DE LA RECHERCHE À L'INDUSTRIE

cea

UPMC SORBONNE UNIVERSITÉS

LIP6

MINES Saint-Étienne

SÉCURISATION DE PROGRAMMES ASSEMBLEUR FACE AUX ATTAQUES VISANT LES PROCESSEURS EMBARQUÉS

Nicolas MORO (CEA)

Thèse encadrée par **Karine HEYDEMANN** (LIP6)
Sous la direction d'**Emmanuelle ENCRENAZ** (LIP6)
et de **Bruno ROBISSON** (CEA)

Des parties du travail présenté ont été effectuées en collaboration avec Amine Dehbaoui

13 NOVEMBRE 2014 – PARIS

LES SYSTÈMES EMBARQUÉS

Les systèmes embarqués :

- Sont des **systèmes électroniques autonomes**
- Sont **très largement utilisés** pour de nombreux besoins

SÉCURITÉ DES SYSTÈMES EMBARQUÉS

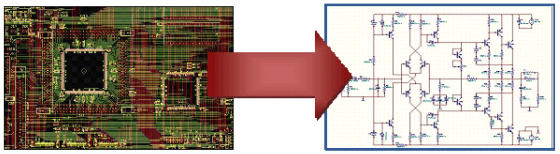
- Ces systèmes embarqués **peuvent être attaqués**

- Ces attaques **visent généralement à :**

Obtenir des données sensibles

Contourner une protection

Faire de la rétro-ingéniérie



La sécurité des systèmes est très importante pour :

Les administrations et gouvernements

Documents d'identité numériques

Les industriels de la carte à puce

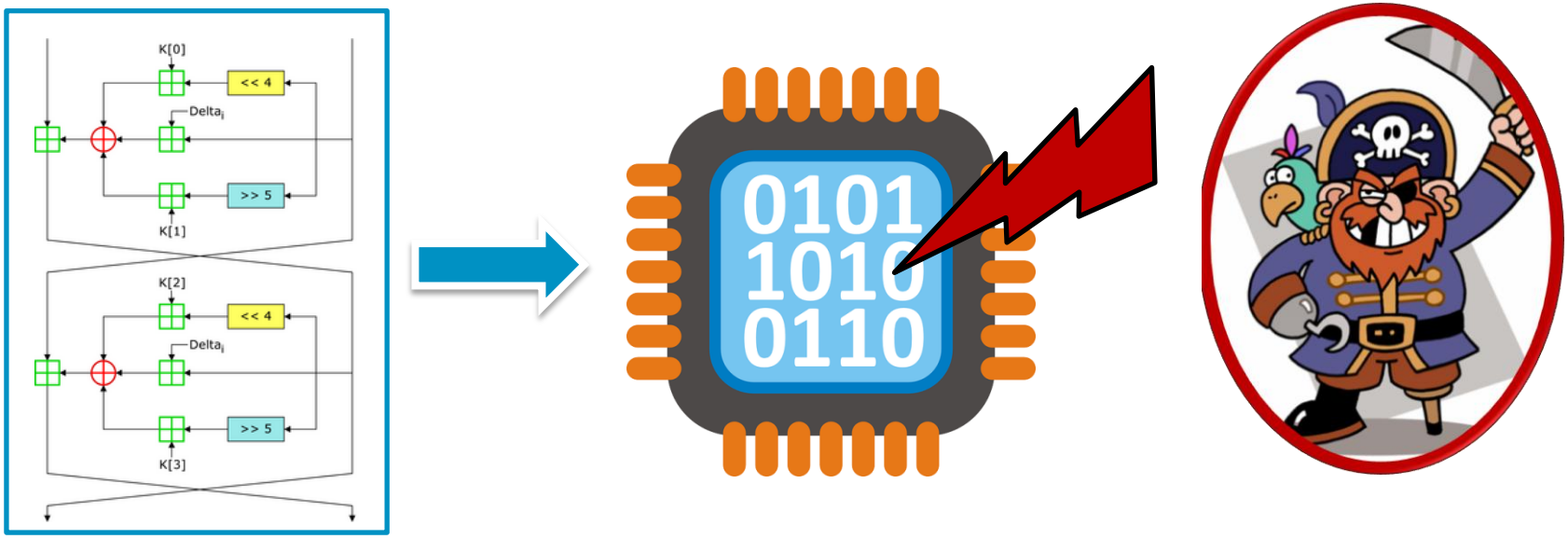
Télévision à péage, cartes bancaires, badges d'accès, …


Les fabricants de produits grand public

Systèmes verrouillés, contenant des systèmes de paiement, …

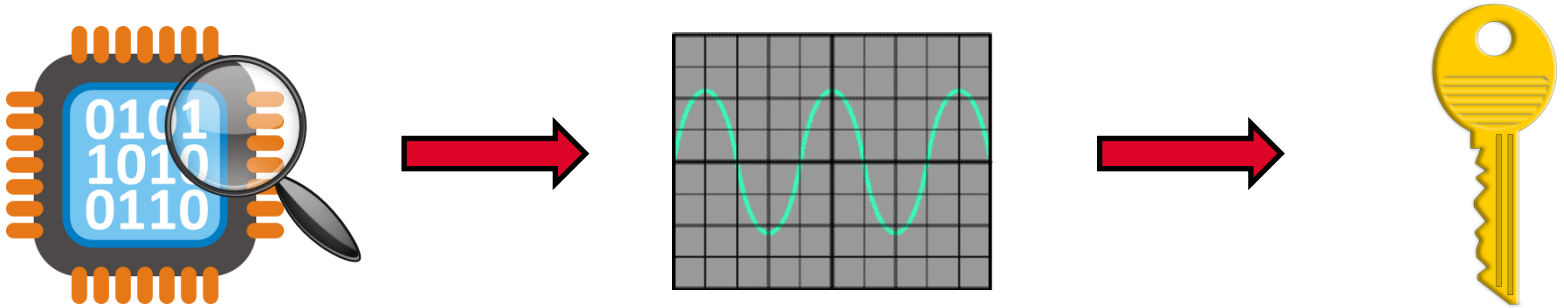
Les **attaques physiques**

- requièrent un **accès au composant**
- visent à exploiter les **vulnérabilités des circuits intégrés**
- sont une **menace sérieuse** pour les systèmes embarqués

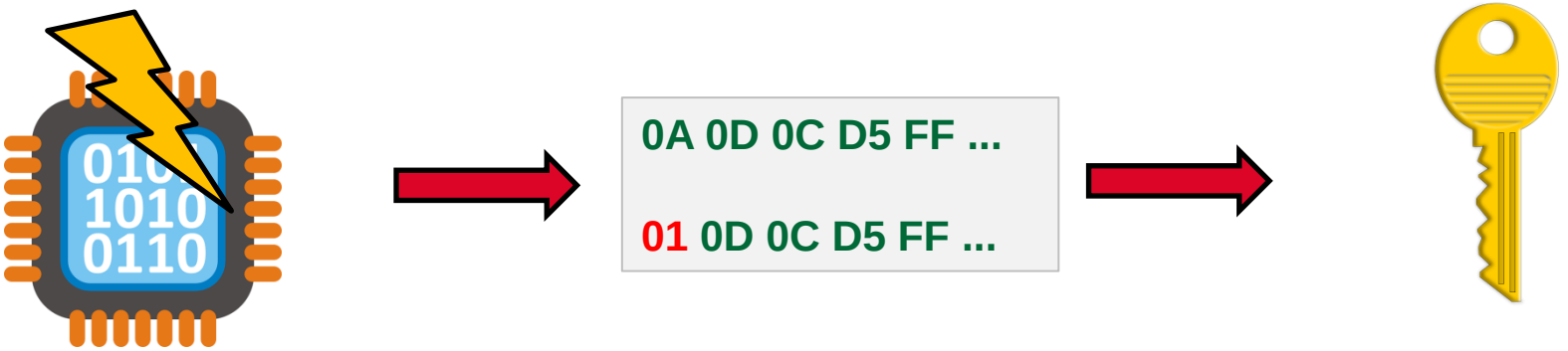


DEUX CATÉGORIES D'ATTAQUES PHYSIQUES

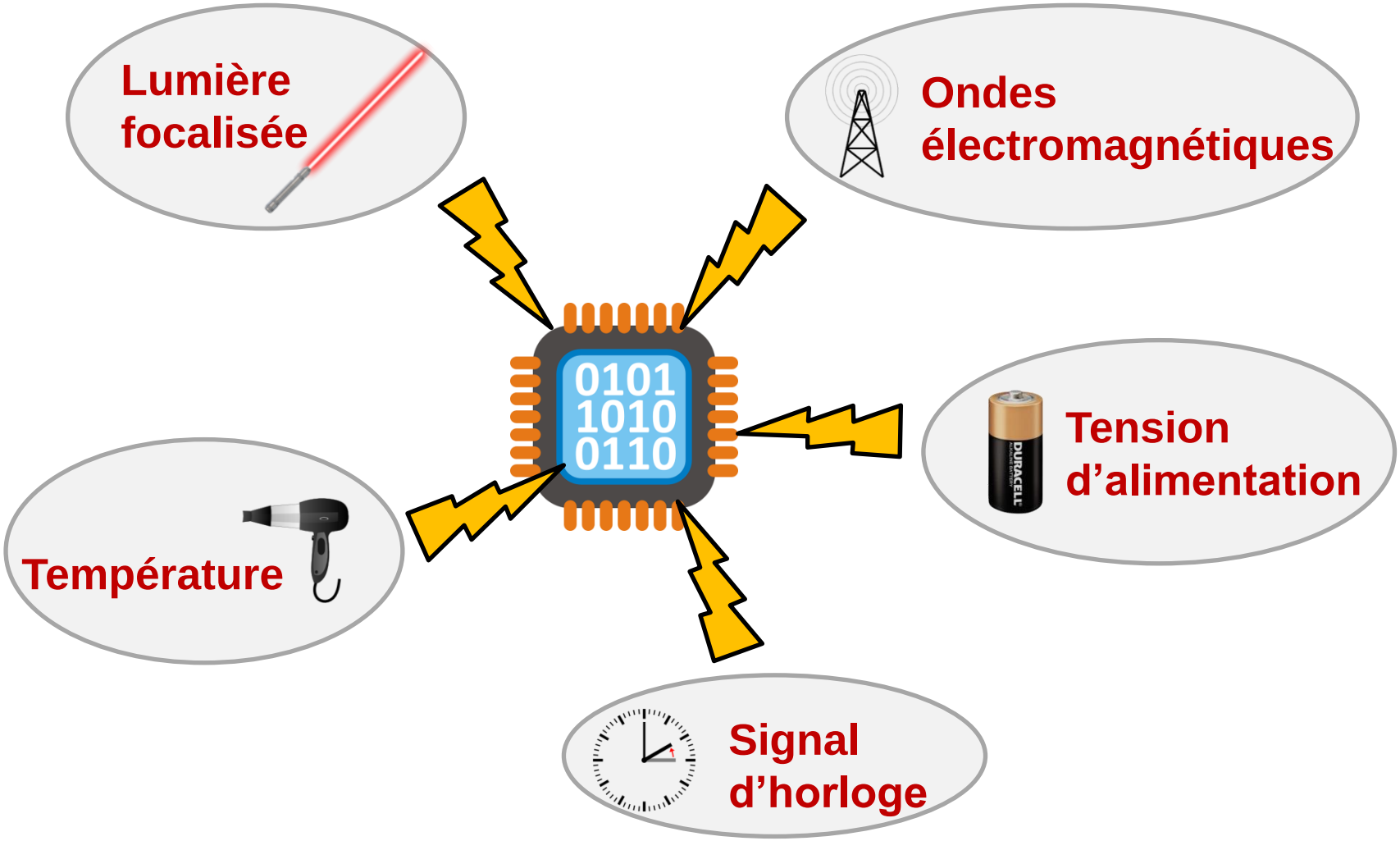
- Les **attaques par observation**



- Les **attaques par injection de fautes**



DIFFÉRENTS MOYENS D'INJECTION DE FAUTES



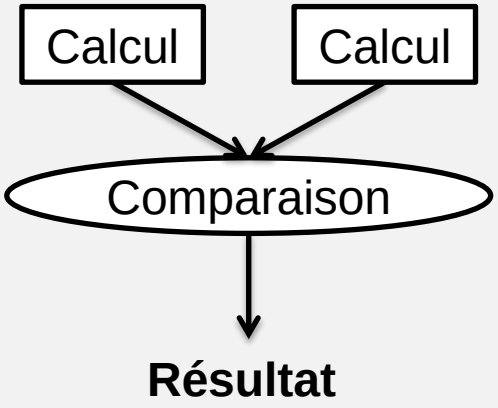
Des contre-mesures à plusieurs niveaux :

