SECURITY OF ASSEMBLY PROGRAMS AGAINST FAULT ATTACKS ON EMBEDDED PROCESSORS

Nicolas MORO (CEA)

Thesis supervised by **Karine HEYDEMANN** (LIP6)
Under the direction of **Emmanuelle ENCRENAZ** (LIP6)
and **Bruno ROBISSON** (CEA)

Some parts of the presented works have been done in cooperation with Amine Dehbaoui

13TH NOVEMBER 2014 – PARIS

Embedded systems :

- Are **autonomous electronic systems**
- Are **widely used** and have many applications

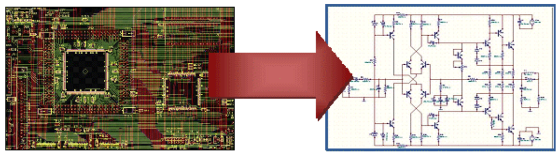
EMBEDDED SYSTEMS SECURITY

- Those systems **can be attacked**
- Those attacks **generally aim at :**

Getting sensitive data

Bypassing a protection

Doing reverse-engineering



EMBEDDED SYSTEMS SECURITY

Embedded systems security is very important for :

Administrations and governments

Digital identity documents

Manufacturers of smart cards

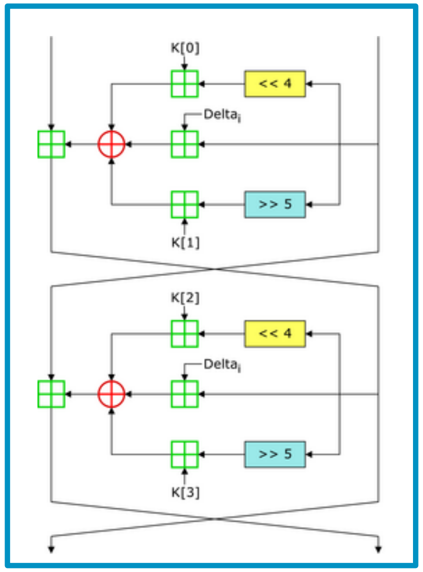
Pay-TV, banking cards, access cards, …

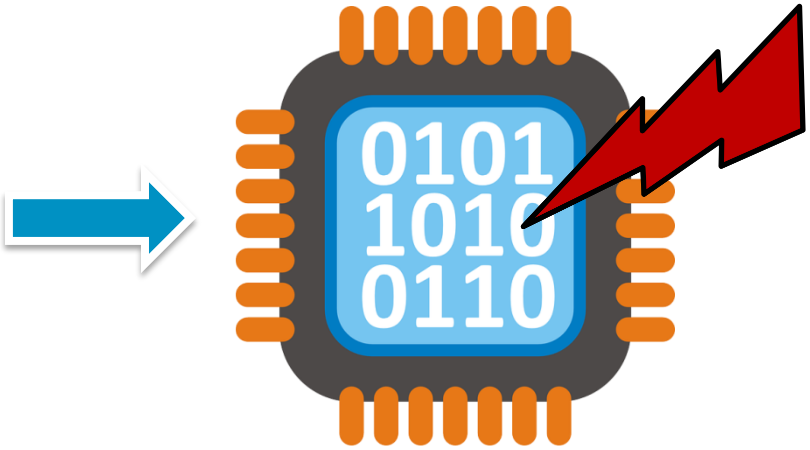
Manufacturers of consumer products

Locked systems, which include payment systems, …

Physical attacks

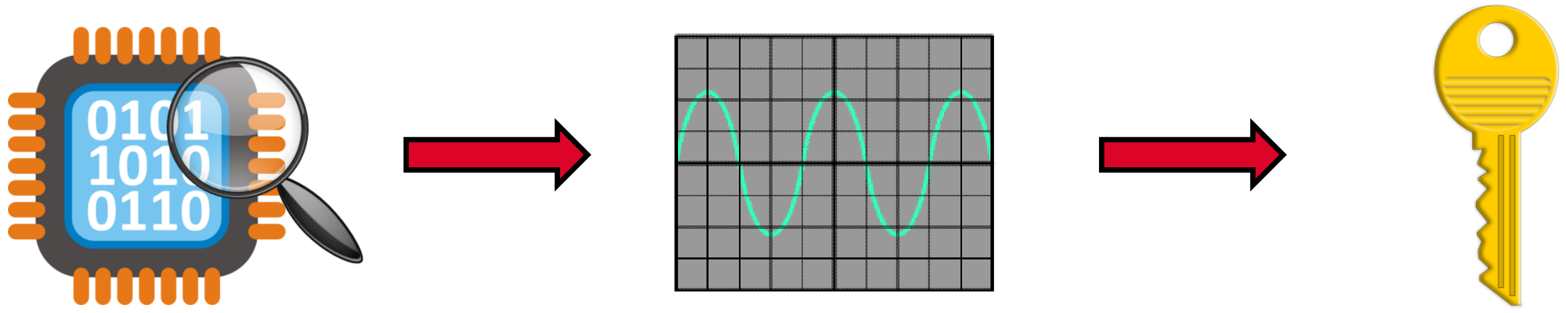
- require an **access to the component**
- aim at exploiting the **vulnerabilities of integrated circuits**
- are a **serious threat** for embedded systems



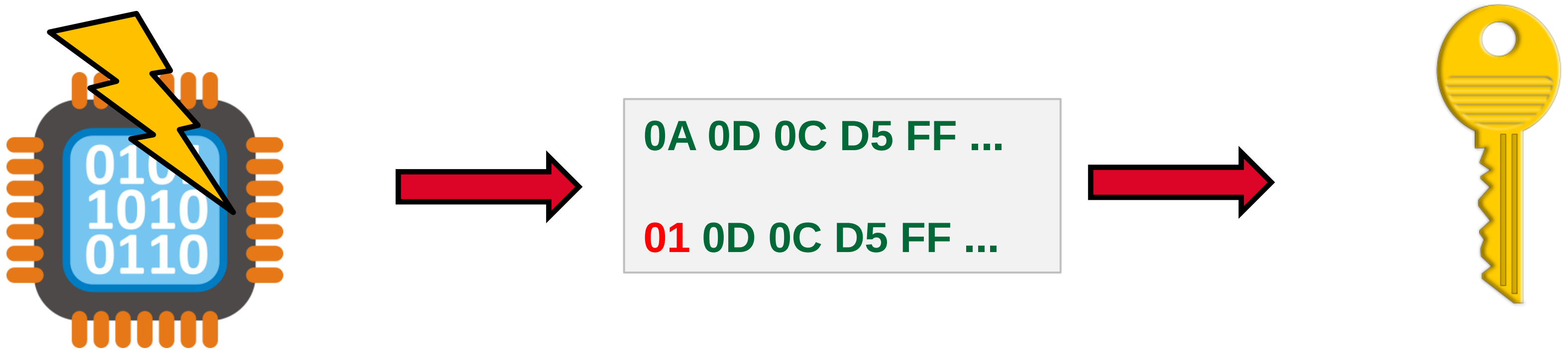


TWO KINDS OF PHYSICAL ATTACKS

- **Side-channel analysis attacks**



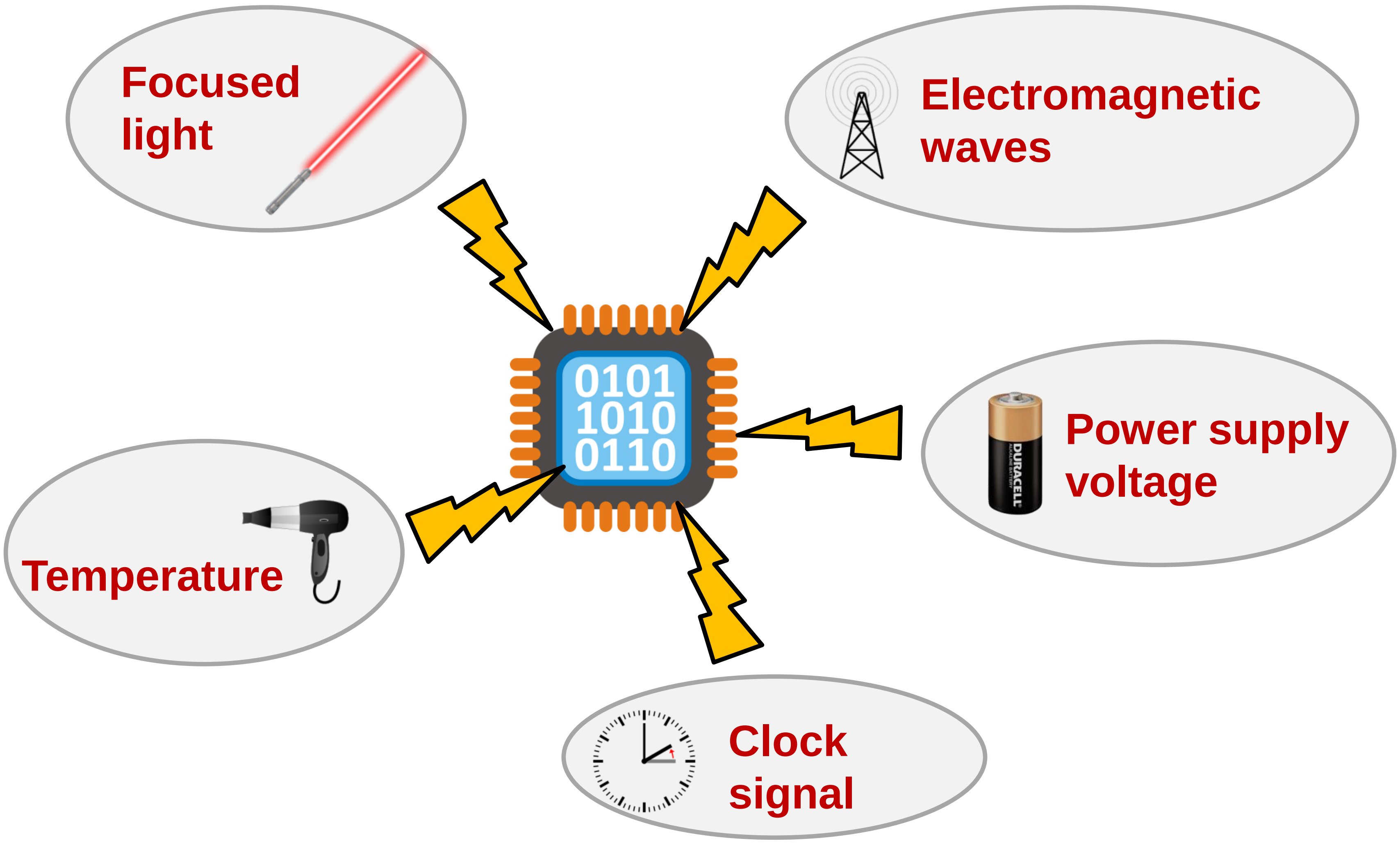
- **Fault injection attacks**



0A 0D 0C D5 FF ...

01 0D 0C D5 FF ...

