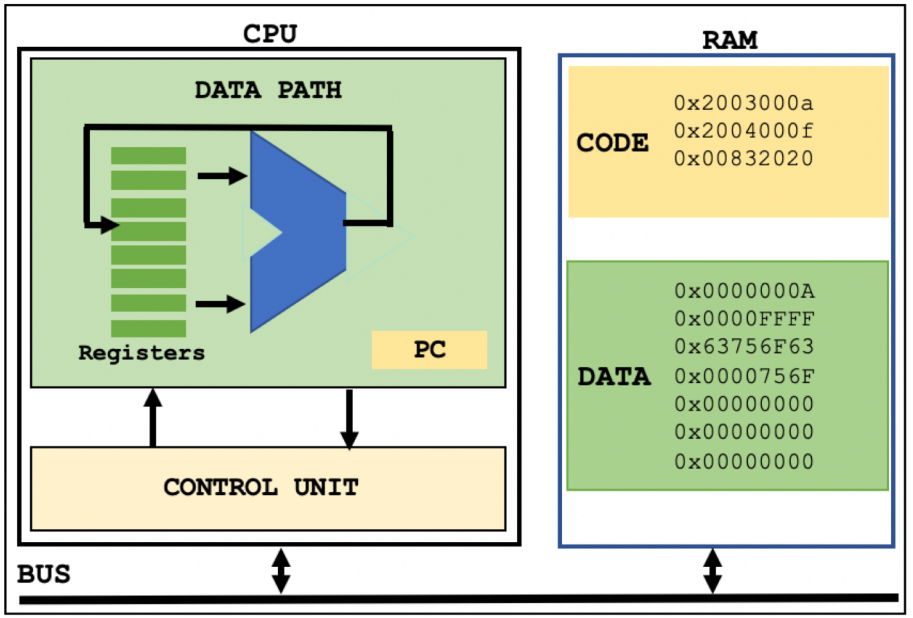


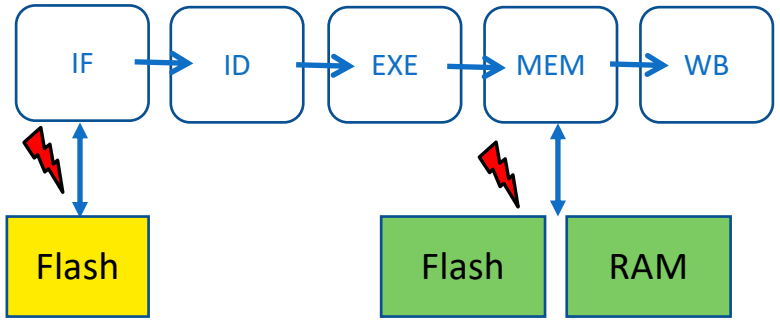
Balasch et al., *An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs* FDTC 2011.



EM pulse effects up to software level

- Corruption of transfers from/to the Flash
- Instruction replacement : 25% equivalent to “skip instruction”
- Loaded data corruption





Moro et al., *Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller*. FDTC 2013.

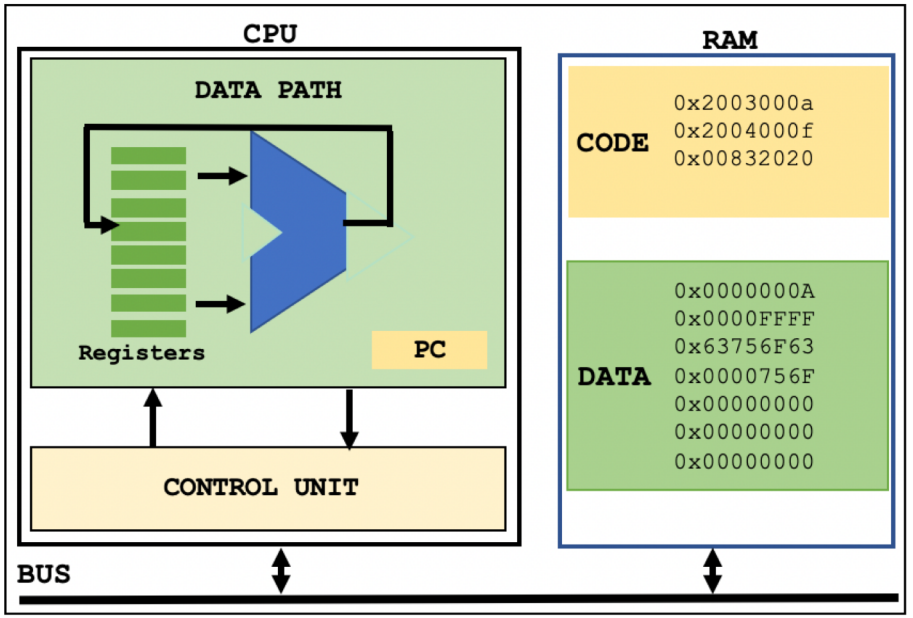
Laser beam effects up to software level

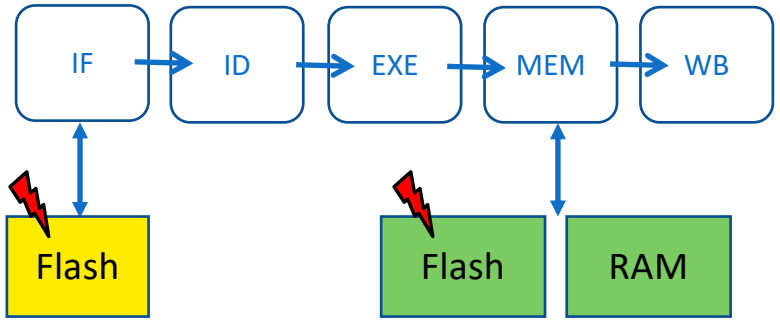
Past work

- Precise fault (bit granularity) in memory or register
- Data or register corruption

Recent work

- Transient faults in Flash memory
- Bit set or bit reset depending on the Flash
- Precision : faulty bit / pair of bits
- Instruction replacement or data corruption

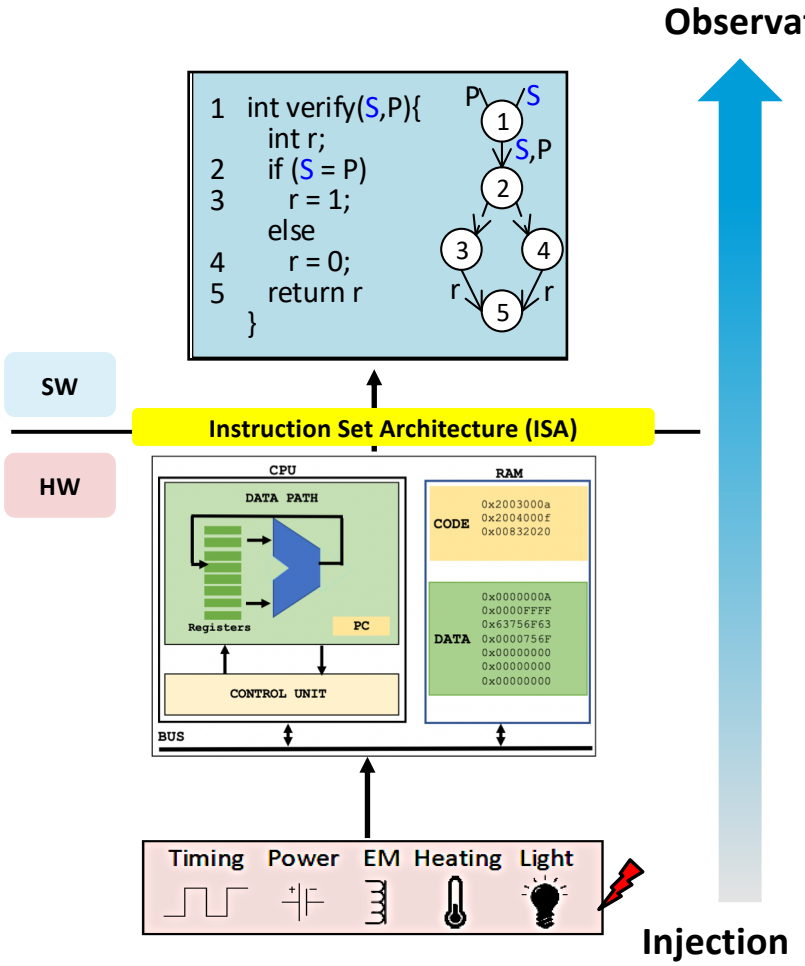




Colombier et al., *Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller,* HOST 2019.

Kumar et al. *An In-depth and Black-BoxCharacterization of the Effects of Laser Pulses on ATmega328P.* CARDIS 2018

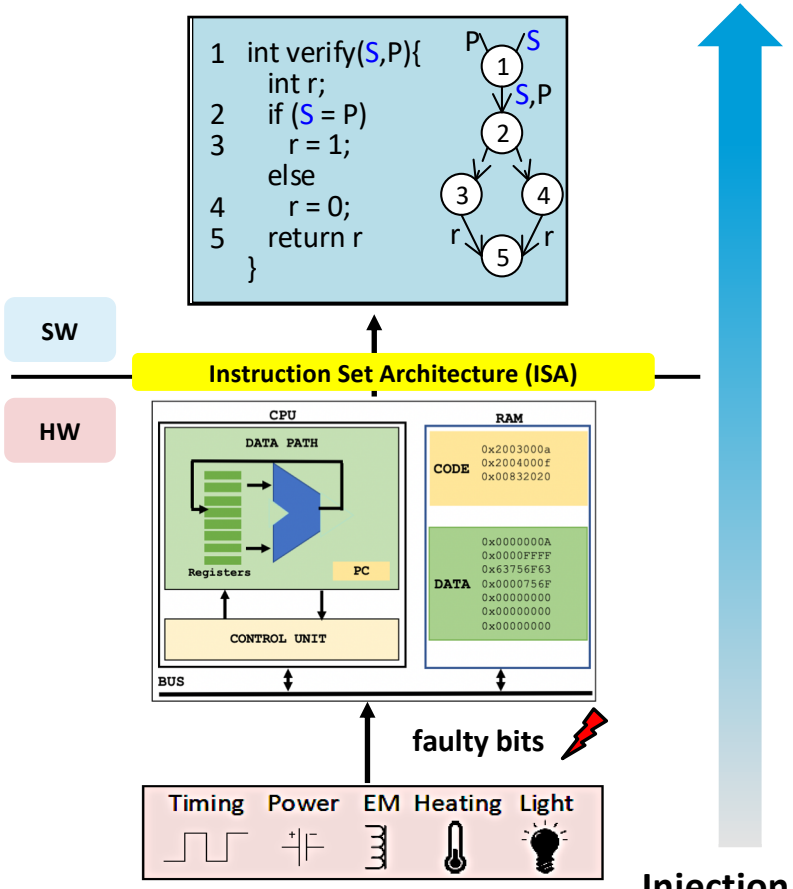
Fault injection effects



[Yuce et al., 2018]

Fault injection effects

Observations/attacker's exploitation



[Yuce et al., 2018]

Injection

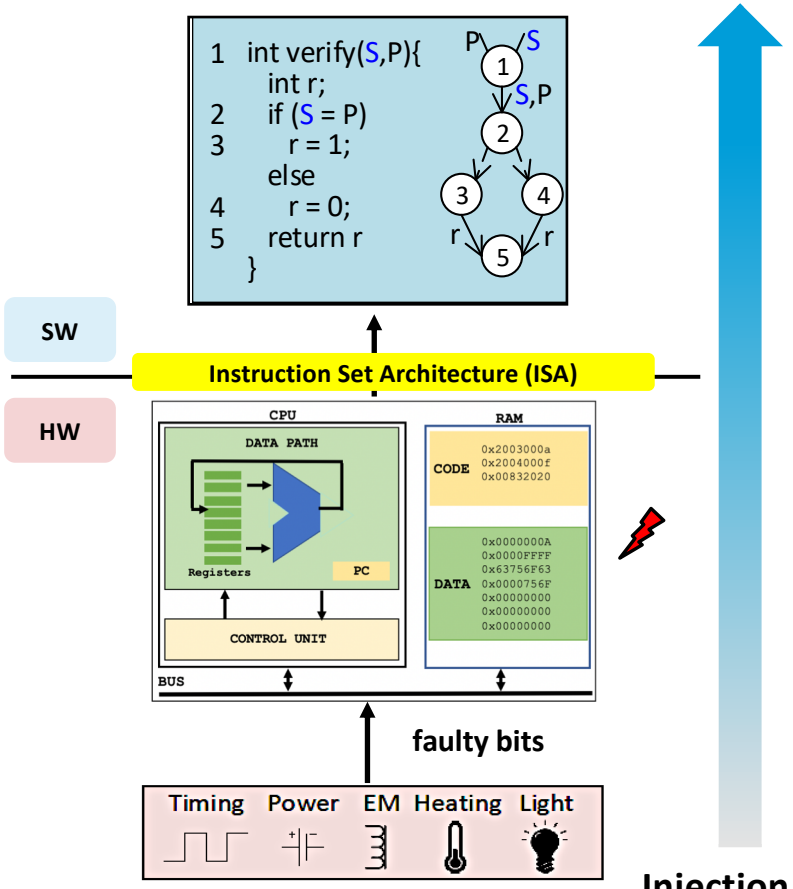
1100101100101110100 1

1100101100101100100 1

Alteration of currents & charges

Fault injection effects

Observations/attacker's exploitation



[Yuce et al., 2018]