1 – Capteurs physiques

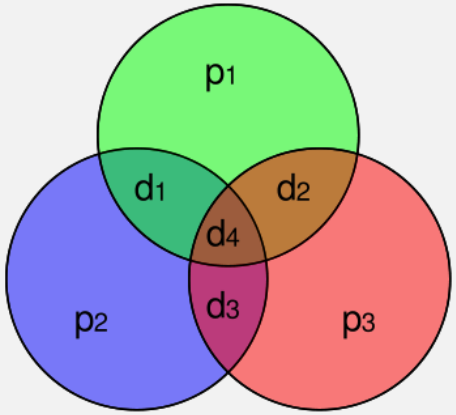
Détecteurs de lumière, de modifications de la tension, …

2 – Mécanismes de détection ou tolérance aux fautes

Redondance



Bits de parité et codes détecteurs d'erreur



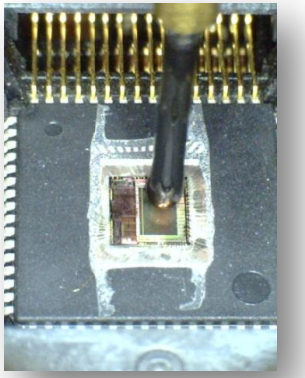
Propriétés mathématiques des algorithmes

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

```
1  begin
2      Generate a random prime r.
3      S_p' ← m^(d mod φ(p·r)) mod p·r.
4      S_q' ← m^(d mod φ(q·r)) mod q·r.
5      if S_p' ≢ S_q' mod r then
6          Return error.
7      end
8      S_p ← S_p' mod p and S_q ← S_q' mod q.
9      Recombine S_p and S_q as explained previously.
10     Return S.
11 end
```

INJECTION ÉLECTROMAGNÉTIQUE PAR IMPULSION

- Technique **relativement récente**
  (théorique en 2002, en pratique depuis 2007)
- Envoi d'une impulsion électrique à travers une **antenne d'injection**
- Couplage électromagnétique avec les **rails d'alimentation** des circuits
- Effet **semi-local**
- **Facilité de réalisation**



Permet de **contourner des contre-mesures existantes**

Markettos, 2011 – Dehbaoui, 2012 – Zussa, 2014

OBJECTIF DE LA THÈSE

- De nouvelles attaques par injection EM ont été réalisées
  ➔ **nouvelles contre-mesures** nécessaires

Contre-mesures **matérielles**

- Lourds changements
- Réservées aux concepteurs
- Besoin d'un circuit fini pour le test

Contre-mesures **logicielles**

- Changements plus souples
- Applicables sur des processeurs
- Plus facilement testables

- Difficulté de modéliser l'effet d'une perturbation EM sur l'exécution d'un programme ➔ **niveau assembleur**

Objectif de la thèse :

Proposer des **contre-mesures logicielles** face aux **attaques par injection électromagnétique**

APPROCHE CHOISIE POUR CETTE THÈSE

Définition d'un modèle de faute

Etude des effets d'une injection de fautes sur un programme assembleur

Définition d'une contre-mesure

Résistante face aux fautes du modèle et vérifiée formellement

Vérification expérimentale de son efficacité

Tests expérimentaux sur des instructions isolées et des codes complexes

APPROCHE CHOISIE POUR CETTE THÈSE

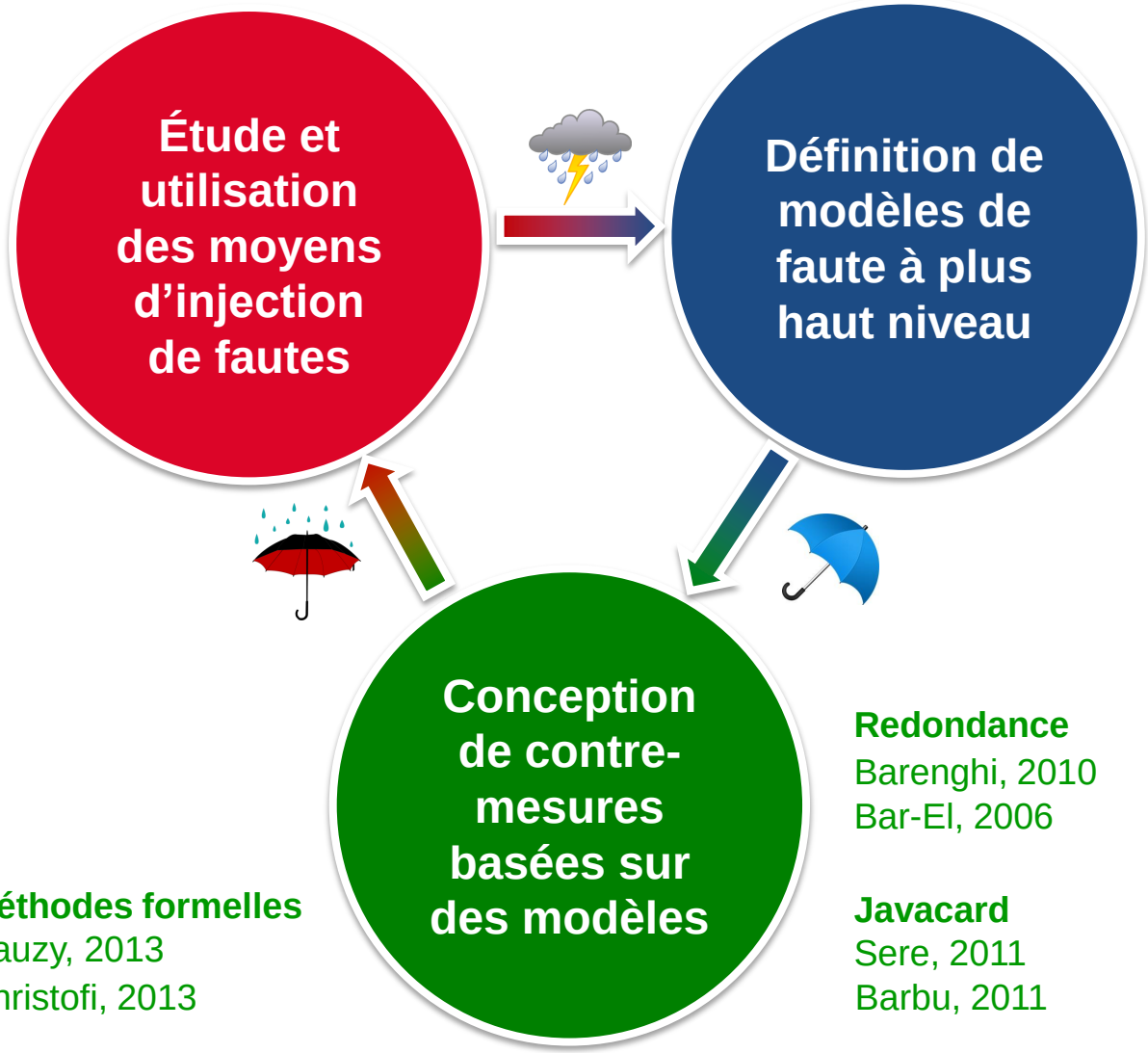
Laser
Roscian, 2013

Ondes EM
Dehbaoui, 2012

Glitch tension
Zussa, 2014

Température
Schmidt, 2014

Glitch horloge
Balasch, 2011



Bit flip
Agoyan, 2010

Saut d'instruction
Barenghi, 2012

Corruption d'instruction
Balasch, 2011
Trichina, 2010

Branchements
Berthomé, 2012

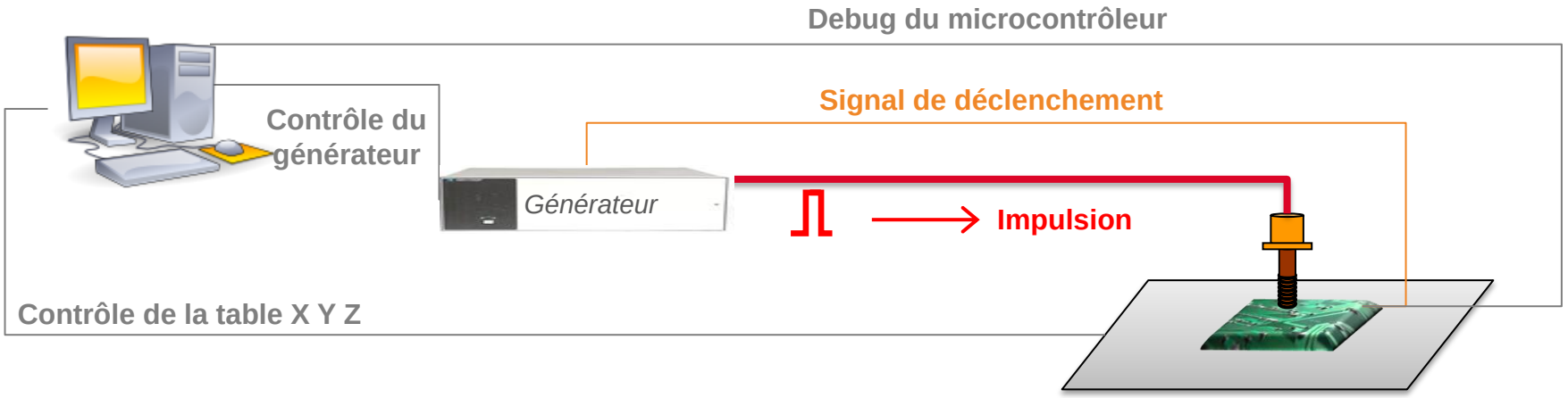
Redondance
Barenghi, 2010
Bar-El, 2006

Javacard
Sere, 2011
Barbu, 2011

Méthodes formelles
Rauzy, 2013
Christofi, 2013

SOMMAIRE

MONTAGE EXPÉRIMENTAL D'INJECTION



1. L'expérimentation est **pilotée depuis le PC**
2. Le code ciblé est **exécuté sur le microcontrôleur**
3. Le microcontrôleur **envoie un signal de déclenchement**
4. Le générateur **envoie une impulsion de tension**
5. Le microcontrôleur est arrêté
6. Les données internes sont récupérées

LE PROCESSEUR ARM CORTEX-M3

- Architecture ARM **majoritaire dans les systèmes embarqués**
- Plusieurs processeurs sécurisés basés sur un ARM Cortex-M
- L'architecture Cortex-M3 est déjà utilisée pour des **cartes à puce** ou des **processeurs de communications RFID**