DIFFERENT FAULT INJECTION MEANS



COUNTERMEASURES AGAINST FAULT ATTACKS

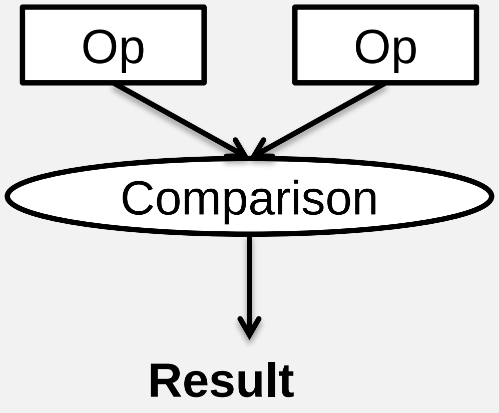
Some countermeasures put at different levels :

1 – Physical sensors

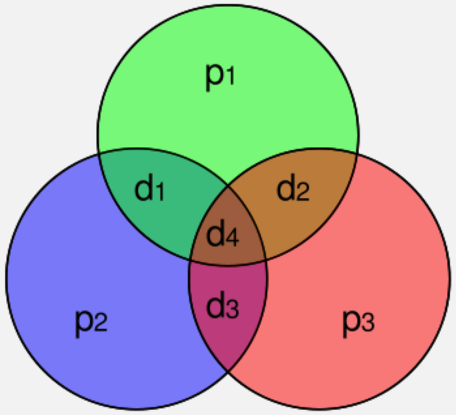
Light detectors, voltage modification detectors, …

2 – Mécanismes de détection ou tolérance aux fautes

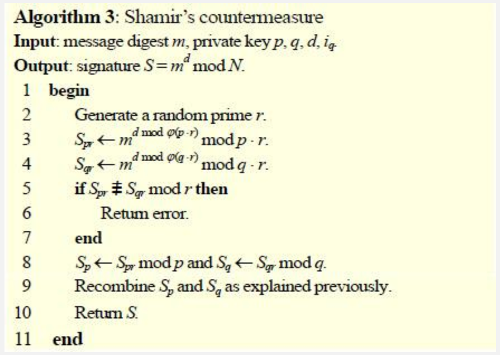
Redundancy



Parity bits and error correcting codes



Mathematical properties of algorithms

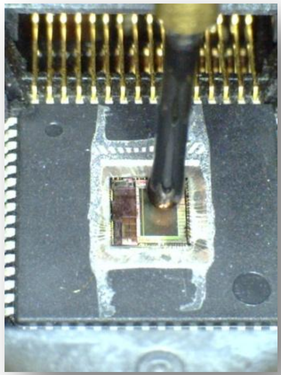


Algorithm 3: Shamir's countermeasure
Input: message digest m, private key p, q, d, i_q.
Output: signature $S = m^d \bmod N$.

```
begin
    Generate a random prime $r$.
    $S_{pr} \leftarrow m^{d \bmod \varphi(p \cdot r)} \bmod p \cdot r$.
    $S_{qr} \leftarrow m^{d \bmod \varphi(q \cdot r)} \bmod q \cdot r$.
    if $S_{pr} \not\equiv S_{qr} \bmod r$ then
        Return error.
    end
    $S_p \leftarrow S_{pr} \bmod p$ and $S_q \leftarrow S_{qr} \bmod q$.
    Recombine $S_p$ and $S_q$ as explained previously.
    Return $S$.
end
```

PULSED ELECTROMAGNETIC FAULT INJECTION

- **Quite recent** technique
  (theoretical in 2002, in practice since 2007)
- An electrical pulse is sent to an **injection antenna**
- Electromagnetic coupling with the **power grid** of the circuits
- **Semi-local effect**
- **Quite easy to set-up**



Can **bypass some existing countermeasures**

Markettos, 2011 – Dehbaoui, 2012 – Zussa, 2014

OBJECTIVE OF THIS THESIS

- Some new attacks with EM injection have been achieved
  ➔ **new countermeasures** are necessary

Hardware countermeasures

- Requires significant changes
- Only for circuit manufacturers
- Need a finished circuit for testing

Software countermeasures

- More flexible changes
- Can be applied to processors
- Easier to test

- Difficult to model the impact of an EM injection on the execution of a program ➔ **assembly level**

Objective of this thesis :
Propose **software countermeasures** against **electromagnetic injection** attacks

CHOSEN APPROACH FOR THIS THESIS

Definition of a fault model

Study of the effects of a fault injection on an assembly program

Definition of a countermeasure

Résistant against the faults of the model and formally verified

Experimental evaluation of its efficiency

Experimental tests on isolated instructions and more complex codes

CHOSEN APPROACH FOR THIS THESIS

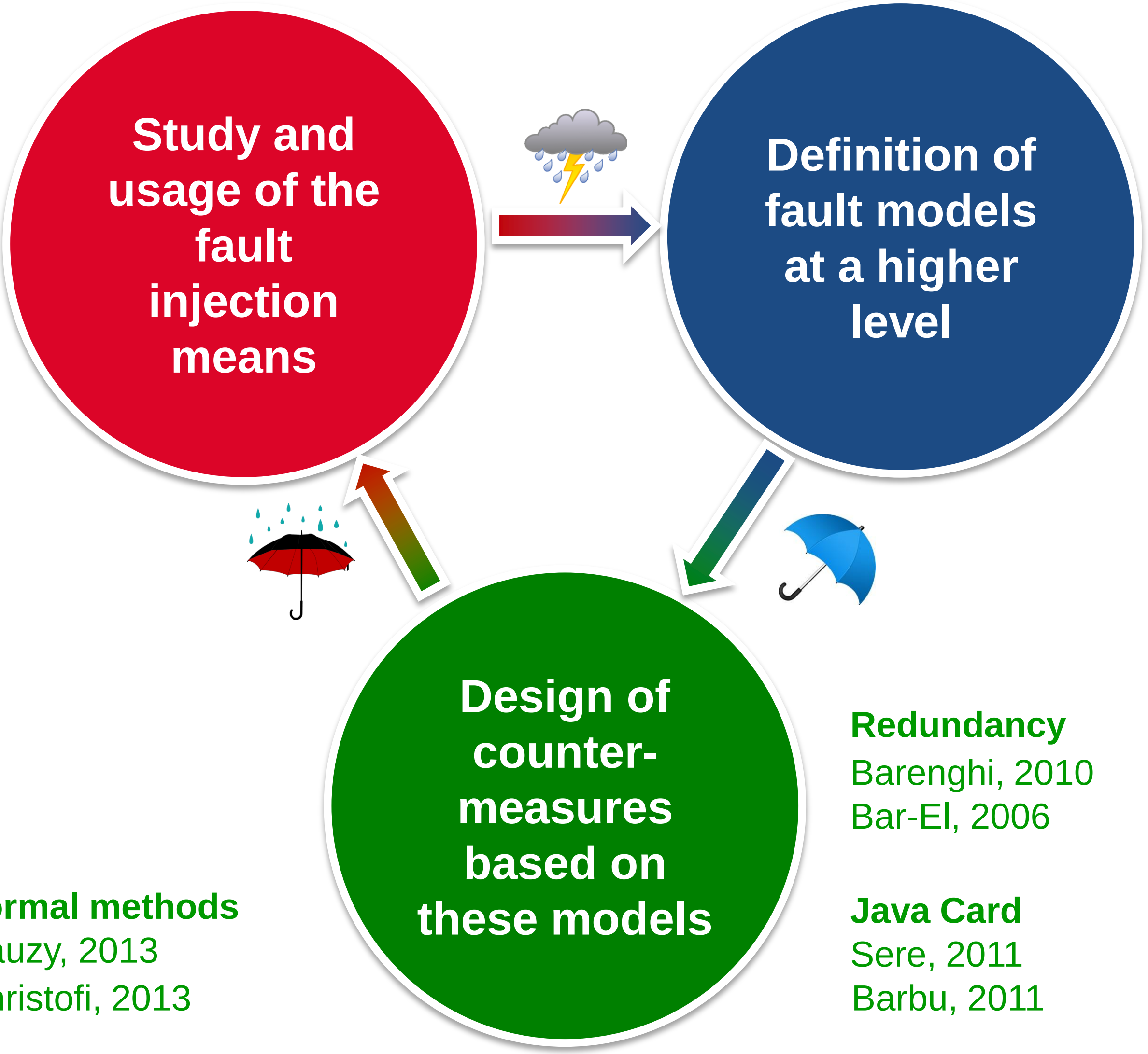
Laser
Roscian, 2013

EM waves
Dehbaoui, 2012

Voltage glitch
Zussa, 2014

Temperature
Schmidt, 2014

Clock glitch
Balasch, 2011


Study and usage of the fault injection means
Definition of fault models at a higher level
Design of counter-measures based on these models

Bit flip
Agoyan, 2010

Instruction skip
Barenghi, 2012

Instruction corruption
Balasch, 2011
Trichina, 2010

Branches
Berthomé, 2012

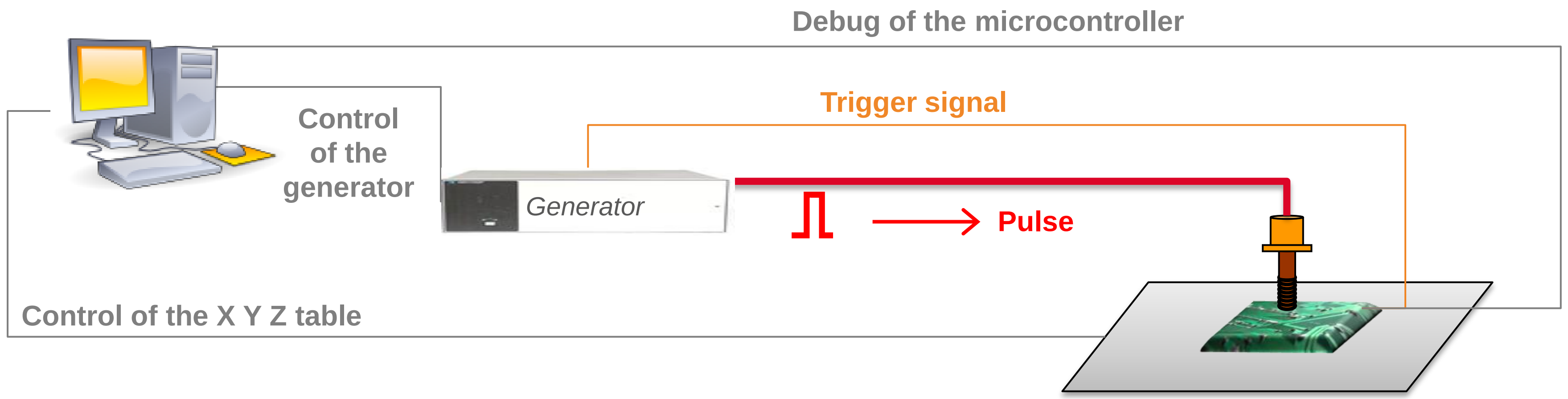
Formal methods
Rauzy, 2013
Christofi, 2013

Redundancy
Barenghi, 2010
Bar-El, 2006

Java Card
Sere, 2011
Barbu, 2011

OUTLINE OF THE PRESENTATION

FAULT INJECTION EXPERIMENTAL SETUP



1. The experiment is **driven from the PC**
2. The target code is **executed on the microcontroller**
3. The microcontroller **sends a trigger signal**
4. The generator **sends a voltage pulse**
5. The microcontroller is stopped
6. The internal data is harvested

THE ARM CORTEX-M3 PROCESSOR

- ARM architecture **in the majority of embedded systems**
- Several secured processors based on an ARM Cortex-M
- The Cortex-M3 architecture is already used for some **smart cards** or some **processors for RFID communications**

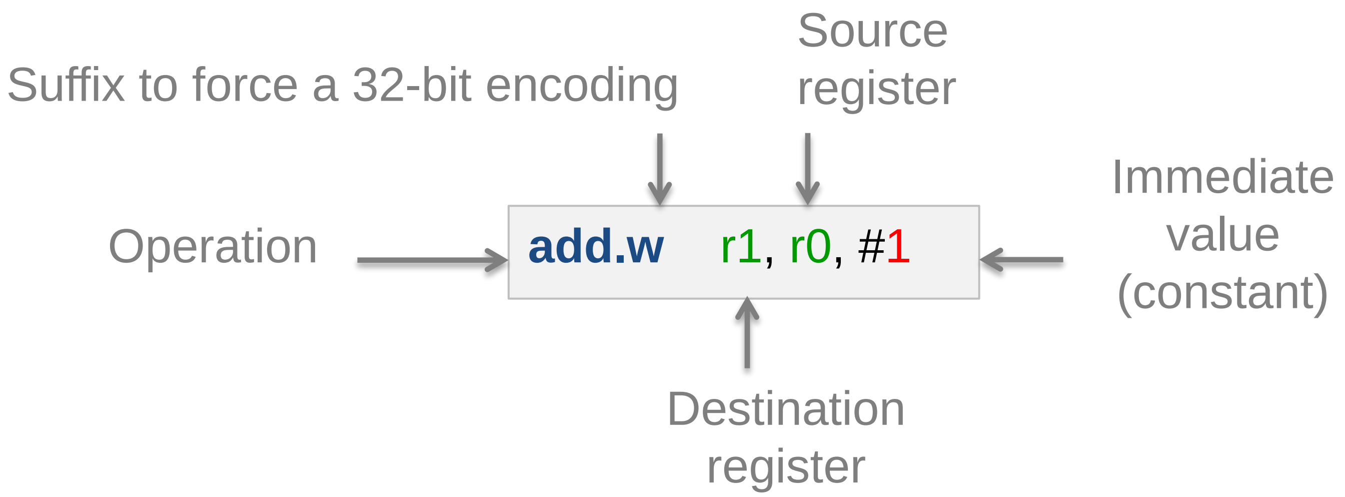
- Fréquency of **56 MHz**, clock period 17.8 ns
- Architecture ARMv7-M 32-bit (Harvard type)

