Attaques par injection de fautes et protections logicielles

Karine Heydemann

LIP6

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

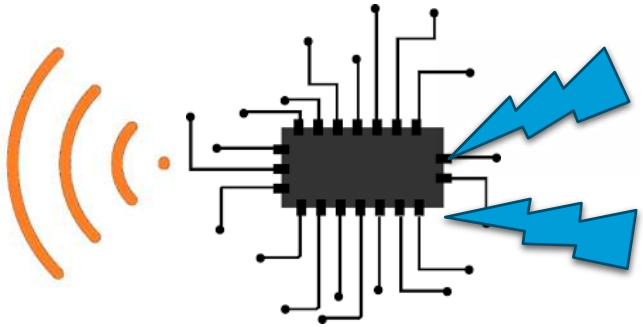
Différentes classes d'attaques

- Exploitation de vulnérabilités logicielles

- Exploitation de vulnérabilités micro-architecturales

- Attaques physiques

Attaques par observation (passives)



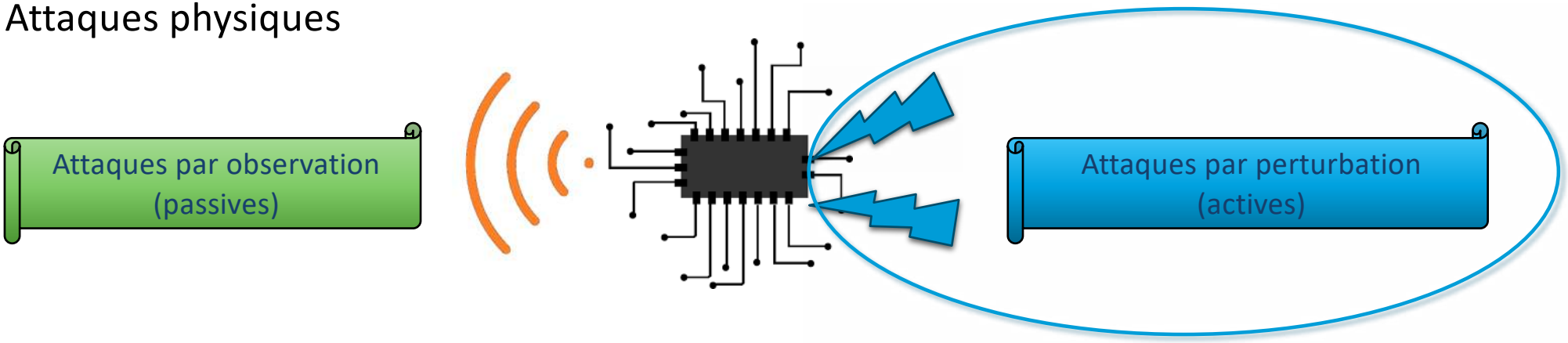
Attaques par perturbation (actives)

Différentes classes d'attaques

- Exploitation de vulnérabilités logicielles

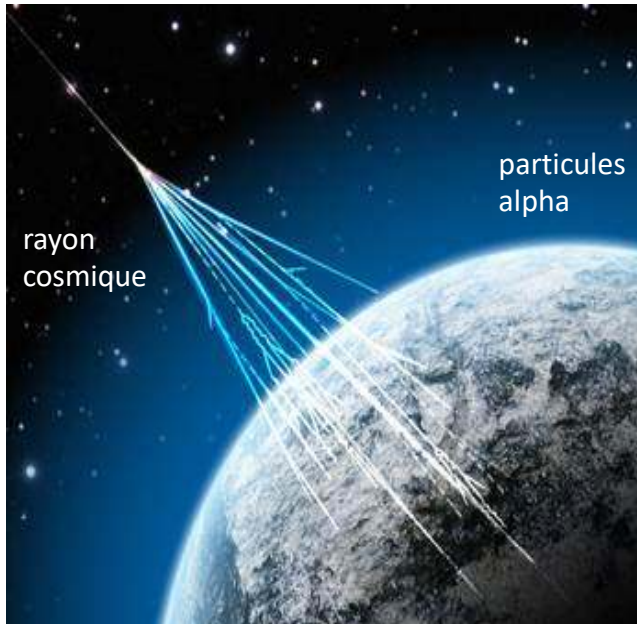
- Exploitation de vulnérabilités micro-architecturales

- Attaques physiques

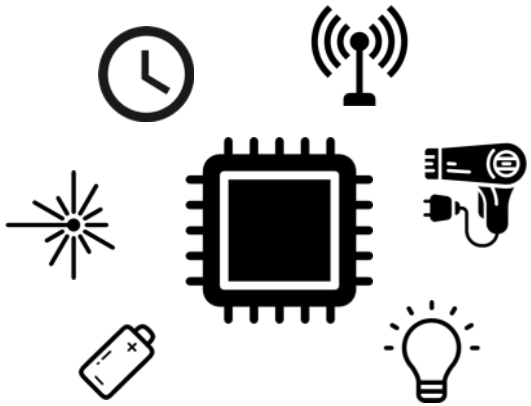


Fautes matérielles

- Fautes matérielles « naturelles »
  - Collision avec particules hautement chargées
  - Fonctionnement dans un environnement « hostile »



- Fautes intentionnelles
  - Perturbation de l'environnement



Cibles des attaques en faute

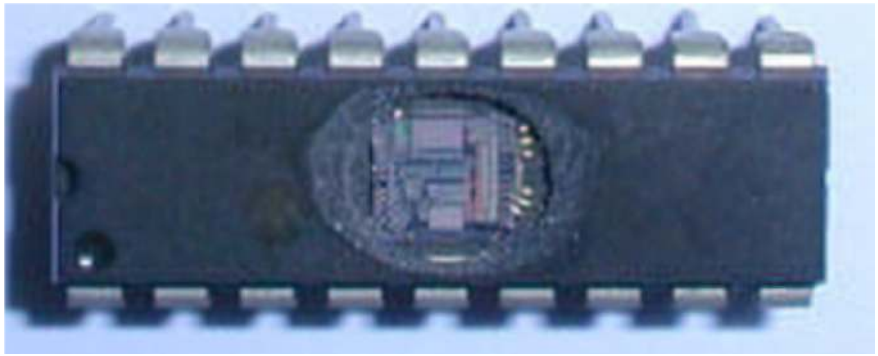
- Historiquement
  - Domaine de la carte à puce : bancaire, identité, télé à péage, santé, porte monnaie électronique
  - Systèmes fortement sécurisés
- Plus récemment
  - Objets connectés, cibles complexes
  - Peu ou pas de protection
- Objectifs d'un attaquant
  - Récupération de données sensibles
  - Contournement de mécanismes d'authentification
  - Escalade de privilèges, prise de contrôle

Plan du séminaire

- Moyens d'injection de faute
- Effets des injections de faute
- Exploitation des fautes
- Contre-mesures
- Conclusion

Attaques semi-invasives

- Besoin d'un accès au silicium
- Ouverture mécanique et/ou chimique du boîtier
- Moyen d'injection : lumière focalisée



Attaques semi-invasives

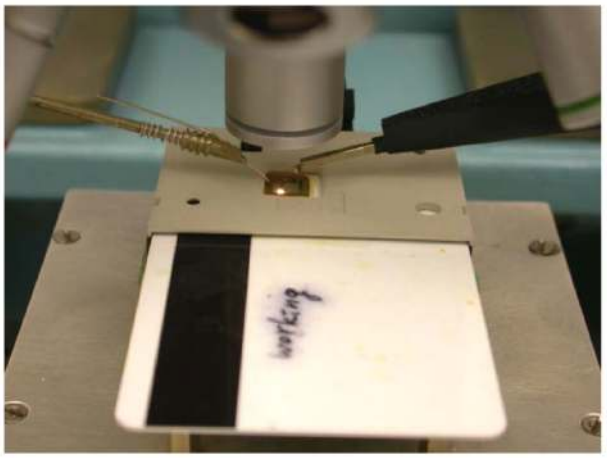
- Premiers travaux utilisant de la lumière focalisée

Optical Fault Induction Attacks

Sergei P. Skorobogatov and Ross J. Anderson

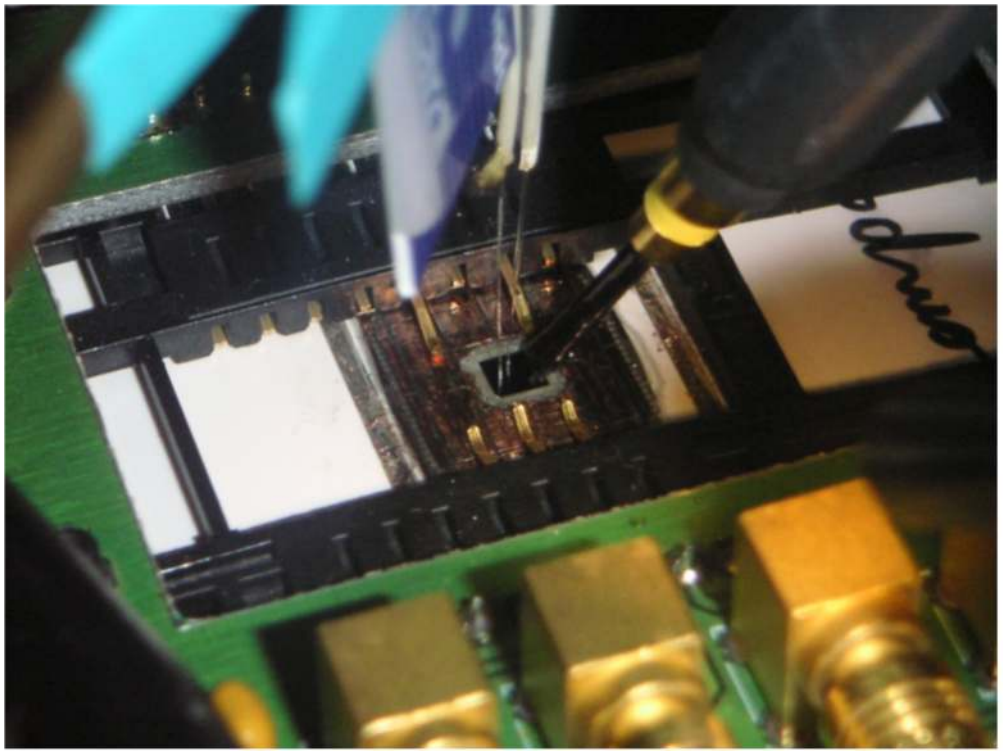
University of Cambridge, Computer Laboratory,
15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
{sps32, rja14}@cl.cam.ac.uk

Abstract. We describe a new class of attacks on secure microcontrollers and smartcards. Illumination of a target transistor causes it to conduct, thereby inducing a transient fault. Such attacks are practical; they do not even require expensive laser equipment. We have carried them out using a flashgun bought second-hand from a camera store for $30 and with an $8 laser pointer. As an illustration of the power of this attack, we developed techniques to set or reset any individual bit of SRAM in a microcontroller. Unless suitable countermeasures are taken, optical probing may also be used to induce errors in cryptographic computations or protocols, and to disrupt the processor's control flow. It thus provides a powerful extension of existing glitching and fault analysis techniques. This vulnerability may pose a big problem for the industry, similar to those resulting from probing attacks in the mid-1990s and power analysis attacks in the late 1990s.

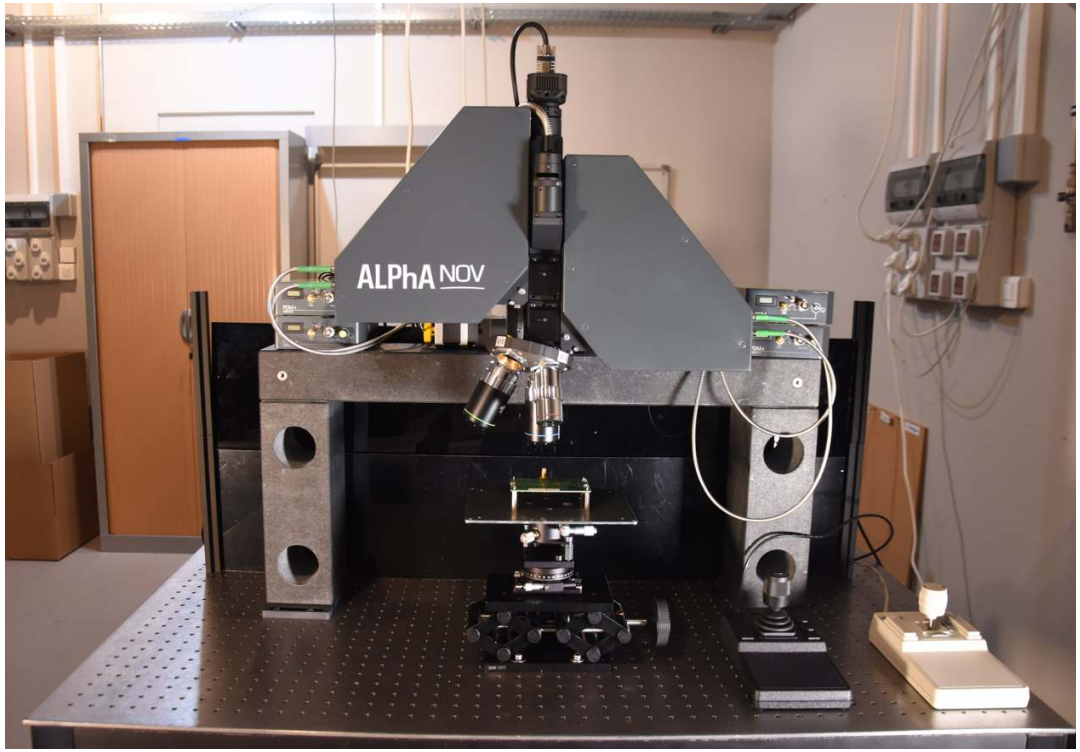


[Skorobogatov et al, CHES 2002]

Attaques semi-invasives



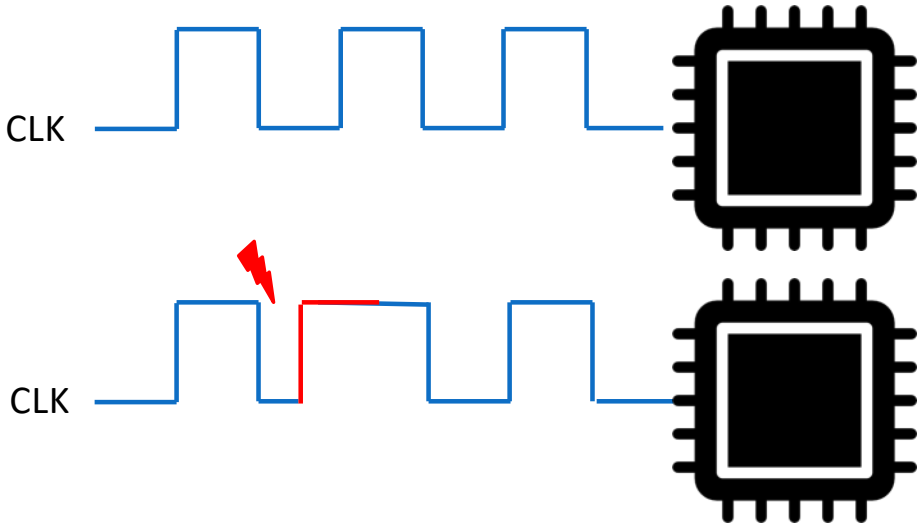
[J. Clédière, cours SCA M2 SU 2022]