Instruction or data corruptions

1100101100101110100 1

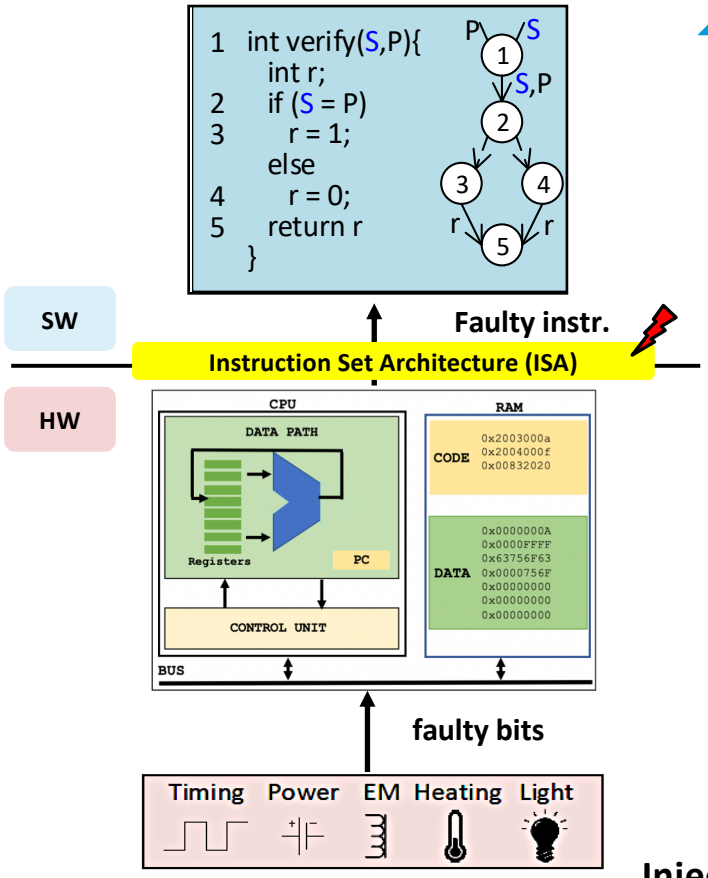
1100101100101100100 1

Alteration of currents & charges

Injection

Fault injection effects

Observations/attacker's exploitation



[Yuce et al., 2018]

Injection

Instruction skip

```
add     r3, r2, r1
strb    r3, [r0]
```

register corruption

```
add     r3, r2, r1
strb    r3, [r0]
```

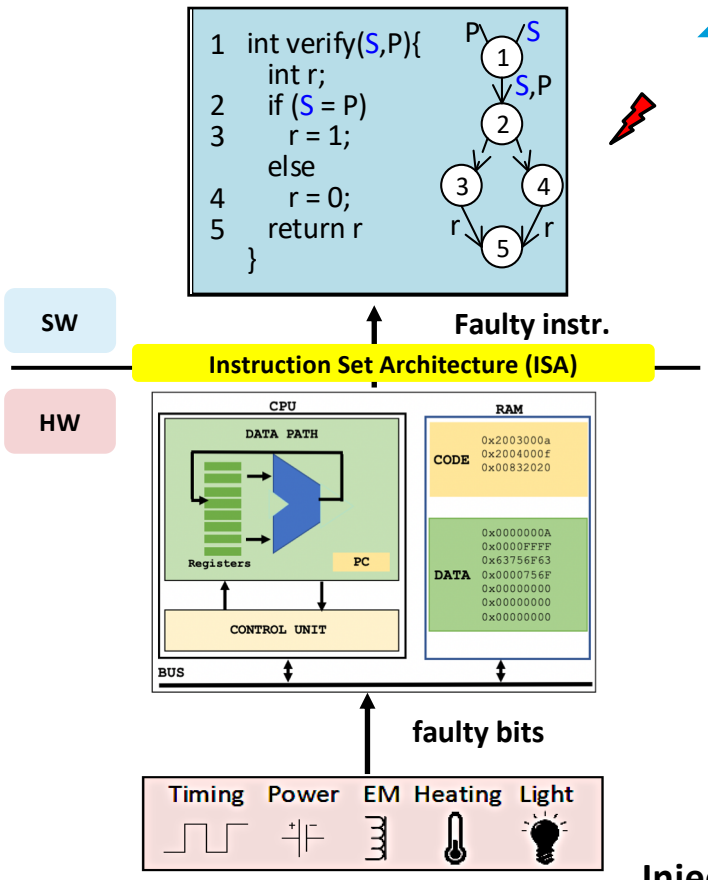
Instruction or data corruptions

```
11001011001011101001
11001011001011001001
```

Alteration of currents & charges

Fault injection effects

Observations/attacker's exploitation



[Yuce et al., 2018]

Injection

?

Instruction skip

```
add     r3, r2, r1
strb    r3, [r0]
```

register corruption

```
add     r3, r2, r1
strb    r3, [r0]
```

Instruction or data corruptions

```
11001011001011101001
11001011001011001001
```

Alteration of currents & charges

Fault impact at source level

Instruction skip at assembly level

- The skipped instruction writes into a **general purpose register** (add, load, …)
  - Next use of faulty register will propagate the fault
  - Data corruption

```
a = b + c;
```

```
add     r3, r2, r1
strb    r3, [r0]
```

```
add     r3, r2, r1
strb    r3, [r0]
```

```
a = attack();
```

Fault impact at source level

Instruction skip at assembly level

- The skipped instruction writes into a **general purpose register** (add, load, …)
  - Next use of faulty register will propagate the fault
  - Data corruption

```
a = b + c;
```

```
add     r3, r2, r1
strb    r3, [r0]
```

```
add     r3, r2, r1
strb    r3, [r0]
```

- **Equivalent to the corruption of destination register**

```
a = attack();
```

Fault impact at source level

Instruction skip at assembly level

- The skipped instruction writes into a **general purpose register** (add, load, …)
  - Next use of faulty register will propagate the fault
  - Data corruption

```
a = b + c;
```

```
add     r3, r2, r1
strb    r3, [r0]
```

```
add     r3, r2, r1
strb    r3, [r0]
```

- **Equivalent to the skip of the store instruction**

```
a = attack();
```

Fault impact at source level

Instruction skip at assembly level

- The skipped instruction writes into a **general purpose register** (add, load, …)
  - Next use of faulty register will propagate the fault
  - Potential branch corruption / test inversion

```
cond = *ch;
if (cond)
  {
  // do something1
  }
else
  {
  // do something2
  }
```

```
        ldr r3, [r0]
        cmp r3, #0
        b.ne then
else:   …
        …
        …
        j next
then:   …
        …
next:
```

```
        ldr r3, [r0]
        cmp r3, #0
        b.ne then
else:   …
        …
        …
        j next
then:   …
        …
next:
```

```
cond = *ch;
goto label_then;
if (!cond)
label_then:
  {
  // do something1
  }
else
  {
  // do something2
  }
```

```
cond = *ch;
goto label_else;
  {
  // do something1
  }
else
  {
label_else:
  // do something2
  }
```

- **Equivalent to a jump insertion**

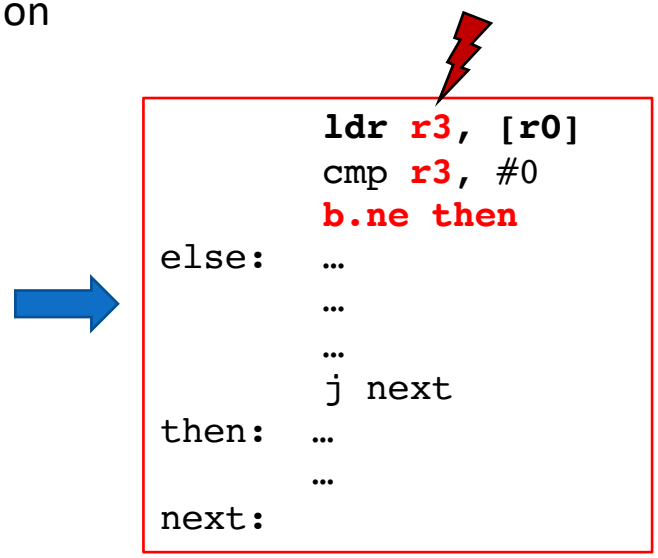
Fault impact at source level

Instruction skip at assembly level

- The skipped instruction writes into a **general purpose register** (add, load, …)
  - Next use of faulty register will propagate the fault
  - Potential branch corruption / test inversion

```
cond = *ch;
if (cond)
  {
  // do something1
  }
else
  {
  // do something2
  }
```

```
        ldr r3, [r0]
        cmp r3, #0
        b.ne then
else:   …
        …
        …
        j next
then:   …
        …
next:
```




```
cond = *ch;
goto label_then;
  {
label_then:
  // do something1
  }
else
  {
  // do something2
  }
```

```
cond = *ch;
goto label_else;
  {
  // do something1
  }
else
  {
label_else:
  // do something2
  }
```

- **Equivalent to a transient memory corruption** (load instruction)

Fault impact at source level

Instruction skip at assembly level

- The skipped instruction writes into a **general purpose register** (add, load, …)
  - Next use of faulty register will propagate the fault
  - Potential branch corruption / test inversion

```
cond = *ch;
if (cond)
  {
  // do something1
  }
else
  {
  // do something2
  }
```

```
        ldr r3, [r0]
        cmp r3, #0
        b.ne then
else:   …
        …
        …
        j next
then:   …
        …
next:
```

```
        ldr r3, [r0]
        cmp r3, #0
        b.ne then
else:   …
        …
        …
        j next
then:   …
        …
next:
```

- **Equivalent to a corruption of the flags** (`cmp` instruction skip or directly)

```
cond = *ch;
goto label_then;
  {
label_then:
  // do something1
  }
else
  {
  // do something2
  }
```

```
cond = *ch;
goto label_else;
  {
  // do something1
  }
else
  {
label_else:
  // do something2
  }
```

Fault impact at source level

Instruction skip at assembly level

- The skipped instruction is a **conditionnal or unconditionnal jump**
  - The fall-through block will be executed
  - Potential control-flow corruption

```
cond = *ch;
if (cond)
  {
   // do something1
  }
else
  {
  // do something2
  }
```

```
        ldr r3, [r0]
        cmp r3, #0
        b.ne then
else:   …
        …
        …
        j next
then:   …
        …
next:
```

```
        ldr r3, [r0]
        cmp r3, #0
        b.ne then
else:   …
        …
        …
        j next
then:   …
        …
next:
```

- **Equivalent to a jump insertion**

```
cond = *ch;
goto label_else;
  {
  // do something1
  }
else
  {
label_else:
  // do something2
  }
```

```
cond = *ch;
if (cond)
  {
label_then:
  // do something1
  }
else
  {
  // do something2
goto label_then;
  }
```

Fault impact at source level

Instruction skip at assembly level

- The skipped instruction is a **conditionnal or unconditionnal jump**
  - Th
  - P

Effect of a instruction skip of a jump depends on the code layout, which is unknown at source level!

```
cond = *ch;
goto label_else;
  {
  // do something1
  }
```

```
cond = *
if (cond
  {
   // do
  }
else
  {
  // do something2
  }
```

```
then:  …
       …
next:
```

```
then:  …
       …
next:
```

```
  }
else
  {
  // do something2
goto label_then;
  }
```

  - **Equivalent to a jump insertion**

Fault impact at source level

Fault impacting a general purpose register

- Next use(s) of faulty register will propagate the fault

- **Consequences**
  - Data corruption(s)
  - Control-flow corruption

Fault impact at source level

Fault impacting a general purpose register

- Next use(s) of faulty register will propagate the fault

- **Consequences / fault models at source level**
  - Data corruption(s): `var = attack();`
  - Control-flow corruption: `goto label;`

Fault impact at source level

Fault impacting a general purpose register

- Next use(s) of faulty register will propagate the fault

- **Consequences / fault models at source level**
  - Data corruption(s): **`var = attack();`**
  - Control-flow corruption: **`goto label;`**

- Fault propagation related to
  - Subsequent uses of the faulty register: « criticality »
  - Initial code and code compilation/optimization

```
        ld    r3, [r0]
        st    r3, [r1]
        …
        bnz   r3, then
else:
        …
        …
        j next
then:
        …
next:
```

Fault impact at source level

Instruction replacement

- One instruction is skipped
- One unexpected instruction is executed
- Combination of instruction skip effects with the one of the extra instruction
- From an attacker point of view
  - Only exploitation matters
  - Need to keep the effect as controllable as possible
    - Instruction skip is the most convenient
    - Can be achieved through different injection means

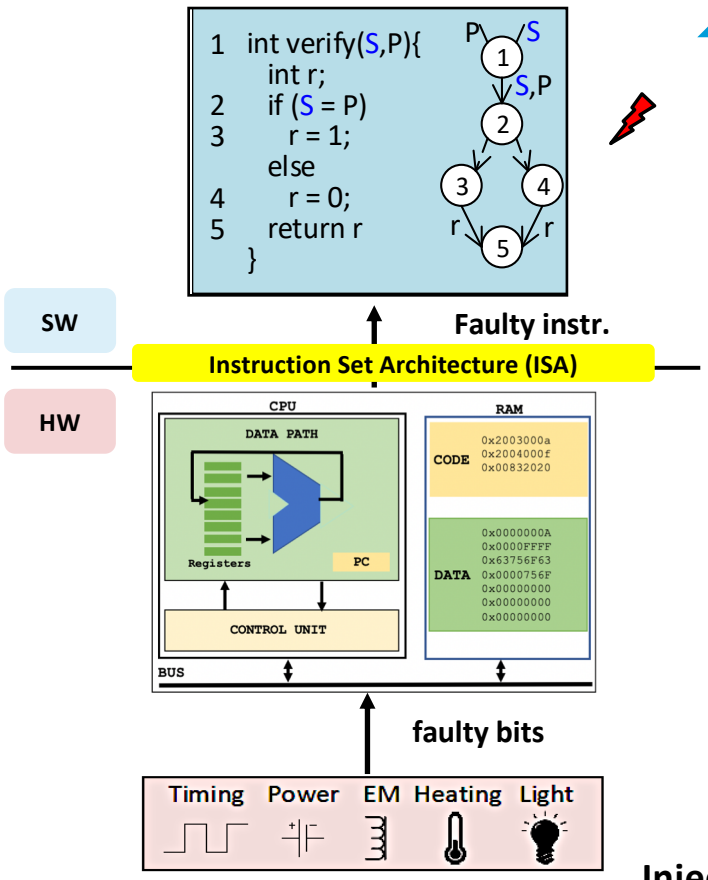
```
mem_cpy:
        push    {r4, r5, lr}
        movs    r3, #0
.L2:
        cmp     r2, r3
        bge     .L7
        ldrb    r5, [r0, r3]
        strb    r5, [r1, r3]
        bne     .L5
        adds    r3, r3, #1
        b       .L2
.L7:
        movs    r0, #0
        pop     {r4, r5, pc}
```

Fault model at source level

- No one-to-one correspondence between fault models at instruction level and source level
  - A statement is translated into several assembly instructions
  - Several faults at assembly level can result into the same fault at source level
  - A fault according to a source code fault model may not exist once the code is compiled
- Some faults at assembly level cannot be directly expressed at source-code level
  - Code placement, code optimization
- Source-code fault models are necessary
  - Source code protection
  - Vulnerability analysis

Fault injection effect

Observations/attacker's exploitation



[Yuce et al., 2018]

Injection

?