The Definitive Guide to the ARM Cortex-M3 – Joseph Yiu, Newnes, 2009

THE ARM CORTEX-M3 PROCESSOR

Thumb-2 instruction set

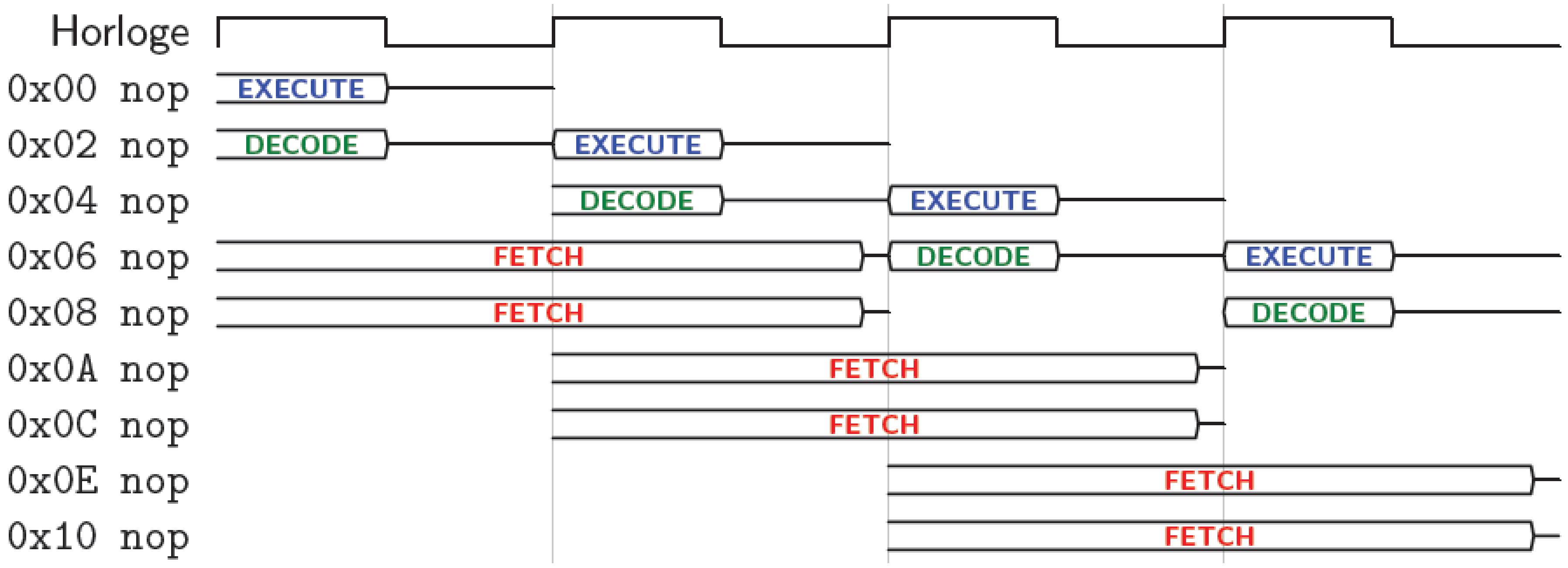
- RISC, 151 instructions encoded on 16 and 32 bits
- Load/store architectures : operations are performed on registers



PIPELINE AND EXECUTION OF THE INSTRUCTIONS

3 levels of pipeline (Fetch – Decode – Execute), no prefetch

Fetch	Loading of the instruction into the instruction register
Decode	Decoding, operand fetch, branch detection
Execute	Execution of the instruction, writing of the results

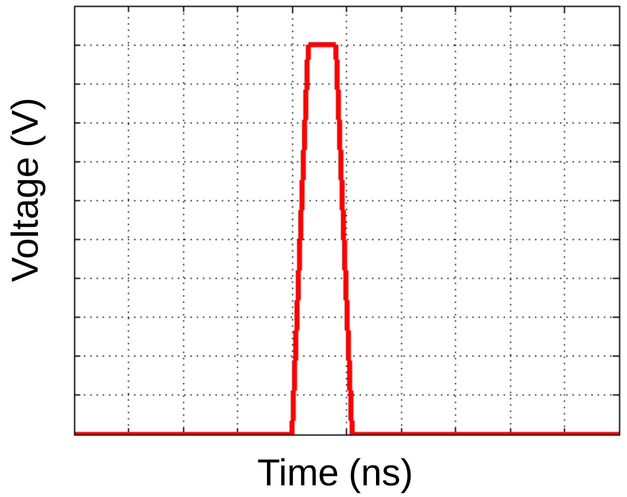


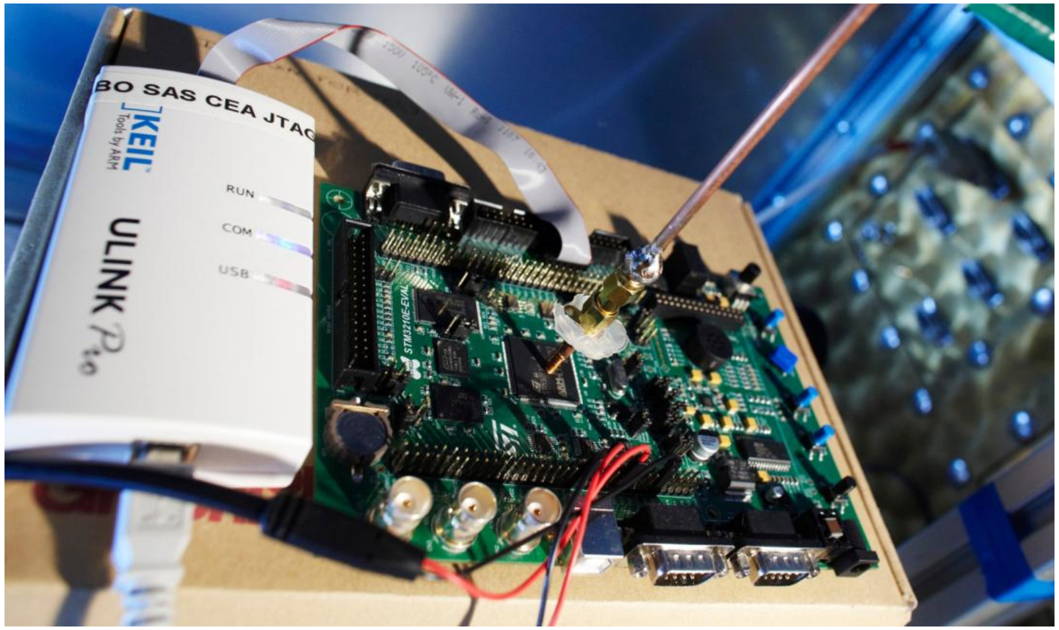
PULSED ELECTROMAGNETIC FAULT INJECTION

- The fault injection antenna is a **copper coil**

- **Keil ULINKpro** JTAG debug probe
  Enables to use the microcontroller in debug mode

- **Pulse** generator
  High voltage, high current





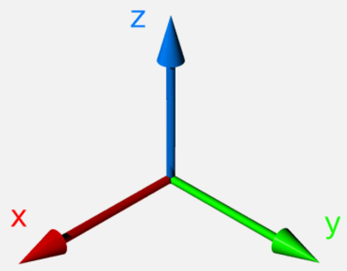
OUTLINE OF THE PRESENTATION

DEFINITION OF A FAULT MODEL

- Enables to better understand the **abilities of an attacker**
- A big number of **experimental parameters**
- Their influence on the obtained faults **must be studied**

Studied parameters

Position of the antenna



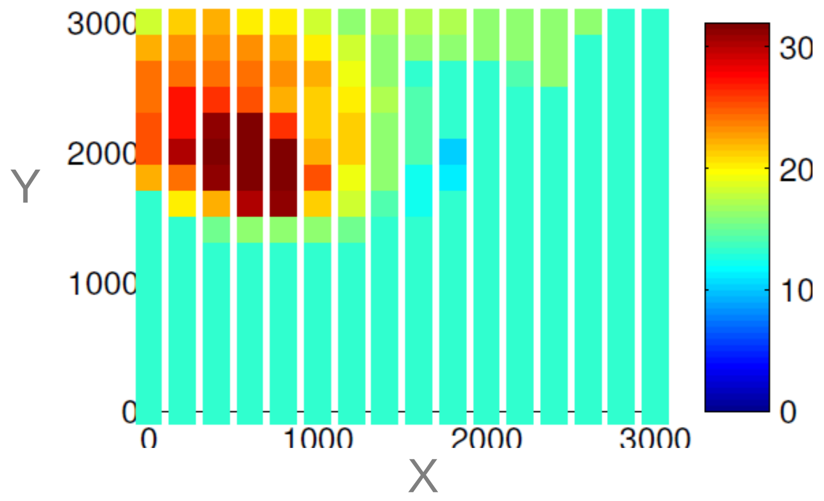
Injection time

Pulse voltage

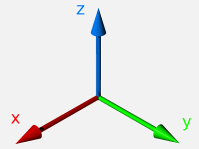
INFLUENCE OF THE POSITION OF THE ANTENNA

`ldr r8,=0x12345678` ➔ loads a 32-bit word from the Flash memory

- **Variation of X and Y** on a square with 3mm side
- Fixed voltage, fixed injection time, fixed position on the Z axis



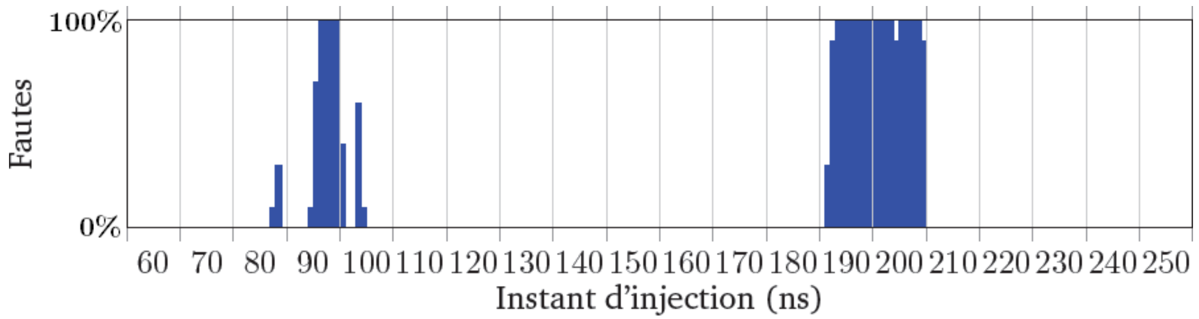
Hamming weight of the loaded word

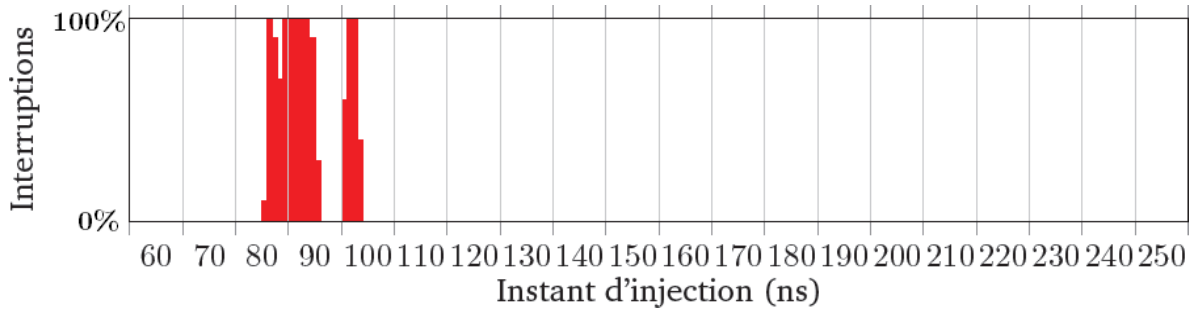


- ➢ A **local effect** of the fault injection technique
- ➢ A **very small area** to inject faults