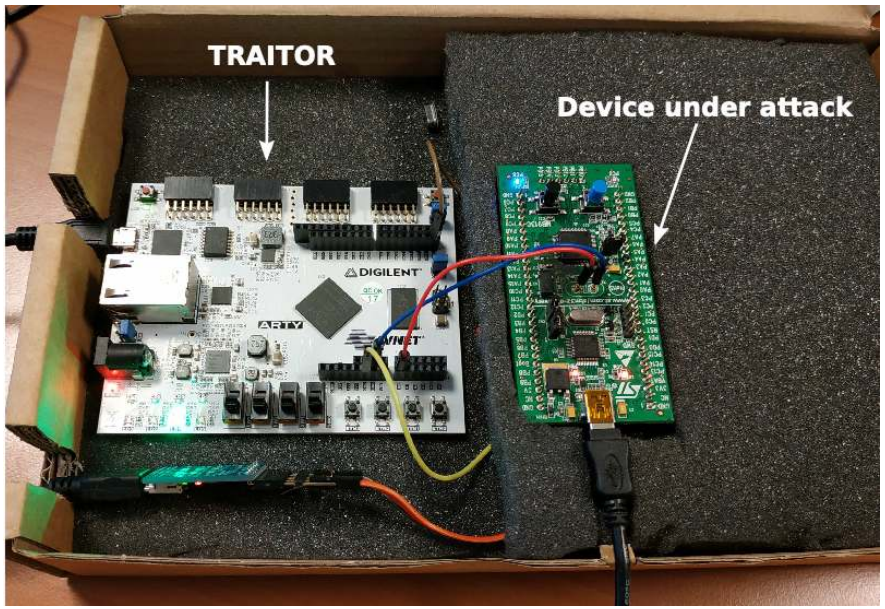


[Journal du CNRS, 30 mars 2022]

Attaques non invasives

- Pas d'ouverture du boîtier
- Exemple
  - Impulsion électrique sur l'horloge ou l'alimentation

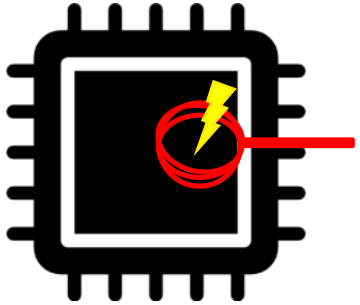


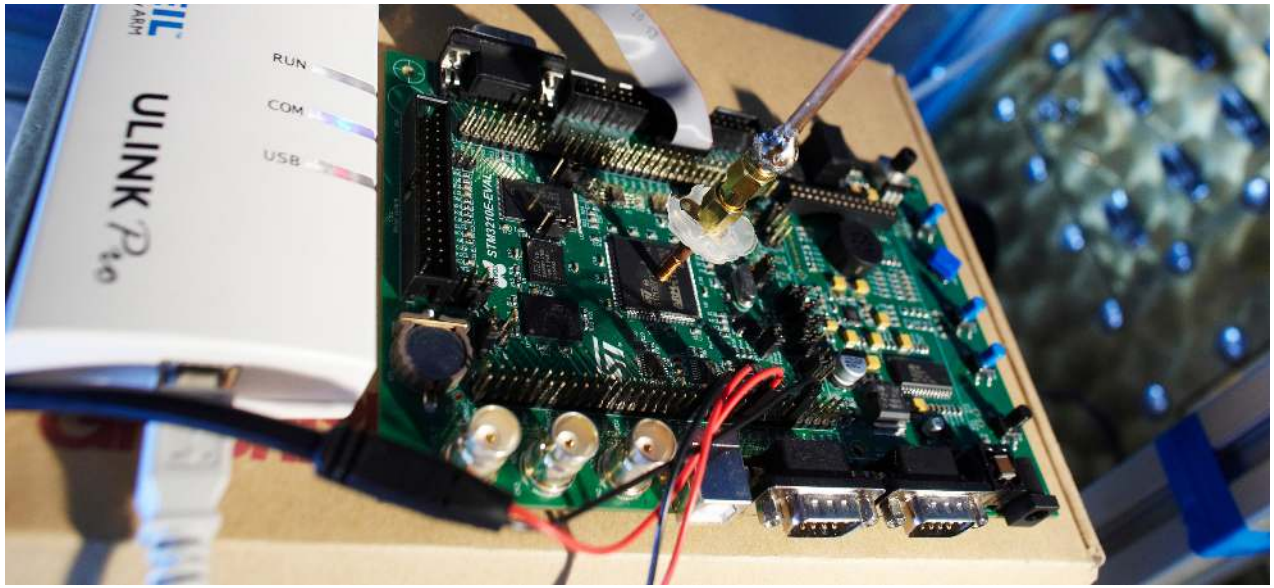


[Claudepierre et al, ASSS 2021]

Attaques non invasives

- Pas d'ouverture du boîtier
- Exemple
  - Impulsion électrique
  - Impulsion électromagnétique





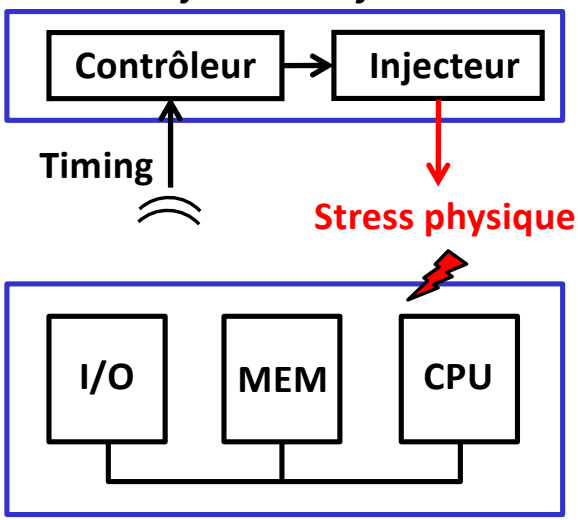
Banc EM [Moro 2014]

Les injections de faute aujourd'hui

Injection "matérielle" de faute

1997 (Bellcore) – aujourd'hui

Plateforme matérielle d'injection de faute

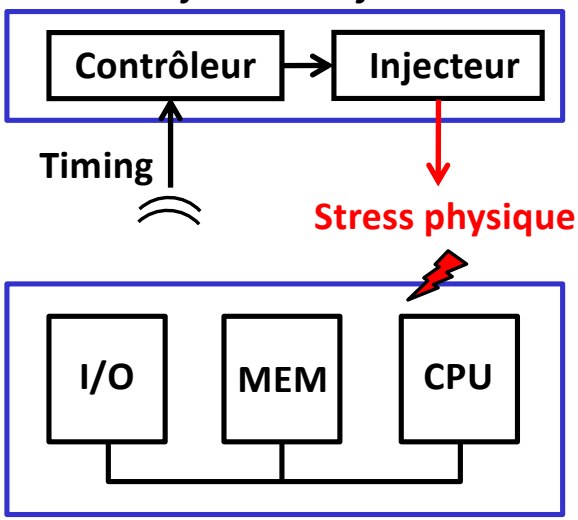


Les injections de faute aujourd'hui

Injection "matérielle" de faute

1997 (Bellcore) – aujourd'hui

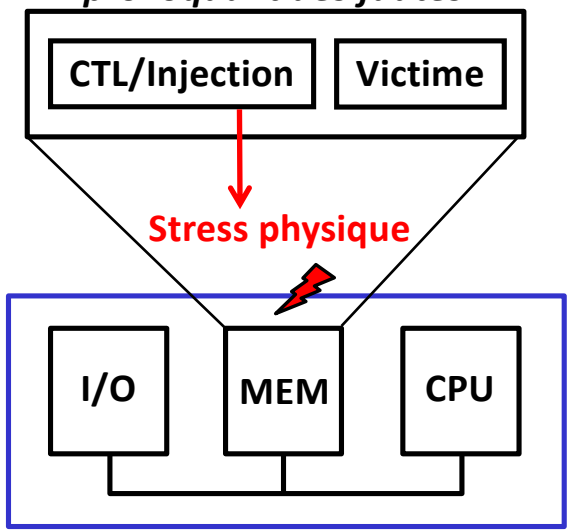
Plateforme matérielle d'injection de faute



Injection "logicielle" de faute

2014 (Rowhammer) – aujourd'hui

Tâches logicielles provoquant des fautes

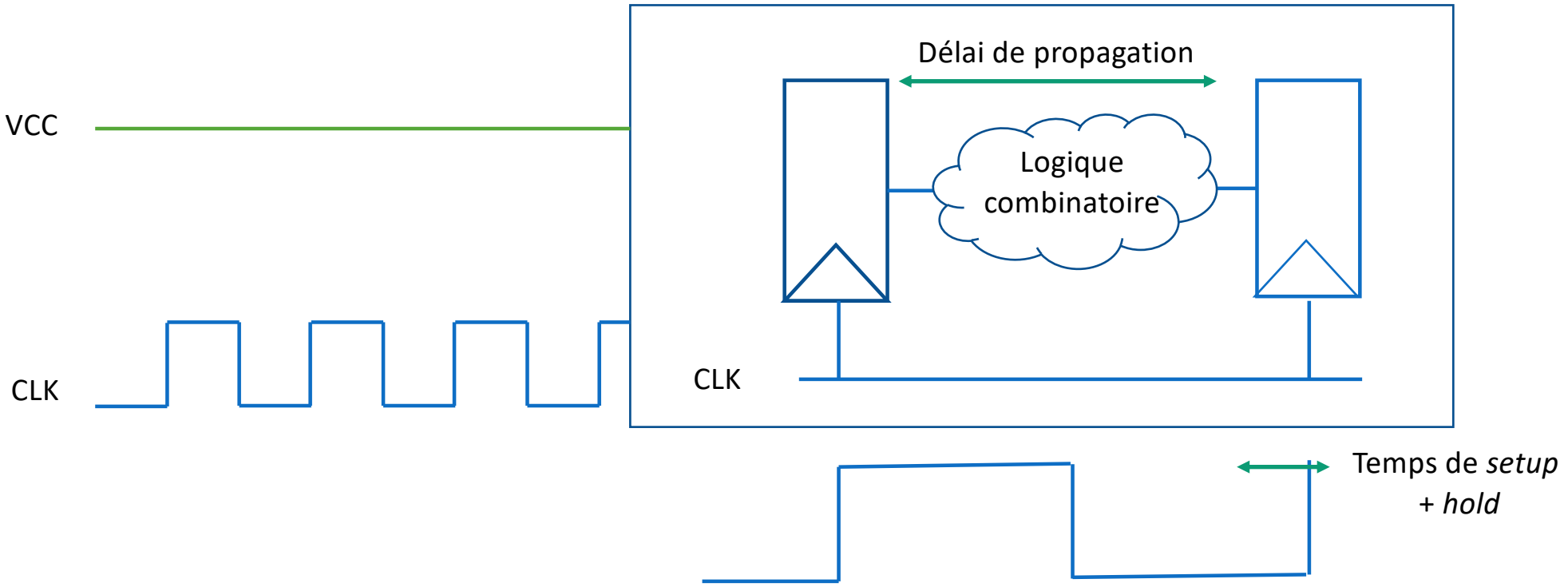


2014 : Rowhammer
2017 : CLKSCREW
2019 : VOLTJOCKEY
2020 : PlunderVolt
…

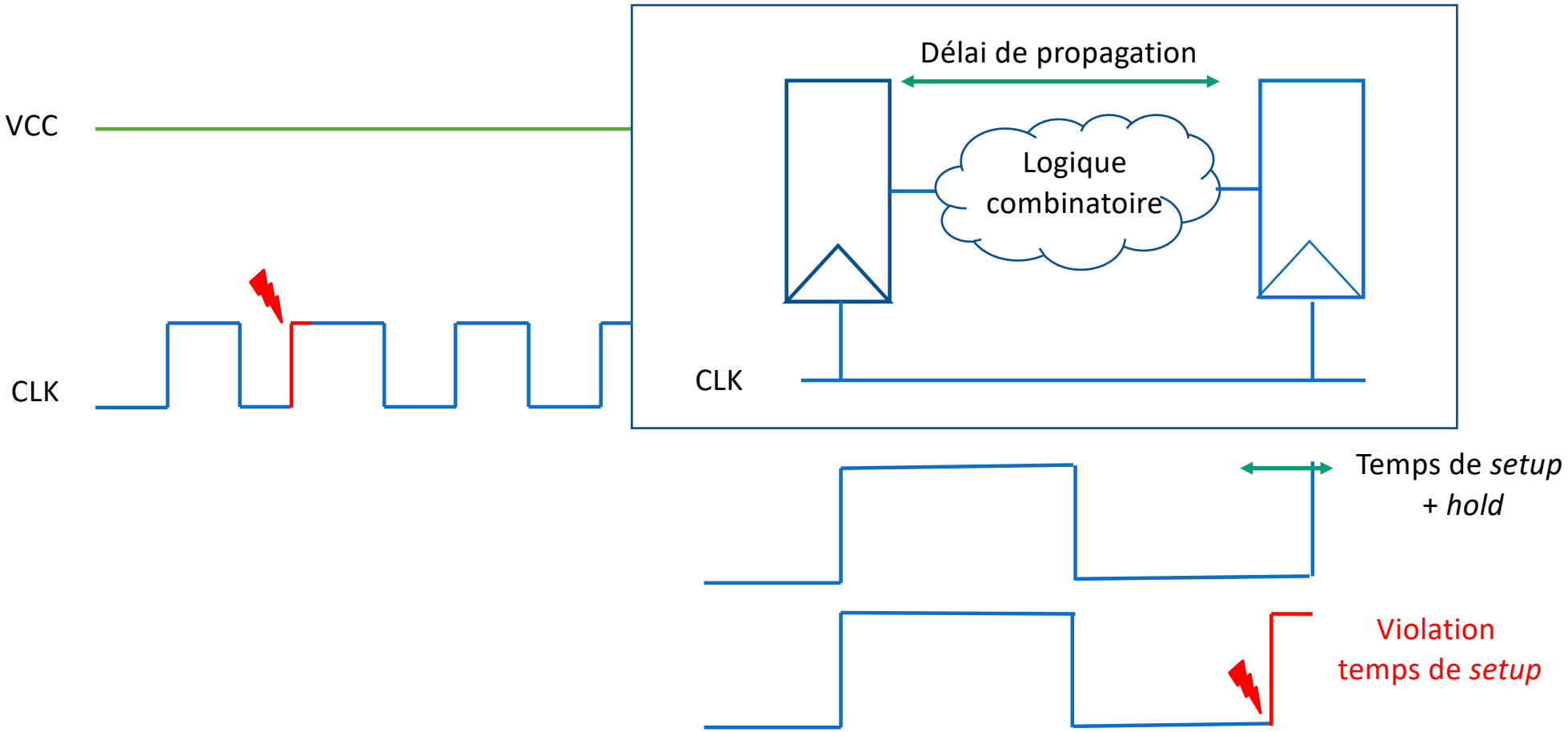
Plan du séminaire

- Moyens d'injection de faute
- **Effets des injections de faute**
- Exploitation des fautes
- Contre-mesures
- Conclusion