Instruction skip

```
add     r3, r2, r1
strb    r3, [r0]
```

register corruption

```
add     r3, r2, r1
strb    r3, [r0]
```

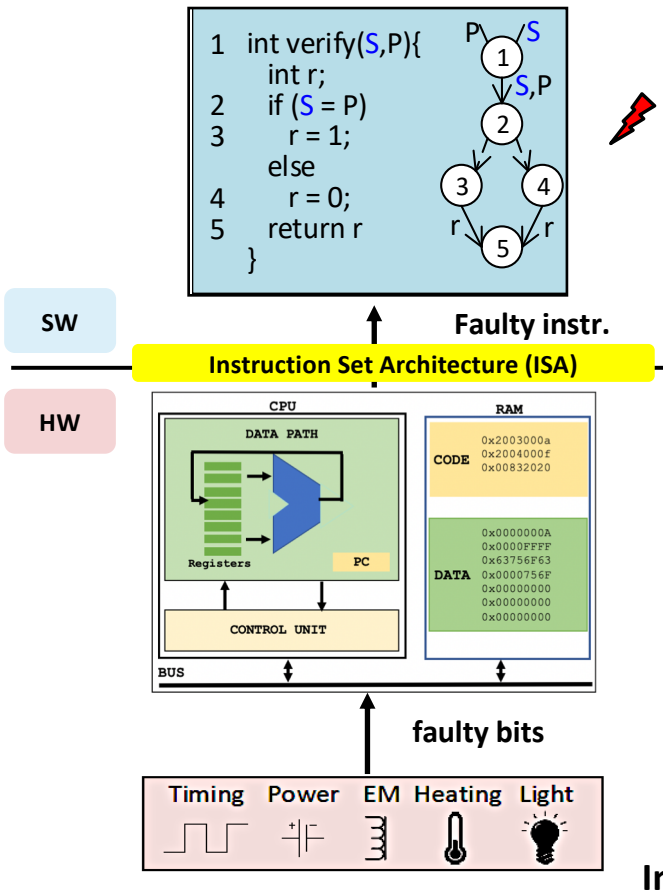
Instruction or data corruptions

```
11001011001011101001
11001011001011001001
```

Alteration of currents & charges

Fault injection effect

Observations/attacker's exploitation



[Yuce et al., 2018]

Injection

Control flow disruption (test inversion, jump insertion) & variable corruption & possible combination

Instruction skip

```
~~add     r3, r2, r1~~
strb    r3, [r0]
```

register corruption

```
add     r3, r2, r1
strb    r3, [r0]
```

Instruction or data corruptions

```
11001011001011101001
11001011001011001001
```

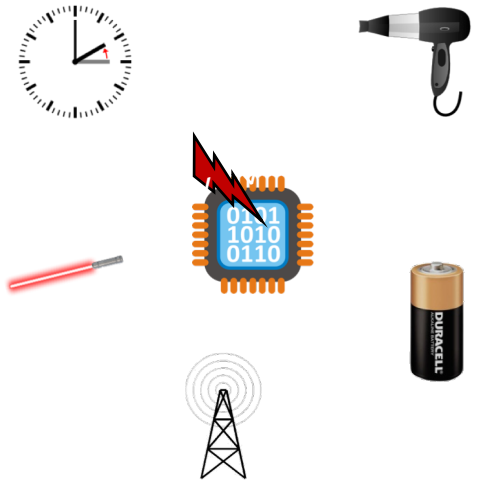
Alteration of currents & charges

Agenda

- Fault exploitation
- Fault effects and modelling
- **Countermeasures**
- Robustness analysis
- Conclusion and perspectives

Protections against fault injection attacks

- **Hardware-based countermeasures** [El Bar et al., 2006]
  - Jitters
  - Light sensor, glitch detectors [Zussa et al., 2014]
  - Redundancy [Karaklajic et al, 2013]
  - Error correcting codes (registers, memory)
- **No full guaranty**
- **Software-based countermeasures** [Verbauhede, 2011] [Rauzy et al., 2015]
  - Redundancy at function level
  - Algorithm-specific protection (e.g. RSA)
  - Ad-hoc protections designed by expert engineers
- In practice combination of both in secure elements



Countermeasures

- Principle of software countermeasures
  - Data integrity
  - Code integrity
  - Control-flow integrity
  - Limitations
- Compiler-assisted code hardening
  - Protection against instruction skip
  - Loop hardening scheme
  - Limitations

Countermeasures

- **Principle of software countermeasures**
  - Data integrity
  - Code integrity
  - Control-flow integrity
  - Limitations
- Compiler-assisted code hardening
  - Protection against instruction skip
  - Loop hardening scheme
  - Limitations

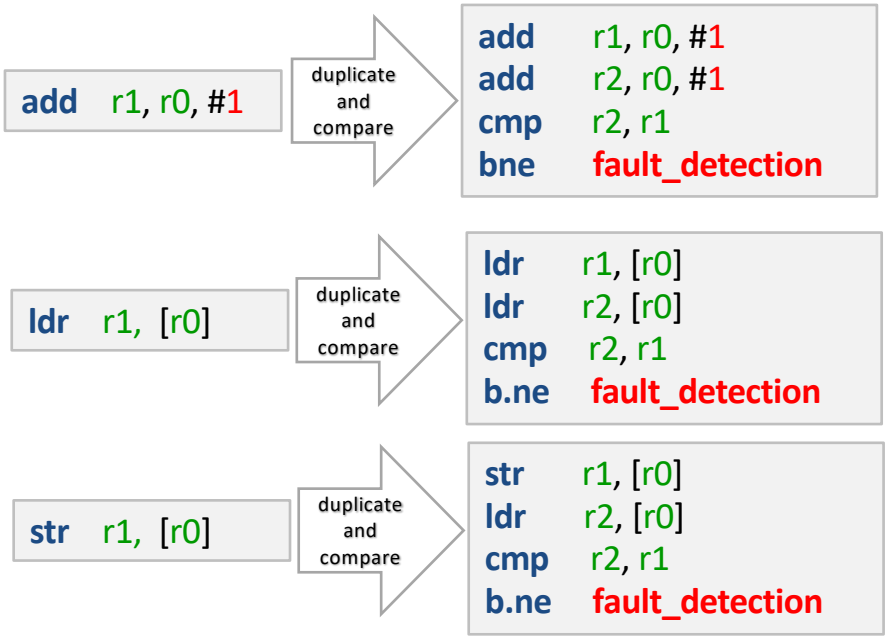
Countermeasures for data integrity

Fault model

- Data corruption: register corruption, load/store corruption

Redundancy-based protections

- Duplication of instructions involved in a computation
- Comparison of results of duplicated computations
- Detection of
  - Register corruption (r1 or r2)
  - Load / store corruption
- Need available registers



A. Barenghi et al. *Countermeasures against fault attacks on software implemented AES*. 5th Workshop on Embedded Systems Security (WESS'10)

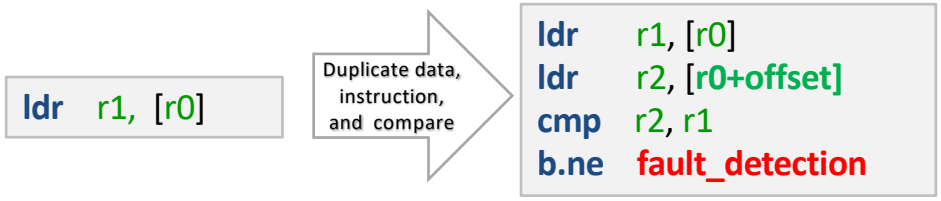
Countermeasures for data integrity

Fault model

- Data corruption: register corruption, load/store corruption and memory corruption

Redundancy-based protections

- Data duplication in addition to instruction duplication
- Detection of
  - Memory corruption
  - Load/store corruption
  - Register corruption
- High overhead: performance and memory footprint



Reis et al. *SWIFT: Software Implemented Fault Tolerance.* International Symposium on Code Generation and Optimization. 2005

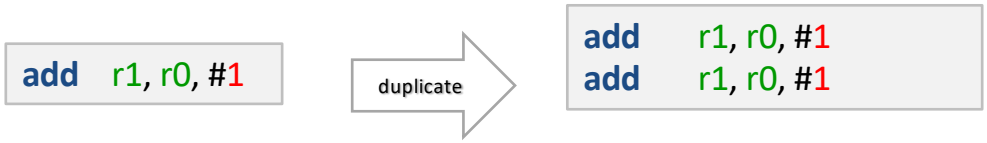
Countermeasures for code integrity

Fault model

- Instruction skip

Redundancy-based protections

- Instruction duplication without detection
  - Tolerance to one instruction skip (n-replication if needed)
  - Only for idempotent instructions
  - Transformation of non-idempotent instructions



Moro et al. *Formal verification of a software countermeasure against instruction skip attacks.* Journal of Cryptographic Engineering 2014.

Countermeasures for code integrity

Redundancy-based protections

- More complex transformation of non-idempotent instructions
- The function call example

Moro et al. *Formal verification of a software countermeasure against instruction skip attacks.* Journal of Cryptographic Engineering 2014.

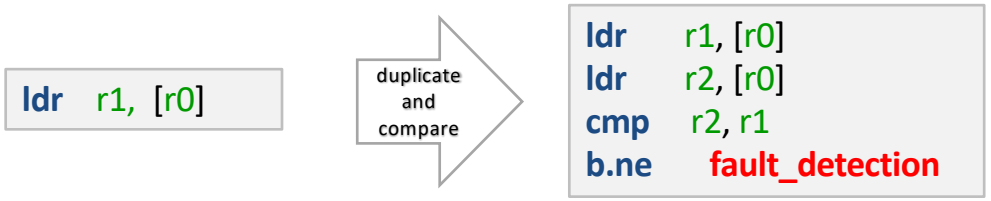
Countermeasures for code integrity

Fault model

- Instruction replacement

Redundancy-based protections

- Instruction duplication with detection
- Detection of
  - One instruction skip
  - Some instruction replacements



A. Barenghi et al. *Countermeasures against fault attacks on software implemented AES*.
5th Workshop on Embedded Systems Security (WESS'10)

Countermeasures for code integrity

Fault model

- Instruction corruption

Redund

- Instru
- Detec
  - C
  - S

No software protection for full code integrity
(i.e. against all kinds of instruction replacement or disruption)

Need HW support !

A. Barenghi et al. *Countermeasures against fault attacks on software implemented AES*.
5th Workshop on Embedded Systems Security (WESS'10)

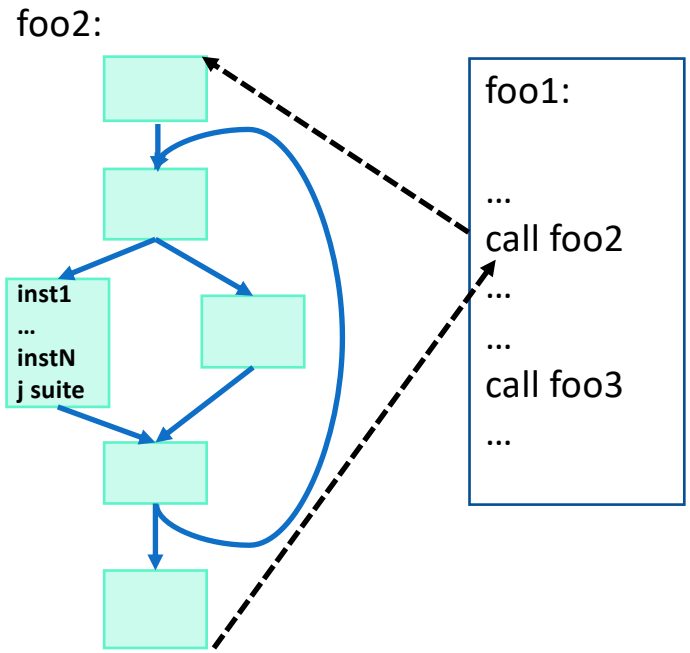
Control flow integrity

Fault model

- Jump insertion

Different levels of control-flow integrity

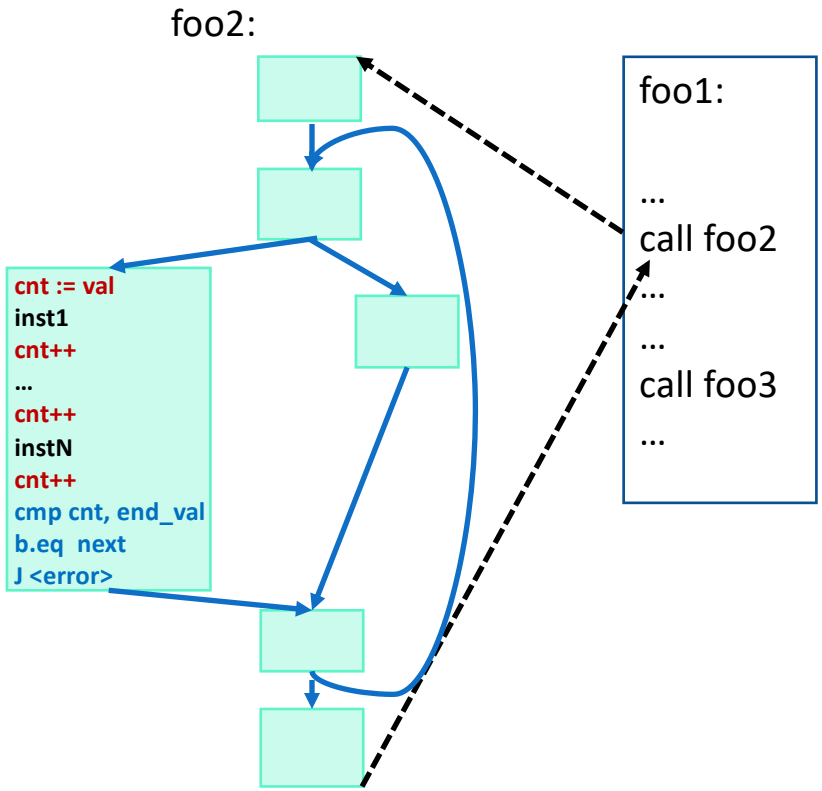
- Intra basic block
  integrity of straight-line code
- Intra procedural
  integrity of control flow transfers inside a function (control flow graph)
- Inter procedural
  integrity of function calls and returns