INFLUENCE OF THE INJECTION TIME

`ldr r8,=0x12345678` ➔ loads a 32-bit word from the Flash memory





- **Variation of the injection time**
- Fixed voltage
- Fixed antenna

➢ Two **distinct time intervals**
➢ Different **kinds of faults**
➢ For some injection times, we obtained **100% of faults**

INFLUENCE OF THE PULSE'S VOLTAGE

`ldr r4,=0x12345678` ➔ loads a 32-bit word from the Flash memory

- **Variation of the pulse voltage**
- Fixed antenna, fixed injection time

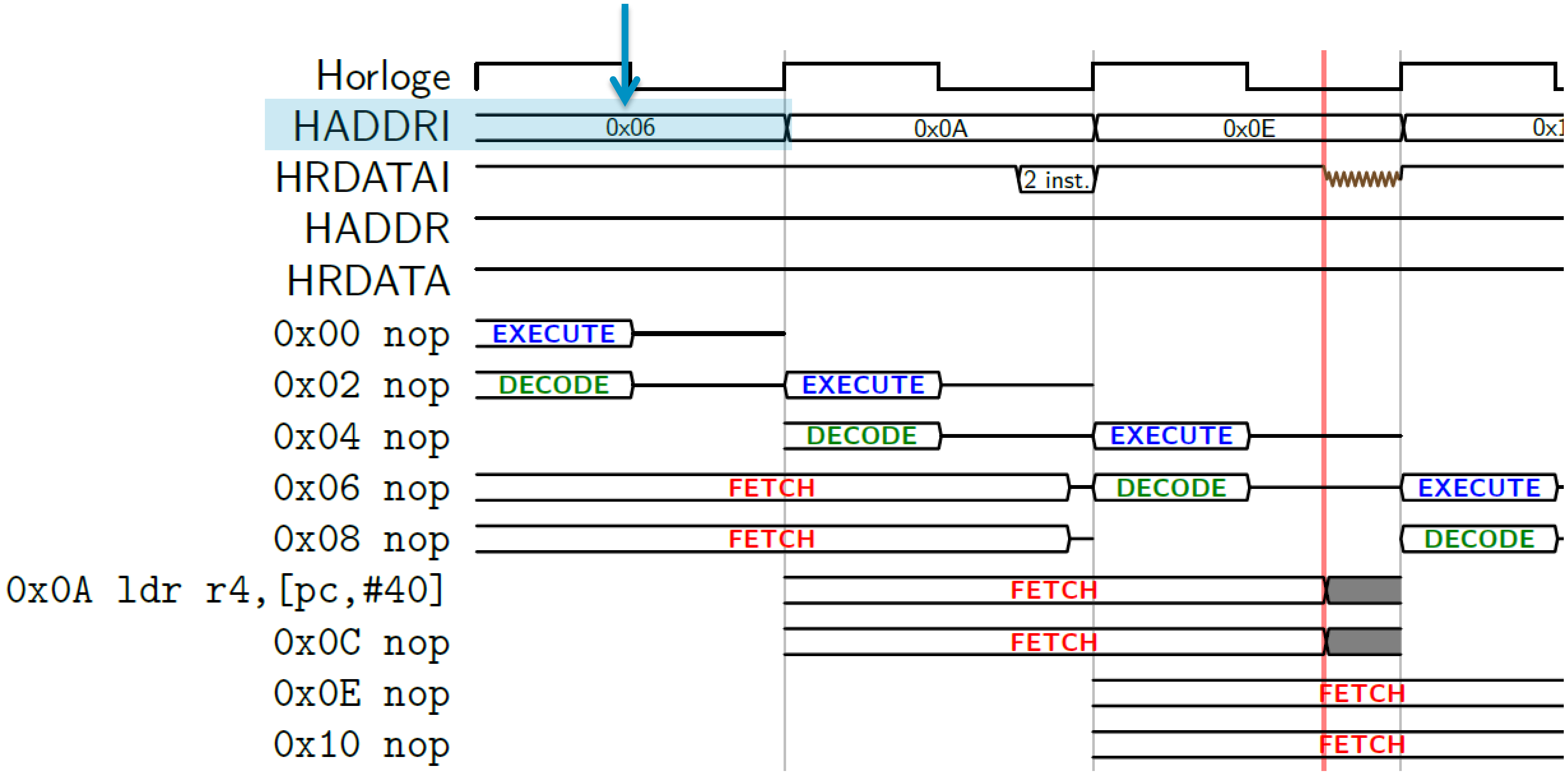
Voltage	Output value
172V	1234 5678
174V	**9**234 5678
176V	**FE**34 5678
178V	**FFF**4 5678
180V	**FFFD** 5678
182V	**FFFF 7F**78
184V	**FFFF FFFD**
186V	**FFFF FFFF**

- ➢ A **set at 1 effect** on the bits
- ➢ The effect is related to the **increase of the voltage**
- ➢ Only obtained when loading data **from the Flash memory**

REGISTER-TRANSFER LEVEL FAULT MODEL

Transfer of an instruction on the HRDATAI instruction bus

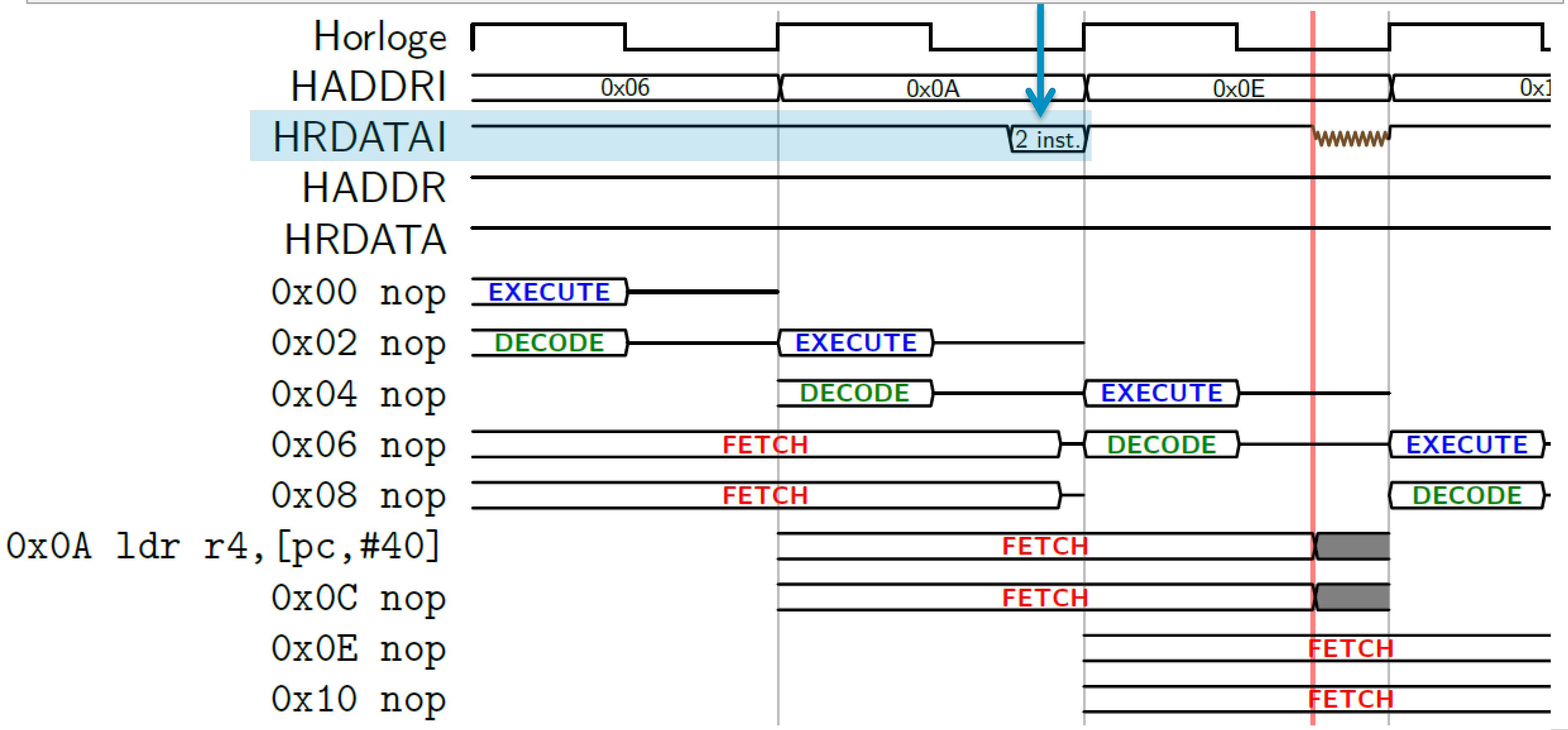
1 – The address of the instruction is put on the HADDRI bus



REGISTER-TRANSFER LEVEL FAULT MODEL

Transfer of an instruction on the HRDATAI instruction bus

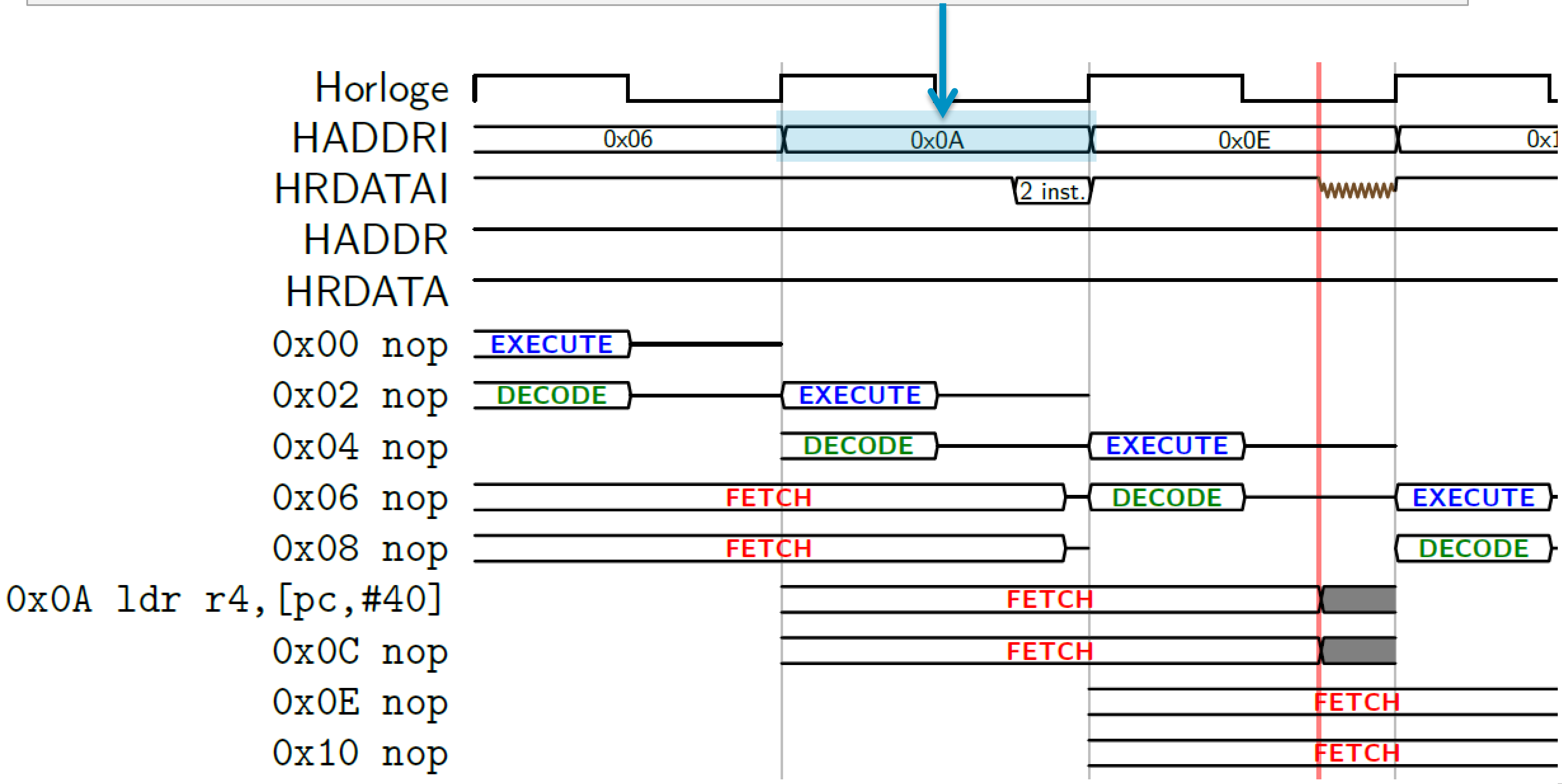
2a – The binary encoding of the instruction is put on the HRDATAI bus
(end of the following clock cycle)