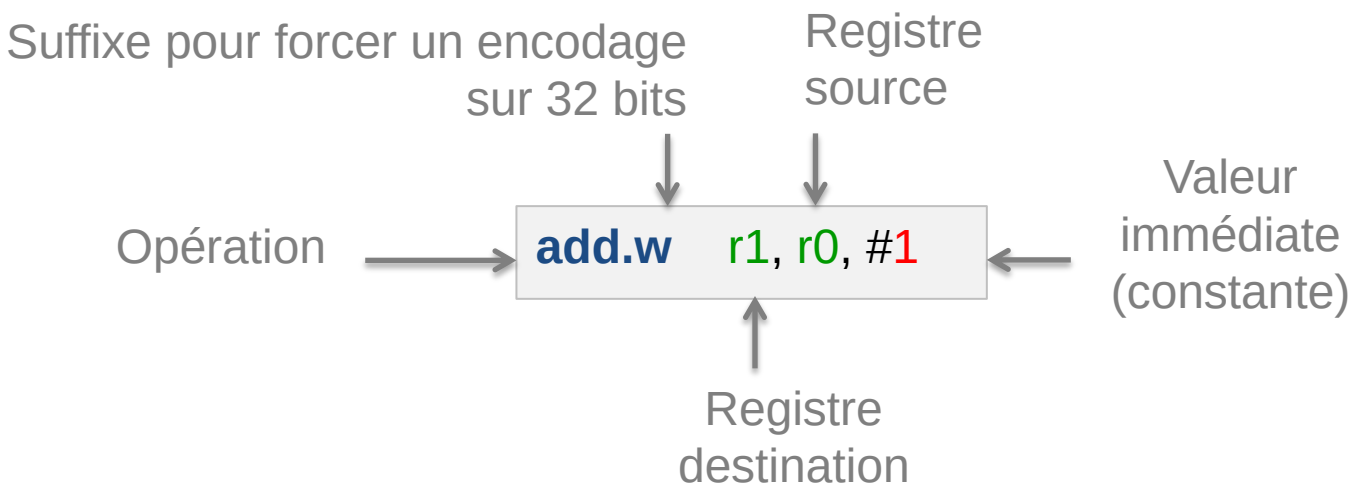
- Fréquence de **56 MHz**, période 17.8 ns
- Architecture ARMv7-M 32 bits type Harvard

Cortex
Low-Power Leadership from ARM

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

Jeu d'instructions Thumb-2

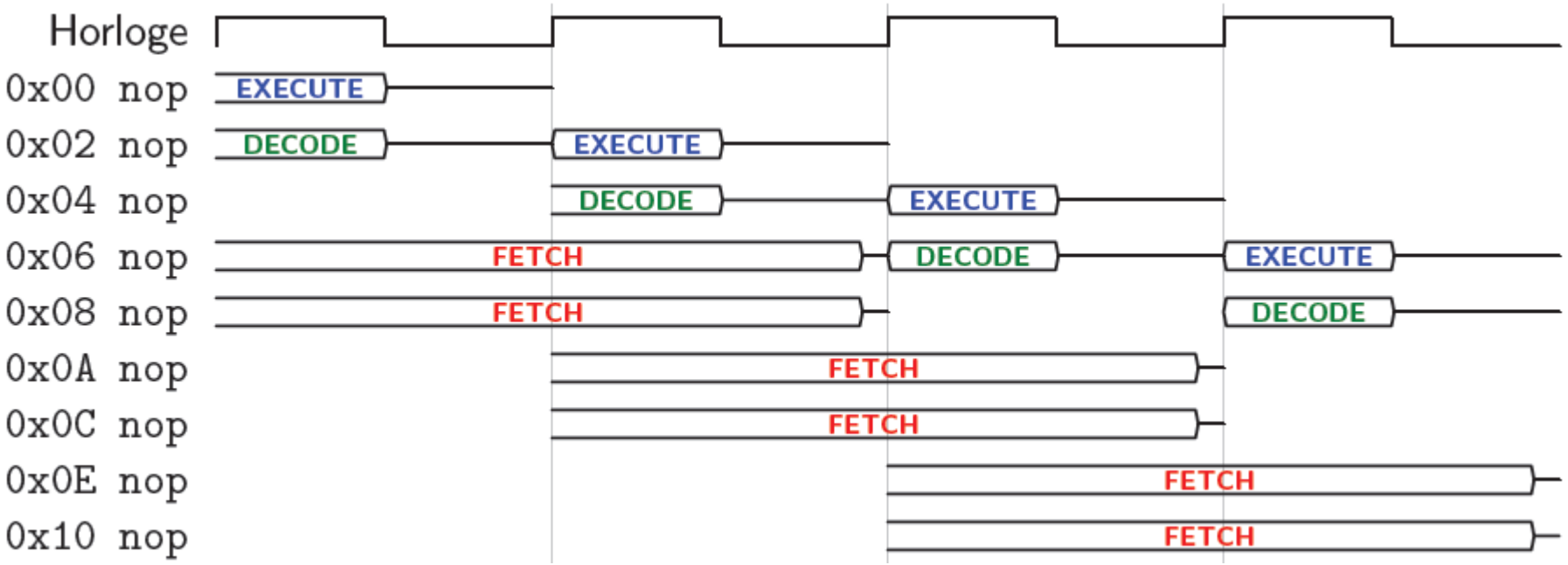
- RISC, 151 instructions sur 16 et 32 bits
- Architecture load/store : opérations effectuées sur des registres



PIPELINE ET EXÉCUTION DES INSTRUCTIONS

3 niveaux de pipeline (Fetch – Decode – Execute), pas de prefetch

Fetch	Chargement de l'instruction dans le registre d'instruction
Decode	Décodage, chargement des opérandes, détection des branchements
Execute	Exécution de l'instruction, écriture des résultats

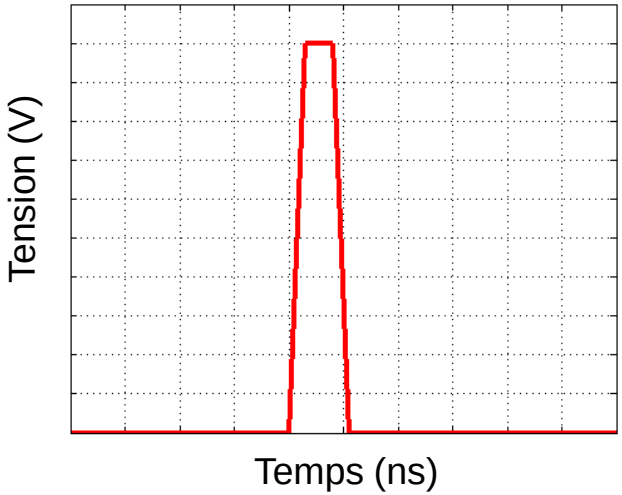


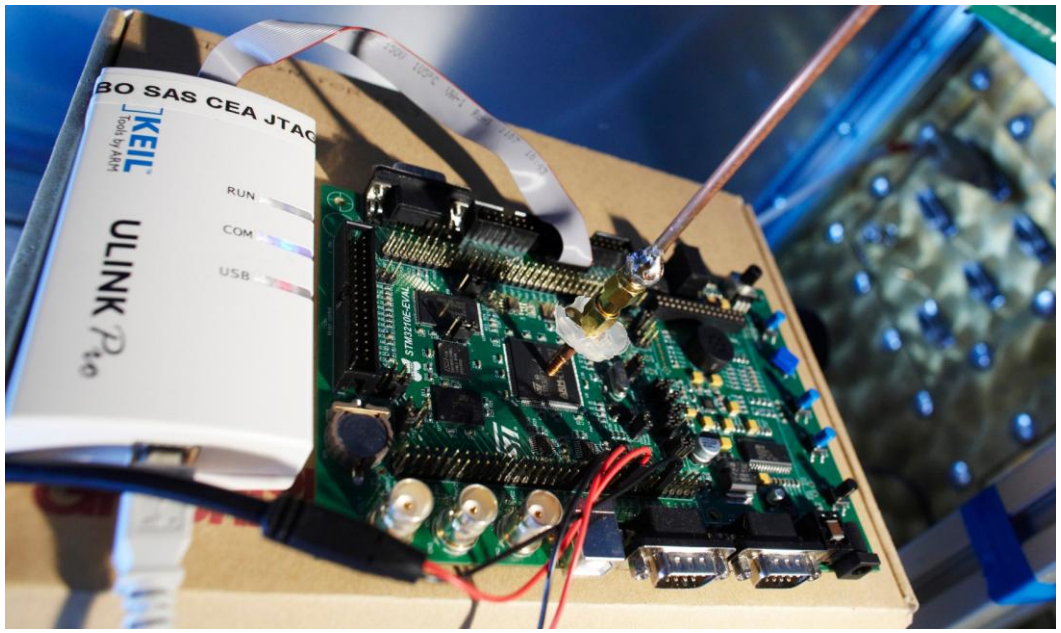
INJECTION PAR IMPULSION ÉLECTROMAGNÉTIQUE

- L'antenne d'injection est un **solénoïde en cuivre**

- Sonde JTAG **Keil ULINKpro**
  Permet d'utiliser le microcontrôleur en mode debug

- Générateur d'**impulsions**
  Forte tension, fort courant





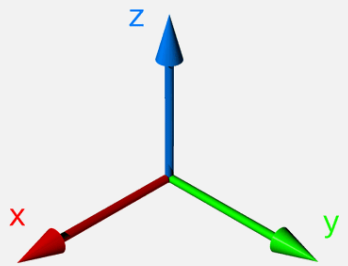
SOMMAIRE

DÉFINITION D'UN MODÈLE DE FAUTES

- Permet de mieux connaître les **capacités d'un attaquant**
- Un grand nombre de **paramètres expérimentaux**
- Leur influence sur les fautes obtenues **doit être étudiée**

Les paramètres étudiés

Position de l'antenne



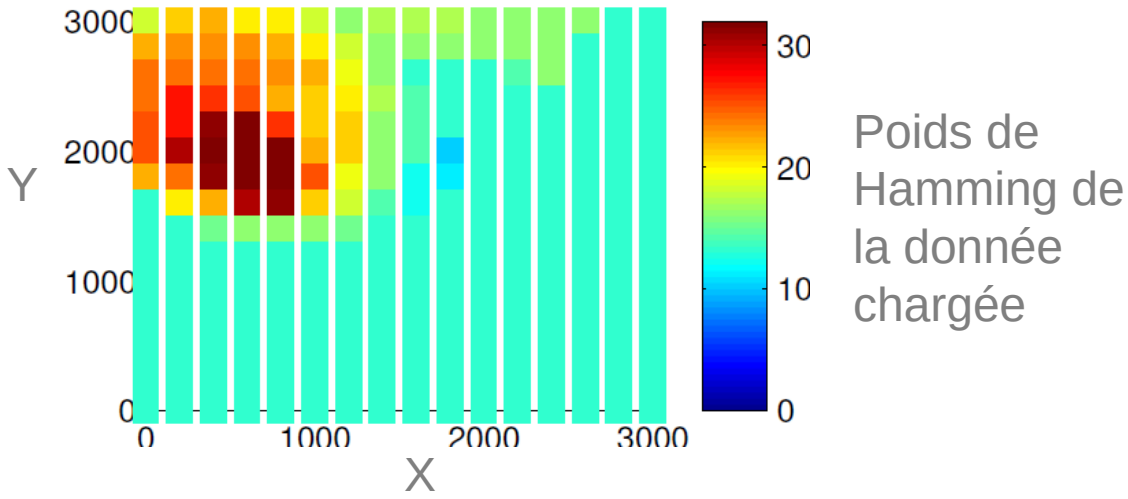
Instant d'injection

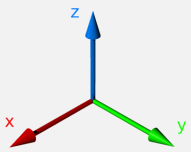
Tension de l'impulsion

INFLUENCE DE LA POSITION D'ANTENNE

`ldr r8,=0x12345678` ➔ charge un mot de 32 bits depuis la mémoire Flash

- **Variation de X et Y** sur un carré de 3mm de côté
- Tension fixée, instant d'injection fixé, position en Z fixée