Intra basic block control flow integrity

Counter-based protections [Akkar et al., 2003]

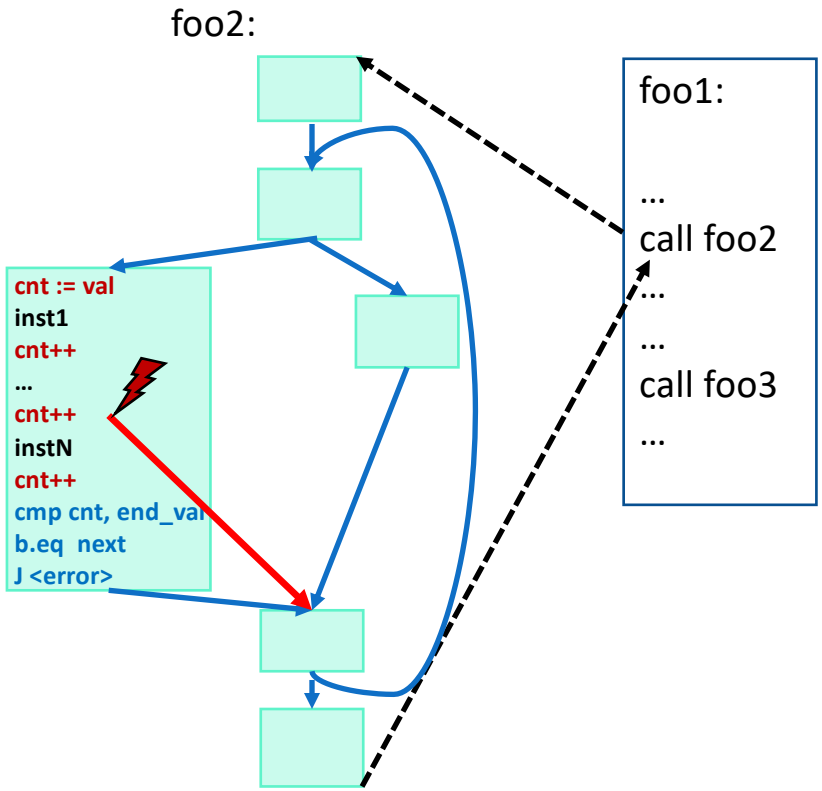
- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB



Intra basic block control flow integrity

Counter-based protections [Akkar et al., 2003]

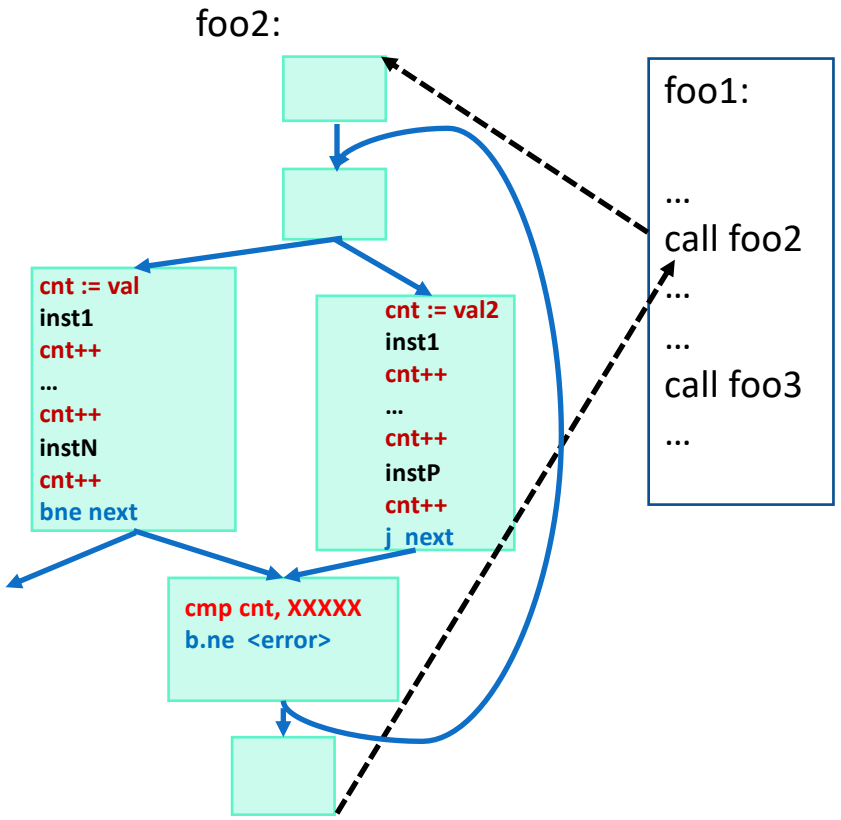
- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps



Control flow integrity

Counter-based protections

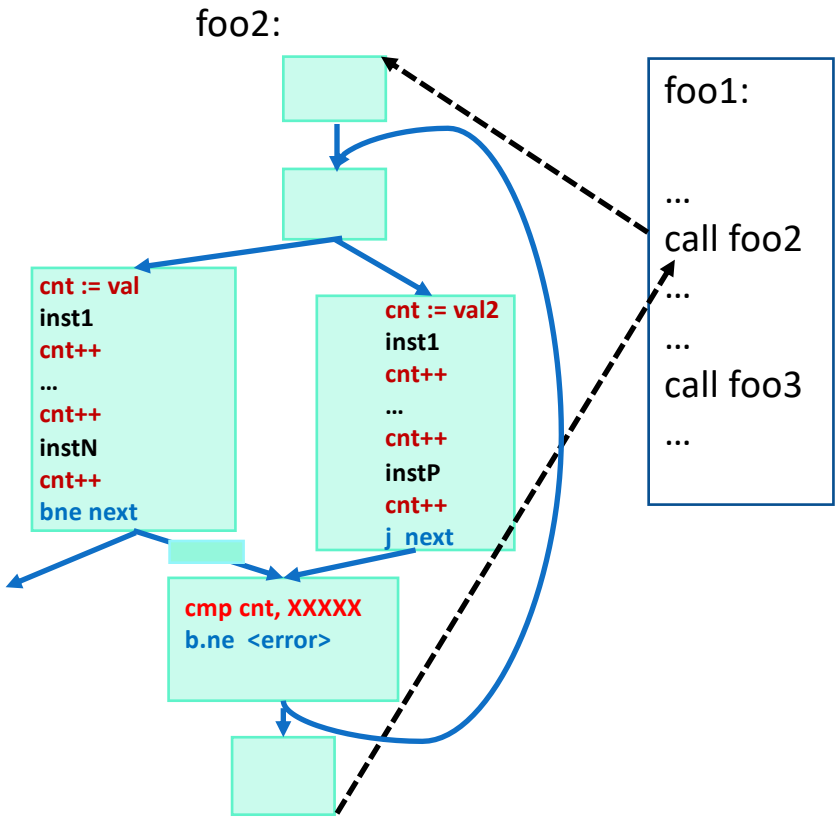
- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps
  - At the beginning of target blocks



Control flow integrity

Counter-based protections

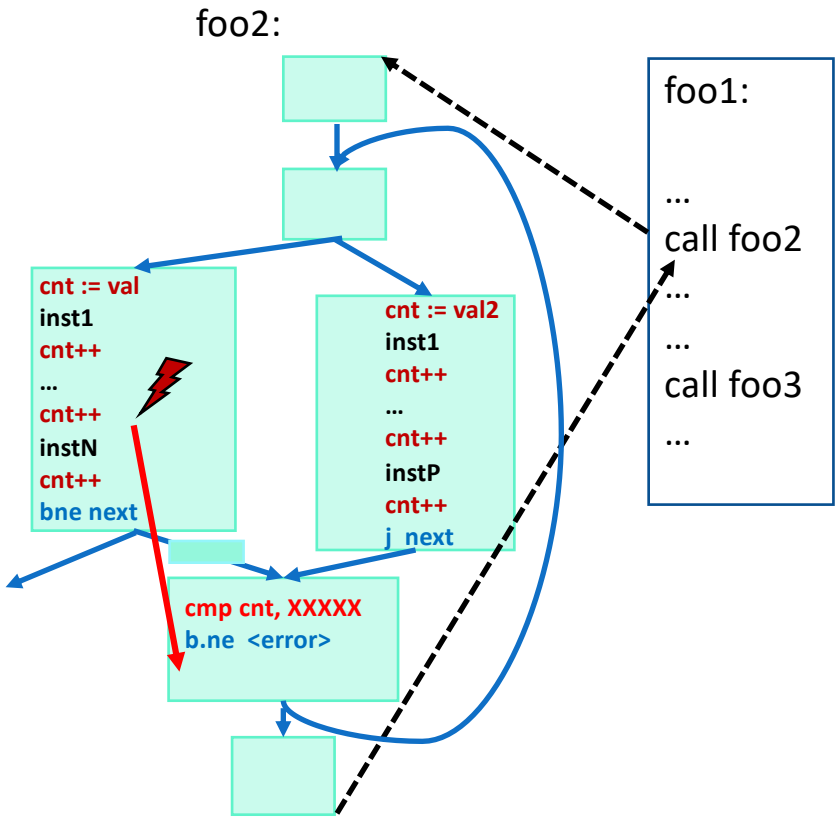
- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps
  - At the beginning of target blocks
    - Need for extra code



Control flow integrity

Counter-based protections

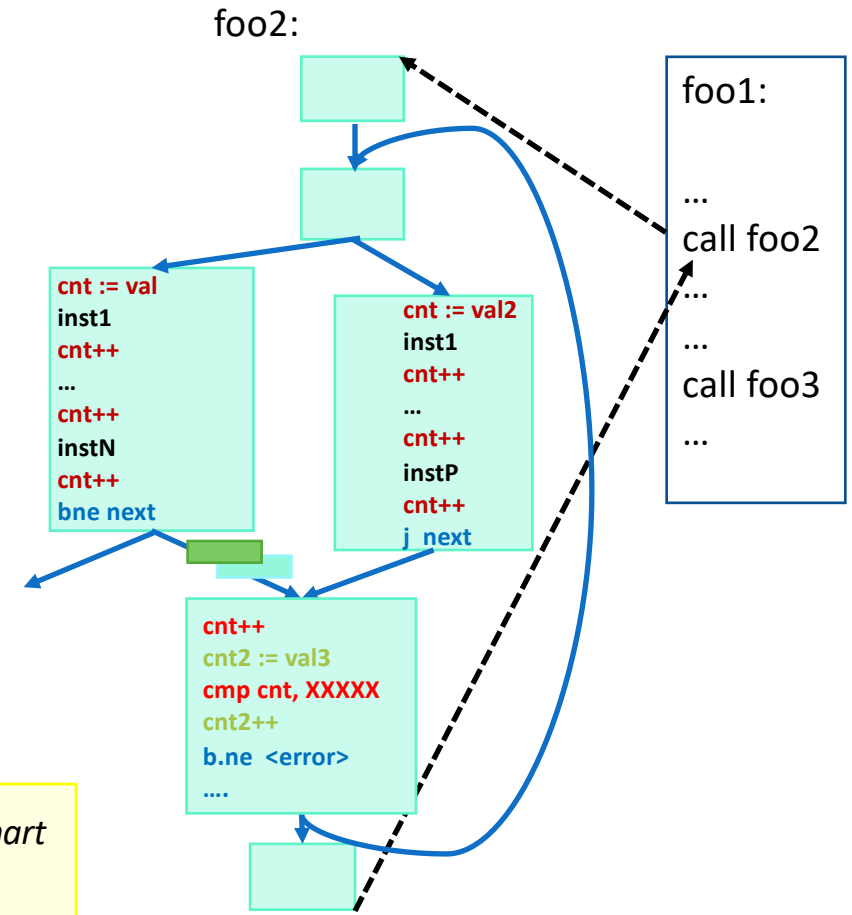
- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps
  - At the beginning of target blocks
    - Need for extra code : still misses some jumps



Control flow integrity

Counter-based protections

- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps
  - At the beginning of target blocks
    - Need for extra code
    - Overlap of counters initialization and check
    - Take into account branch outcome



J-F. Lalande et al. *Software countermeasures for control flow integrity of smart card C codes.* ESORICS 2014.