REGISTER-TRANSFER LEVEL FAULT MODEL

Transfer of an instruction on the HRDATAI instruction bus

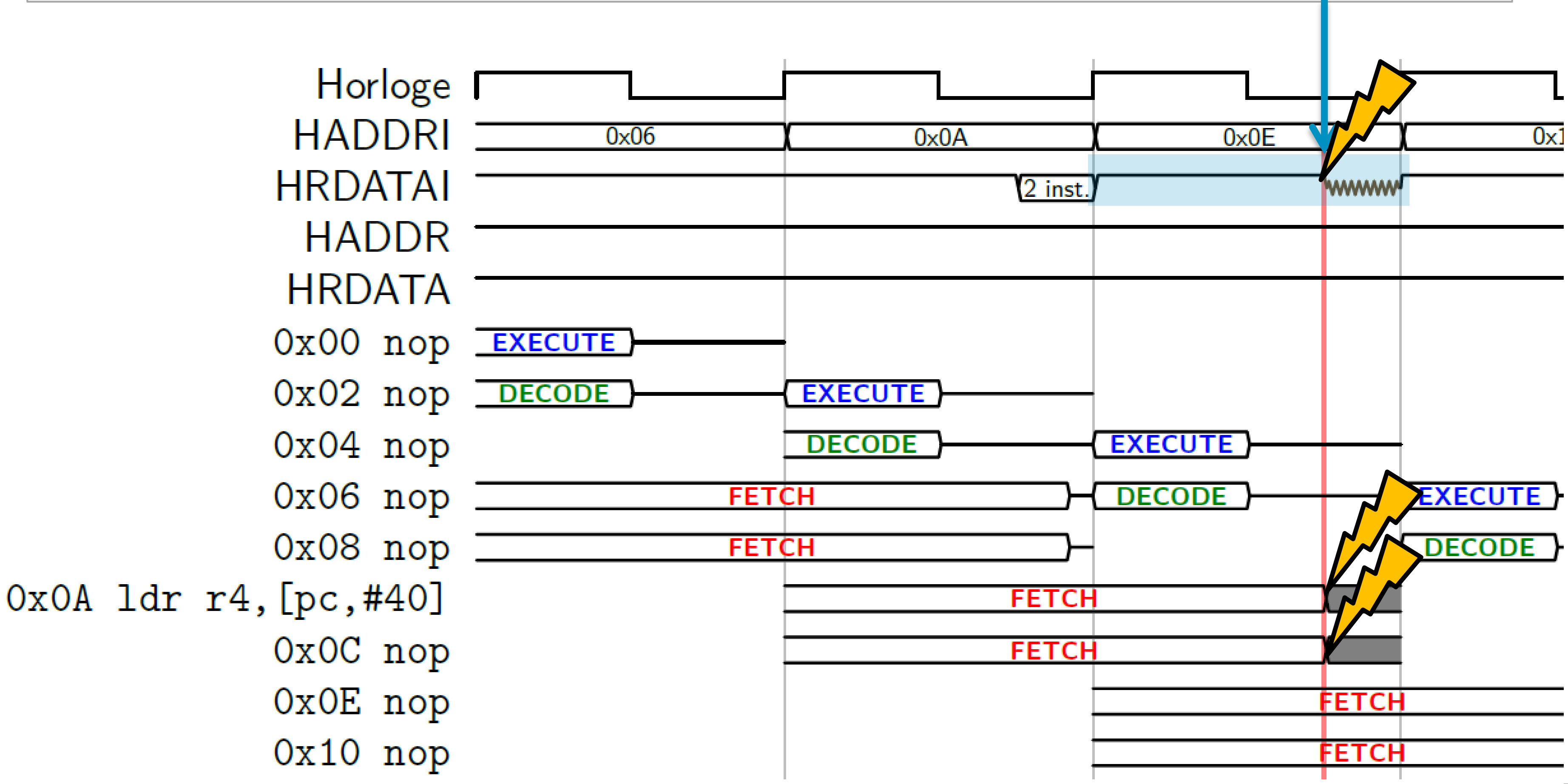
2b – The address of the next instruction is put on the HADDRI bus



REGISTER-TRANSFER LEVEL FAULT MODEL

Transfer of an instruction on the HRDATAI instruction bus

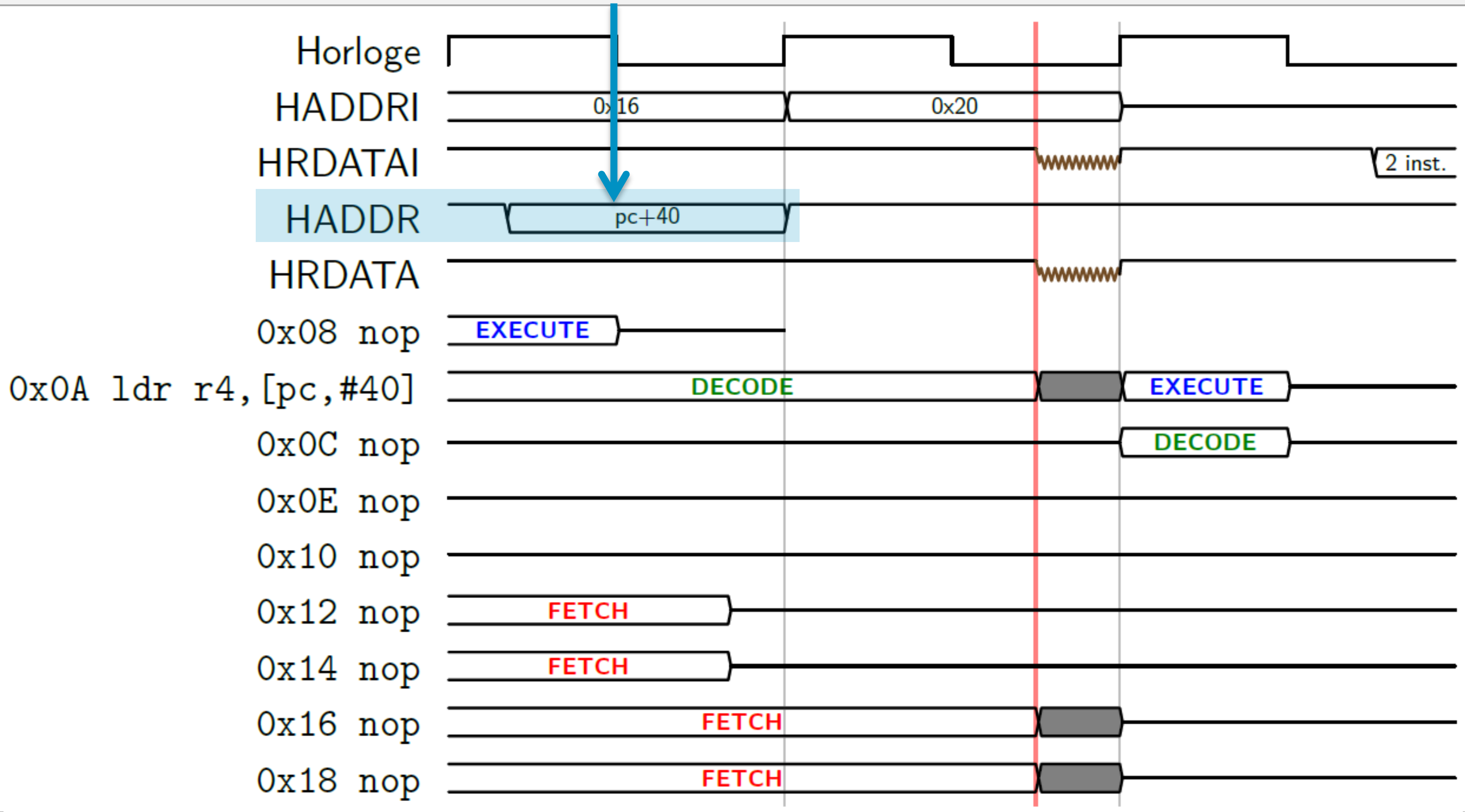
3 – Corruption of the transfer of the instruction on the HRDATAI bus



REGISTER-TRANSFER LEVEL FAULT MODEL

Transfer of a piece of data on the HRDATA data bus

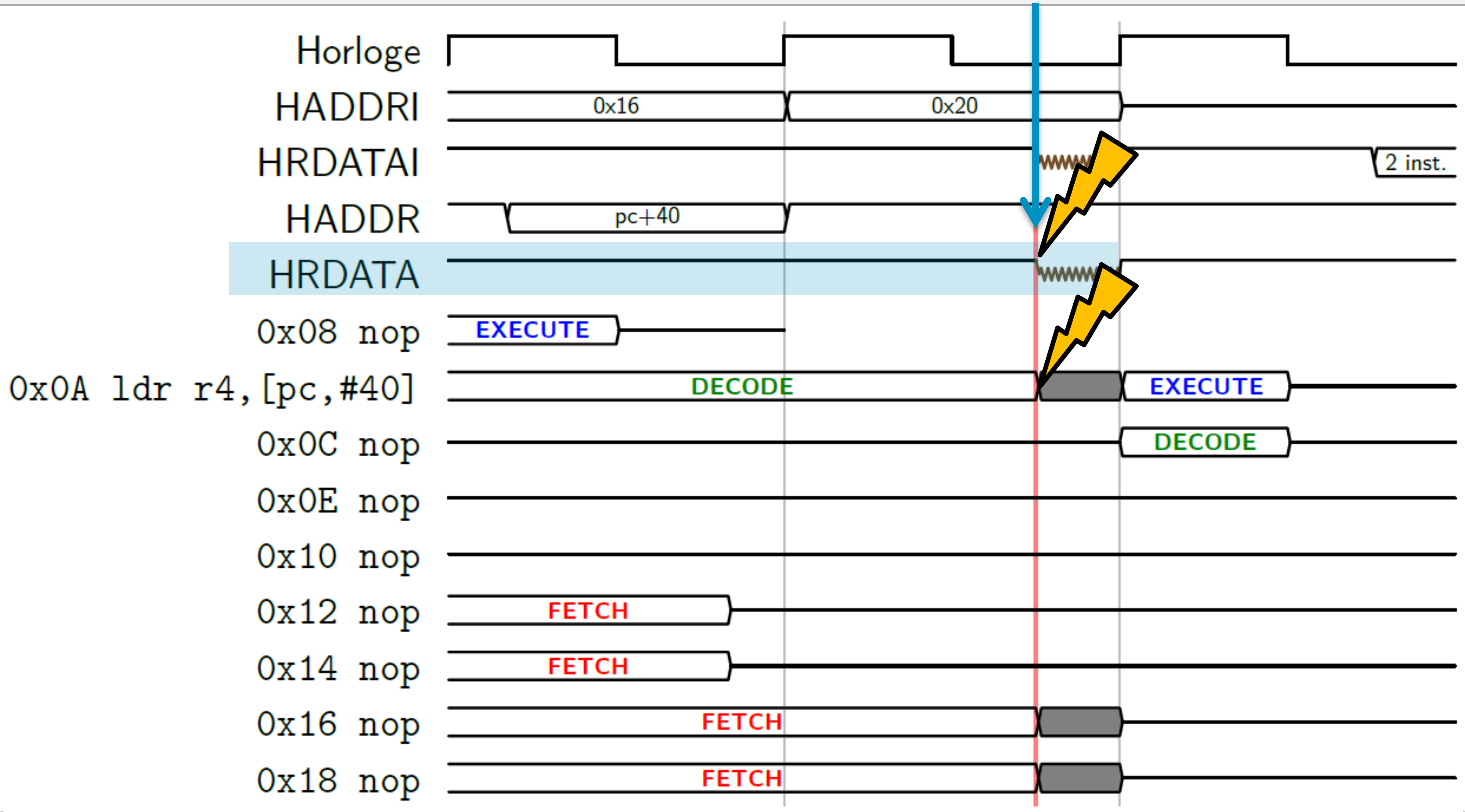
1 – The address of the piece of data is put on the HADDR bus