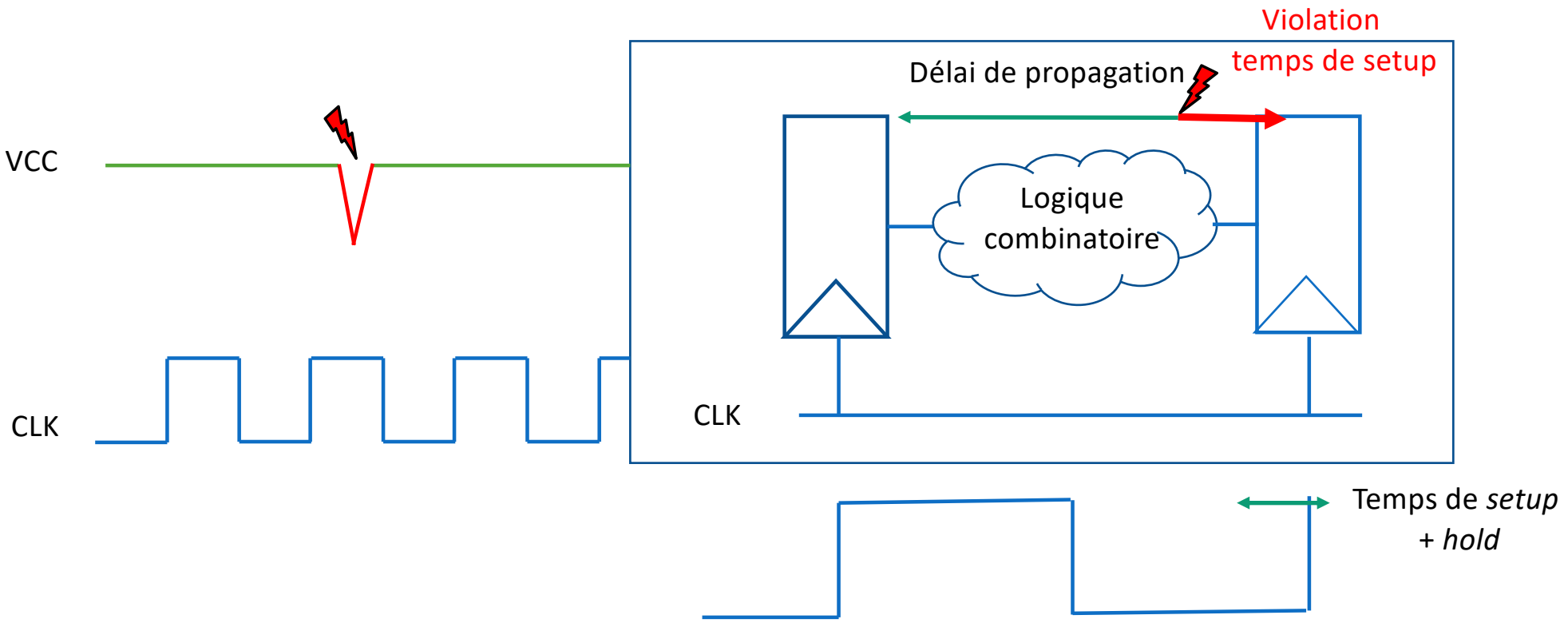
Effet d’une impulsion électrique



Effet d’une impulsion électrique

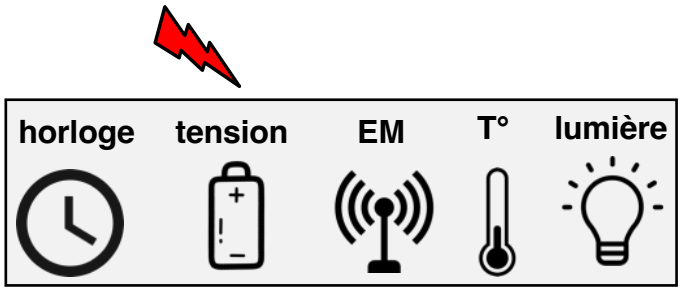


Effet d’une impulsion électrique



Effets des injections de faute

Niveau physique

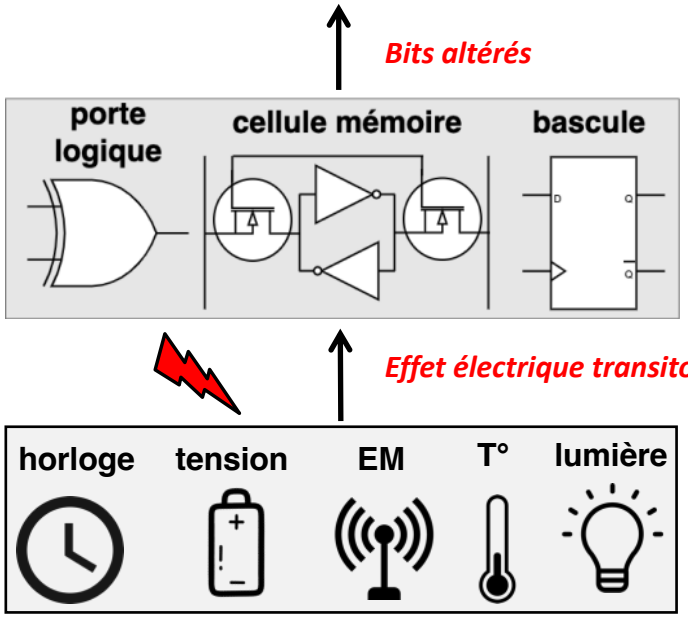


Injection

Effet physique

- Génération de courant induit (lumière, EM)
- Altération des niveaux électriques
- Violation des temps d'horloge

Effets des injections de faute



Effet logique : altération de bits

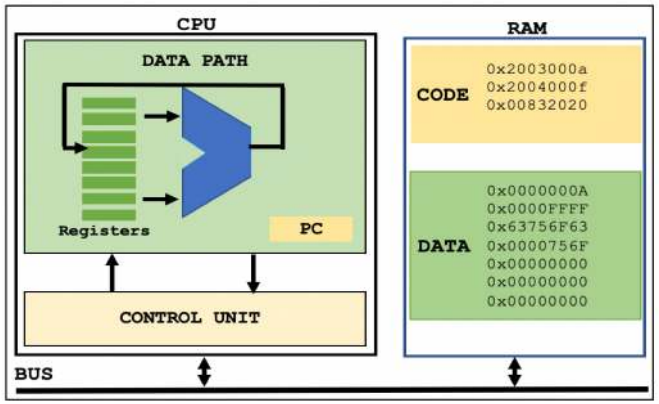
- Granularité : monobit, multibits, octet, mot
- Type : inversion, mise à 0 ou 1, aléatoire
- Durée : transitoire, permanente, destructive

Effet physique

- Génération de courant induit (lumière, EM)
- Altération des niveaux électriques
- Violation des temps d'horloge

Effets des injections de faute

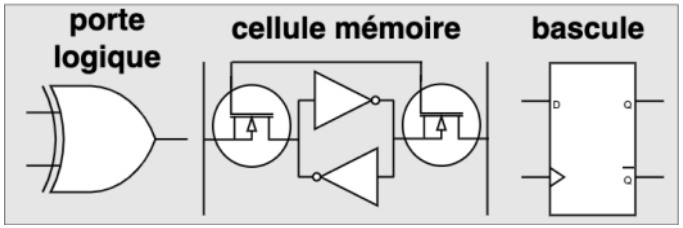
Niveau micro-architecture



Effet au niveau micro-architectural ?

Bits altérés

Niveau circuit



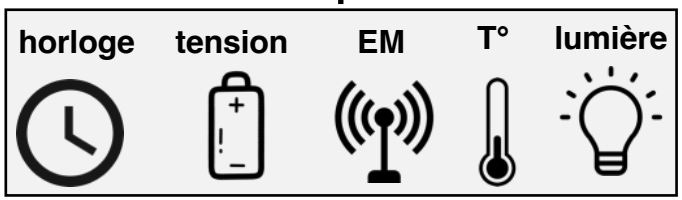
Manifestation

Effet logique : altération de bits

- Granularité : monobit, multibits, octet, mot
- Type : inversion, mise à 0 ou 1, aléatoire
- Durée : transitoire, permanente, destructive

Effet électrique transitoire

Niveau physique

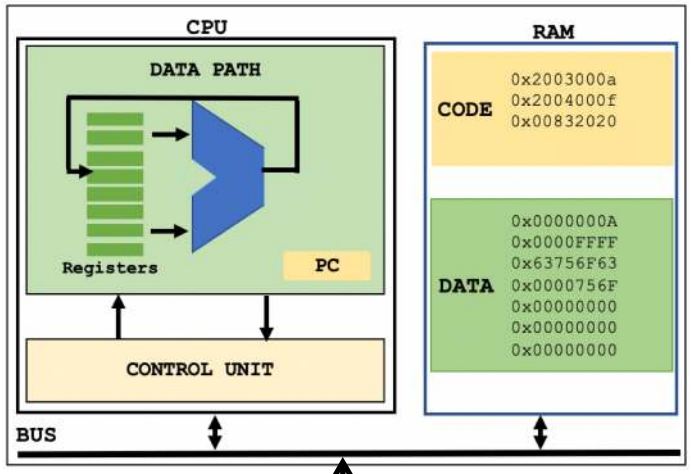


Injection

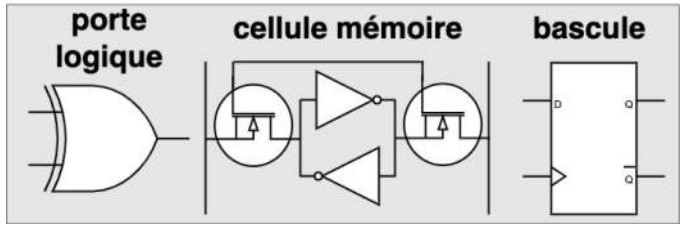
Effet physique

- Génération de courant induit (lumière, EM)
- Altération des niveaux électriques
- Violation des temps d'horloge