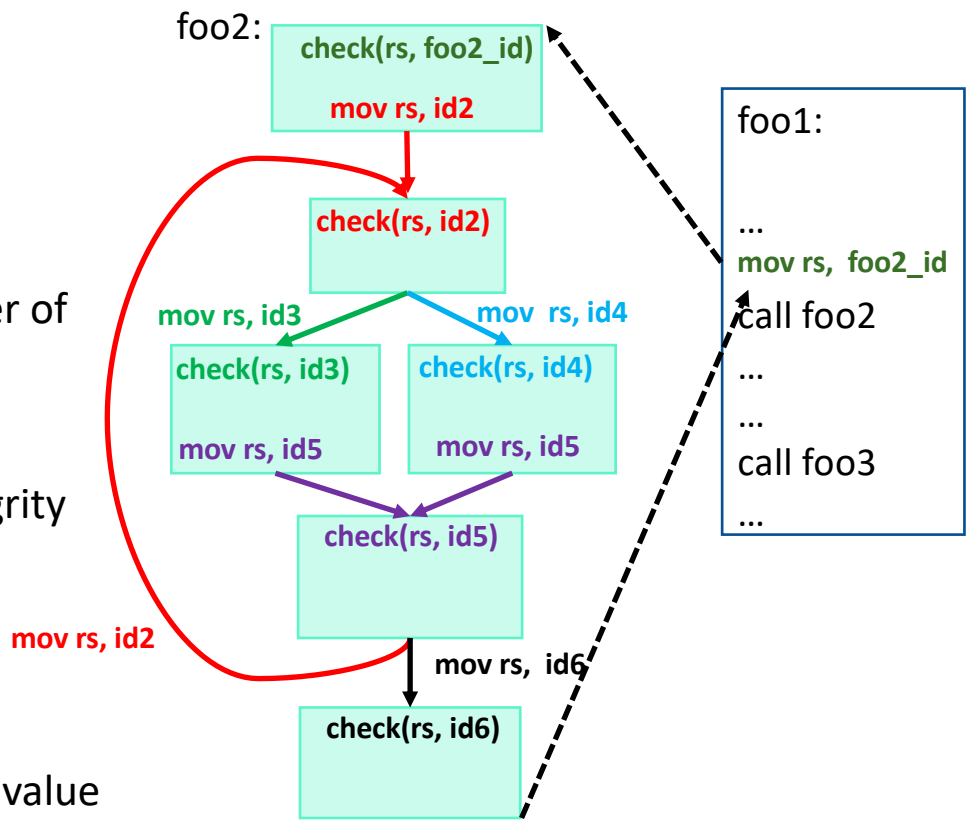
Countermeasures for control flow integrity

Signature-based protections [Oh et al. 2002] [Goloubeva et al., 2005]

- Unique identifier / signature assigned to every basic block (and function)
- Use to check every single control flow transfer
- Global signature computation limits the number of checks
- Ensure the CFG integrity
- Need for branch condition integrity / data integrity

Combination [SIED, 2003]

- Step counters inside basic blocks
- Signature for control flow transfers
- Signature computed with the branch condition value



Summary on SW protection

- Mostly based on
  - Redundant (or complementary) data
  - Duplication of computations
  - Step counters or signatures
  - Consistency checks
- Typical usages at source-code level
  - Duplication of conditions and tests
  - Duplication of critical variables
  - Tracers (step counters)
- Difficult part of code hardening: Which protections ? Where ?
- Protection are manually deployed, considering a specific threat model i.e. attacker's capabilities

A small example

```
#define BOOL_FALSE 0xAA     // specific values for Booleans
#define BOOL_TRUE  0x55

int verifyPIN(char *cardPin , char *userPin , unsigned size){
  unsigned i;
  unsigned diff = 0;
  /********** Comparison loop **********/
  for (i = 0; i < size ; i++)
     if (userPin[i] != cardPin[i])
          diff = 1;

  if (diff == 0) // PIN codes match
      return BOOL_TRUE;
  else           // PIN codes differ
      return BOOL_FALSE;

return BOOL_FALSE;
}
```

A better small example

```
#define BOOL_FALSE 0xAA     // specific values for Booleans
#define BOOL_TRUE  0x55

int verifyPIN(char *cardPin , char *userPin , unsigned size){
  unsigned i;
  unsigned diff = 0;
  /********** Comparison loop **********/
  for (i = 0; i < size ; i++)
     diff += userPin[i] ^ cardPin[i]); // constant-time loop body

  if (diff == 0) // PIN codes match
      return BOOL_TRUE;
  else           // PIN codes differ
      return BOOL_FALSE;

return BOOL_FALSE;
}
```

A small example

```
#define BOOL_FALSE 0xAA     // specific values for Booleans
#define BOOL_TRUE  0x55

int verifyPIN(char *cardPin , char *userPin , unsigned size){
  unsigned i;
  unsigned diff = 0;
  /********** Comparison loop **********/
  for (i = 0; i < size ; i++)
     diff += userPin[i] != cardPin[i]);

  if (diff == 0) // PIN codes match
      return BOOL_TRUE;
  else           // PIN codes differ
      return BOOL_FALSE;

return BOOL_FALSE;
}
```

Attacker's goal : bypass authentication check using a wrong `userPin`

Possible means:

- Loop corruption (0 iteration)
- Final check corruption
- Return value corruption

Protecting the small example

```
#define BOOL_FALSE 0xAA     // specific values for Booleans
#define BOOL_TRUE  0x55

int verifyPIN(char *cardPin , char *userPin , unsigned size){
  unsigned i, j = 0; // redundant iteration variable
  unsigned diff = 0;
  /********** Comparison loop **********/
  for (i = 0; i < size ; i++) {
      diff += userPin[i] ^ cardPin[i];
      j++;
  }
  if (j < size) error();

  if (diff == 0){    // PIN codes match
      if (diff != 0) error(); // redundant check
      return BOOL_TRUE;
  }
  else               // PIN codes differ
      return BOOL_FALSE;

return BOOL_FALSE;
}
```

Attacker's goal : bypass authentication check using a wrong **`userPin`**

Possible means:

- Loop corruption (0 iteration)
- Final check corruption
- Return value corruption

Issue related to software protection


```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

Issue related to software protection

Source code

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

Binary code

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

Issue related to software protection

Source code

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
    i
```

Binary code

Issue related to software protection

Source code

Binary code

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
    i

unsigned i;
for (i=0; i<size; i++) {
    foo(i);
}
if (i<size) error();
```

Issue related to software protection

Source code

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
    i

unsigned i;
for (i=0; i<size; i++) {
    foo(i);
}
if (i<size) error();
```

Binary code

Issue related to software protection

Compilation/optimisation

Source code

Binary code

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
    i

unsigned i;
for (i=0; i<size; i++) {
    foo(i);
}
if (i<size) error();
```

```
unsigned i;
for (i=0; i<size; i++)
{
    foo(i);
}
```

Issue related to software protection

Compilation/optimisation

Source code

Binary code

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

i

```
unsigned i;
for (i=0; i<size; i++) {
    foo(i);
}
if (i<size) error();
```

```
unsigned i;
for (i=0; i<size; i++)
{
    foo(i);
}
```

Issue related to software protection

Compilation/optimisation

Source code

Binary code

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

```
unsigned i, j=0;
for (i=0; i<size; i++) {
    foo(i);
    j++;
}
if (j<size) error();
```

i

```
unsigned i;
for (i=0; i<size; i++) {
    foo(i);
}
if (i<size) error();
```

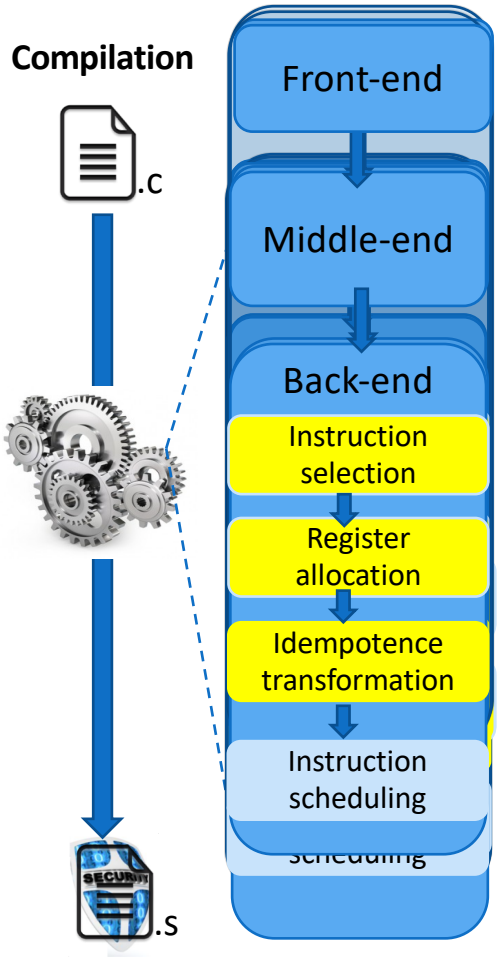
```
unsigned i;
for (i=0; i<size; i++)
{
    foo(i);
}
```

Countermeasures

- Principle of software countermeasures
  - Data integrity
  - Code integrity
  - Control-flow integrity
  - Limitations
- **Compiler-assisted code hardening**
  - Compilation of an instruction-skip protection
  - Compile-time loop hardening
  - Limitations

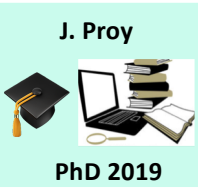
Instruction-skip protection at compilation-time

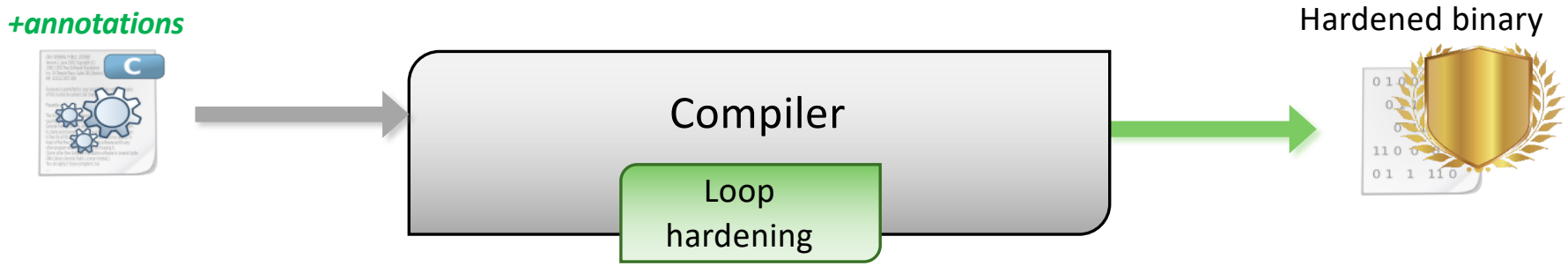
- **Protection scheme against instruction skip** [Moro et al. 2014]
- Main principle: **duplication of idempotent instructions**
- **Take advantage of compilation flow to**
  - Force the generation of idempotent instructions
    - Modification of the instruction selection
    - Modification of the register allocation
    - Additional transformation for remaining non-idempotent instructions (e.g. push and pop instruction that use and modify the stack pointer)
  - Add an instruction duplication pass
  - Let the scheduler optimize the resulting protected code
- Results in automatically protected code with better code size and performance



T. Barry et al. *Compilation of a Countermeasure Against Instruction-Skip Fault Attacks.* CS2 2016.

Compile-time loop hardening





Objective: Expected iteration count and right exit

Fault models: Instruction skip and register corruption (in the loop)

```
unsigned i;
#pragma sensitive_loop
for (i=0; i<size; i++){
    foo(i);
}
```

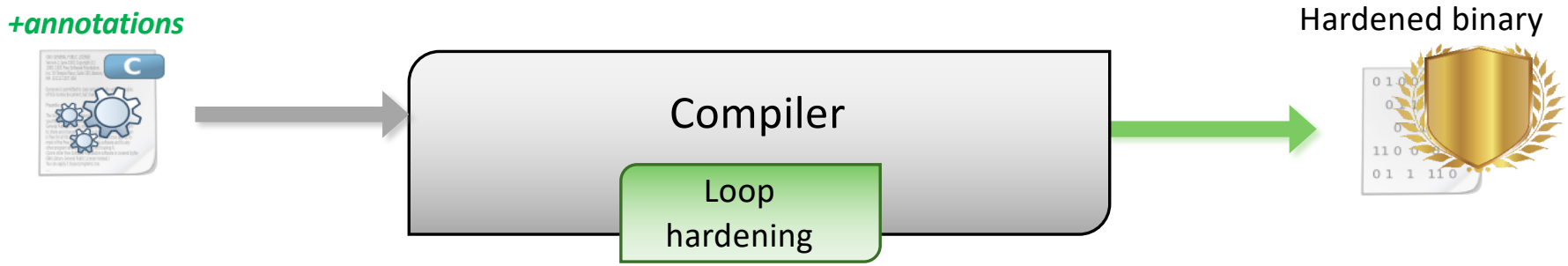
INVIA Inria informatics mathematics

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Compile-time loop hardening

J. Proy

PhD 2019



Objective: Expected iteration count and right exit

Fault models: Instruction skip and register corruption (in the loop)

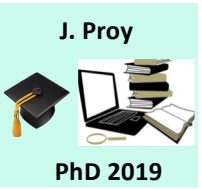
```
unsigned i;
#pragma sensitive_loop
for (i=0; i<size; i++){
    foo(i);
}
```

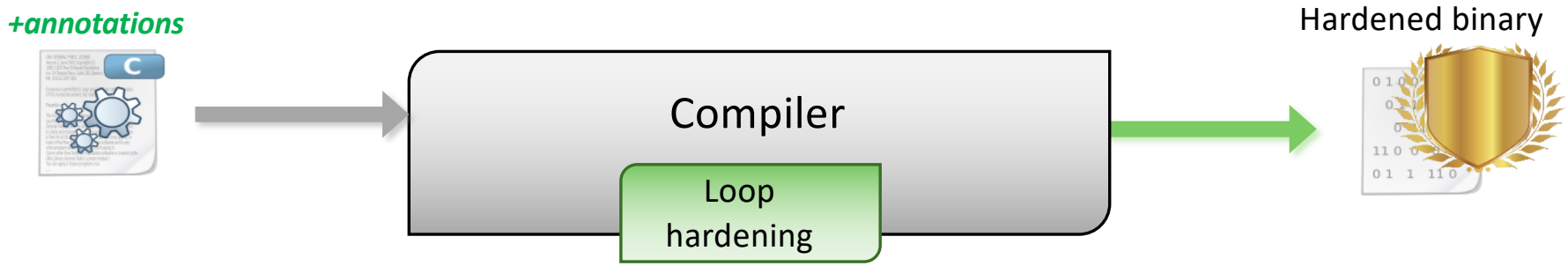
```
unsigned i;

for (i=0; i<size; i++){
    foo(i);
}
```

INVIA Inria informatics mathematics

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Compile-time loop hardening





Objective: Expected iteration count and right exit

Fault models: Instruction skip and register corruption (in the loop)