REGISTER-TRANSFER LEVEL FAULT MODEL

Transfer of a piece of data on the HRDATA data bus

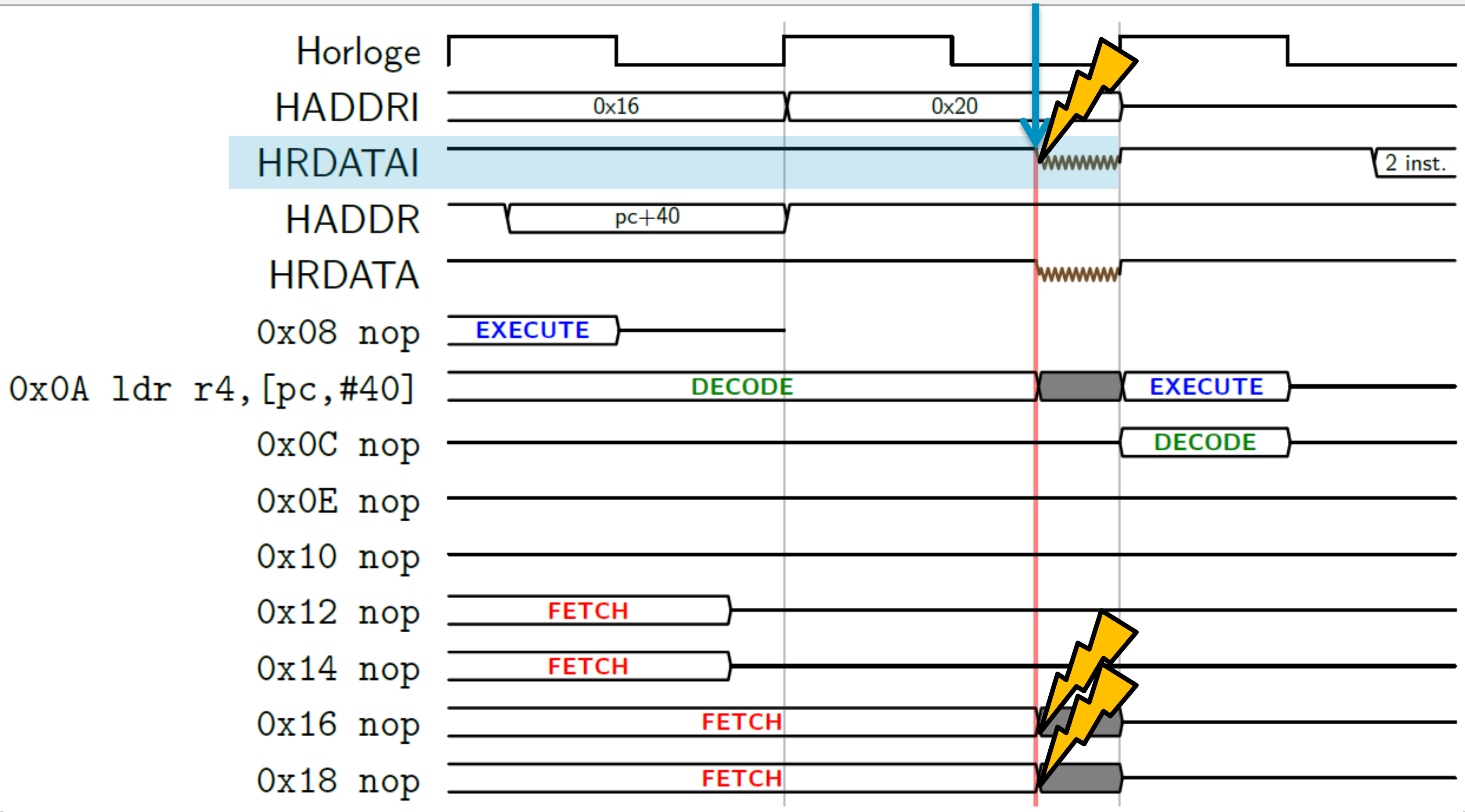
2a – Corruption of the transfer of the pice of data on the HRDATA bus



REGISTER-TRANSFER LEVEL FAULT MODEL

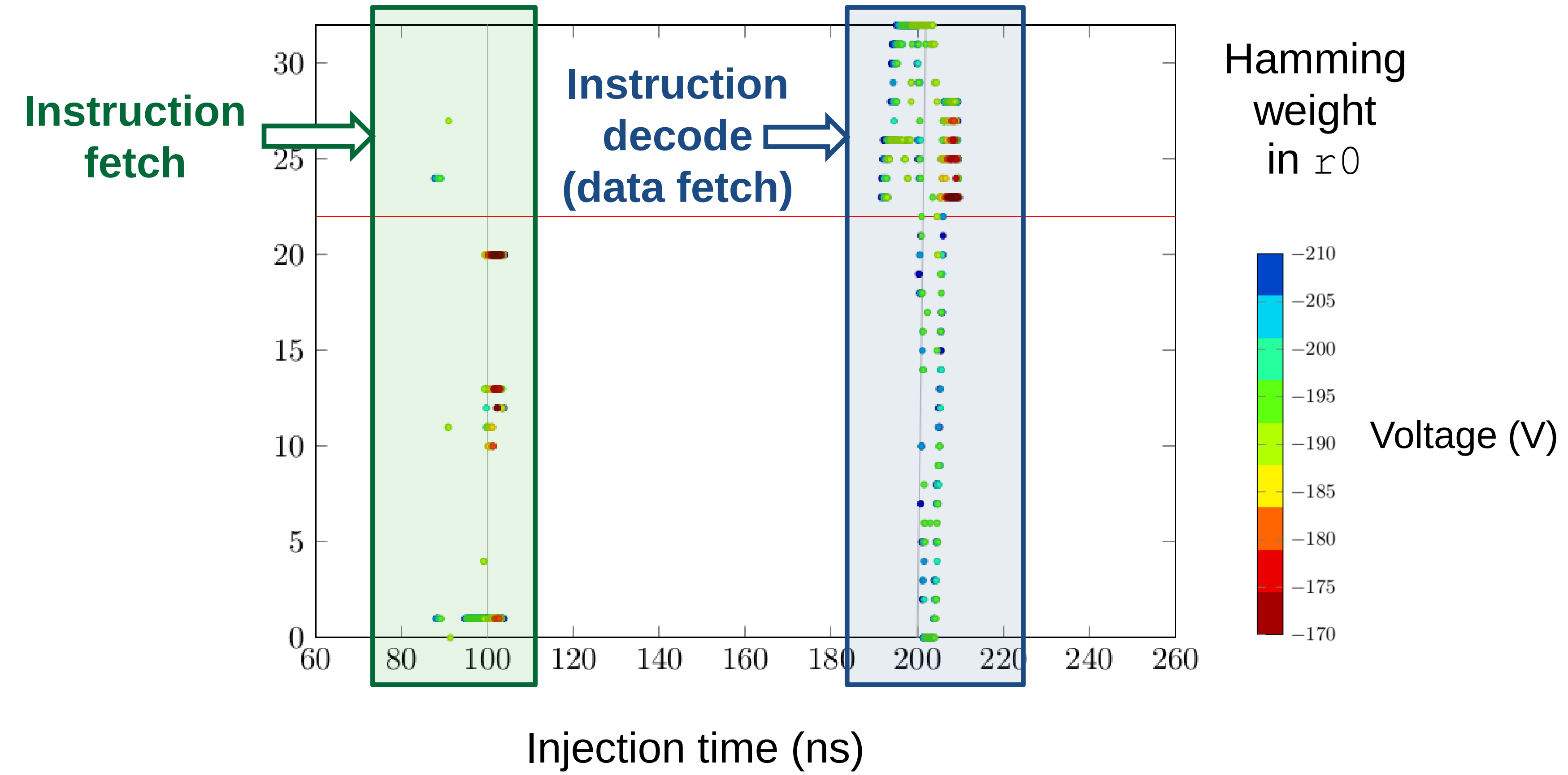
Transfer of a piece of data on the HRDATA data bus

2b – Corruption of the transfer of other instructions on the HRDATAI bus



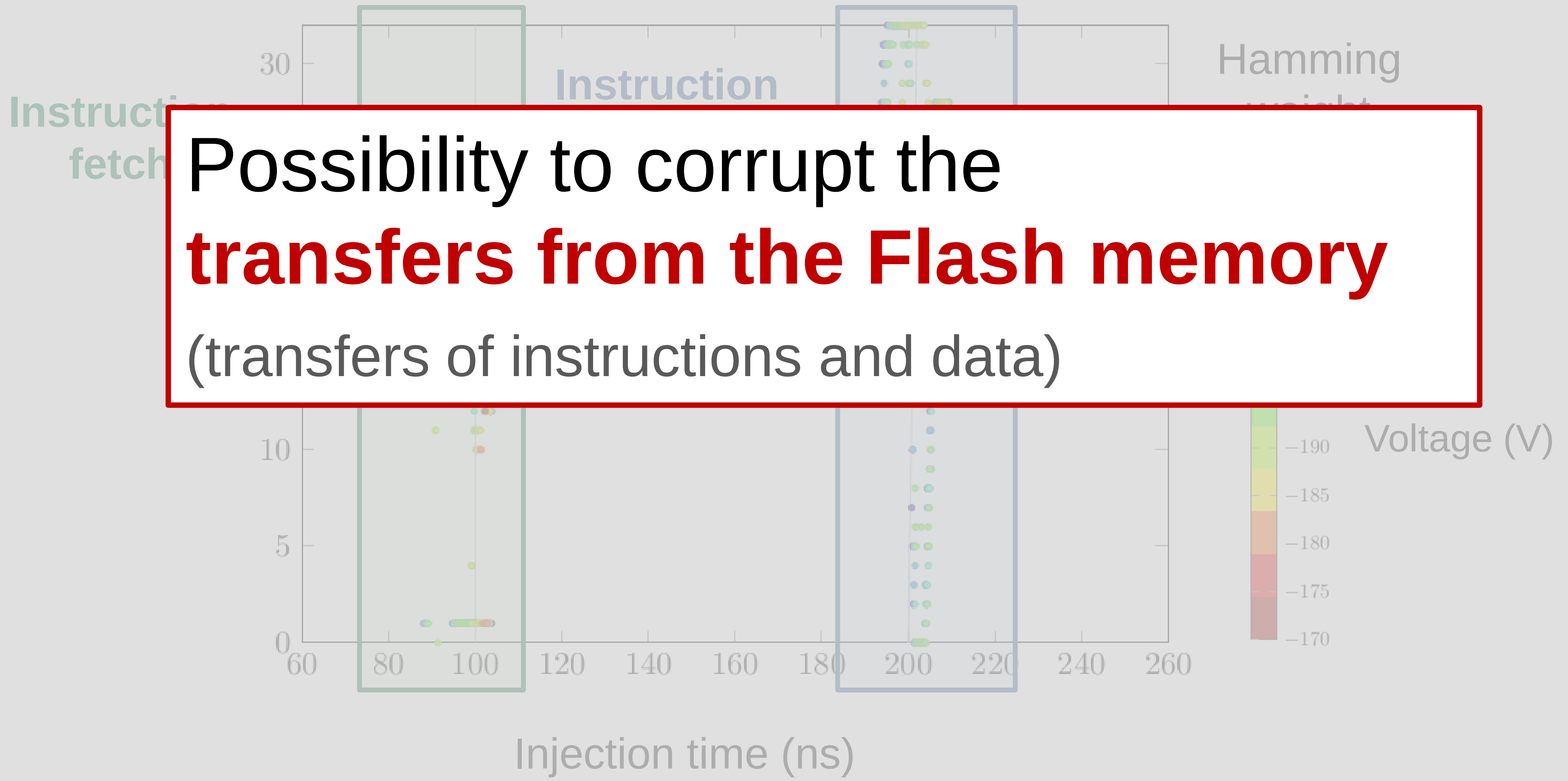
EXPERIMENTAL VALIDATION OF THIS RTL FAULT MODEL