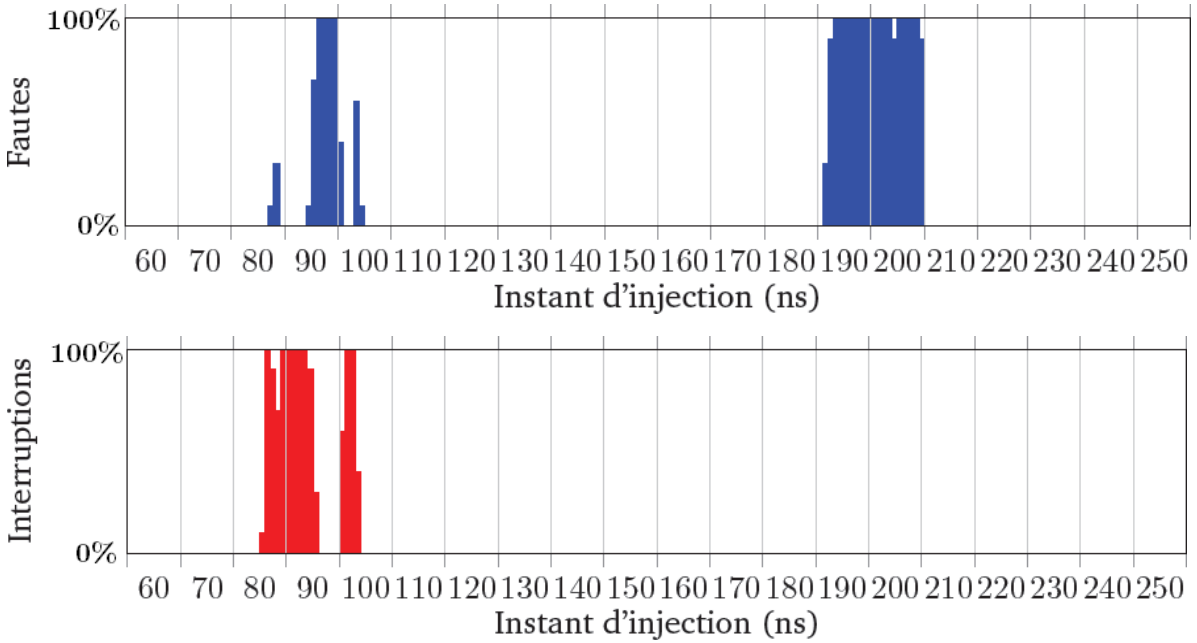




- Un **effet localisé** de la technique d'injection
- Une **zone très réduite** pour injecter des fautes

INFLUENCE DE L'INSTANT D'INJECTION

`ldr r8,=0x12345678` ➔ charge un mot de 32 bits depuis la mémoire Flash



- **Variation de l'instant d'injection**
- Tension fixée
- Antenne fixée

- Deux **intervalles temporels distincts**
- Différents **types de fautes**
- Pour certains instants, près de **100% de fautes**

INFLUENCE DE LA TENSION DE L'IMPULSION

`ldr r4,=0x12345678` ➔ charge un mot de 32 bits depuis la mémoire Flash

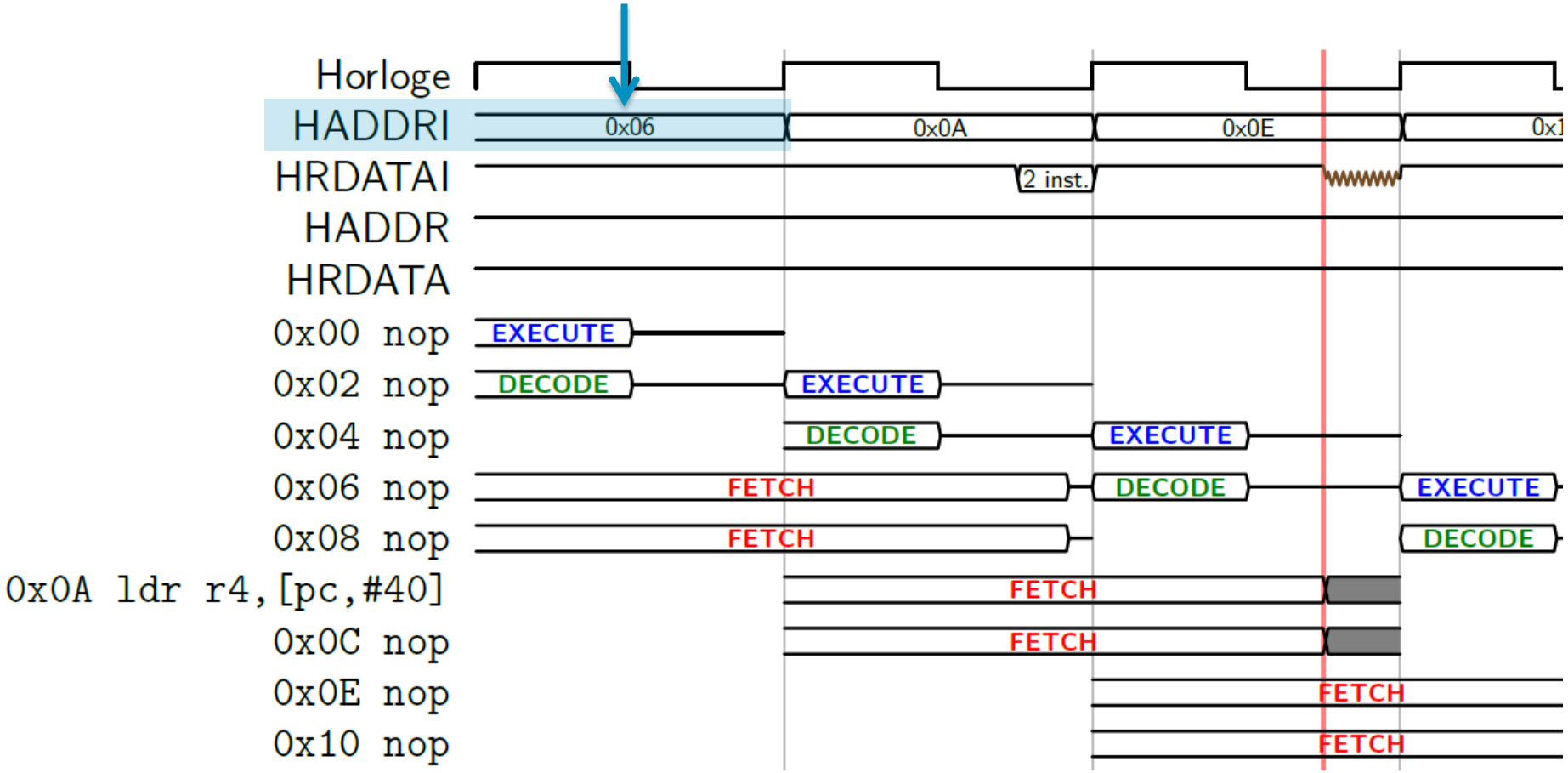
- **Variation de la tension d'injection**
- Position d'antenne fixée, instant d'injection fixé

Tension	Valeur de sortie
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**

- Un **effet de mise à 1 des bits**
- L'effet est lié à **l'augmentation de la tension**
- Obtenu seulement lors d'un **chargement depuis la Flash**

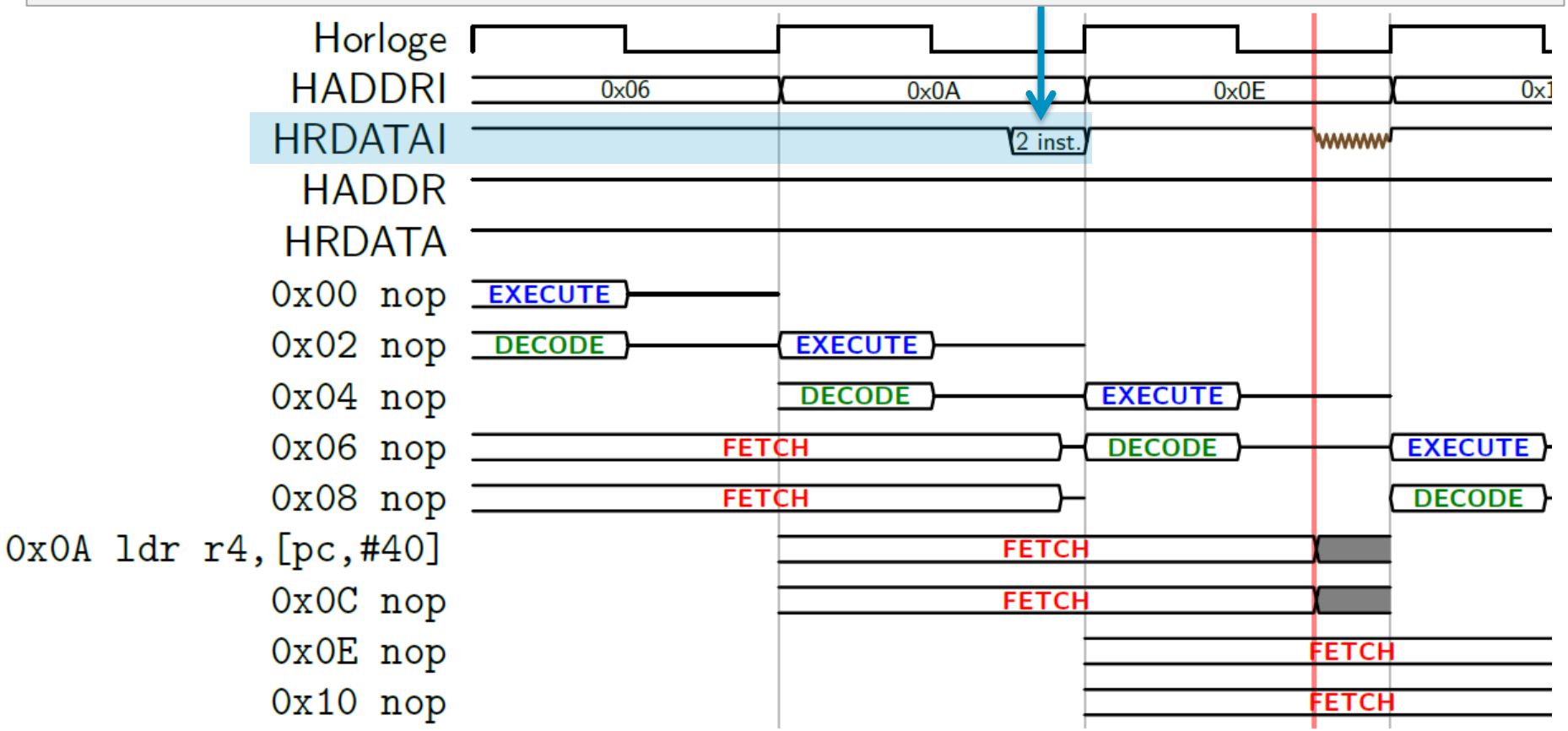
Passage d'une instruction sur le bus HRDATAI