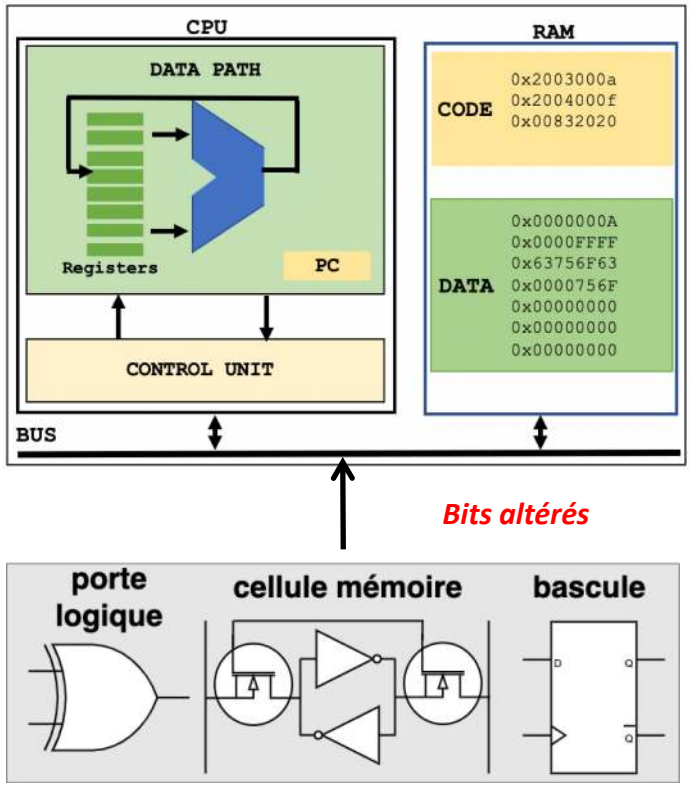
Effets au niveau micro-architectural

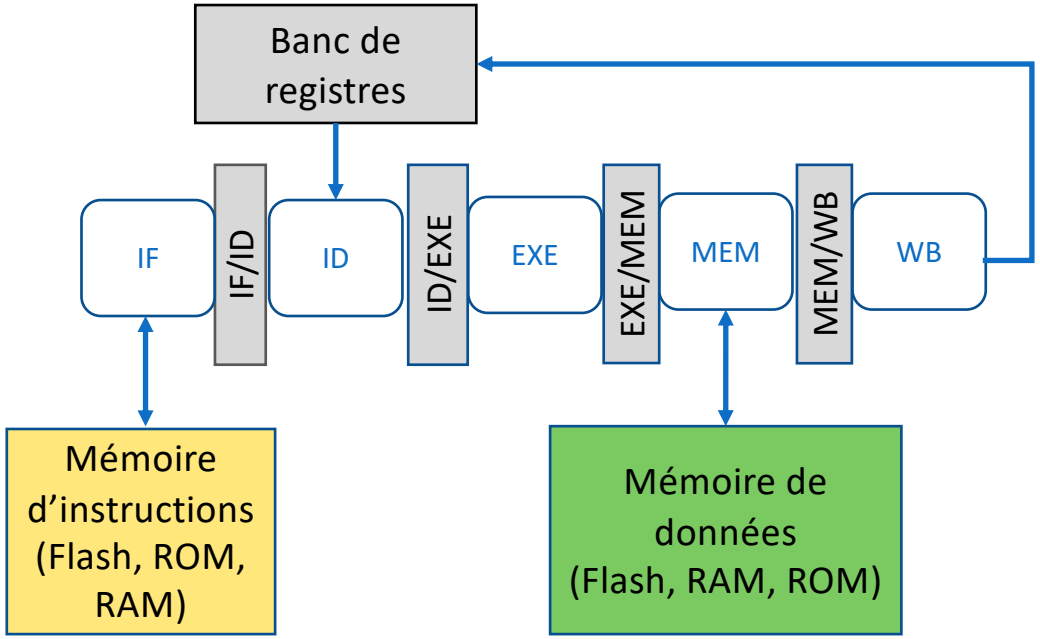


Bits altérés

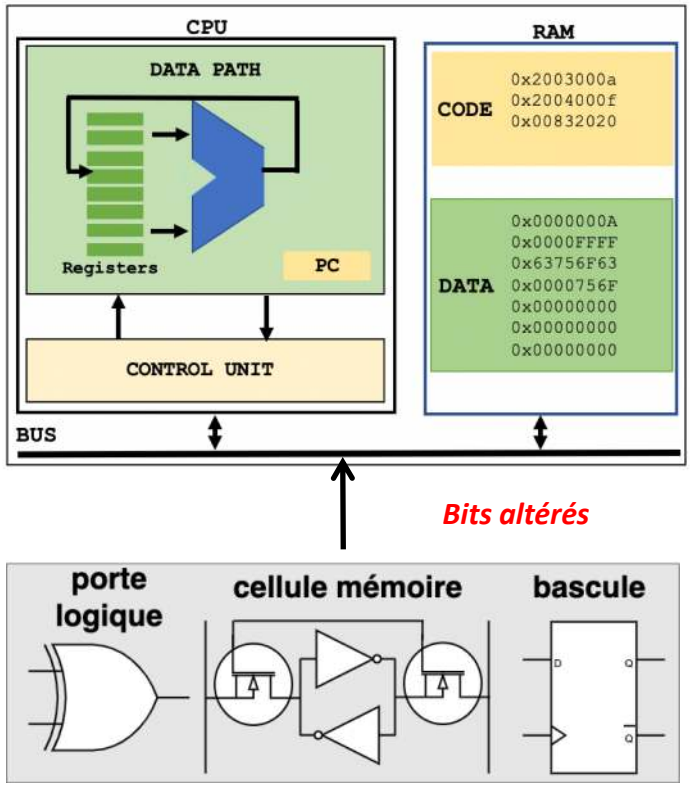


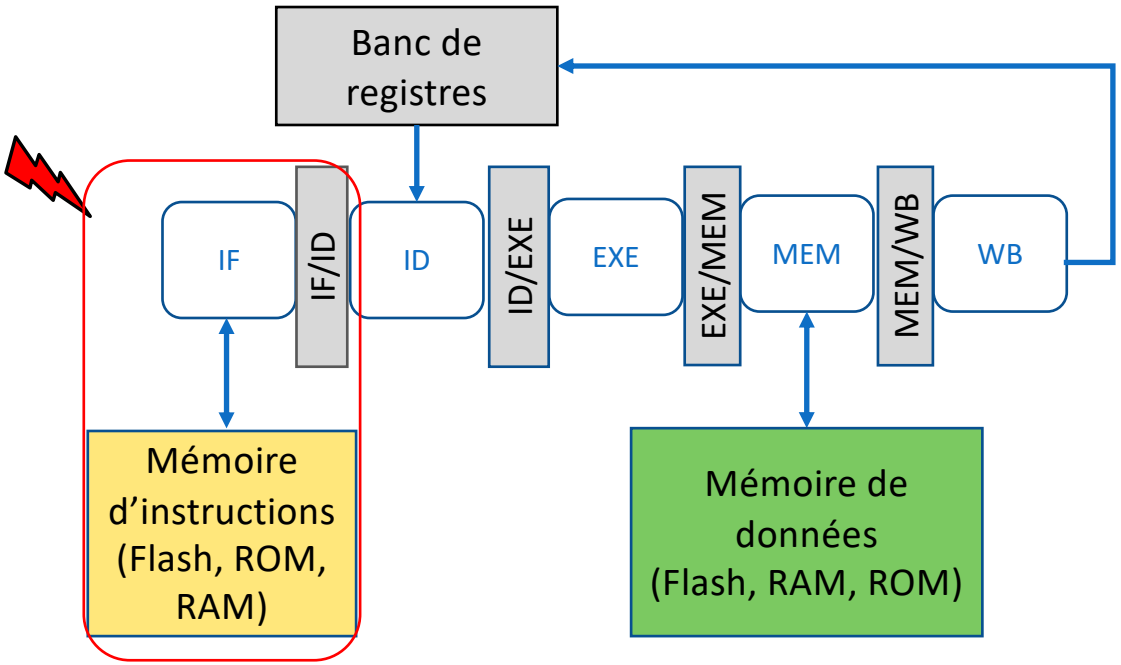
Effets au niveau micro-architectural





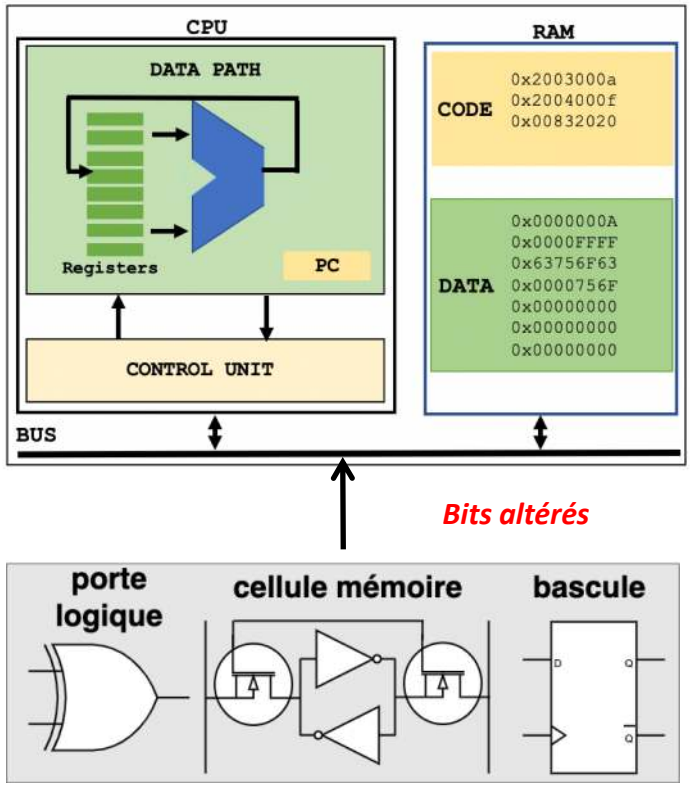
Effets au niveau micro-architectural

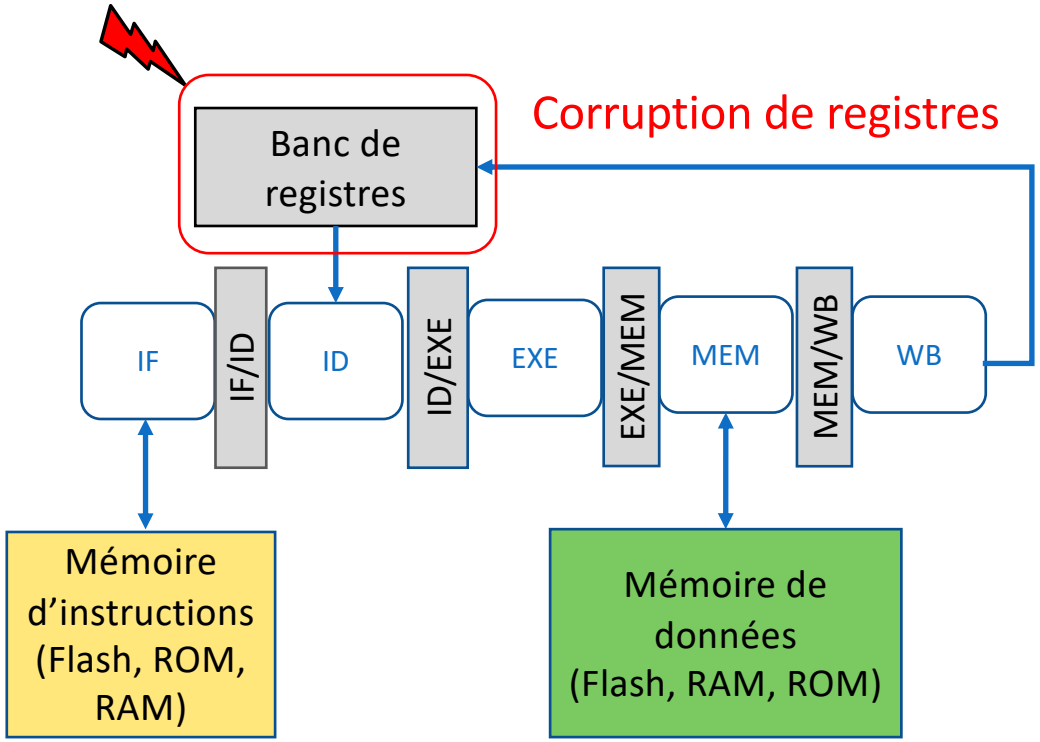




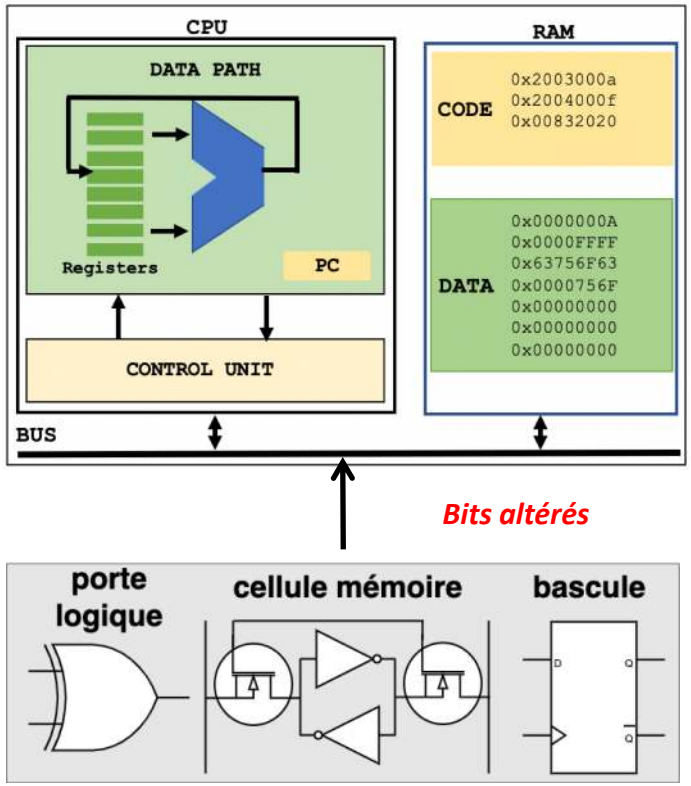
Corruption d'instructions

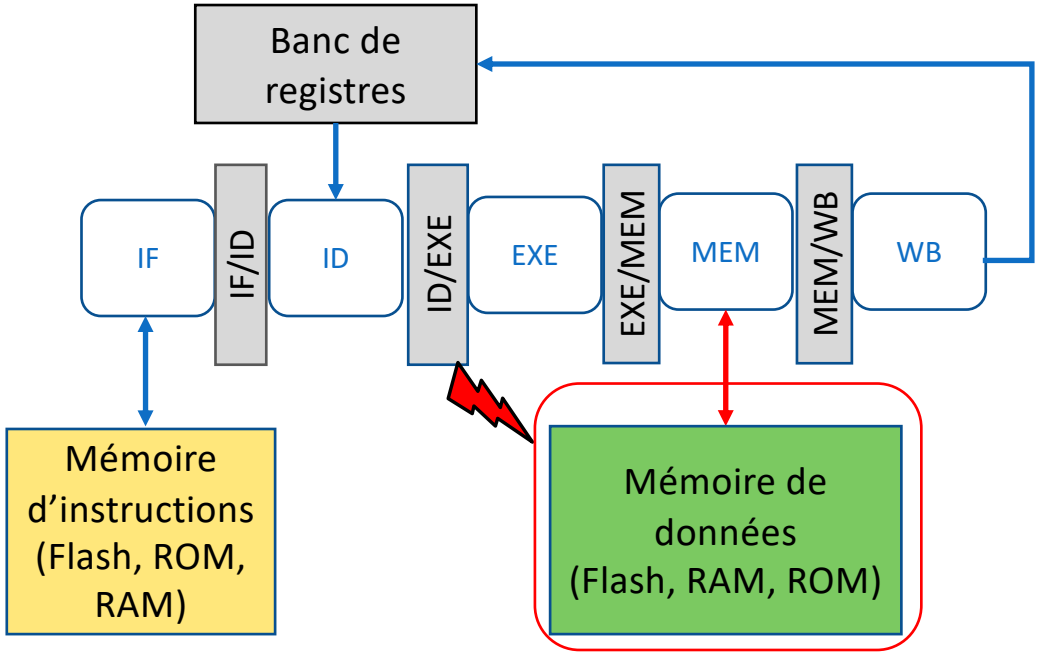
Effets au niveau micro-architectural





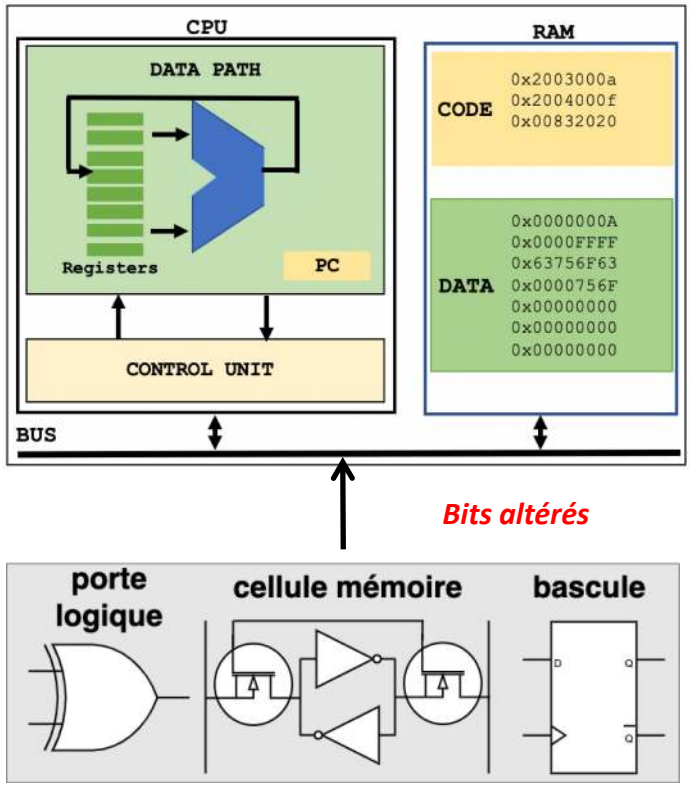
Effets au niveau micro-architectural

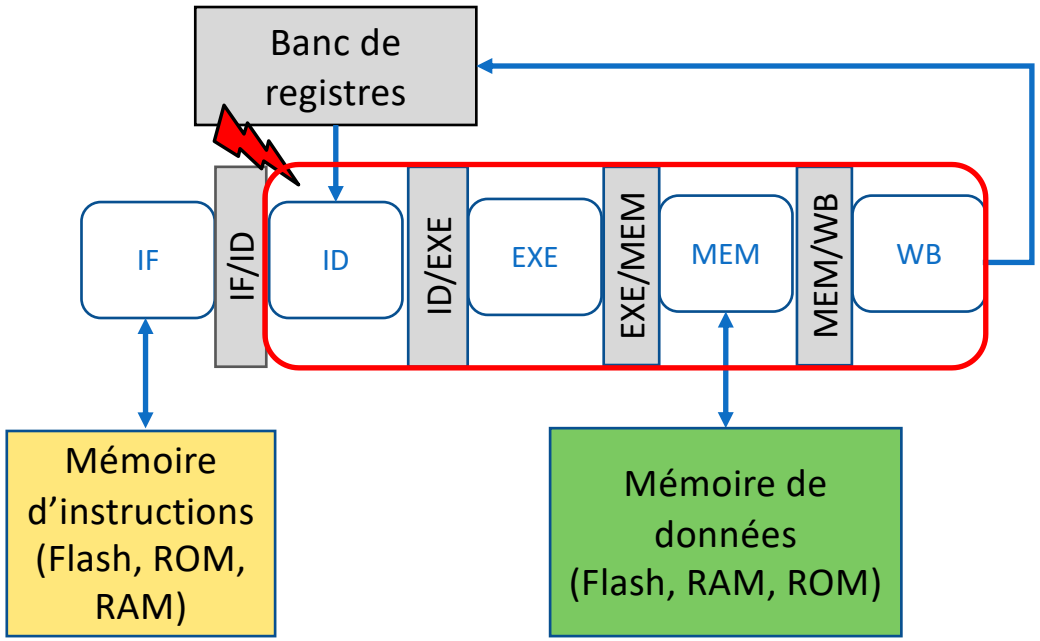




Corruption de données

Effets au niveau micro-architectural

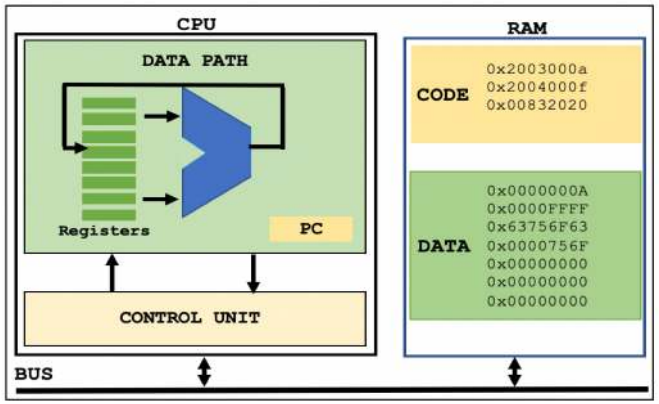




Corruption de l'exécution : calculs ou flot de contrôle

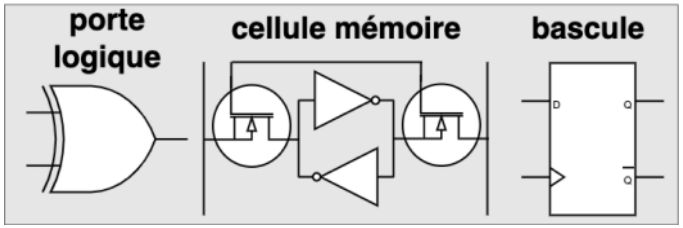
Effets des injections de faute

Niveau micro-architecture



Bits altérés

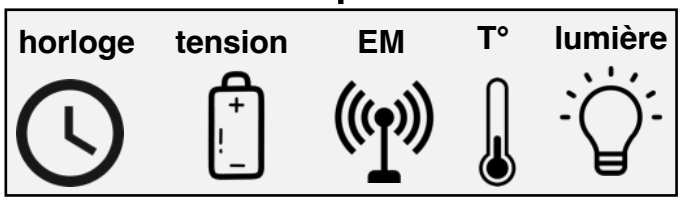
Niveau circuit



Manifestation

Effet électrique transitoire

Niveau physique



Injection

Effet au niveau architectural

- Corruption d'instruction(s)
- Corruption de registre(s), de donnée(s)
- Corruption de calcul(s)
- Corruption du flot de contrôle

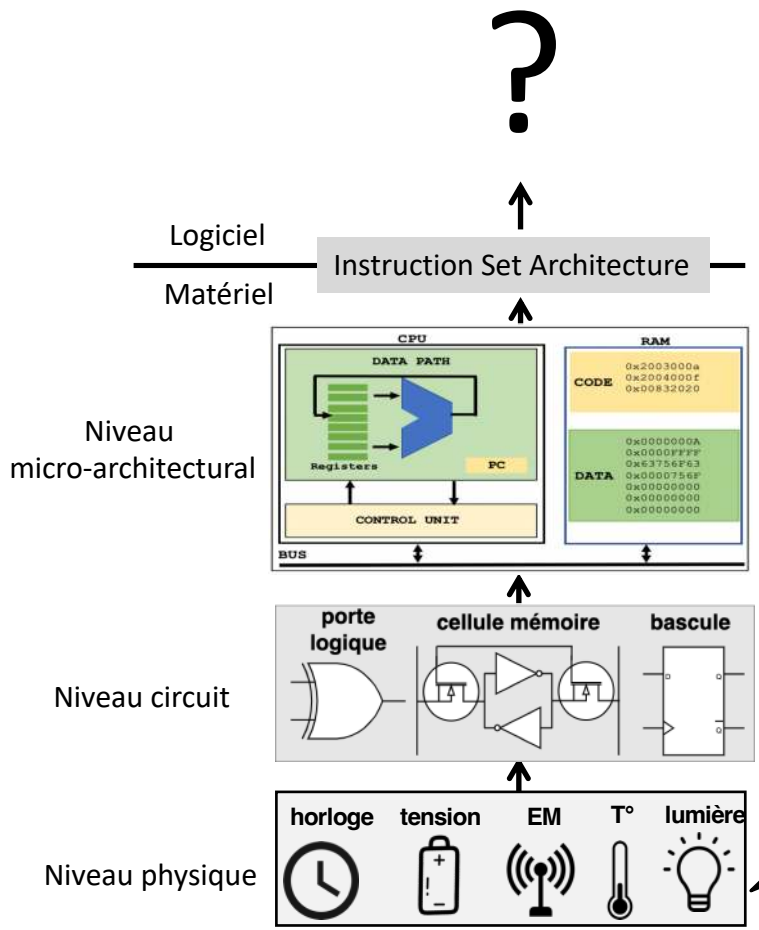
Effet logique : altération de bits

- Granularité : monobit, multibits, octet, mot
- Type : inversion, mise à 0 ou 1, aléatoire
- Durée : transitoire, permanente, destructive

Effet physique

- Génération de courant induit (lumière, EM)
- Altération des niveaux électriques
- Violation des temps d'horloge

Effets au niveau logiciel



Corruptions : intructions, registres, données, calculs ou flot d'exécution

Propagation

bits altérés

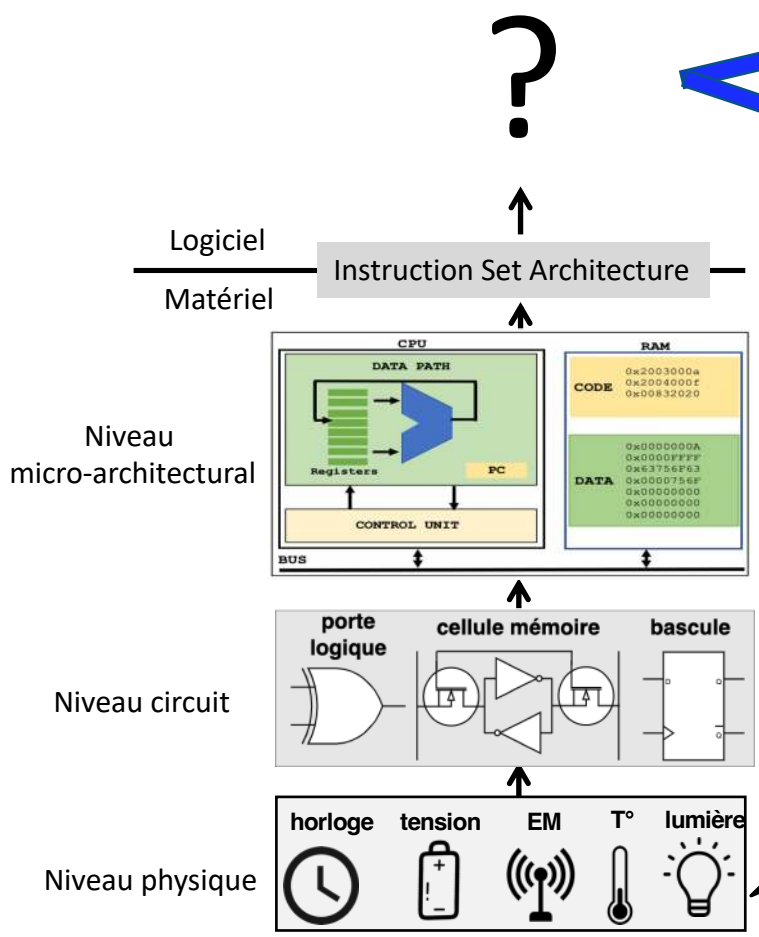
Manifestation

effet électrique transitoire

Injection

[Yuce et al., 2018]

Effets au niveau logiciel



Exploitation : vue “attaquant”

Caractérisation : vue “défenseur”

Corruptions : intructions, registres, données, calculs ou flot d’exécution

Propagation

bits altérés

Manifestation

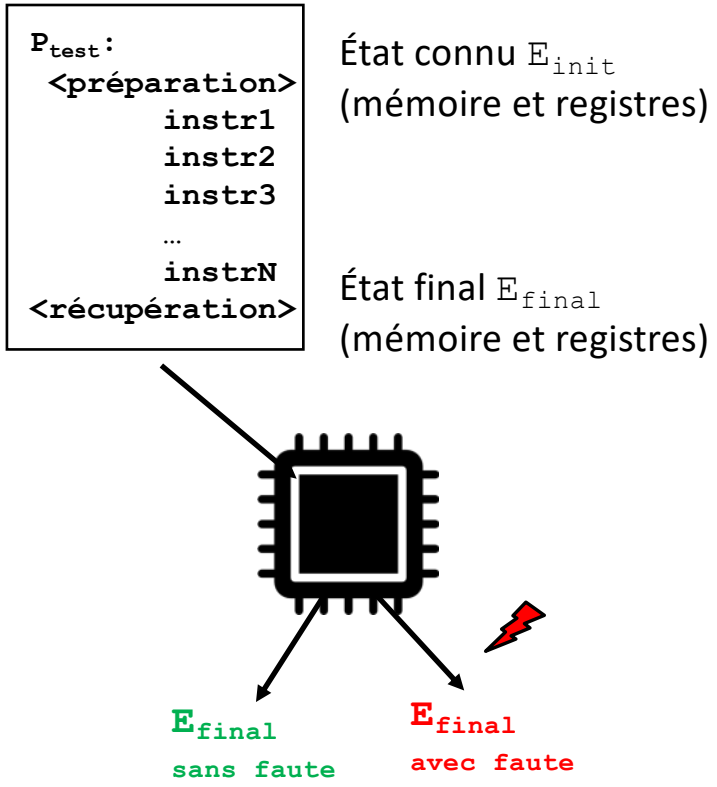
effet électrique transitoire

Injection

[Yuce et al., 2018]

Caractérisation des effets jusqu'au niveau logiciel

- Calibrage du banc d'injection
  - Paramétrage : forme et durée de l'impulsion, position sur la puce, etc.
- Campagnes d'injection
  - Programmes dédiés P_{test}
  - État initial E_{init} « bien choisi »
- Comparaison des E_{final}
  - Inférence de modèles de faute
  - Remplacement d'instruction,
    saut d'instruction(s),
    corruption d'opérande,
    rejeu d'instruction(s),
    …



Plan du séminaire

- Moyens d'injection de faute
- Effets des injections de faute
- **Exploitation des fautes**
- Contre-mesures
- Conclusion

Attaque BellCoRe

- Algorithme de signature RSA
  - $N = p \times q$, p et q deux grands entiers premiers
  - e clé publique
  - d clé privée
  - Signature s d'un message m : $s = m^d \bmod N$
- Variante RSA-CRT: moins d'exponentiations