```
unsigned i;
#pragma sensitive_loop
for (i=0; i<size; i++){
    foo(i);
}
```

```
unsigned i, j=0;

for (i=0; i<size; i++){
    if (j>=size) error();
    foo(i);
    j++;
}
if (j<size) error();
```

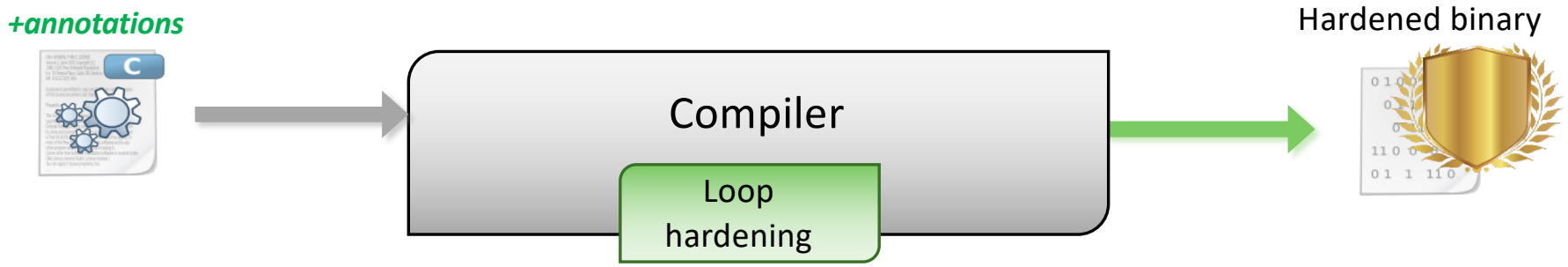
INVIA Inria informatics mathematics

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Compile-time loop hardening

J. Proy

PhD 2019



Objective: Expected iteration count and right exit

Fault models: Instruction skip and register corruption (in the loop)

```
unsigned i;
#pragma sensitive_loop
for (i=0; i<size; i++){
    foo(i);
}
```

```
unsigned i, j=0;

for (i=0; i<size; i++){
    if (j>=size) error();
    foo(i);
    j++;
}
if (j<size) error();
```

```
unsigned i, j=0;

for (i=0; i<size; i++){
    if (j>=size) error();
    foo(i);
    j++;
}
if (j<size) error();
```

INVIA Inria informatics mathematics

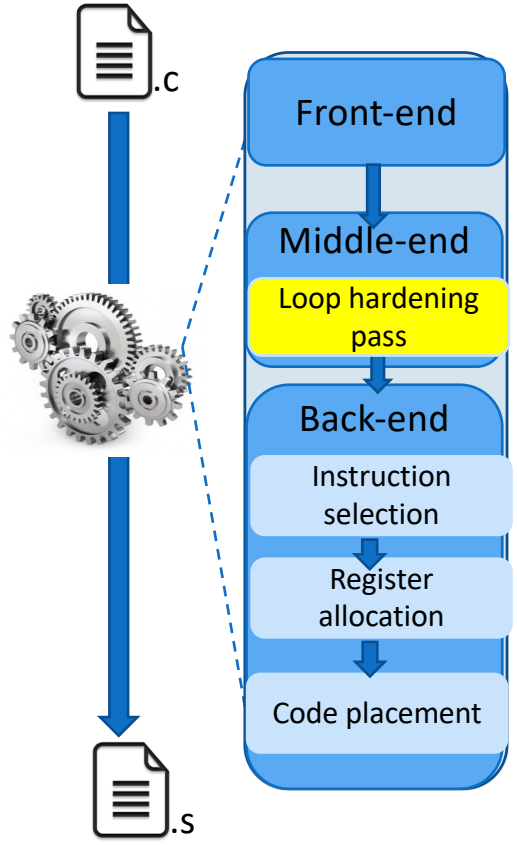
LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Compile-time loop hardening

J. Proy

PhD 2019

- Target-independant pass in clang/LLVM (LLVM IR level)
- Limited overhead in performance and code size (average <20%)
- **Fault injection simulations : detection rate 99%**
- **1% undetected faults**
  - Harmfull downstream optimisation passes
  - Need to desactivate (when possible) or adapt them



Compiler-Assisted Loop Hardening Against Fault Attacks
- J. Proy et al. ACM TACO 2017.

INVIA Inria informatics mathematics

SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257 LIP6

Compile-time loop hardening

J. Proy

PhD 2019

- Target-independant pass in clang/LLVM (LLVM IR level)
- Limited o
- **Fault inj**
- **1% unde**
  - Harr
  - Nee

More effective than source-level protections
But
Still needs a binary code analysis to verify that the downstream passes did not alter the protection

Compiler-Assisted Loop Hardening Against Fault Attacks
- J. Proy et al. ACM TACO 2017.

INVIA Inria informatics mathematics

SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

LIP6

Agenda

- Fault exploitation
- Fault effects and modelling
- Countermeasures
- **Robustness analysis**
- Conclusion and perspectives

Robustness analysis at binary level

Need for such analysis

- Binary code : final code, post-hardening, post-compilation

Objective : verify that the application behaves as intended in presence of a fault attack or detects it

- Specification of « intended behaviour » : security property
- Specification of possible faults : fault models

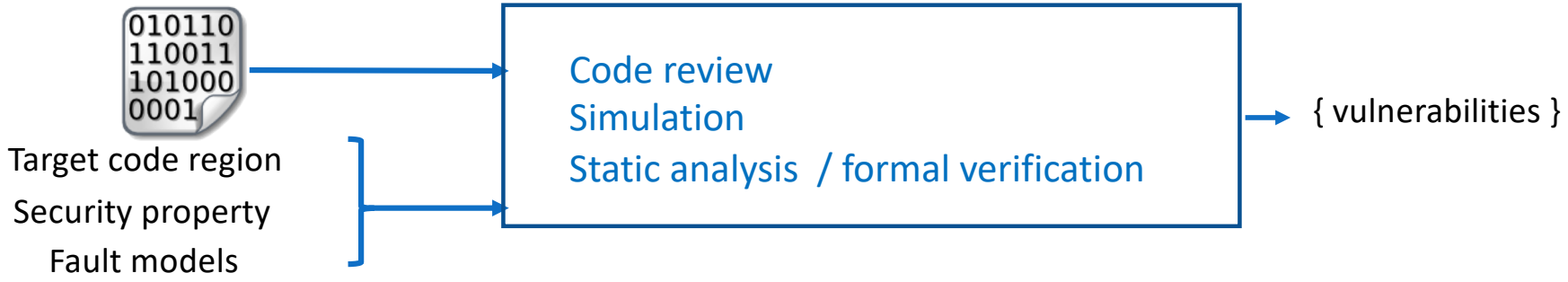
Robustness analysis at binary level

Need for such analysis

- Binary code : final code, post-hardening, post-compilation

Objective : verify that the application behaves as intended in presence of a fault attack or detects it

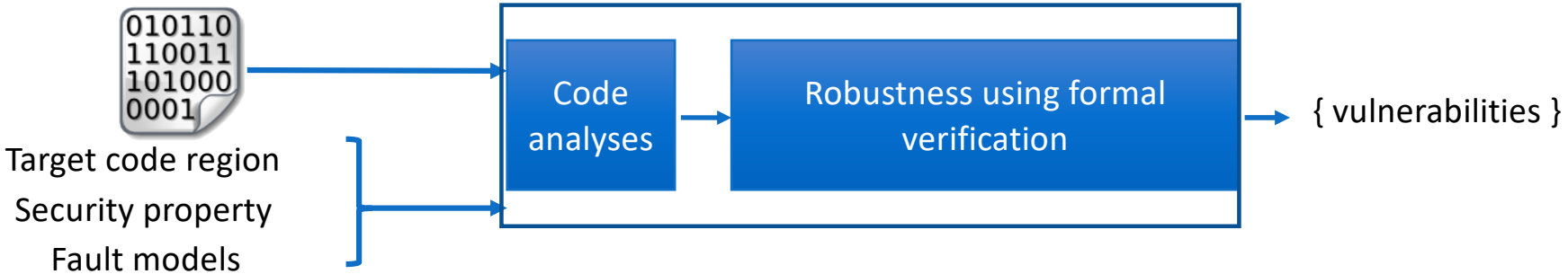
- Specification of « intended behaviour » : security property
- Specification of possible faults : fault models



J.B Bréjon

PhD 2020

RobustB : robustness analysis at binary level

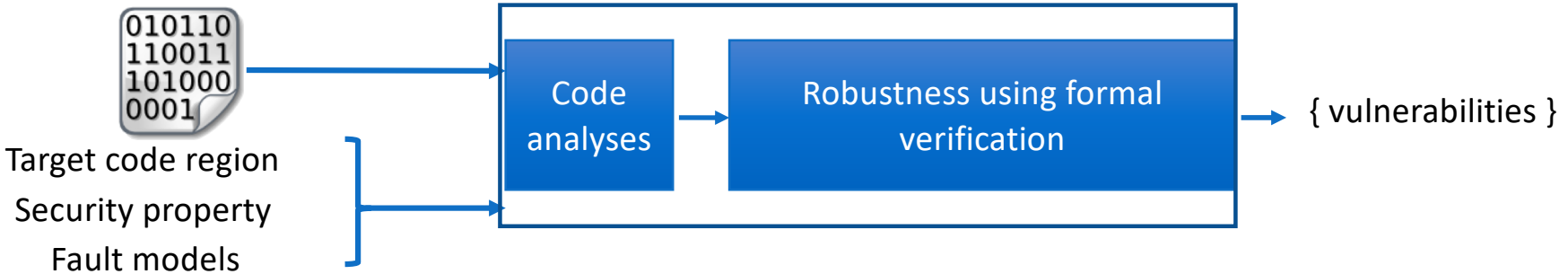


Security property

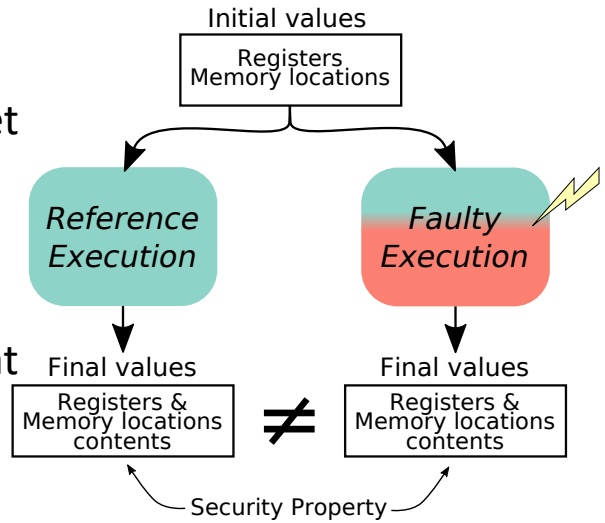
- Integrity of some registers and memory locations at the end of the target region execution

  Example : integrity of the return value of verifyPIN

RobustB : robustness analysis at binary level



Verification scheme



Security property

- Integrity of some registers and memory locations at the end of the target region execution

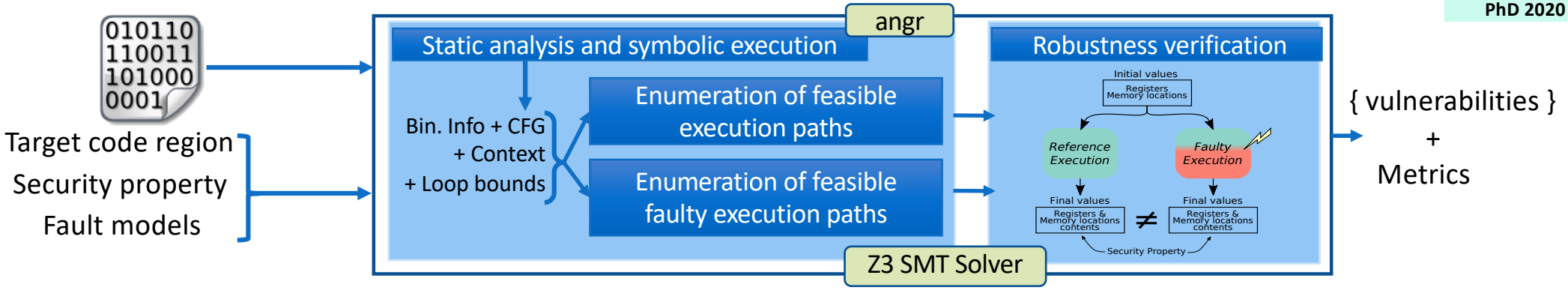
  Example : integrity of the return value of verifyPIN

Property verification

- Comparison of the content of selected registers and memory locations at the end of the target region execution with and without fault injection

RobustB : robustness analysis at binary level

J.B Bréjon
PhD 2020



Overview

- Preliminary code analyses
- Determination of feasible execution paths $\{P_{i\_ref}\}$
  - For each P_{i_ref}, determination of faulty feasible faulty execution paths $\{P_{i\_faulty\_j}\}$
  - Robustness verification $(P_{i\_ref}, P_{i\_faulty\_j})$
- Metrics summarising all the results (attack surface, attack density, instruction sensitivity)

PROSECCO
AGENCE NATIONALE DE LA RECHERCHE ANR

Formal models for the path feasibility analysis

Feasibility of an execution path *P* composed of instructions $i_1, \ldots, i_n$

- **Satisfiability of** $P_{ref}\ (context) = init \wedge inst_1 \wedge \ldots \wedge inst_n$

  init defines initial variables according to *context* (constraints on initial values)

  $$init \Leftrightarrow r0_0 = val_0 \wedge \ldots \wedge r16_0 = val_{16} \wedge mem[\ldots] = \ldots$$

  $inst_i$ defines new variables (*cf.* SSA), *if* $i_i \equiv$ **add r4, r2, r3** then

  $$inst_i \Leftrightarrow r4_i = r2_{i-1} + r3_{i-1} \wedge_{for\ all\ i \neq 4} rX_i = rX_{i-1}$$

Feasibility of a faulty execution path P_{faulty} resulting from one fault injection targeting i_j

- **Satisfiability of**

  $$P_{faulty}\ (context) = init \wedge inst_1 \wedge \ldots \wedge inst_{j-1} \wedge fault(inst_j) \wedge inst_{j+1} \wedge \ldots \wedge inst_p$$

  with $fault(inst_j)$ depends on the considered fault model:

  skip $i_j \Leftrightarrow (fault(inst_j) \Leftrightarrow \wedge_{for\ all\ X}\ rX_j = rX_{j-1})$

  register *(rY)* **corruption right before** $i_j \Leftrightarrow (fault(inst_j) \Leftrightarrow rY_{j-1} = ??? \wedge inst_j)$

Formal models for the path feasibility analysis

Feasibility of an execution path *P* composed of instructions $i_1, \ldots, i_n$

- **Satisfiability of** $P_{ref}\ (context) = init \wedge inst_1 \wedge \ldots \wedge inst_n$

  init defines initial variables according to *context* (constraints on initial value)

  $$init \Leftrightarrow r0_0 = val_0 \wedge \ldots \wedge r16_0 = val_{16} \wedge mem[\ldots] = \ldots$$

  $inst_i$ defines new variables (*cf.* SSA), *if* $i_i \equiv$ **add r4, r2, r3** then

  $$inst_i \Leftrightarrow r4_i = r2_{i-1} + r3_{i-1} \wedge_{for\ all\ i\ \neq 4} rX_i = rX_{i-1}$$

Feasibility of a faulty execution path P_{faulty} resulting from one fault injection targeting i_j

- **Satisfiability of**

  $$P_{faulty}\ (context) = init \wedge inst_1 \wedge \ldots \wedge inst_{j-1} \wedge fault(inst_j) \wedge inst_{j+1} \wedge \ldots \wedge inst_p$$

  with $fault(inst_j)$ depends on the considered fault model:

  skip $i_j \Leftrightarrow (fault(inst_j) \Leftrightarrow \wedge_{for\ all\ X} rX_j = rX_{j-1})$

  register *(rY)* **corruption right before** $i_j \Leftrightarrow (fault(inst_j) \Leftrightarrow rY_{j-1} = ??? \wedge inst_j)$

Formal models for robustness analysis

Feasibility of an execution path composed of instructions $i_1, \ldots, i_n$

- **Satisfiability of** $P_{ref}(context) = init \wedge inst_1 \wedge \ldots \wedge inst_n$

Feasibility of a faulty execution path P_{faulty} resulting from a fault injection targeting i_j

- **Satisfiability of**

$$P_{faulty}(context) = init \wedge inst_1 \wedge \ldots \wedge inst_{j-1} \wedge fault(inst_j) \wedge inst_{j+1} \wedge \ldots \wedge inst_p$$

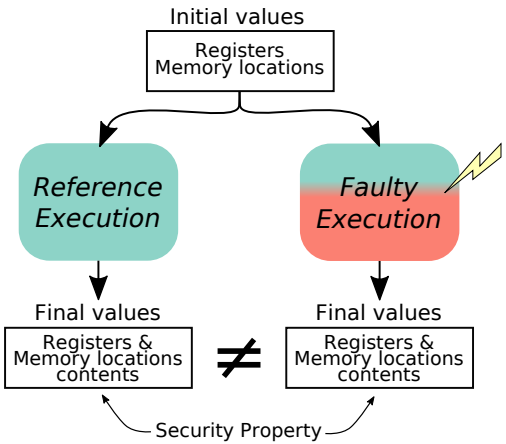
Vulnerability search

- **Satisfiability of**

$$VULN = P_{ref}(context) \wedge P_{faulty}(context) \wedge vuln$$

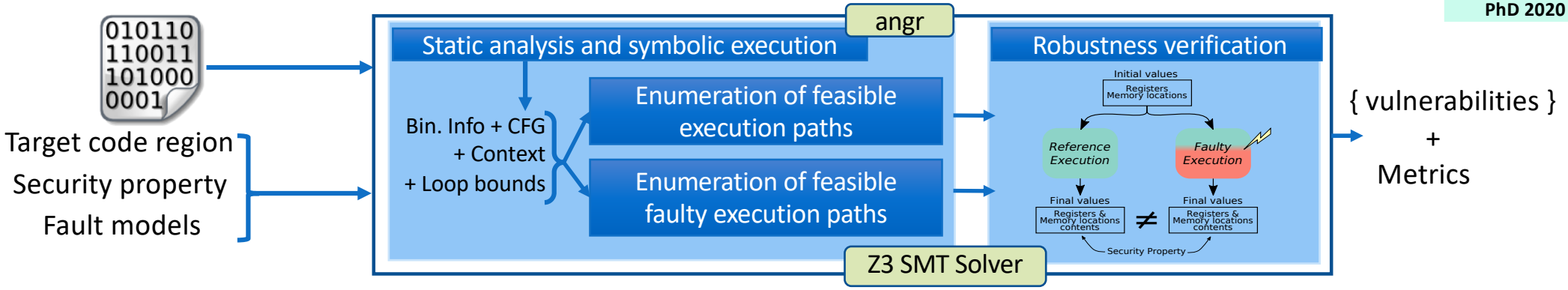
with

$$vuln \Leftrightarrow \text{non-equivalence of registers and memory locations content at the end of the execution}$$



RobustB : robustness analysis at binary level

J.B Bréjon
PhD 2020



Use-case 1: source-level protected codes (VerifyPIN from FISSC [Dureuil 2016])

1. Effects of compilation options, compiler impact → Metrics help analysing and comparing different versions
2. Elimination of redundant protections

Use-case 2: compiler-hardened code

1. Hardened loop (memcpy) [Proy et al, ACM TACO 2017] → Effective protection w.r.t considered attacker model
   → But one vulnerability due to code placement
1. Compiler-assisted instruction-skip [Barry et al., CS2@HIPEAC2016] → Effective protection w.r.t attacker model

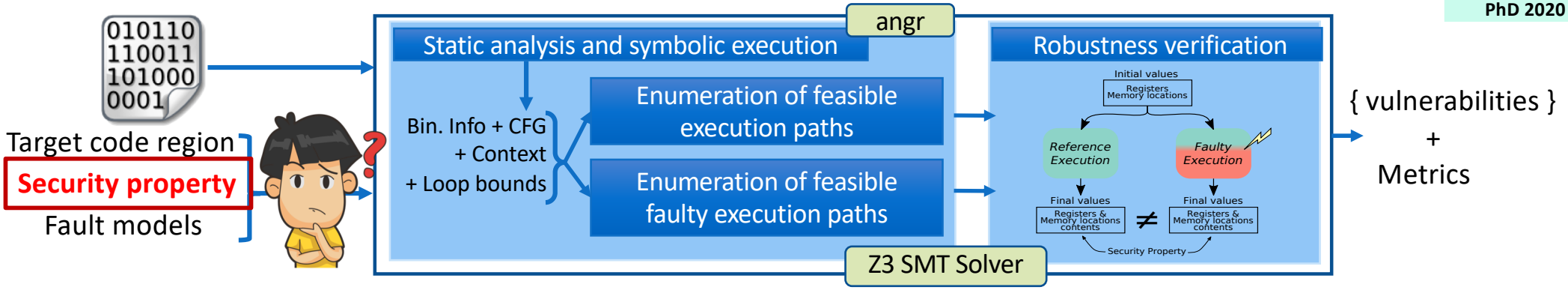
PROSECCO

AGENCE NATIONALE DE LA RECHERCHE ANR

Fault attack vulnerability assessment of binary code - J-B. Bréjon et al. CS2 2019.

RobustB : robustness analysis at binary level

J.B Bréjon
PhD 2020



Use-case 1: source-level protected codes (VerifyPIN from FISSC [Dureuil 2016])

1. Effects of compilation options, compiler impact ➜ Metrics help analysing and comparing different versions
2. Elimination of redundant protections

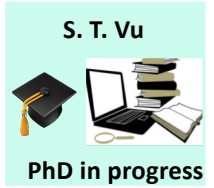
Use-case 2: compiler-hardened code

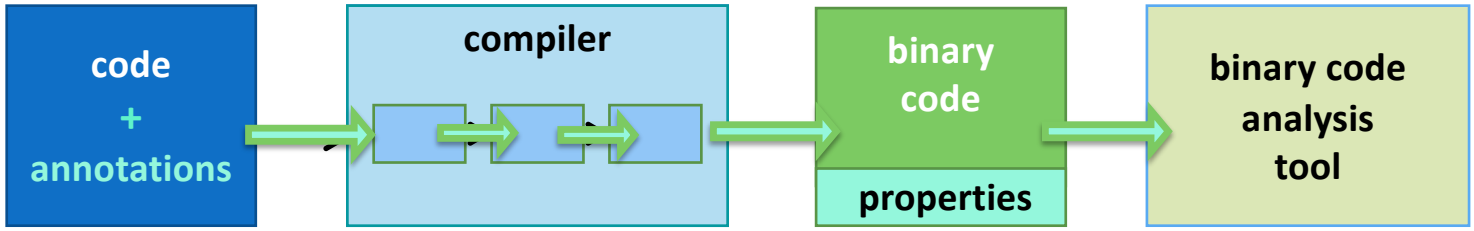
1. Hardened loop (memcpy) [Proy et al, ACM TACO 2017] ➜ Effective protection w.r.t considered attacker model
   ➜ But one vulnerability due to code placement
1. Compiler-assisted instruction-skip [Barry et al., CS2@HIPEAC2016] ➜ Effective protection w.r.t attacker model

PROSECCO
AGENCE NATIONALE DE LA RECHERCHE ANR

Fault attack vulnerability assessment of binary code - J-B. Bréjon et al. CS2 2019.

Compilation of (security) properties