1 – Envoi de l'adresse de l'instruction sur le bus d'adresse HADDRI



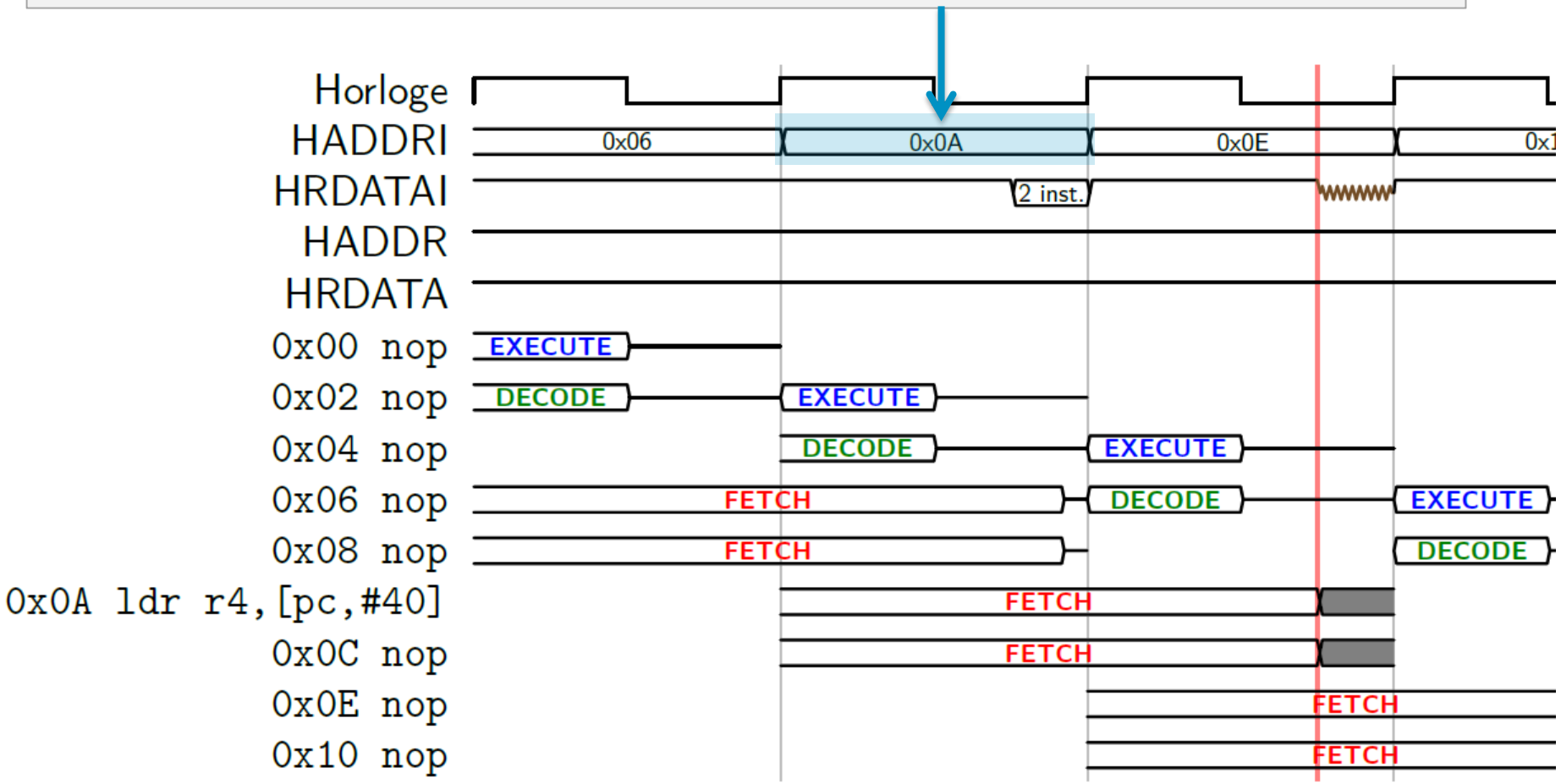
Passage d'une instruction sur le bus HRDATAI

2a – Envoi du code binaire de l'instruction sur le bus d'instructions HRDATAI
(fin du cycle d'horloge suivant)



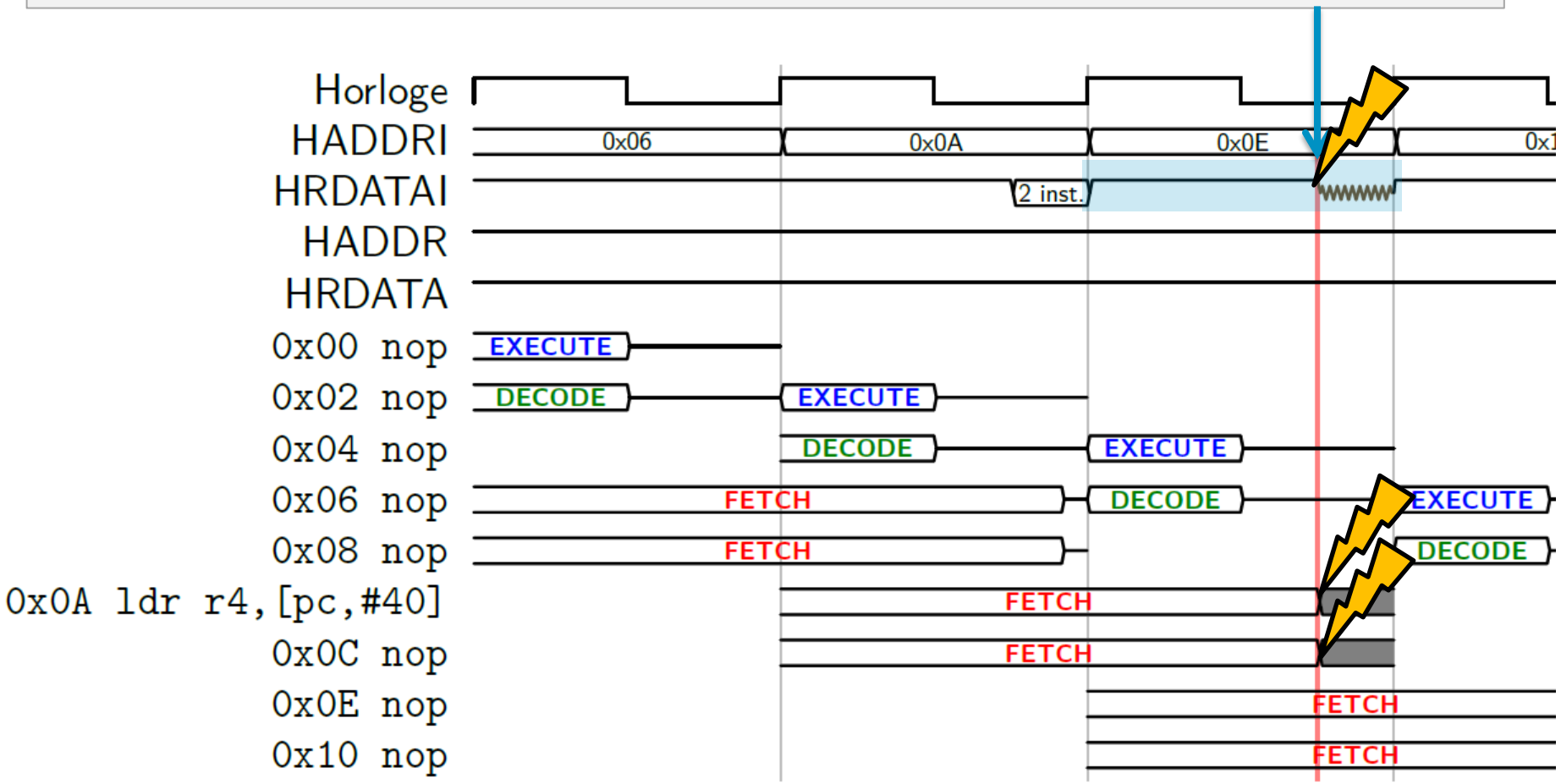
Passage d'une instruction sur le bus HRDATAI

2b – Envoi de l'adresse de l'instruction sur le bus d'adresse HADDRI



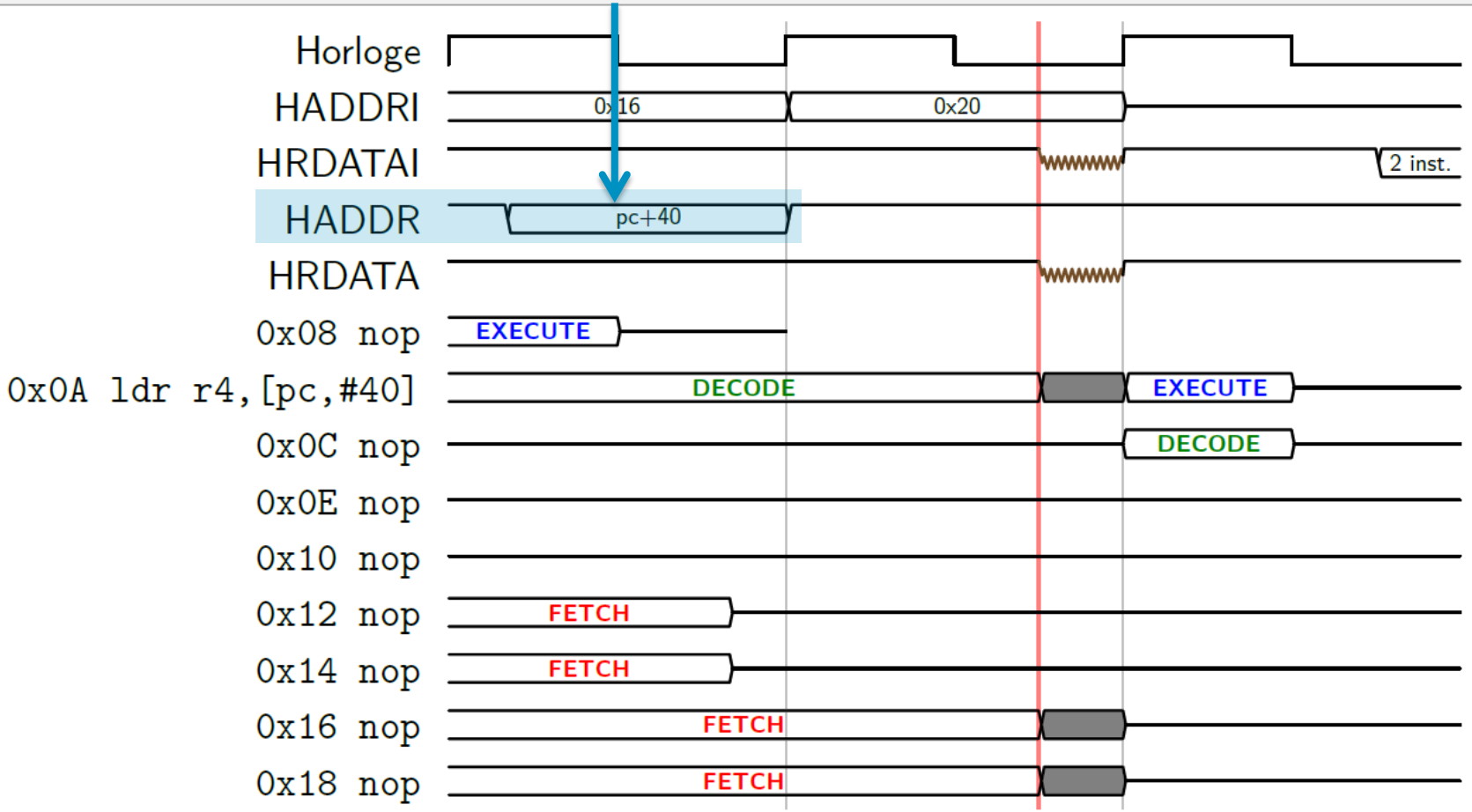
Passage d'une instruction sur le bus HRDATAI