  - Deux calculs

    $s_p = m^{d \bmod (p-1)} \bmod p$

    $s_q = m^{d \bmod (q-1)} \bmod q$
  - Recombinaison

$$s = (s_p \times q^{-1} \bmod p) \times q + (s_q \times p^{-1} \bmod q) \times p$$
$$= a(s_p) \times q + b(s_q) \times p$$

On the Importance of Checking Cryptographic Protocols for Faults*

Dan Boneh† dabo@bellcore.com
Richard A. DeMillo rad@bellcore.com
Richard J. Lipton‡ lipton@bellcore.com

Security and Cryptography Research Group, Bellcore,
445 South Street, Morristown, NJ, 07960

Abstract

We present a theoretical model for breaking various cryptographic schemes by taking advantage of random hardware faults. We show how to attack certain implementations of RSA and Rabin signatures. An implementation of RSA based on the Chinese Remainder Theorem can be broken using a single erroneous signature. Other implementations can be broken using a larger number of erroneous signatures. We also analyze the vulnerability to hardware faults of two identification protocols: Fiat-Shamir and Schnorr. The Fiat-Shamir protocol can be broken after a small number of erroneous executions of the protocol. Schnorr's protocol can also be broken, but a larger number of erroneous executions is needed.

[Eurocrypt 1997]

Attaque BellCoRe

- s signature sans faute :

$$s = a(s_p) \times q + b(s_q) \times p$$

- s' signature avec une faute dans le calcul de s_p devenue $s_{p'}$

$$s' = a(s_{p'}) \times q + b(s_q) \times p$$

- Différence entre s et s' : $s - s' = a(s_p) \times q - a(s_{p'}) \times q$

- $s - s'$ est un multiple de q donc $q = pgcd(s-s', N)$, on retrouve ensuite aisément p puis d

- Déduction identique de p si faute lors du calcul de s_q

Exploitation des injections de fautes

- Cryptanalyse
  - Analyse différentielle : *Differential Fault Analysis*
  - Réduction du nombre de rondes : *Round Reduction*
  - Faute sans effet : *Safe Error Analysis*
- Attaques logiques

Analyse des fautes sans effet (*safe-error)*

```
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]
```

Multiplication rapide
dite “à la russe”
+ contre-mesure “*double-add always*”

Analyse des fautes sans effet (*safe-error)*

```
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]
```

Opération factice quand k_i égal 0

Analyse des fautes sans effet (*safe-error)*

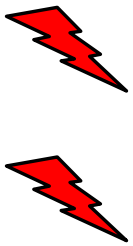
```
Input: Elliptic Curve Point P
       secret integer k = {k_{n-1}k_{n-2}…k_1k_0}
Output: k.P
  R[0] = 0
  for i = n – 1 down to 0 do
     R[0] = 2.R[0]
     R[1] = R[0] + P
     R[0] = R[k_i]
  end for
  return R[0]
```

Faute sans effet si dans une opération factice

Attaques logiques

- Attaques généralistes
  - Extraction de contenu mémoire
  - Détournement du flot de contrôle
  - Escalade de privilèges
  - Court-circuit du démarrage sécurisé (secure boot)

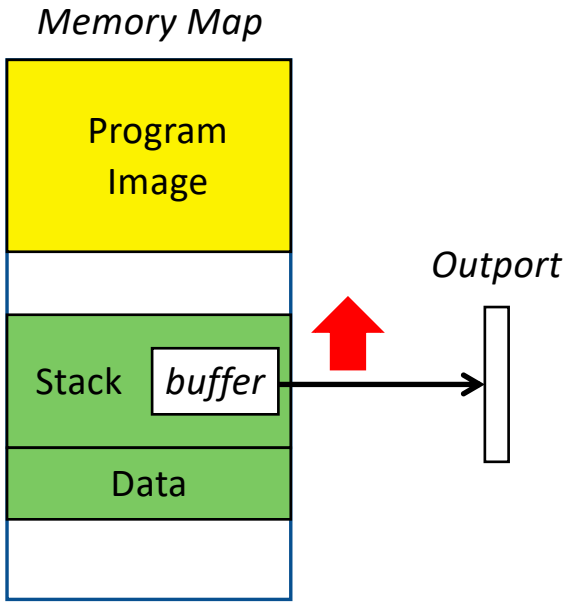
Extraction de données en mémoire



```
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  ...
```