`ldr r0, [pc,#40]` ➔ loads a 32-bit word from the Flash memory



EXPERIMENTAL VALIDATION OF THIS RTL FAULT MODEL

`ldr r0, [pc,#40]` ➔ loads a 32-bit word from the Flash memory



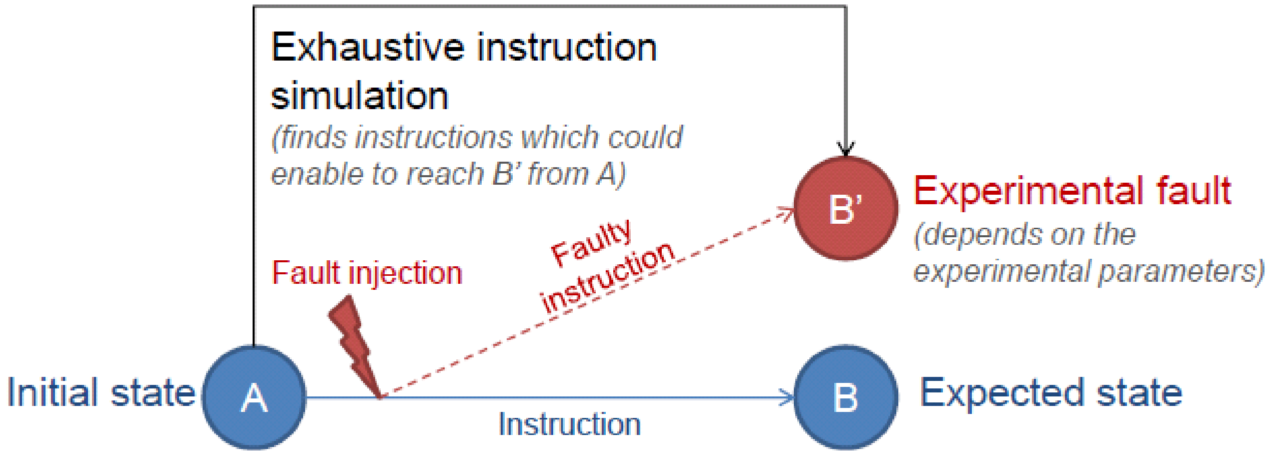
Possibility to corrupt the
transfers from the Flash memory
(transfers of instructions and data)

FAULTS ON THE DATA FLOW AND THE CONTROL FLOW

How can we extract from the previous results a **fault model** for the instruction replacements ?

➔ Search for replacements
that can qui peuvent **explain the obtained faults**

- **Simulation** of instruction corruptions
- **Comparison** with the experimental results


Exhaustive instruction simulation
(finds instructions which could enable to reach B' from A)
Fault injection
Faulty instruction
B'
Experimental fault
(depends on the experimental parameters)
Initial state
A
Instruction
B
Expected state

OVERVIEW OF THE FAULT MODEL AT RTL LEVEL

Consequences regarding instructions

- Instruction **replacements**
- Instruction **skips** in some cases

nop	1011 1111 0000 0000	**str** r0, [r0, #0]	0110 0000 0000 0000
nop	1011 1111 0000 0000	**nop**	1011 1111 0000 0000

Consequences regarding data

- Corruption of the `ldr` from the Flash memory (encryption keys, …)
- Values with high Hamming weight easier to obtain (on this target)

Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller
N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, E.Encrenaz – FDTC 2013, Santa-Barbara, USA

FAULT MODEL AT ASSEMBLY LEVEL

« nop » fault model (instruction skip)

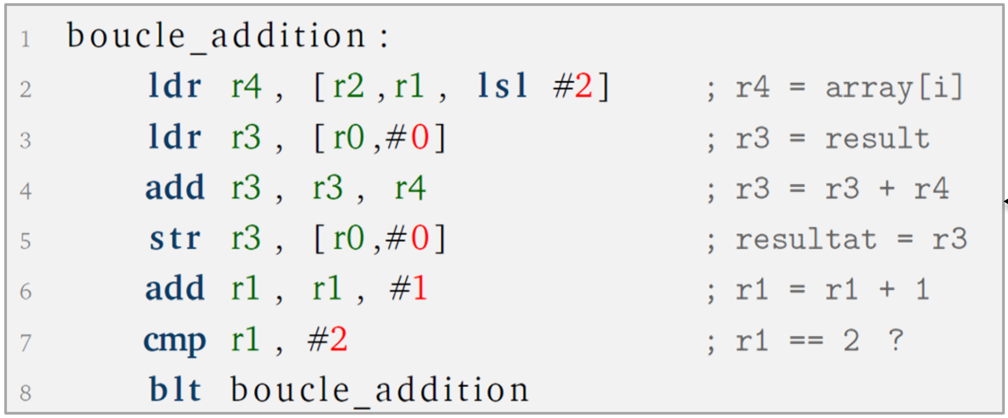
- Can enable to skip a subroutine call
- Possible to **detect some vulnerabilities** on a program

Replacement by un nop statistically more frequent

- Writing into a dead register or a unused memory address
- Re-execution of a previous idempotent instruction (`add r1,r2,r3` …)
- Replacement by an instruction without any effect (`mov r0,r0` …)

In which proportion do the injected faults have an effect that is **similar to an instruction skip** (nop) ?

EVALUATION THE NOP MODEL’S COVERAGE RATE

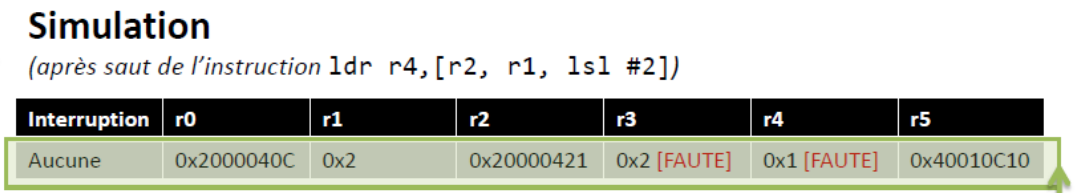


```
boucle_addition :
      ldr  r4 , [r2 ,r1 , lsl #2]      ; r4 = array[i]
      ldr  r3 , [r0 ,#0]               ; r3 = result
      add  r3 , r3 , r4                ; r3 = r3 + r4
      str  r3 , [r0 ,#0]               ; resultat = r3
      add  r1 , r1 , #1                ; r1 = r1 + 1
      cmp  r1 , #2                     ; r1 == 2 ?
      blt  boucle_addition
```

1 **Experiment** on a program that sums the elements of a 2-value array

tab[0] = 1 ; tab[1] = 2
Expected result : 3

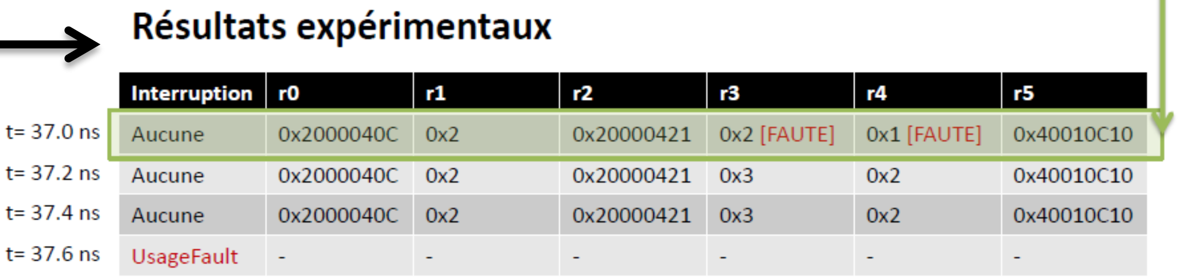
2 **Simulation** of the skip of every instruction



Simulation
(après saut de l’instruction `ldr r4,[r2, r1, lsl #2]`*)*

Interruption	r0	r1	r2	r3	r4	r5
Aucune	0x2000040C	0x2	0x20000421	0x2 [FAUTE]	0x1 [FAUTE]	0x40010C10

3 **Comparison** between the output values and the experimental results



Résultats expérimentaux

	Interruption	r0	r1	r2	r3	r4	r5
t= 37.0 ns	Aucune	0x2000040C	0x2	0x20000421	0x2 [FAUTE]	0x1 [FAUTE]	0x40010C10
t= 37.2 ns	Aucune	0x2000040C	0x2	0x20000421	0x3	0x2	0x40010C10
t= 37.4 ns	Aucune	0x2000040C	0x2	0x20000421	0x3	0x2	0x40010C10
t= 37.6 ns	UsageFault	-	-	-	-	-	-

EVALUATION THE NOP MODEL’S COVERAGE RATE

```
boucle_addition:
      ldr r4, [r2,r1, lsl #2]      ; r4 = array[i]
      ldr r3, [r0,#0]              ; r3 = result
      add r3, r3, r4               ; r3 = r3 + r4
      str r3, [r0,#0]              ; resultat = r3
      add r
      cmp r
      blt b
```

1 **Experiment** on a program that sums the elements of a 2-value

On the tested program, about **25% of the faults obtained in practice** can be seen as an instruction skip

2 **Sim** skip of every instruction

						r5
Aucune	0x2000040C	0x2	0x20000421	0x2 [FAUTE]	0x1 [FAUTE]	0x40010C10

3 **Comparison** between the output values and the experimental results

Résultats expérimentaux

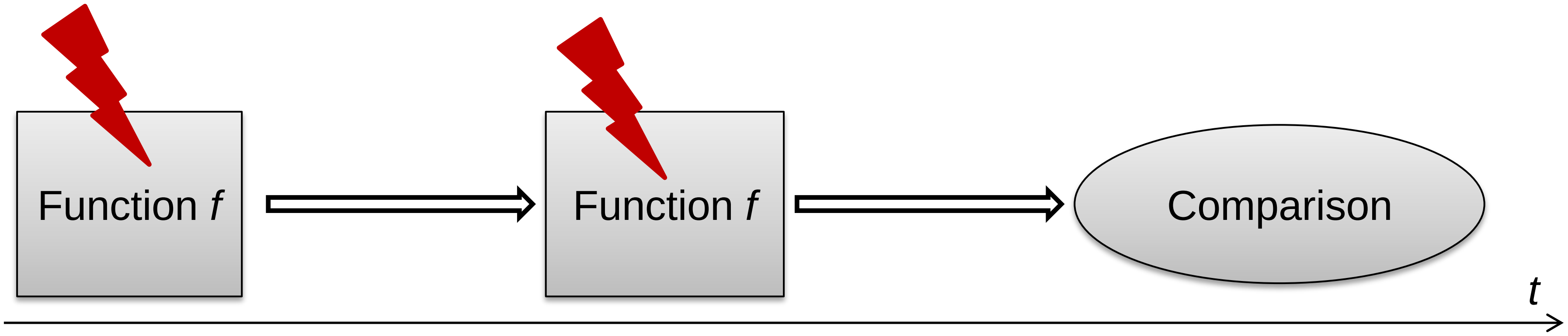
	Interruption	r0	r1	r2	r3	r4	r5
t= 37.0 ns	Aucune	0x2000040C	0x2	0x20000421	0x2 [FAUTE]	0x1 [FAUTE]	0x40010C10
t= 37.2 ns	Aucune	0x2000040C	0x2	0x20000421	0x3	0x2	0x40010C10
t= 37.4 ns	Aucune	0x2000040C	0x2	0x20000421	0x3	0x2	0x40010C10
t= 37.6 ns	UsageFault	-	-	-	-	-	-

OUTLINE OF THE PRESENTATION

ATTACK PATHS ENABLED BY THE MODEL

- Many **instruction replacements** have an effect that is equivalent to an **instruction skip**

- Some **double faults** are possible if the time between both injections is high enough (a few µs for our bench)



How can we **guarantee a correct execution** with a potential instruction skip by an attacker ?