3 – Corruption du transfert du code de l'instruction sur le bus HRDATAI



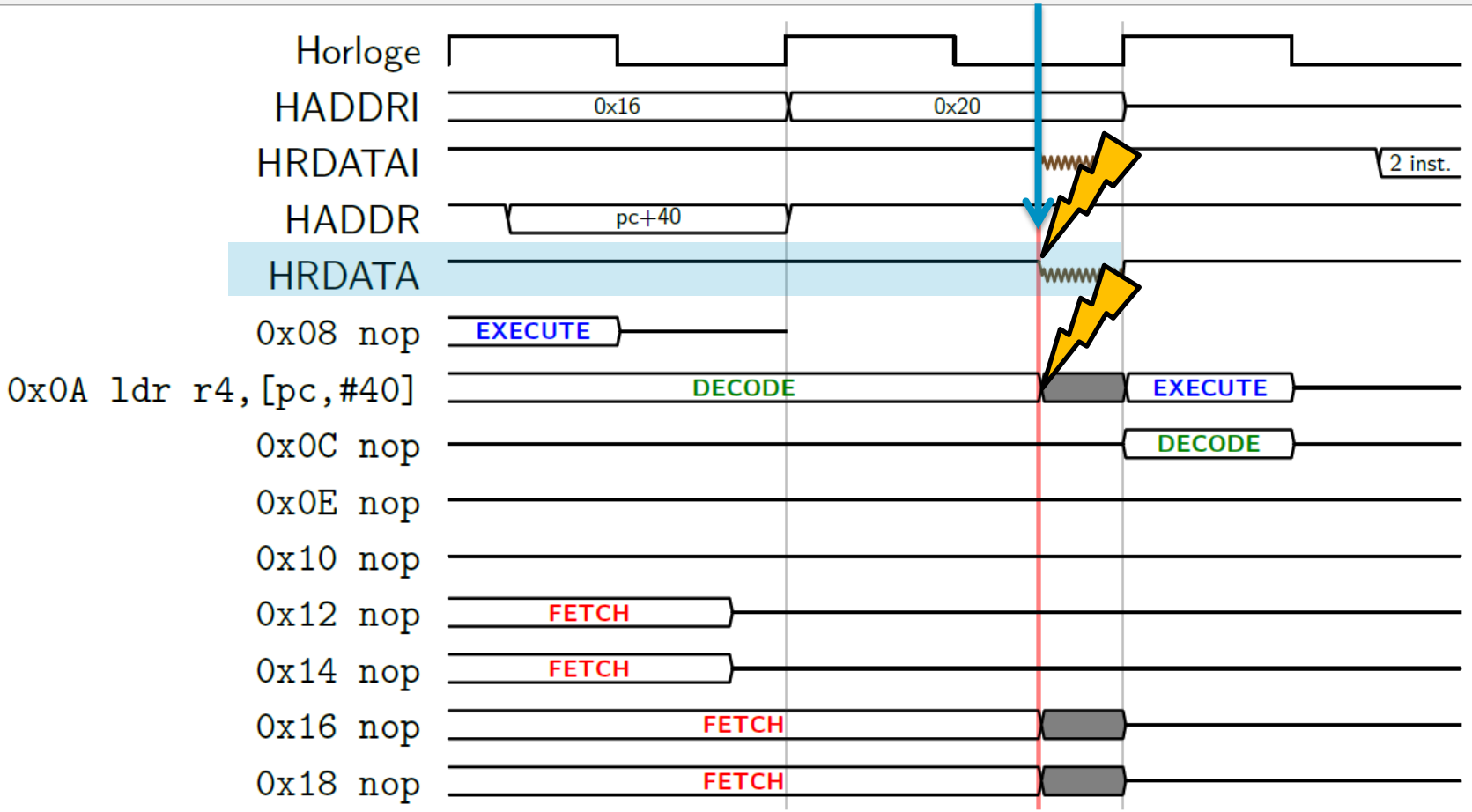
Passage d'une donnée sur le bus HRDATA

1 – Envoi de l'adresse de la donnée sur le bus d'adresse HADDR



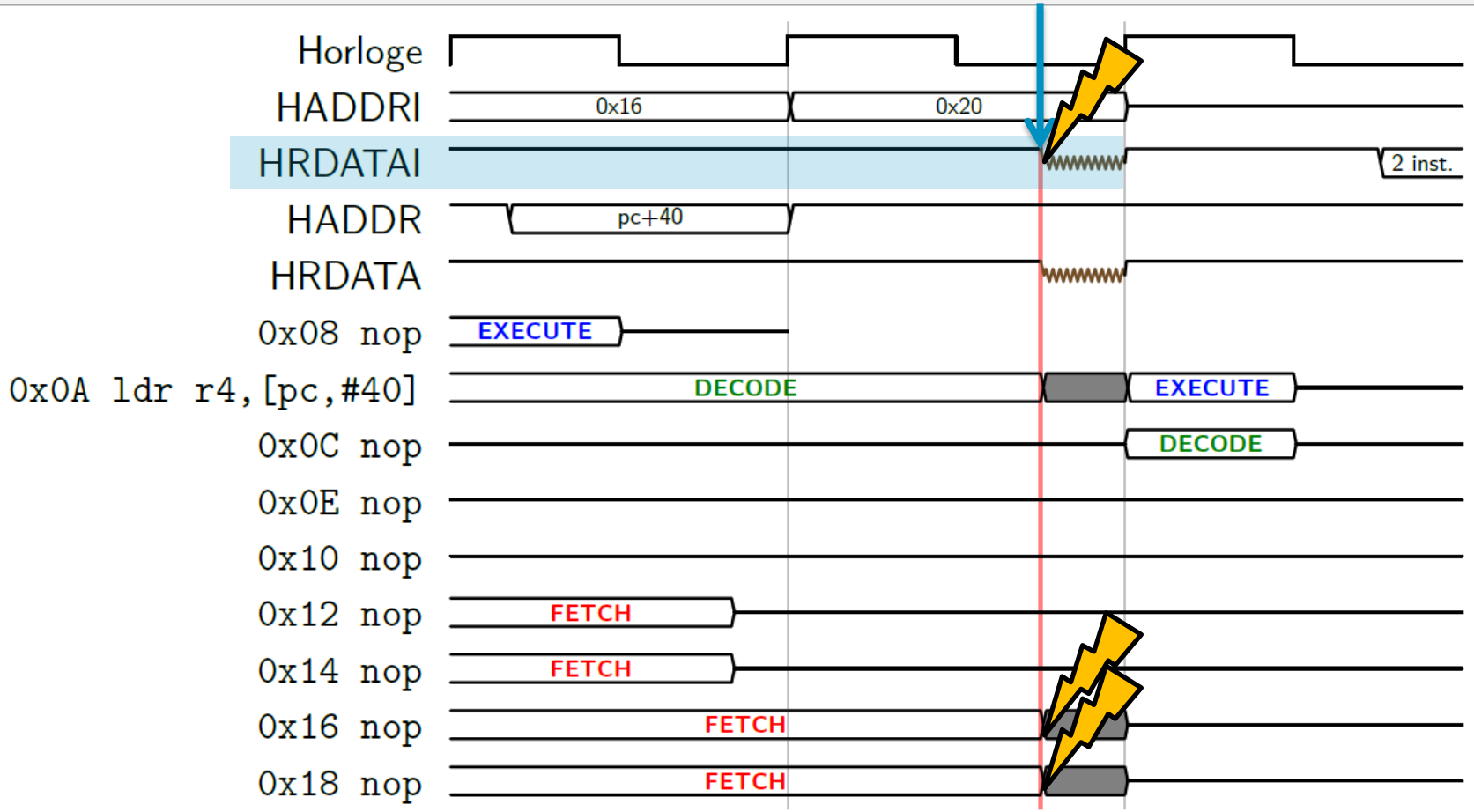
Passage d'une donnée sur le bus HRDATA

2a – Corruption du transfert de la donnée sur le bus de données HRDATA



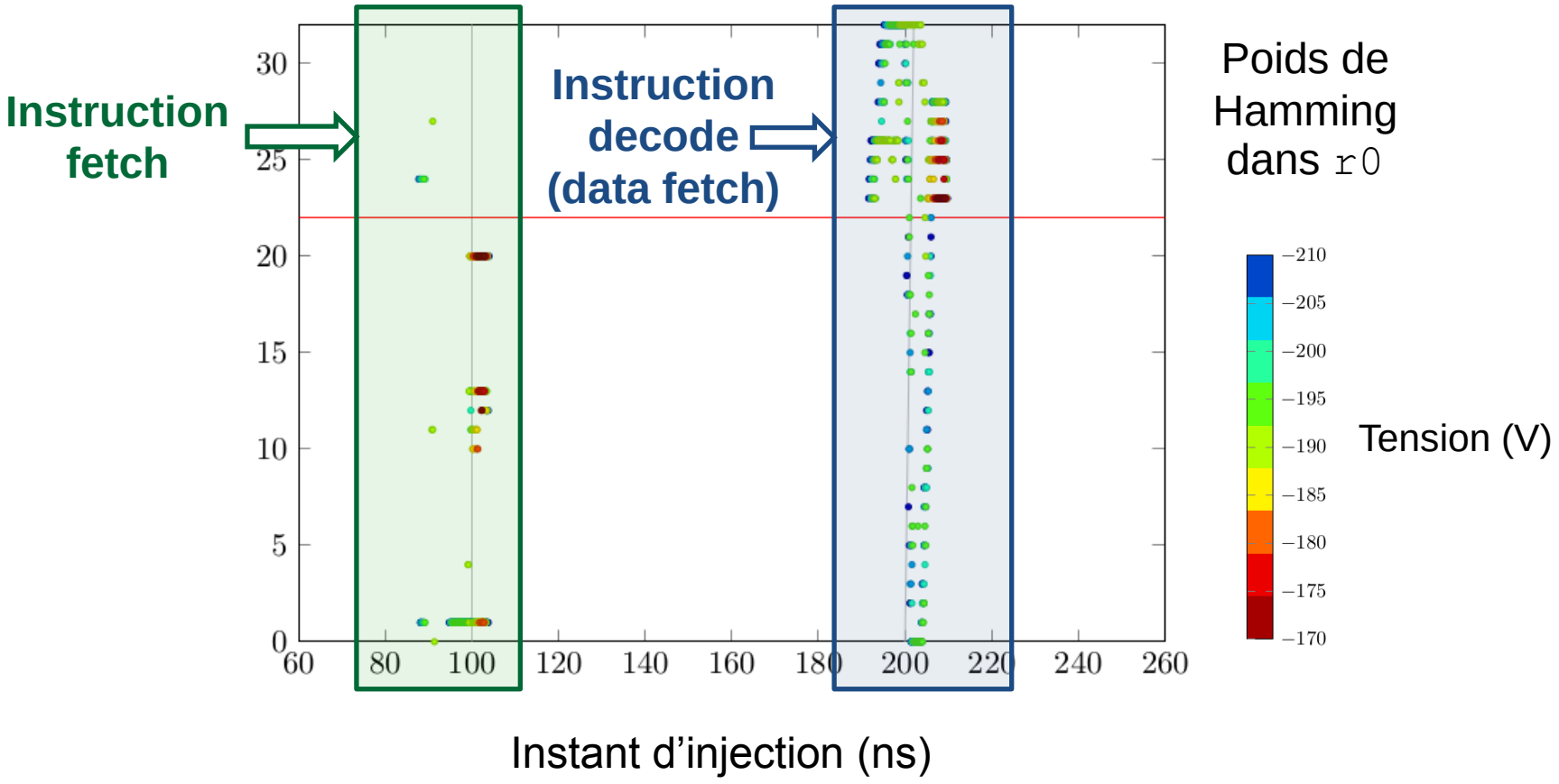
Passage d'une donnée sur le bus HRDATA

2b – Corruption du transfert d'autres instructions sur le bus HRDATAI



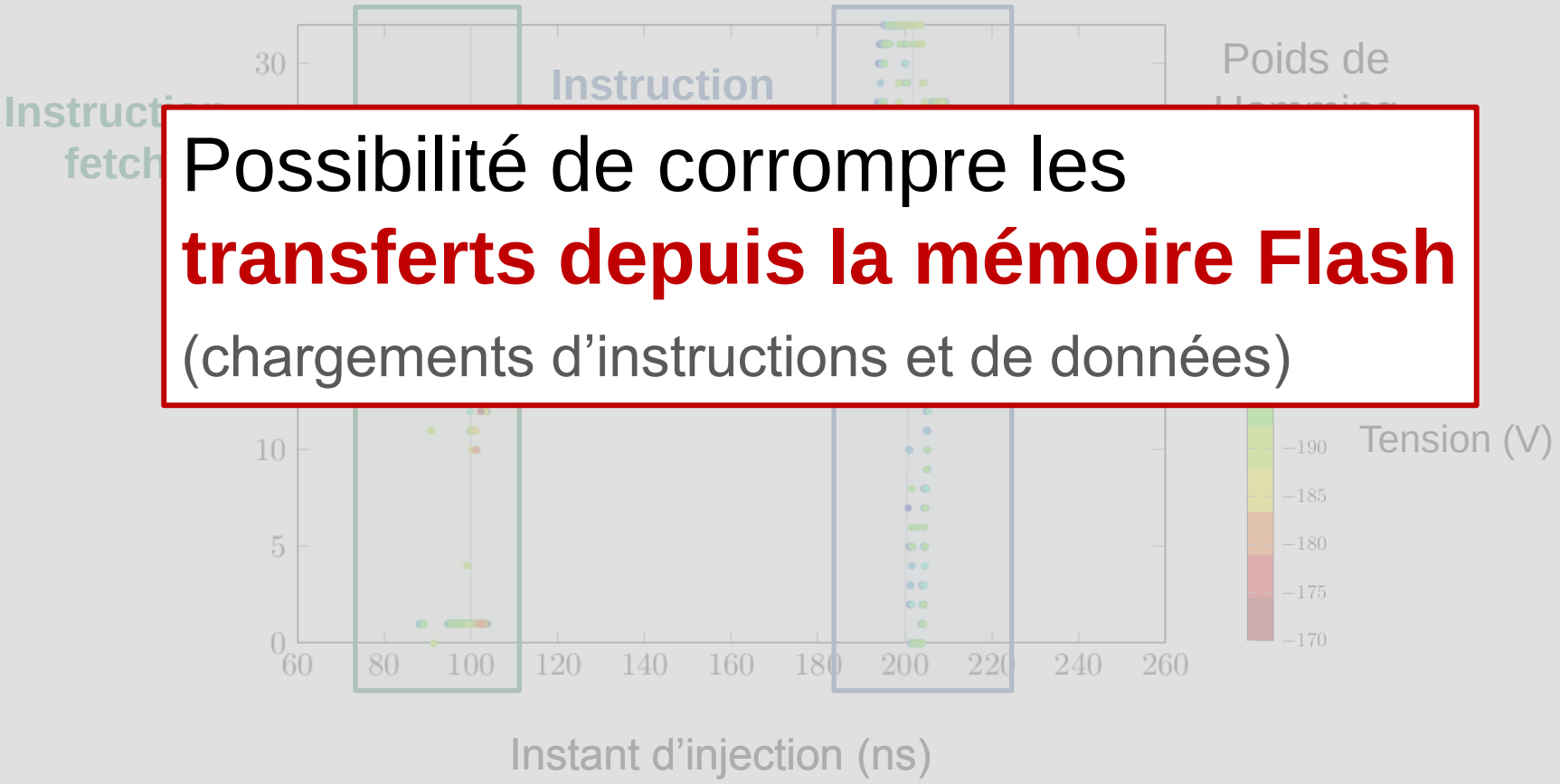
VALIDATION EXPÉRIMENTALE DE CE MODÈLE RTL

`ldr r0, [pc,#40]` ➔ charge un mot de 32 bits depuis la mémoire Flash



VALIDATION EXPÉRIMENTALE DE CE MODÈLE RTL

`ldr r0, [pc,#40]` ➔ charge un mot de 32 bits depuis la mémoire Flash



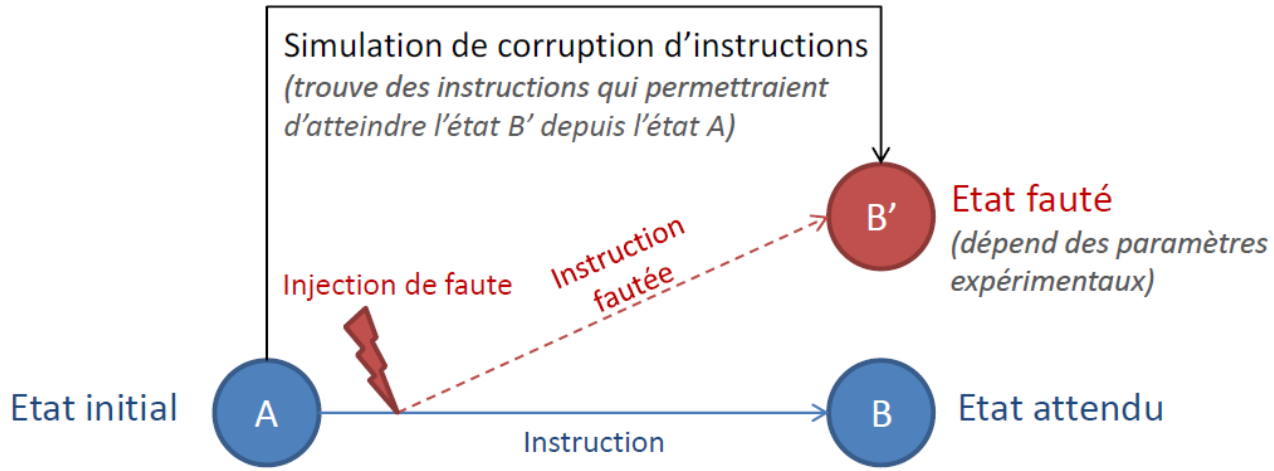
Possibilité de corrompre les
transferts depuis la mémoire Flash
(chargements d'instructions et de données)

FAUTES SUR LE FLOT DE DONNÉES ET LE FLOT DE CONTRÔLE

Comment extraire des résultats précédents un **modèle de fautes** pour les remplacements d'instructions ?

➔ Recherche de remplacements qui peuvent **expliquer les fautes obtenues**

- **Simulation** de corruption d'instructions
- **Comparaison** avec les résultats expérimentaux



BILAN SUR LE MODÈLE DE FAUTES AU NIVEAU RTL

Conséquences au niveau des instructions

- **Remplacement** d'instructions
- **Saut** d'une instruction assembleur dans certains cas

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

Conséquences au niveau des données

- Corruption des `ldr` depuis la mémoire Flash (clé de chiffrement, …)
- Valeurs à poids de Hamming élevé plus faciles à obtenir (sur cette archi.)

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

MODÈLE DE FAUTES AU NIVEAU ASSEMBLEUR

Modèle de nop (saut d'une instruction)
- Peut permettre d'empêcher un appel de sous-fonction
- Assez facile de **détecter des vulnérabilités** sur un programme

Remplacement par un nop statistiquement plus fréquent
- Ecriture dans un registre non-vivant ou une case mémoire non-utilisée
- Ré-exécution d'une instruction précédente idempotente
- Remplacement par une instruction sans effet

Dans quelle proportion les fautes injectées ont-elles un effet **équivalent à un saut d'instruction** (nop) ?

EVALUATION DU TAUX DE COUVERTURE DU MODÈLE DE NOP

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

1 **Expérimentation** sur un programme de somme des éléments d'un tableau à 2 cases

tab[0] = 1 ; tab[1] = 2
Résultat attendu : 3

2 **Simulation** du saut de chaque 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 **Comparaison** des valeurs de sortie avec les résultats expérimentaux

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 DU TAUX DE COUVERTURE DU MODÈLE DE NOP

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