Saut d'instruction

Saut d'instruction ou autre remplacement

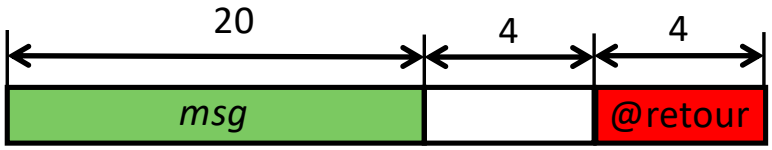


R. Anderson and M. Kuhn, "Tamper resistance: a cautionary note," *2nd USENIX Workshop on Electronic Commerce*. 1996.

Attaque par débordement de tampon

```
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;
}
```

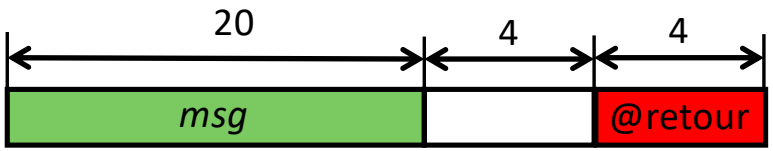


Pile dans `myfunc`

Attaque par débordement de tampon

```
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;
 }
```

Saut d'instruction

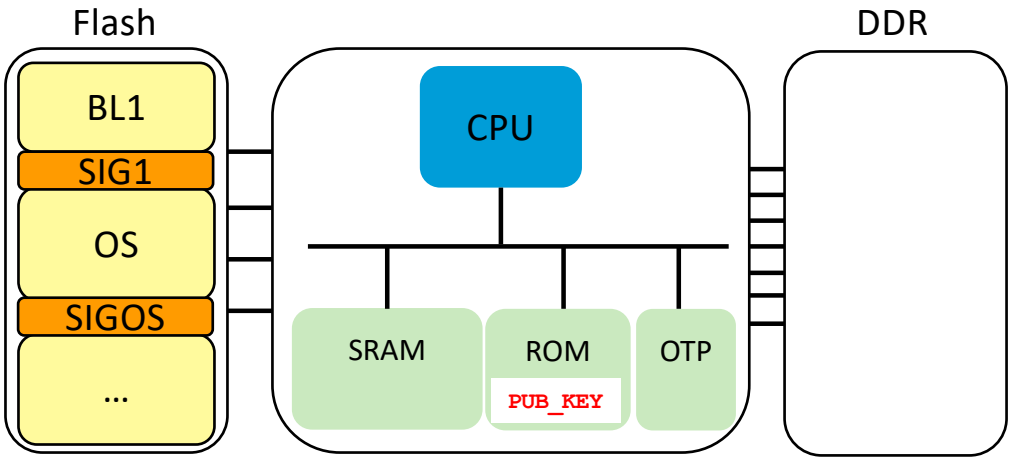


Pile dans `myfunc`

ARM Cortex M0

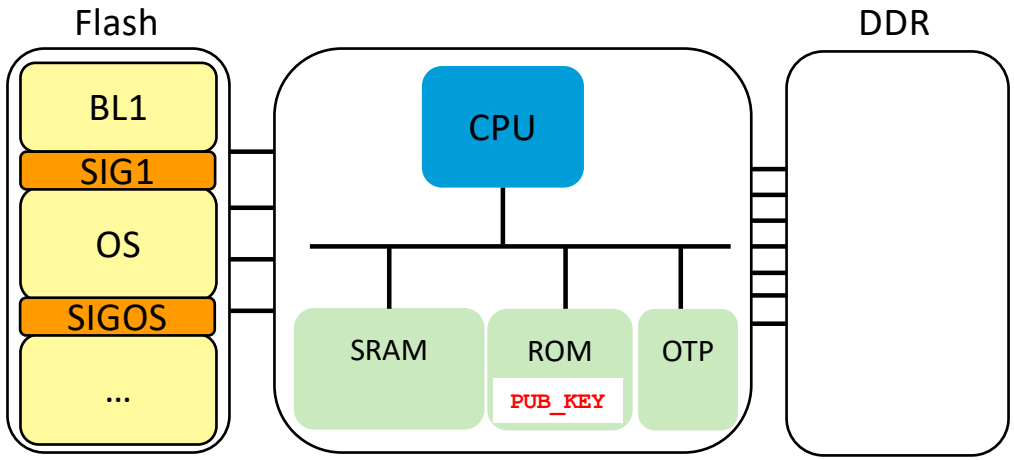
S. Nashimoto et al. *Buffer overflow attack with multiple fault injection and a proven countermeasure*. J. Cryptographic Engineering 7(1): 35-46 (2017)

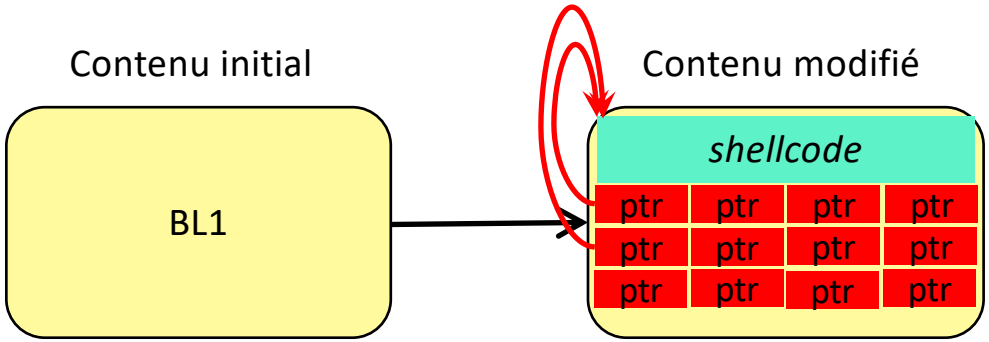
Bypassing secure boot




```
/* 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 from signature */
    rsa(PUB_KEY, SIG_RAM, SIG_HASH)

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

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

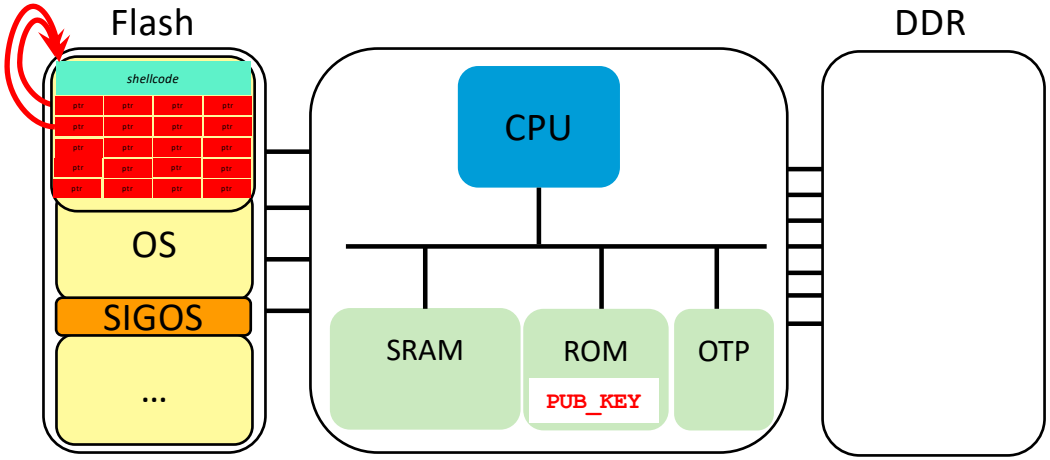
Bypassing secure boot






```
/* 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 from signature */
    rsa(PUB_KEY, SIG_RAM, SIG_HASH)

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

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

Bypassing secure boot

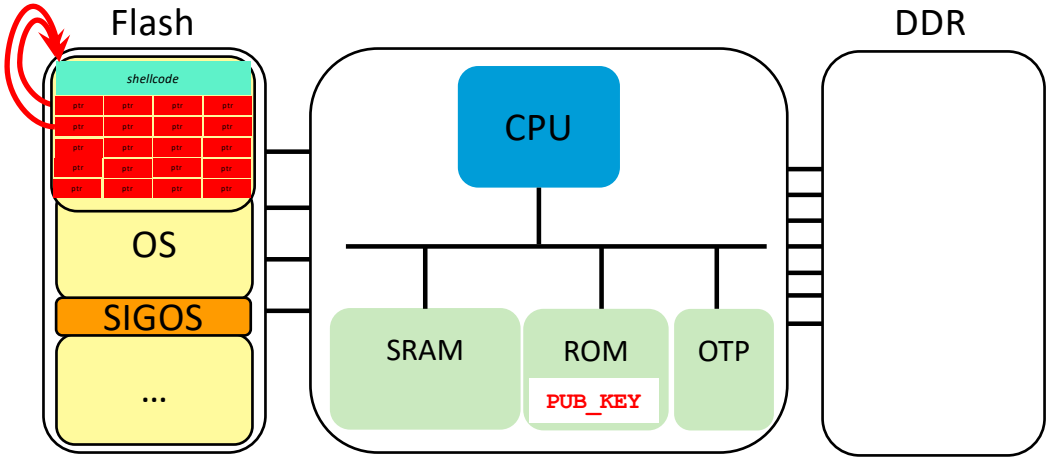



```
MultiWordCopy:
  LDMIA r1!, {r3 - r10}
  STMIA r0!, {r3 - r10}
  SUBS r2, r2, #32
  BGE MultiWordCopy
```

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

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

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

Bypassing secure boot

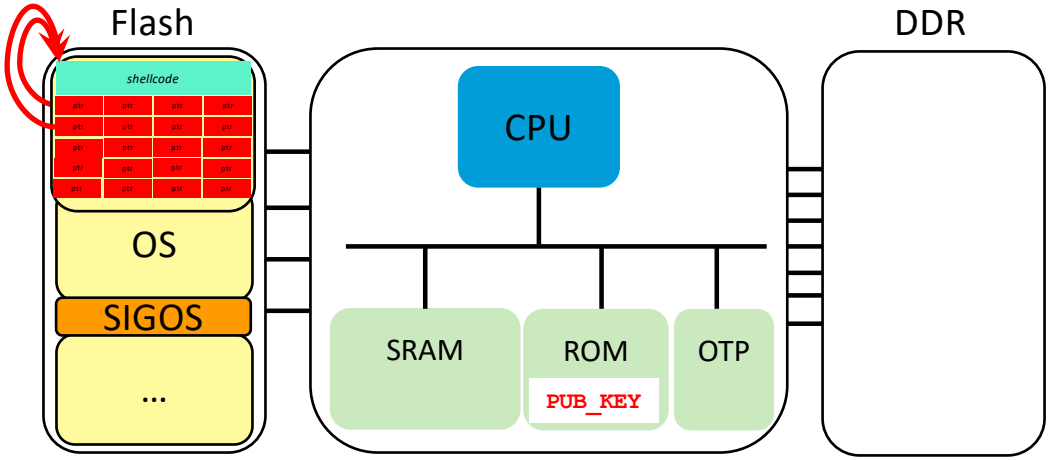



```
MultiWordCopy:
  LDMIA r1!, {r3 - r10}
  STMIA r0!, {r3 - r10}
  SUBS r2, r2, #32
  BGE MultiWordCopy
```

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

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

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

Bypassing secure boot




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

→ **`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 from signature */
    rsa(PUB_KEY, SIG_RAM, SIG_HASH)

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

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

Attaques réelles

- *Xbox reset glitch hack*[1] -- 2011
  - *Launch your own code* (Linux kernel)
- *Glitching the (Nintendo) Switch*[2] -- 2018
- Extraction de firmware sur plusieurs microcontrôleurs répandus -- 2019[3] & 2020[4]
- Extraction de la clé secret d'un portefeuille de crypto-monnaie -- 2020 & 2021[5]
- Contournement du *secure boot* d'un SoC Nvidia Tegra X2[6] utilisé pour le pilotage automatique de voitures Tesla -- 2021

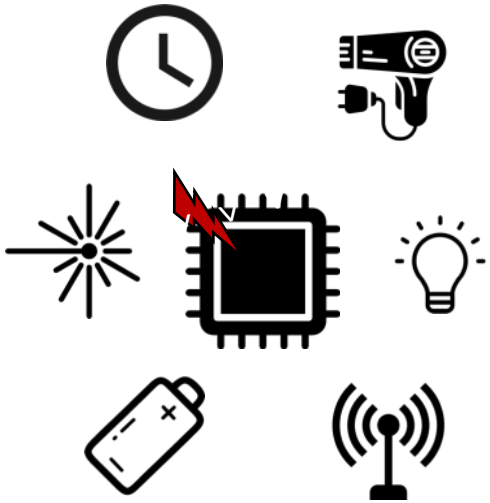
[1] http://www.logic-sunrise.com/news-341321-the-reset-glitch-hack-a-new-exploit-on-xbox-360-en.html
[2] https://media.ccc.de/v/c4.openchaos.2018.06.glitching-the-switch
[3] https://tches.iacr.org/index.php/TCHES/article/view/7390
[4] https://tches.iacr.org/index.php/TCHES/article/view/8727
[5] https://www.sstic.org/user/oheriveaux/
[6] https://arxiv.org/pdf/2108.06131.pdf

Plan du séminaire

- Moyens d'injection de faute
- Effets des injections de faute
- Exploitation des fautes
- **Contre-mesures**
- Conclusion

Protections contre les injections de faute

- **Contre-mesures matérielles**
  - Mécanisme d'introduction de gigue *: jitters*
  - Horloge interne, détecteur de *glitch*
  - Régulateur de tension
  - Bouclier métallique
  - Détecteur de lumière
  - Redondance
  - Code détecteur d'erreur (registres, mémoire)
- **Countre-mesures logicielles**
  - Spécifiques à un algorithme (RSA-CRT)
  - Génériques
- En pratique, combinaison dans les composants sécurisés



Protections logicielles

- Propriétés de sécurité visées
  - Intégrité des données
  - Intégrité du code
  - Intégrité du flot de contrôle
  - Intégrité des calculs
- Applicables (en majorité) à différents niveaux de présentation du code
  - Source
  - Assembleur

Durcissement de valeur

Règle 1

- Pas de valeurs triviales pour des valeurs décisionnelles critiques
- Valeurs avec des distances de Hamming élevées

Règle 2

- Détecter les valeurs dans les zones blanches

Corollaires

- Deux valeurs Booléennes non triviales
- Pas de comportement par défaut

```
#define TRUE  0xA596
#define FALSE 0x5A69

if (check == TRUE)
    // cas TRUE

if (check == FALSE)
    // cas FALSE

// zone blanche douteuse
fault_handler()
```

Redondance à gros grain

Principe général

- Duplication
- Vérification de la cohérence des résultats

Efficacité

- Détection de plusieurs erreurs non identiques

Règles

- Préférer
  - $f^{-1}(f(input)) == input$
  - $f_{implem1}(input) == f_{implem2}(input)$

à $f(input) == f(input)$

- Protéger le test de cohérence

```
encode(pt, key, ct)
encode(pt, key, ct_dup)

if (memcmp(ct, ct_dup) != 0){
// réponse arbitraire, effacement clé, …
   fault_handler();
}
// suite…
```

```
encode(pt, key, ct)
decode(ct, key, text)

if (memcmp(pt, text) != 0){
// réponse arbitraire, effacement clé, …
   fault_handler();
}
// suite…
```