COUNTERMEASURE AGAINST AN INSTRUCTION SKIP

1. **Able to resist to a fault injection**
   ➔ Based on a temporal redundancy principle

2. **Resistant to double faults sufficiently far apart**
   ➔ Instruction-level redundancy

3. **Can be automatically applied**
   ➔ Replacement sequence for every instruction
   ➔ Semantic equivalence regarding the initial instruction
   ➔ Principle to reinforce a full program

DIFFERENT CLASSES OF INSTRUCTIONS

Idempotent instructions

```
add     r1, r0, #1
```

```
add     r1, r0, #1
add     r1, r0, #1
```

Separable instructions

```
add     r1, r1, #1
```

Duplication not correct if no fault

➔ Separation then duplication

```
add     r12, r1, #1
add     r12, r1, #1
mov     r1, r12
mov     r1, r12
```

DIFFERENT CLASSES OF INSTRUCTIONS

Idempotent instructions

```
add     r1, r0, #1
```

↓

```
add     r1, r0, #1
add     r1, r0, #1
```

Separable instructions

```
push    {r1, r2, r3, lr}
```

↓

➔ Separation then duplication

```
stmdb   sp, {r1, r2, r3, lr}
stmdb   sp, {r1, r2, r3, lr}
sub     r12, sp, #16
sub     r12, sp, #16
mov     sp, r12
mov     sp, r12
```

DIFFERENT CLASSES OF INSTRUCTIONS

Idempotent instructions

```
add     r1, r0, #1
```

```
add     r1, r0, #1
add     r1, r0, #1
```

```
umlal   r1, r2, r3, r4
```

r1:r2 = r3*r4 + r1:r2
Multiplication and addition over 64 bits

```
mrs     r12, apsr
mrs     r12, apsr
umull   r10, r11, r3, r4
umull   r10, r11, r3, r4
adds    r9, r10, r1
adds    r9, r10, r1
addc    r10, r11, r2
addc    r10, r11, r2
mov     r1, r9
mov     r1, r9
mov     r2, r10
mov     r2, r10
msr     apsr, r12
msr     apsr, r12
```

SUBROUTINE CALL INSTRUCTION (BL)

- Branch instruction can be duplicated…
  but not subroutine call instructions
  (otherwise every subroutine would be executed twice)

```
    bl          function
```

```
    adr         r12, return_label
    adr         r12, return_label
    add         lr, r12, #1
    add         lr, r12, #1
    b           function
    b           function

return_label
```

Puts the return addess into r12

Updates the return pointer lr

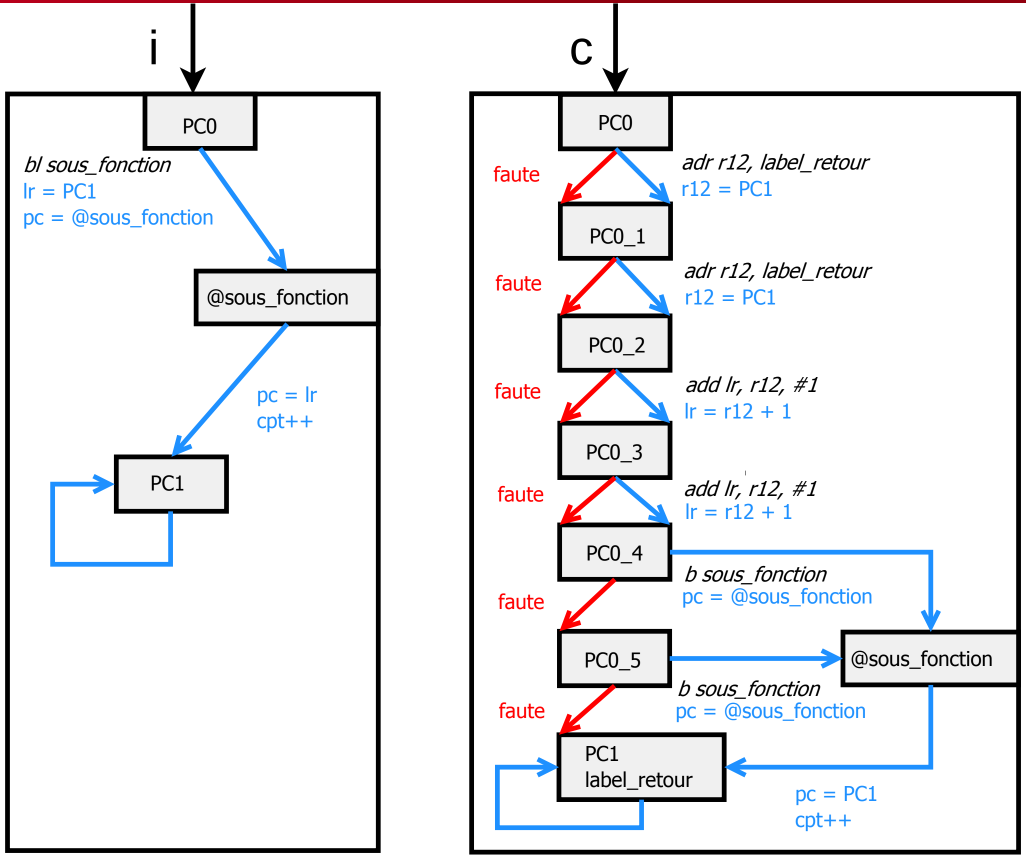
Branches to the subfunction

VERIFICATION OF THE COUTERMEASURE WITH MODEL-CHECKING

Properties to verify :

1. Every replacement sequence has the **same semantics than the instruction it replaces**

2. Every replacement sequence is **tolerant to an instruction skip**

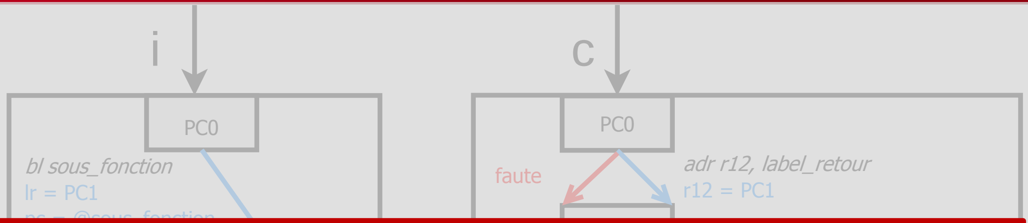


$$AG[((i.pc = pc\_init\_i) \wedge (c.pc = pc\_init\_c)) \Rightarrow$$
$$AF((i.pc = pc\_final\_i) \wedge (c.pc = pc\_final\_c) \wedge \forall x \in D, (i.x = c.x))]$$

VERIFICATION OF THE COUTERMEASURE WITH MODEL-CHECKING

Properties to verify :

1. Every replacement



The verification step has a **double purpose** :

1. Verification of the equivalence **of sequences**
(with and without fault injections)

2. Help for the **design of sequences**
(enables to highlight the challenging cases)

$$AG[((i.pc = pc\_init\_i) \wedge (c.pc = pc\_init\_c)) \Rightarrow$$
$$AF((i.pc = pc\_final\_i) \wedge (c.pc = pc\_final\_c) \wedge \forall x \in D, (i.x = c.x))]$$

AUTOMATIC APPLICATION OF THE COUNTERMEASURE

- An **automatic application algorithm** has been designed

Implementation	Overhead (cycles)	Overhead (code size)
AES	**+ 113.7%**	**+ 202%**
MiBench AES	**+ 186.4%**	**+ 189.9%**
MiBench SHA0	**+ 122.8%**	**+ 178.2%**
AES with CM on the last two rounds	**+ 18.6%**	**+ 282.5%**

➔ **High overhead cost**,

but **comparable** to the overhead of usual redundancy approaches

Formal verification of a software countermeasure against instruction skip fault attacks
N. Moro, K. Heydemann, E.Encrenaz, B. Robisson - Journal of Cryptographic Engineering, 2014

OUTLINE OF THE PRESENTATION

EXPERIMENTAL EVALUATION OF THE COUNTERMEASURE

- Countermeasure against a **simplified model of attacker** (skip of an assembly instruction)

- Does not protect against the faults on the **data flow**

Can be **complemented** with
- a fault **detection** countermeasure
- that also protects **data loads**

Countermeasures against fault attacks on software implemented AES
A. Barenghi, L. Breveglieri, I.Koren, G. Pelosi, F. Regazzoni – WESS 2010, New-York, USA

FAULT DETECTION COUNTERMEASURE

- Detection of **single faults**
  Instruction skip, some replacements, fault on the data flow

- Proposed for a **restricted set of instructions**
  Arithmetic and logic, load-store
  … but not branches, stack manipulation or flags use

- **High** overhead
  In registers, code size and number of cycles

```
ldr     r0, [pc, #34]
```

⇨

```
ldr     r0, [pc, #40]
ldr     r1, [pc, #38]
cmp     r0, r1
bne     error
```

EXPERIMENTAL EVALUATION METHOD

Study of the impact of the countermeasures for :

- **some isolated instructions**
- **some complex codes**

Chosen isolated instructions :

Fault tolerance countermeasure

➢ **`bl` instruction**

```
    adr     r12, return_label
    adr     r12, return_label
    add     lr, r12, #1
    add     lr, r12, #1
    b       function
    b       function

return_label
```

Fault detection countermeasure

➢ **`ldr` instruction**

```
ldr         r0, [pc, #40]
ldr         r1, [pc, #38]
cmp         r0, r1
bne         error
```

EVALUATION ON ISOLATED INSTRUCTIONS

For **both countermeasures** :

- It is necessary to **force a 32-bit encoding**

```
    adr     r12, return_label
    adr     r12, return_label
```

```
ldr          r0, [pc, #40]
ldr          r1, [pc, #38]
```

For the **tolerance countermeasure** :

- On a subroutine call, **97% reduction** of the output faults

For the **detection countermeasure** :

- On a data load, **98% reduction** of the output faults

EVALUATION ON COMPLEX CODES

Evaluated on a **FreeRTOS-MPU implementation**

Tolerance CM	Function that changes the privilege level
Detection CM	Function that initializes a task and sets its priority

For the tolérance countermeasure :

→ **26% reduction** of the faults in the output register
→ Effect that is probably more complex than an instruction skip

For the detection countermeasure :

→ **98% reduction** of the faults in the output register

Experimental evaluation of two software countermeasures against fault attacks
N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, E. Encrenaz – IEEE HOST 2014, Arglinton, USA

COMBINATION OF THE TWO COUNTERMEASURES

➢ **Both countermeasures** are used to **reinforce the same code** (an AES addRoundKey function)

```
    bl.w    addRoundKey         ; branchement vers la fonction
    bx      lr                  ; sortie de la fonction de chiffrement

addRoundKey
    ldrb.w  r2, [r0, #0]        ; chargement du premier octet de texte
    ldrb.w  r3, [r1, #0]        ; chargement du premier octet de clé
    eors.w  r2, r2, r3          ; OU EXCLUSIF entre les deux octets
    strb.w  r2, [r0, #0]        ; stockage du résultat en mémoire
```

- **Detection CM**: more efficient but cannot be applied everywhere
- **With proposed CM**: protection of the other instructions

➔ **90% reduction** for the faults, among the remaining ones **no usable fault** for a cryptanalysis

OUTLINE OF THE PRESENTATION

CONCLUSION

- An **accurate fault model** at assembly level
  - Corruption of the transfers from the Flash memory
  - Double-faults possible under certain conditions
  - High percentage of instruction skips

- A **countermeasure tolerant to an instruction skip**
  - Local redundancy at instruction-scale
  - Formally verified with model-checking tools
  - Automatically applicable to a generic code
  - Reinforces especially branches and stack

CONCLUSION

- An **experimental evaluation** of two countermeasures
  - With another fault detection countermeasure
  - Importance of the encoding of instructions
  - Combination of the two CM very efficient on an AES code
  - First tests with DPA-like side-channel analysis techniques

➔ The proposed fault tolerance countermeasure **is a good complement for the detection countermeasure** for the subroutine calls and the instructions for which it does not apply

PERSPECTIVES

- Improvement of the **accuracy of fault models**
  - Towards a better understanding of instruction replacements
  - Investigate the effects in the different levels of the pipeline
  - Could enable to improve the definition of countermeasures

- Test of those countermeasures with **side-channel analysis**
  - Do they introduce vulnerabilities ?
  - How to combine those countermeasures with countermeasures against side-channel analysis ?

- Automatic application of the countermeausres **by the compiler**
  - Generate reinforced code for some specified functions

DIFFUSION OF THE RESULTS OF THIS THESIS

- **4 articles in conferences with proceedings**
  - **COSADE** 2013
  - **FDTC** 2013
  - **IEEE HOST** 2014
  - **IFIP/IEEE VLSI-SoC** 2014

- **1 article in a workshop without proceedings**
  - **PROOFS** 2013
- **1 article in a peer-reviewed journal**
  - **Journal of Cryptographic Engineering** 2014
- **3 communications in workshops without proceedings**
  - **Crypto'Puces** 2013
  - **Chip-To-Cloud Security Forum** 2013
  - **TRUDEVICE Workshop** 2014

THANK YOU FOR YOUR ATTENTION

Any questions ?

Download the presentation

PDF file