1 **Expérimentation** sur un programme de somme des éléments

Sur le programme testé, environ **25% des fautes réalisées** sont interprétables comme un saut d'une instruction assembleur

2 Simu… de c…

3 **Comparaison** des valeurs de sortie avec les résultats expérimentaux

Résultats expérimentaux

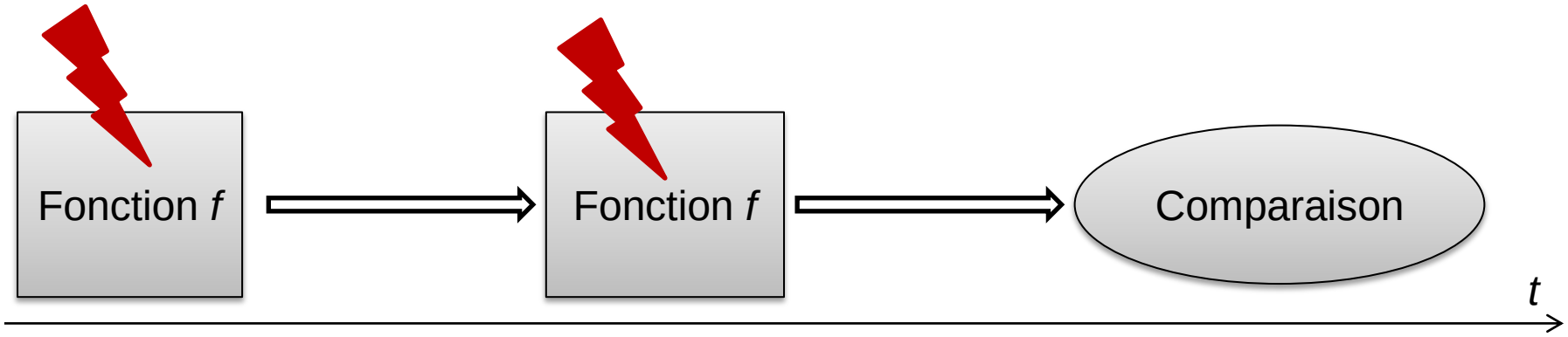
	Interruption	r0	r1	r2	r3	r4	r5
t= 37.0 ns	Aucune	0x2000040C	0x2	0x20000421	0x2 [FAUTE]	0x1 [FAUTE]	0x40010C10
t= 37.2 ns	Aucune	0x2000040C	0x2	0x20000421	0x3	0x2	0x40010C10
t= 37.4 ns	Aucune	0x2000040C	0x2	0x20000421	0x3	0x2	0x40010C10
t= 37.6 ns	UsageFault	-	-	-	-	-	-

SOMMAIRE

CHEMINS D'ATTAQUE PERMIS PAR LE MODÈLE

- Beaucoup de **remplacements d'instruction** ont un effet équivalent à celui d'un **saut d'instruction**

- Des **double fautes** sont possibles si l'écart entre les deux injections est suffisamment important (quelques µs pour notre banc)



Comment **garantir une exécution correcte** avec un potentiel saut d'instruction par un attaquant ?

CONTRE-MESURE CONTRE UN SAUT D'UNE INSTRUCTION

1. **Capable de résister à une injection de fautes**
   - ➔ Basée sur un principe de redondance temporelle

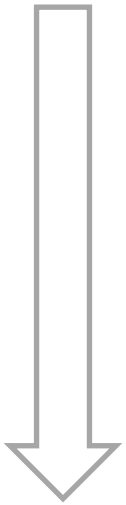
2. **Résistante aux double fautes suffisamment espacées**
   - ➔ Redondance à l'échelle de l'instruction

3. **Applicable automatiquement**
   - ➔ Séquence de remplacement pour chaque instruction
   - ➔ Equivalence par rapport à l'instruction initiale
   - ➔ Principe de renforcement d'un programme complet

DIFFÉRENTES CLASSES D'INSTRUCTIONS

Instructions idempotentes

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



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

Instructions séparables

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

Duplication fausse si pas de faute

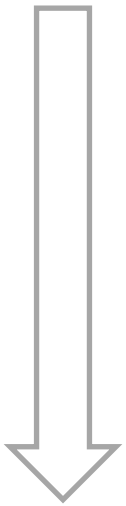
➔ Séparation puis duplication

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

DIFFÉRENTES CLASSES D'INSTRUCTIONS

Instructions idempotentes

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



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

Instructions séparables

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

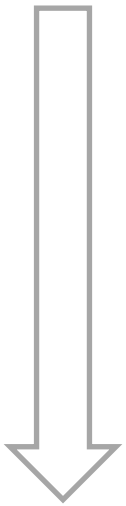
➔ Séparation puis duplication

```
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
```

DIFFÉRENTES CLASSES D'INSTRUCTIONS

Instructions idempotentes

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



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

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

r1:r2 = r3*r4 + r1:r2
Multiplication et addition sur 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
```

INSTRUCTION D'APPEL DE SOUS-FONCTION (BL)

- Les instructions de branchement peuvent être dupliquées **mais pas les appels de sous-fonctions**
  (sinon chaque sous-fonction serait exécutée deux fois)