Redondance à grain fin

Protection des tests

- Duplication simple
- Test complémentaire

```
if (cond){
    // sensitive op
}
```

Redondance à grain fin

Protection des tests

- Duplication simple
- Test complémentaire

```
if (cond){
   // sensitive op
}
```

```
if (cond){
   if (cond){
     // sensitive op
   }
   else {
     fault_handler();
   }
}
```

```
if (cond){
   if (!cond){
      fault_handler();
   }
   else {
      // sensitive op
   }
}
```

Redondance à grain fin

Protection des tests

- Duplication simple
- Test complémentaire

```
if (cond){
   // sensitive op
}
```

```
if (cond){
   if (cond){
     // sensitive op
   }
   else {
     fault_handler();
   }
}
```

```
if (cond){
   if (!cond){
      fault_handler();
   }
   else {
      // sensitive op
   }
}
```

Duplication du calcul de condition

- Détermination des opérations impliquées
- Dupliquer les opérations et les variables impliquées

```
a = … ;

cond = foo(a);

if (cond){

   // sensitive op
   }
```

Redondance à grain fin

Protection des tests

- Duplication simple
- Test complémentaire

```
if (cond){
   // sensitive op
}
```

```
if (cond){
   if (cond){
     // sensitive op
   }
   else {
     fault_handler();
   }
}
```

```
if (cond){
   if (!cond){
      fault_handler();
   }
   else {
      // sensitive op
   }
}
```

Duplication du calcul de condition

- Détermination des opérations impliquées
- Dupliquer les opérations et les variables impliquées

```
a = … ;

cond = foo(a);

if (cond){

   // sensitive op
   }
```

```
a = … ;
a_dup = … ;
cond = foo(a);
cond_dup = foo(a_dup);
if (cond){
   if (cond_dup){
      // sensitive op
   }
   else {
      fault_handler();
   }
}
```

Intégrité du flot de contrôle

Cas des attaques mémoire

- Besoin de protéger les transferts de flot de contrôle dépendant des entrées
  → sauts et appels de fonctions indirects
- De nombreux travaux autour du CFI
  → [Abadi et al, CCS 2005] [Burow et al, ACM Survey 2017]

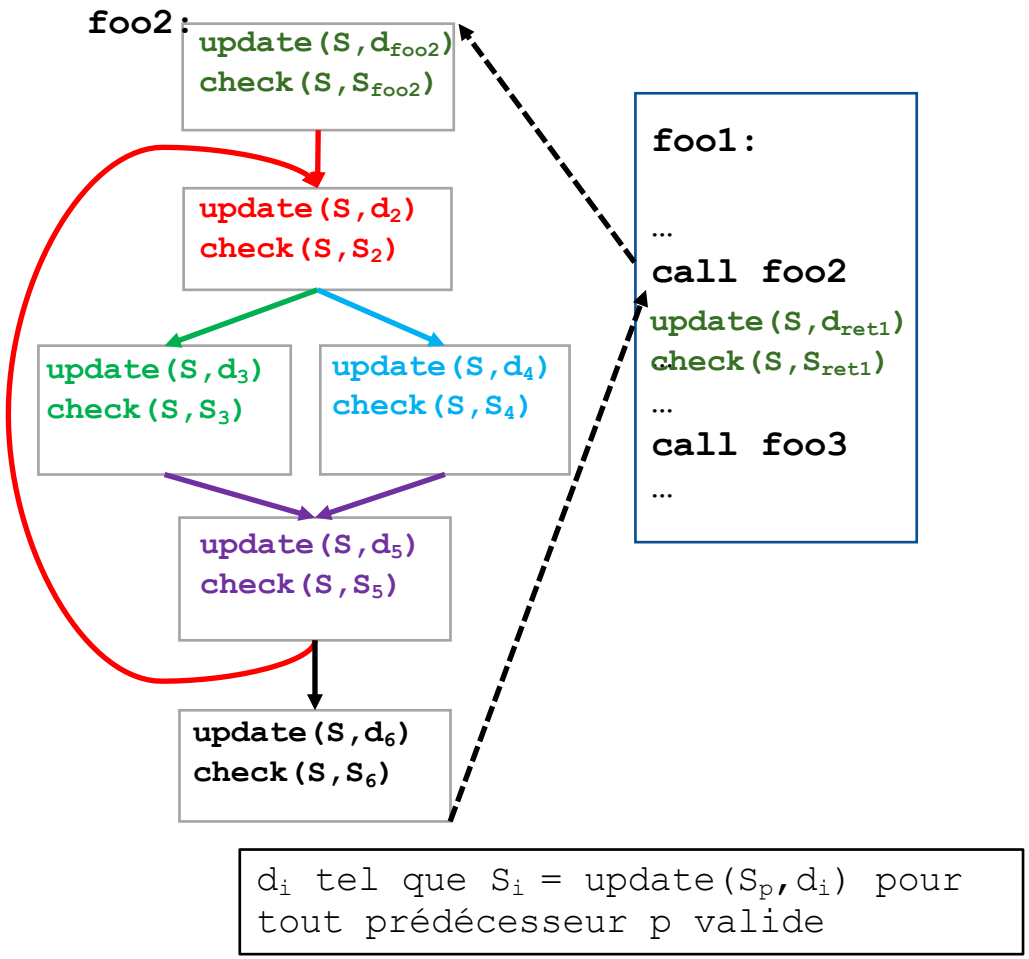
Cas des fautes matérielles

- Tous les sauts sont potentiellement à protéger
- Le code séquentiel aussi
- Premiers travaux pour la tolérance aux fautes proposés début 2000

Intégrité du flot de contrôle

Approche à base de signature [Oh et al. 2002] [Goloubeva et al., 2005]

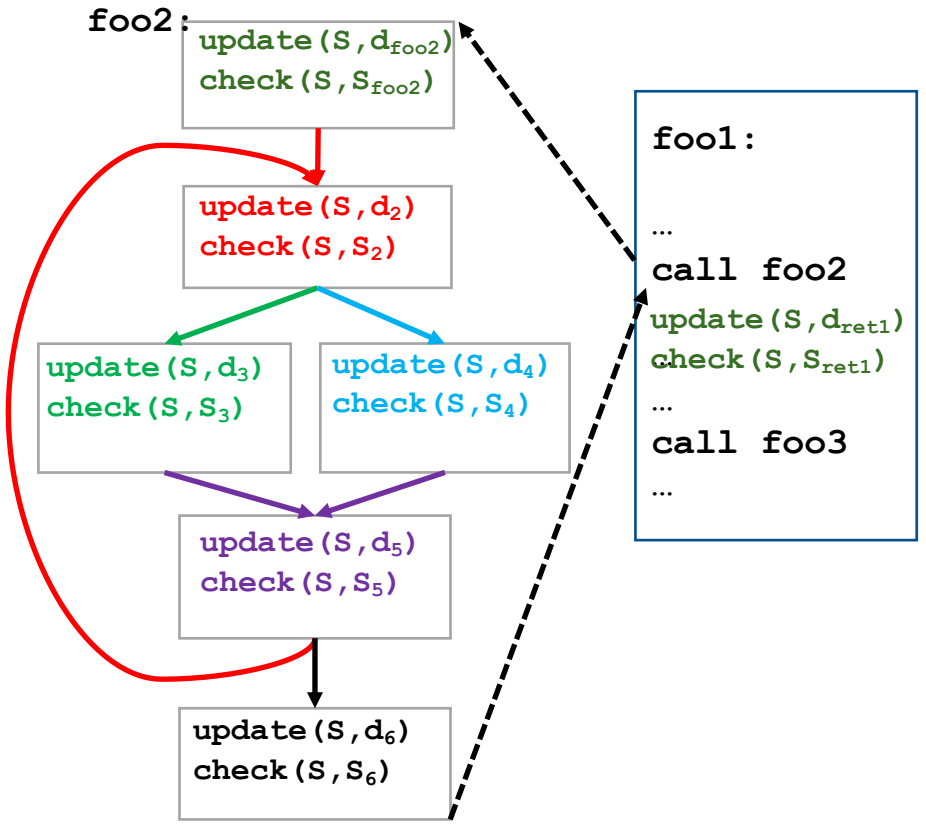
- Signature associée à chaque BB ou fonction
- Vérification au début de chaque bloc de base après mise à jour de la signature courante
- Intégrité du CFG



Intégrité du flot de contrôle

Approche à base de signature [Oh et al. 2002] [Goloubeva et al., 2005]

- Signature associée à chaque BB ou fonction
- Vérification au début de chaque bloc de base après mise à jour de la signature courante
- Intégrité du CFG
- Intégrité de la direction des sauts

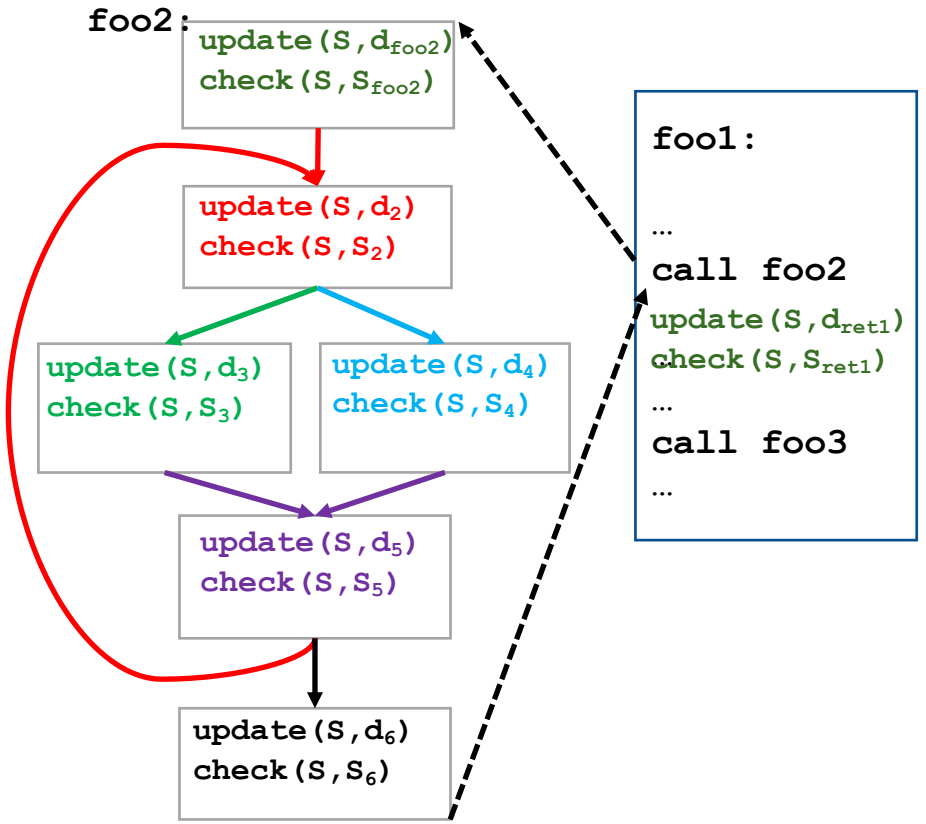


d_i tel que S_i = update(S_p, d_i) pour tout prédécesseur p valide

Intégrité du flot de contrôle

Approche à base de signature [Oh et al. 2002] [Goloubeva et al., 2005]

- Signature associée à chaque BB ou fonction
- Vérification au début de chaque bloc de base après mise à jour de la signature courante
- Intégrité du CFG
- Intégrité de la direction des sauts
  - prise en compte de la valeur de la condition dans la signature/mise à jour [SIED, 2003]

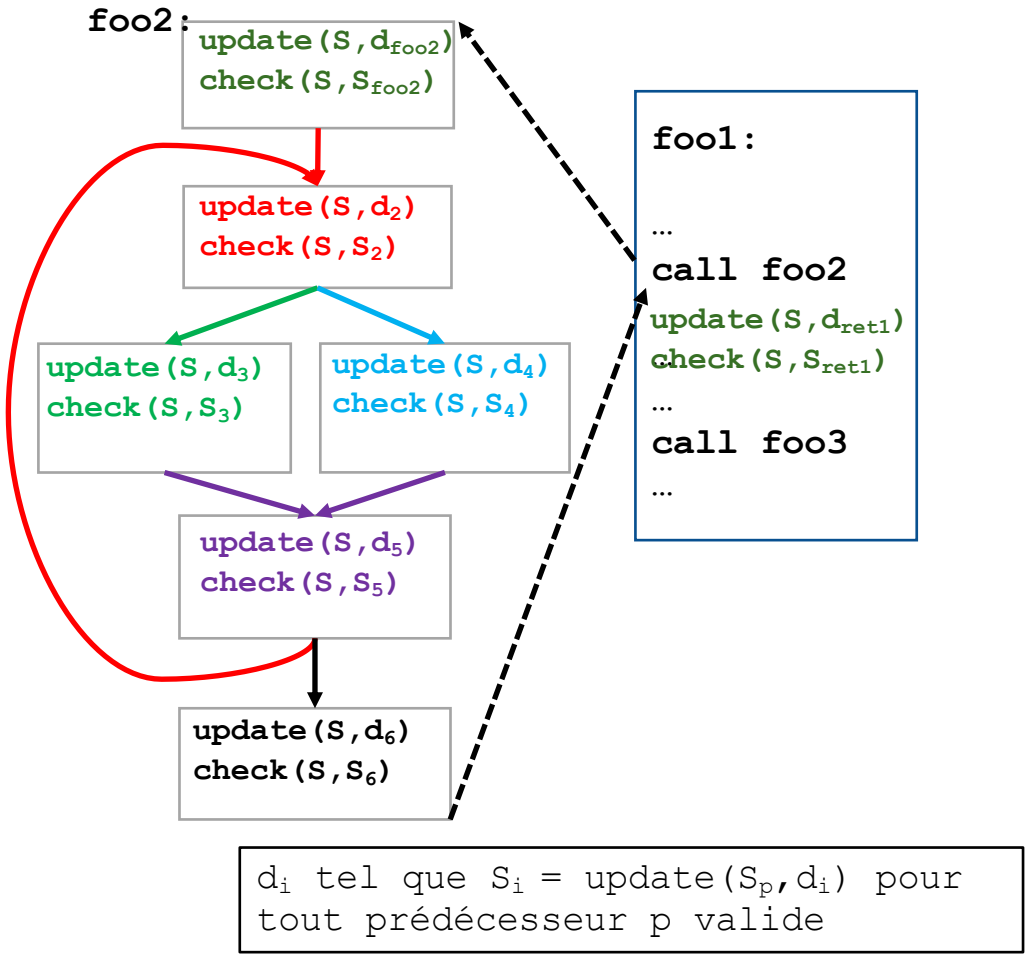


d_i tel que S_i = update(S_p, d_i) pour tout prédécesseur p valide

Intégrité du flot de contrôle

Approche à base de signature [Oh et al. 2002] [Goloubeva et al., 2005]

- Signature associée à chaque BB ou fonction
- Vérification au début de chaque bloc de base après mise à jour de la signature courante
- Intégrité du CFG
- Intégrité de la direction des sauts
  - prise en compte de la valeur de la condition dans la signature/mise à jour [SIED, 2003]
  - intégrité des données pour une couverture complète