```
int i;
// sensitive loop
for (i=0; i<size; i++){
    foo(i);
}
P:__attribute__("i == size");
```

Objectives

- Expression of functional properties at source-code level (predicates on variables)
- Compilation of these properties in concert with the code (but not included in the code)
- Production of properties in the binary code (in a dedicated section)

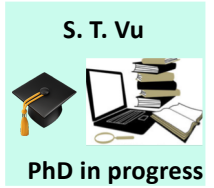
Major issue

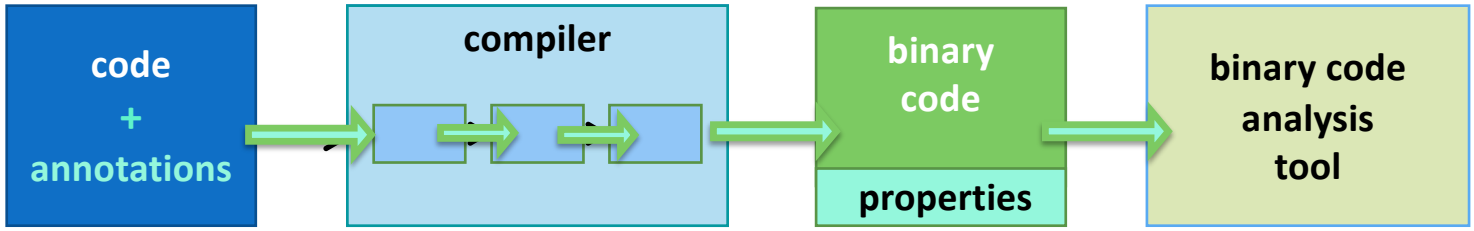
- Preservation and correctness of properties all the way down in an optimizing compiler

arm Google

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Compilation of (security) properties






```
int i;
// sensitive loop
for (i=0; i<size; i++){
    foo(i);
}
P:__attribute__("i == size");
```

Objectives

- Expression of functional properties at source-code level (predicates on variables)
  → annotations based on *ANSI/ISO C Specification Language* (ACSL)
- Compilation of these properties in concert with the code (but not included in the code)
- Production of properties in the binary code (in a dedicated section)

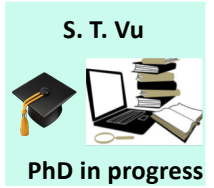
Major issue

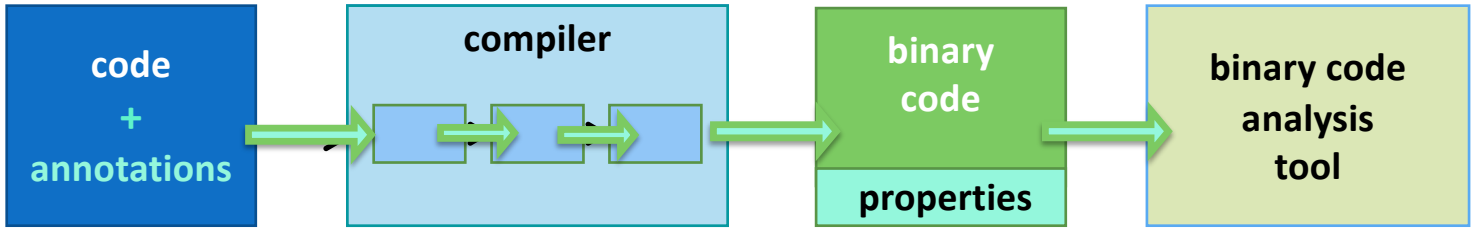
- Preservation and correctness of properties all the way down in an optimizing compiler

arm Google

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Compilation of (security) properties






```
int i;
// sensitive loop
for (i=0; i<size; i++){
    foo(i);
}
P:__attribute__("i == size");
```

Objectives

- Expression of functional properties at source-code level (predicates on variables)
  → annotations based on *ANSI/ISO C Specification Language* (ACSL)
- Compilation of these properties in concert with the code (but not included in it)
  → notion of property preservation (observation point and involved variables and memory locations)
- Production of properties in the binary code (in a dedicated section)

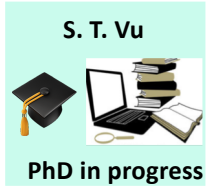
Major issue

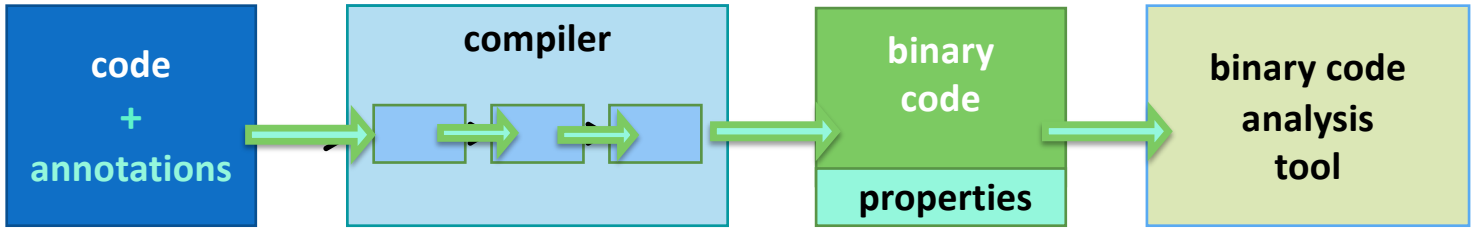
- Preservation and correctness of properties all the way down in an optimizing compiler

arm Google

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Compilation of (security) properties