```
    bl      fonction
```

```
    adr     r12, label_retour
    adr     r12, label_retour
    add     lr, r12, #1
    add     lr, r12, #1
    b       fonction
    b       fonction

label_retour
```

Place l'adresse de retour dans r12

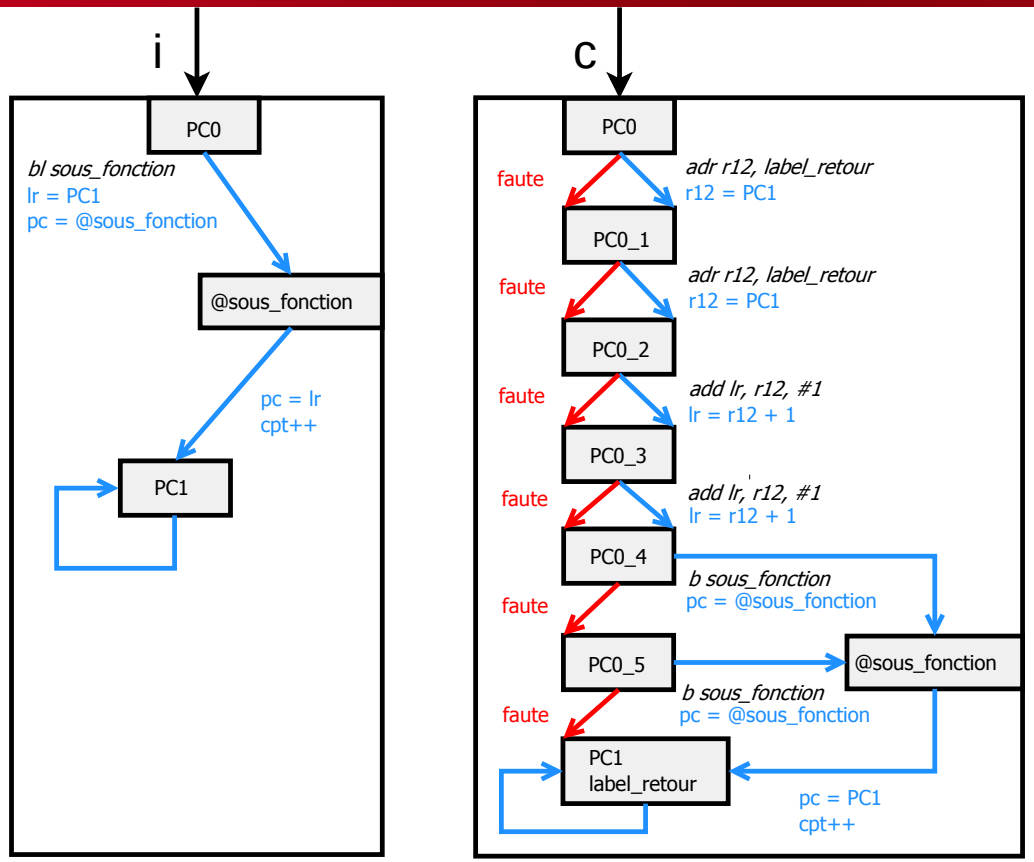
Met à jour le pointeur de retour lr

Branche vers la sous-fonction

Propriétés à vérifier :

1. Chaque séquence de remplacement a la **même sémantique que l'instruction qu'elle remplace**

2. Chaque séquence de remplacement est **tolérante au saut d'une instruction**

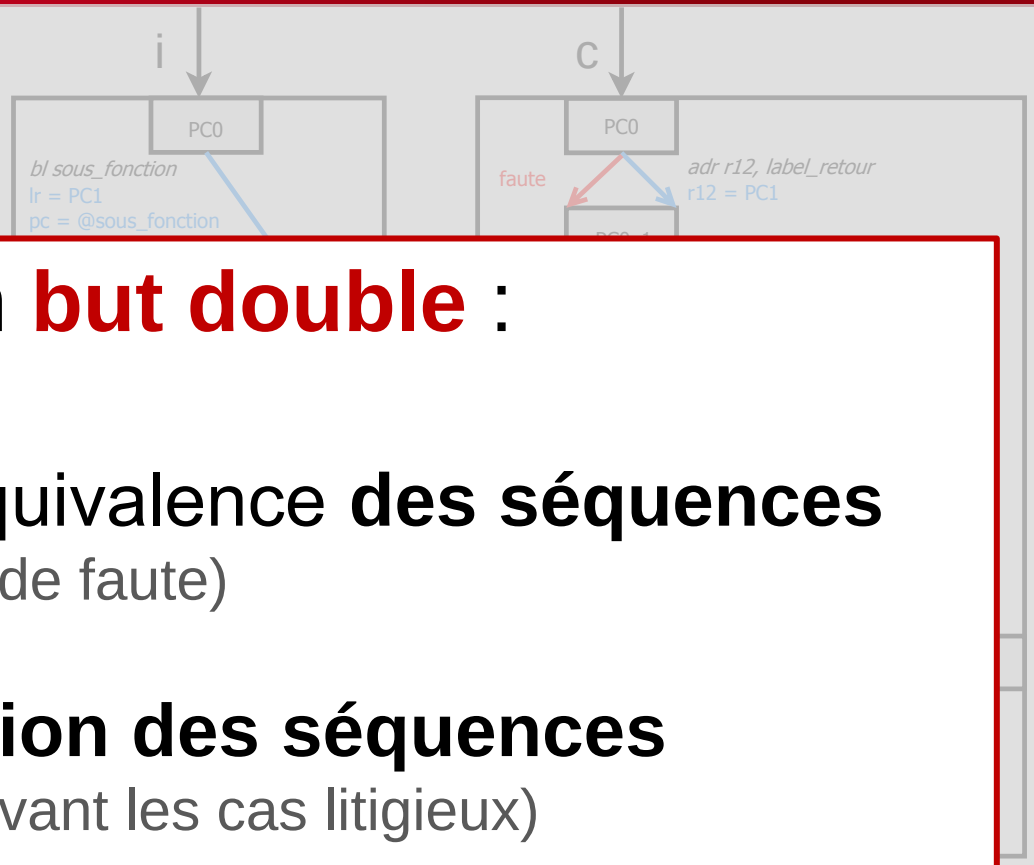


$$AG[((i.pc = pc\_init\_i) \wedge (c.pc = pc\_init\_c)) \Rightarrow$$
$$AF((i.pc = pc\_final\_i) \wedge (c.pc = pc\_final\_c) \wedge \forall x \in D, (i.x = c.x))]$$

VÉRIFICATION DE LA CONTRE-MESURE PAR MODEL-CHECKING

Propriétés à vérifier :

1. Chaque séquence de



La vérification a un **but double** :

1. Vérification de l'équivalence **des séquences**
   (avec et sans injection de faute)
2. Aide à la **conception des séquences**
   (permet de mettre en avant les cas litigieux)

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

APPLICATION AUTOMATIQUE DE LA CONTRE-MESURE

- Un **algorithme d'application automatique** a été conçu

Implémentation	Surcoût en cycles	Surcoût en taille de code
AES	**+ 113.7%**	**+ 202%**
MiBench AES	**+ 186.4%**	**+ 189.9%**
MiBench SHA0	**+ 122.8%**	**+ 178.2%**
AES avec CM sur les deux dernières rondes	**+ 18.6%**	**+ 282.5%**

➔ **Surcoût élevé**,

mais **comparable** à ceux d'approches classiques par redondance

Formal verification of a software countermeasure against instruction skip fault attacks

N. Moro, K. Heydemann, E.Encrenaz, B. Robisson - Journal of Cryptographic Engineering, 2014

SOMMAIRE

EVALUATION EXPÉRIMENTALE DE LA CONTRE-MESURE

- Contre-mesure face à un **modèle simplifié d'attaquant**
  (saut d'une instruction assembleur)

- Ne protège pas face aux fautes sur le **flot de données**

Peut être **complétée** par
- une contre-mesure de **détection**
- qui protège aussi les **chargements de données**

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

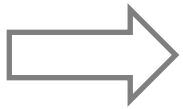
EVALUATION EXPÉRIMENTALE DE LA CONTRE-MESURE

- Détection de **fautes uniques**
  Saut d'instruction, certains remplacements, faute sur les données

- Proposée pour un **ensemble restreint d'instructions**
  Arithmétiques et logiques, load-store
  … mais pas branchements, manipulation de pile ou utilisation de flags

- Surcoût **élevé**
  En registres, taille de code et nombre de cycles

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



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

MÉTHODE D'ÉVALUATION EXPÉRIMENTALE

Etude de l'impact de la contre-mesure pour :

- des **instructions isolées**
- des **codes complexes**

Instructions isolées choisies :

Contre-mesure de tolérance

➢ **Instruction `bl`**

```
adr     r12, label_retour
adr     r12, label_retour
add     lr, r12, #1
add     lr, r12, #1
b       fonction
b       fonction

label_retour
```

Contre-mesure de détection

➢ **Instruction `ldr`**

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

ÉVALUATION POUR DES INSTRUCTIONS ISOLÉES

Pour les deux contre-mesures :

- Il est nécessaire de **forcer un encodage sur 32 bits**

```
adr     r12, label_retour
adr     r12, label_retour
```

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

Pour la contre-mesure de tolérance :

- Sur un branchement, **diminution de 97%** des fautes

Pour la contre-mesure de détection :

- Sur un chargement de donnée, **diminution de 98%**

Évaluées sur une **implémentation de FreeRTOS-MPU**

CM de tolérance	Fonction de changement du niveau de privilège
CM de détection	Fonction d'initialisation d'une tâche du système

Pour la contre-mesure de tolérance :

→ **Diminution de 26%** des fautes dans le registre en sortie

→ Effet probablement plus complexe qu'un saut d'instruction

Pour la contre-mesure de détection :

→ **Diminution de 98%** des fautes dans le registre en sortie

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

COMBINAISON DES DEUX CONTRE-MESURES

➢Les **deux contre-mesures** sont utilisées pour **renforcer un même code** (une fonction addRoundKey d'AES)

```
    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
```

- **CM de détection** : plus efficace mais ne s'applique pas partout
- **Avec CM proposée** : protection des autres instructions

➔ **Diminution de 90% des fautes**, parmi les restantes **aucune faute exploitable** pour une cryptanalyse

SOMMAIRE

CONCLUSION

- Un **modèle de fautes précis** au niveau assembleur
  - Corruption des transferts depuis la mémoire Flash
  - Double-fautes possibles sous certaines conditions
  - Pourcentage élevé de sauts d'instruction

- Une **contre-mesure** de **tolérance au saut d'une instruction**
  - Redondance locale à l'échelle de l'instruction
  - Vérifiée formellement à l'aide d'outils de model-checking
  - Applicable automatiquement à un code générique
  - Renforce notamment branchements et pile

CONCLUSION

- Une **évaluation expérimentale** de deux contre-mesures
  - Avec une deuxième contre-mesure de détection
  - Importance de l'encodage des instructions
  - Combinaison des deux CM très efficace sur un code AES
  - Premiers tests d'attaques par observation type DPA

➔ La contre-mesure de tolérance proposée **complète efficacement la contre-mesure de détection** pour les appels de sous-fonctions et les instructions où celle-ci ne s'applique pas

PERSPECTIVES

- Amélioration de la **précision des modèles de fautes**
  - Pour mieux comprendre les remplacements d'instructions
  - Investiguer les effets au niveau du pipeline
  - Permettrait d'améliorer la définition de contre-mesures

- Test des contre-mesures face aux **attaques par observation**
  - Introduisent-elles des vulnérabilités face à ces attaques ?
  - Comment combiner ces contre-mesures avec des contre-mesures face aux attaques par observation ?

- Application automatique de contre-mesures **par le compilateur**
  - Générer du code renforcé pour des fonctions spécifiées

DIFFUSION DES RÉSULTATS DE CETTE THÈSE

- **4 articles dans des conférences avec actes**
  - **COSADE** 2013
  - **FDTC** 2013
  - **IEEE HOST** 2014
  - **IFIP/IEEE VLSI-SoC** 2014

- **1 article dans un workshop sans actes**
  - **PROOFS** 2013

- **1 article dans un journal à comité de lecture**
  - **Journal of Cryptographic Engineering** 2014

- **3 communications dans des workshops sans actes**
  - **Crypto'Puces** 2013
  - **Chip-To-Cloud Security Forum** 2013
  - **TRUDEVICE Workshop** 2014

Des questions ?

Télécharger la présentation

Fichier PDF