```
int i;
// sensitive loop
for (i=0; i<size; i++){
    foo(i);
}
P:__attribute__("i == size");
```

Objectives

- Expression of functional properties at source-code level (predicates on variables)
  → annotations based on *ANSI/ISO C Specification Language* (ACSL)
- Compilation of these properties in concert with the code (but not included in it)
  → notion of property preservation (observation point and involved variables and memory locations)
- Production of properties in the binary code (in a dedicated section)
  → debug information - DWARF

Major issue

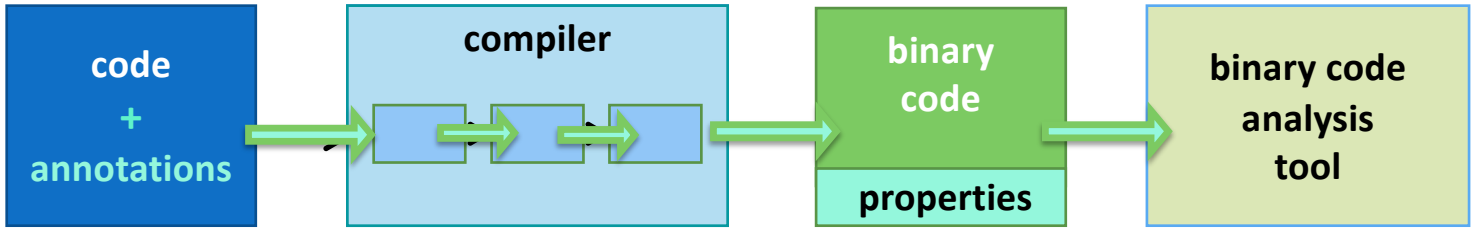
- Preservation and correctness of properties all the way down in an optimizing compiler

arm Google

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Compilation of (security) properties

S. T. Vu
PhD in progress




```
int i;
// sensitive loop
for (i=0; i<size; i++){
    foo(i);
}
P:__attribute__("i == size");
```

Objectives

- Expression of functional properties at source-code level (predicates on variables)
  → annotations based on *ANSI/ISO C Specification Language* (ACSL)
- Compilation of these properties in concert with the code (but not included in it)
  → notion of property preservation (observation point and involved variables and memory locations)
- Production of properties in the binary code (in a dedicated section)
  → debug information - DWARF

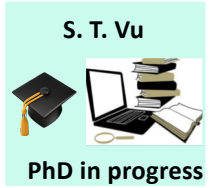
Major issue

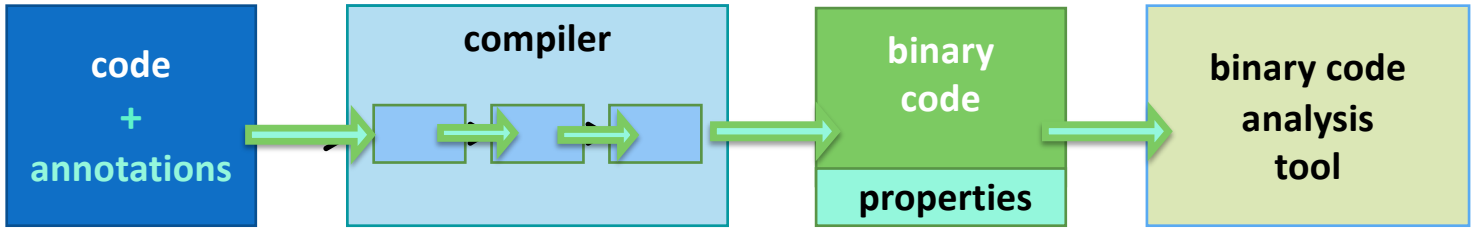
- Preservation and correctness of properties all the way down in an optimizing compiler
  → insertion of barriers i.e I/O & side-effecting instructions to ensure the propagation and preservation
  → implemented in clang/LLVM

arm Google

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Compilation of (security) properties






```
int i;
// sensitive loop
for (i=0; i<size; i++){
    foo(i);
}
P:__attribute__("i == size");
```

Objectives

- Expression of functional properties at source-code level (predicates on variables)
  → annotations based on *ANSI/ISO C Specification Language* (ACSL)
- Compilation of these properties in concert with the code (but not included in it)
  → notion of property preservation (observation point and involved variables and memory locations)
- Production of properties in the binary code (in a dedicated section)
  → debug information - DWARF

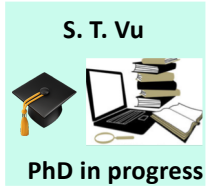
Major issue

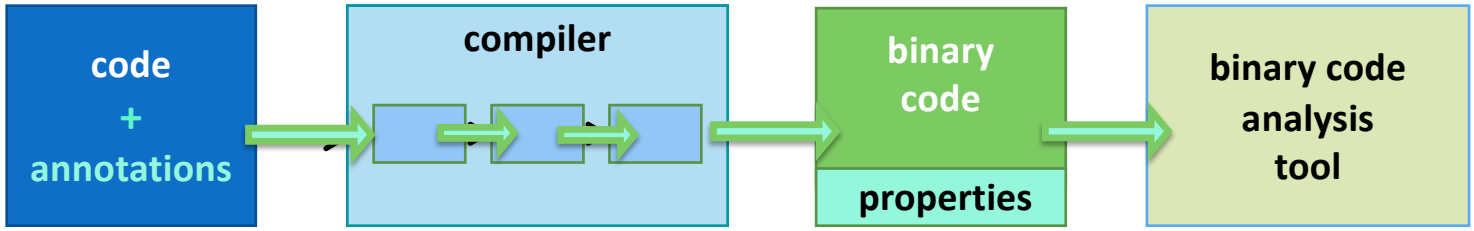
- Preservation and correctness of properties all the way down in an optimizing compiler
  → insertion of barriers i.e I/O & side-effecting instructions to ensure the propagation and preservation
  → implemented in clang/LLVM
- **Validation using 30 tests / 558 annotations** from the test suite ACSL/Frama-C

arm Google

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Protecting the protections



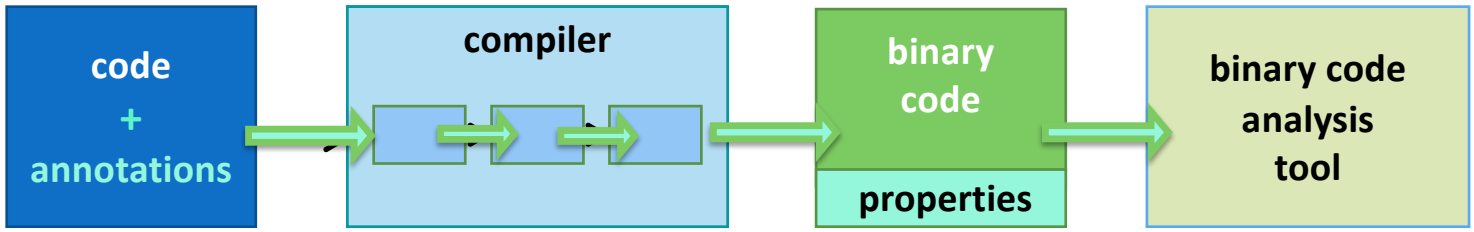


- Preservation of protections by expressing properties related to the protections ?

arm Google

LIP6 SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Protecting the protections



S. T. Vu

PhD in progress

- Expression of functional (observational) properties related to protections
- Preservation of protections = preservation of properties
- Application to 4 protection types : against fault attacks (verifyPIN-like) and against data leakage (considering AES and RSA)

→ Enable to compile and optimize source-level protected code

→ Enable to verify presence/effectiveness of protection at binary level

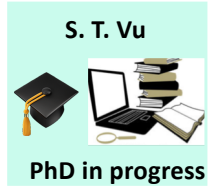
Secure delivery of program properties through optimizing compilation

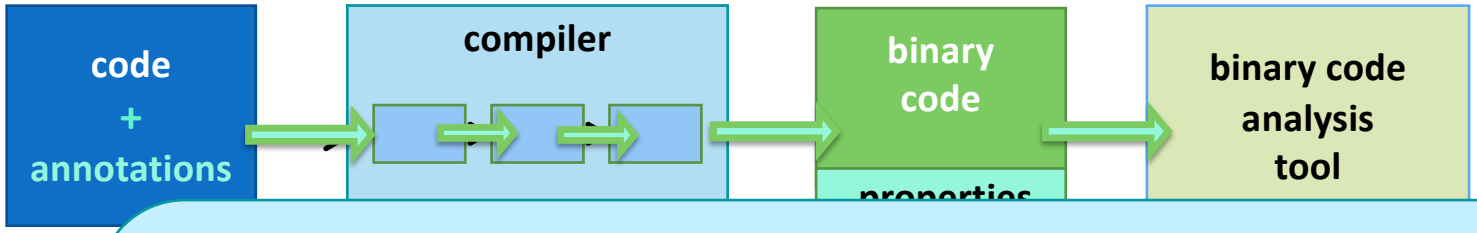
S. T. Vu et al. Compiler Construction 2020.

arm Google

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Protecting the protections





- Expressi
- Preserva
- Applicati … age (consider

More on this ?

1. Talk « Secure optimization through opaque observation »
   from Son T. Vu at the GT MFS Day – Tuesday, March 16
2. Son's PhD defense planned on Friday, April 2

→ Enable to

→ Enable to verify presence/effectiveness of protection at binary level

Secure delivery of program properties through optimizing compilation
S. T. Vu et al. Compiler Construction 2020.

arm Google

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Conclusion

Fault attacks

- Powerful and particularly harmful

Sofware hardening issues

- Related to the hardening process : fault models, sensitive region/assets, design and combination of protections
- Related to the compilation flow and its optimisations
- Robustness analysis of the final code required

Investigated solutions

- Compile-time code hardening
- Robustness analysis at binary level
- Compilation (propagation and preservation) of properties
- Expression of properties for protecting the protections

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Open questions and research lines

Faults attacks and faults effects

- Availability or potential building of low cost injection means : Riscure[1], NewAE[2], [Kelly 2020]
- Multiple faults are there ! Several temporal ones [Bozzato 2019] & few consecutive instruction skips to one hundred of consecutive instruction skips ! [Dutertre 2019, Menu 2020]
- Many precise faults are highly dangerous : « instruction skip oriented programming » [Péneau 2020]
- Complex targets are unprotected and vulnerable [Proy 2019][Trouchkine 2019a] [Trouchkine 2019b]
- Precise faults inside the processor induce effects invisible at ISA-level [Laurent 2019]

Countermeasures

- Need to take into account the increase in complexity, multiplicity and diversity of faults
- Need for protection-aware and/or hardening compilers
- HW/SW protection solutions for a better coverage and performance trade-off [ANR COFFI]
- Robustness verification methods and tools to help designers / developers

[1] https://www.riscure.com
[2] https://www.newae.com

LIP6 SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

References

[Akkar 2003] M. Akkar, Automatic Integration of Counter-Measures Against Fault Injection AttacksE-Smart'2003.

[ANR COFFI] https://www.mines-stetienne.fr/coffi-en/

[Bozzato 2019] C. Bozzato. Shaping the glitching: Optimizing Voltage Fault Injection. TCHES 2020.

[Dehbaoui 2013] A. Dehbaoui et al. Electromagnetic Glitch on the AES Round Counter. COSADE 2013.

[Dutertre 2019] J-M Dutertre et al. Experimental Analysis of the Laser-Induced Instruction Skip Fault Model. Nordsec 2019

[Dureuil 2016] L. Dureuil et al. FISSC: a Fault Injection and Simulation Secure Collection. SAFECOMP 2016.

[El Bar 2006] H. Bar-El et al. The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE. 2006

[Kelly 2020] M. Kelly et al. High precision Laser Fault Injection using Low-cost Components IEEE HOST 2020

[Goloubeva 2005] O. Goloubeva et al. Improved software-based processor control-flow errors detection technique. Annual Reliability and Maintainability Symposium, 2005.

[Karaklajic 2013] D. Karaklajic et al, I. Hardware Designer's Guide to Fault Attacks. IEEE Trans. on VLSI Systems. 2013.

[Kumar 2017] S. V. Dilip Kumar et al. A Practical Fault Attack on ARX-Like Ciphers with a Case Study on ChaCha20. FDTC 2017.

[Laurent 2019] J. Laurent et al. Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures. DATE 2019

[Menu 2020] A. Menu et al. Experimental Analysis of the Electromagnetic Instruction Skip Fault Model. DTIS 2020

[Moro 2014] N. Moro et al. A formally verified countermeasure against instruction skip. JCEN 2014

References

[Oh 2002] N. Oh et al. Control-flow checking by software signatures. IEEE Transactions on Reliability, 2002.

[Péneau 2020] P-Y Péneau et al. NOP-Oriented Programming: Should we Care? SILM@EuroS&P 2020

[Proy 2019] J. Proy et al. A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures: A Secure Software Perspective. ARES 2019

[Rauzy et al., 2015] P. Rauzy et al. A formal proof of countermeasures against fault injection attacks on CRT-RSA. JCEN 2014

[Reis et al., 2005] G. Reis et al. SWIFT: Software Implemented Fault Tolerance. CGO 2005.

[SIED, 2003] B. Nicolescu et al. SIED: Software Implemented Error Detection. Defect and Fault Tolerance in VLSI System 2003.

[Timmers 2016] N. Timmers et al. Controlling PC on ARM Using Fault Injection. FDTC 2016.

[Timmers 2017] N. Timmers et al. Escalating Privileges in Linux Using Voltage Fault Injection. FDTC 2017

[Trouchkine 2019a] T. Trouchkine et al. Fault injection characterization on modern CPUs. WISTP 2019.

[Trouchkine 2019b] T. Trouchkine et al. Electromagnetic fault injection against a System-on-Chip, toward new micro-architectural fault models. CoRR abs/1910.11566, 2019

[Vasselle 2017] A. Vasselle et al. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot. FDTC 2017.

[Verbauwhede 2011] I. Verbauwhede et al. FDTC 2011.

[Yuce 2018] B. Yuce t al. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. JHSS 2018.

[Zussa 2014] L. Zussa et al. A. Efficiency of a glitch detector against electromagnetic fault injection. DATE 2014.