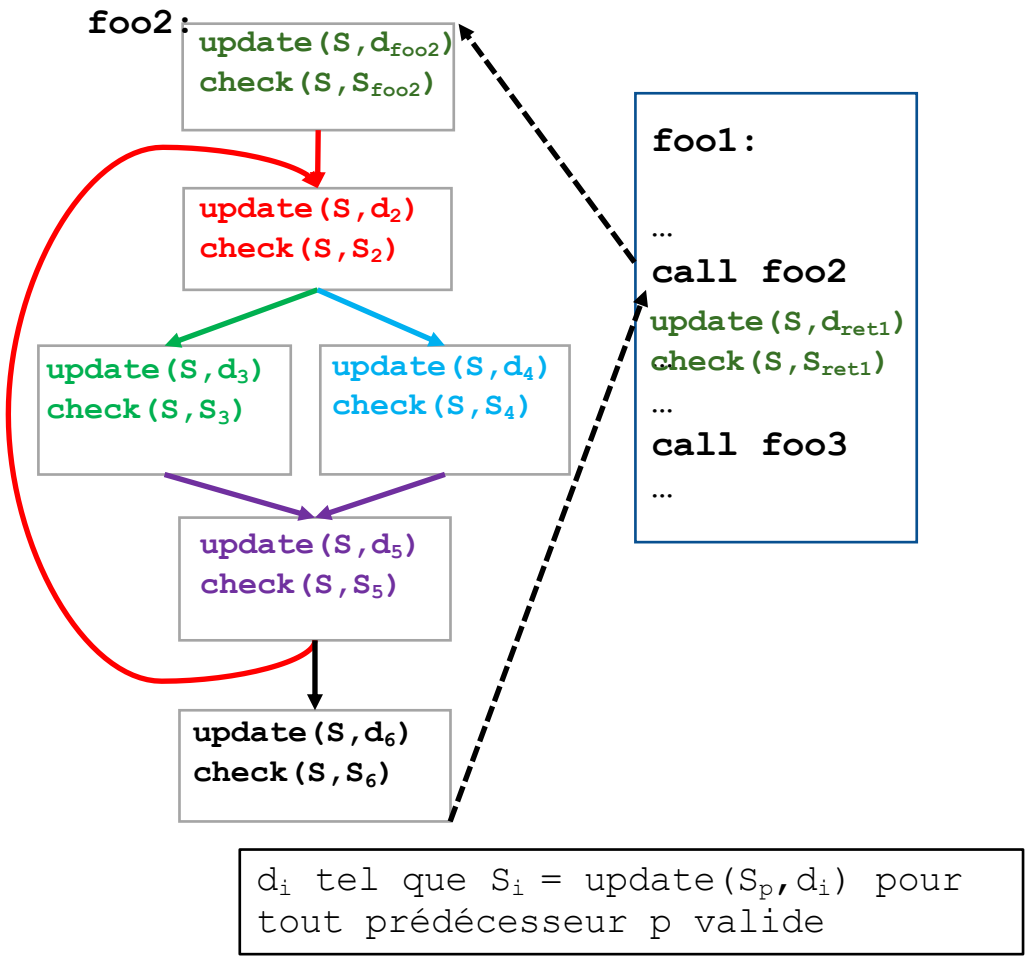
Intégrité du flot de contrôle

Approche à base de signature [Oh et al. 2002] [Goloubeva et al., 2005]

- Signature associée à chaque BB ou fonction
- Vérification au début de chaque bloc de base après mise à jour de la signature courante
- Intégrité du CFG
- Intégrité de la direction des sauts
  - prise en compte de la valeur de la condition dans la signature/mise à jour [SIED, 2003]
  - intégrité des données pour une couverture complète

Intégrité du flot séquentiel

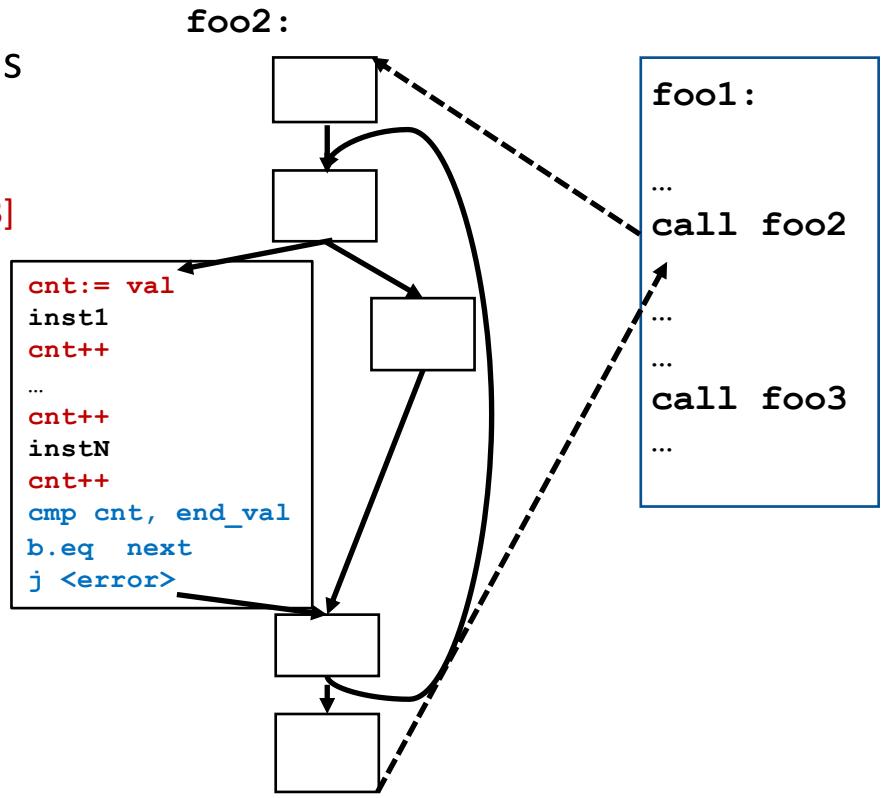
- Mise à jour de la signature tout au long d'un bloc



Suivi de l'exécution avec des traceurs

Protections à base de compteurs

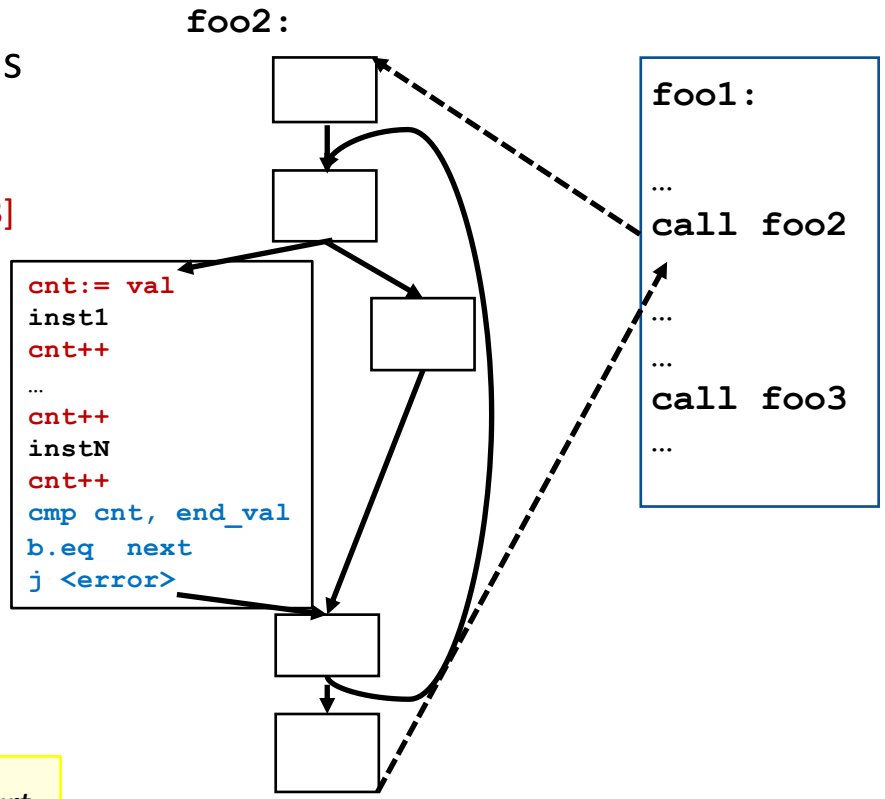
- Initialisation, mise à jour et vérification à des points précis du programme
- Protection de l'intégrité de code séquentiel [Akkar et al., 2003]
  - Initialisation au début d'un bloc séquentiel
  - Mise à jour tout au long du bloc
  - Vérification à la fin du bloc



Suivi de l'exécution avec des traceurs

Protections à base de compteurs

- Initialisation, mise à jour et vérification à des points précis du programme
- Protection de l'intégrité de code séquentiel [Akkar et al., 2003]
  - Initialisation au début d'un bloc séquentiel
  - Mise à jour tout au long du bloc
  - Vérification à la fin du bloc
- Extensible à la protection d'une fonction entière, appels de fonction inclus

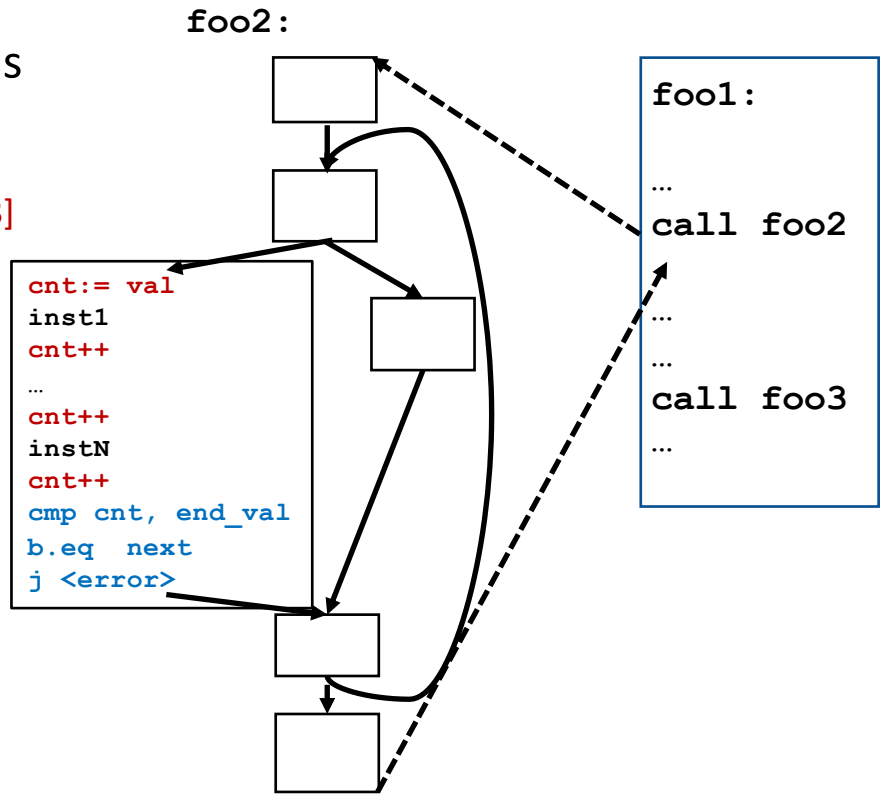


J-F. Lalande et al. *Software countermeasures for control flow integrity of smart card C codes.* ESORICS 2014.

Suivi de l'exécution avec des traceurs

Protections à base de compteurs

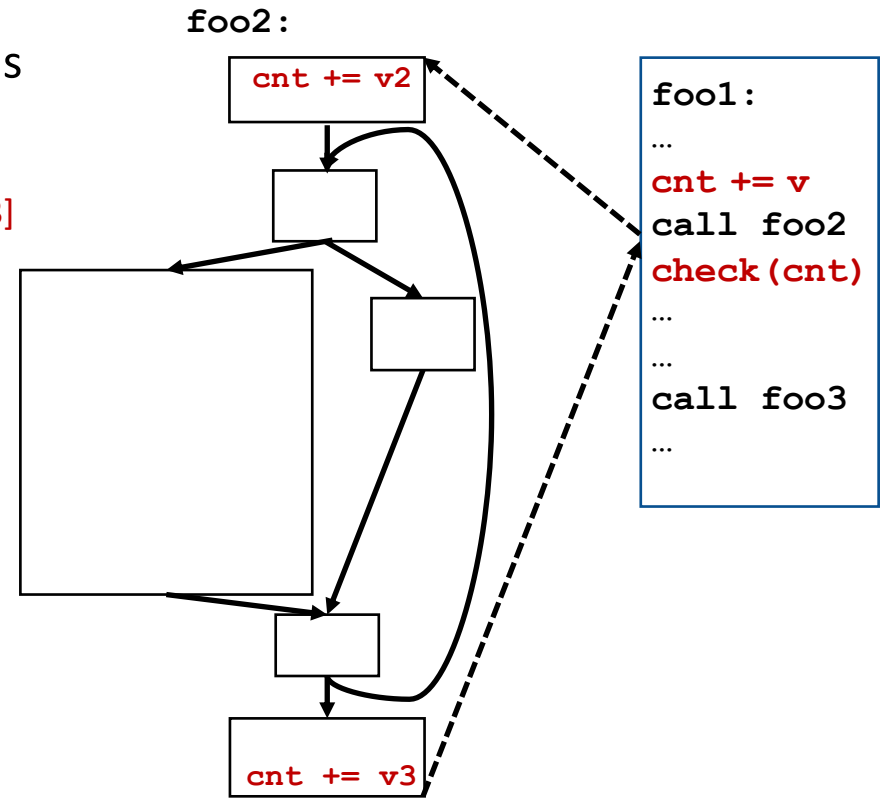
- Initialisation, mise à jour et vérification à des points précis du programme
- Protection de l'intégrité de code séquentiel [Akkar et al., 2003]
  - Initialisation au début d'un bloc séquentiel
  - Mise à jour tout au long du bloc
  - Vérification à la fin du bloc
- Extensible à la protection d'une fonction entière, appels de fonction inclus
- Alternative à la duplication de code quand impossible (code avec effet de bord)



Suivi de l'exécution avec des traceurs

Protections à base de compteurs

- Initialisation, mise à jour et vérification à des points précis du programme
- Protection de l'intégrité de code séquentiel [Akkar et al., 2003]
  - Initialisation au début d'un bloc séquentiel
  - Mise à jour tout au long du bloc
  - Vérification à la fin du bloc
- Extensible à la protection d'une fonction entière, appels de fonction inclus
- Alternative à la duplication de code quand impossible (code avec effet de bord)
- Version "allégée", peu chère : points de passage



Résumé sur les protections logicielles

- Des principes relativement simples
  - Durcissement de valeurs
  - Redondance de données (complémentaires ou masquées)
  - Duplication de tests, de calculs
  - Mécanisme à base de traceurs ou signatures
  - Et des vérifications de cohérence
- Difficulté
  - Mise en œuvre efficace complexe
  - Quelles protections sur quelles parties du code ?
- En pratique
  - Guidée par un modèle de menace qui dépend du produit final

Exemple classique

```
#define FALSE 0
#define TRUE  1

int verifyPIN(char *cardPin, char *userPin, unsigned size){
  unsigned i;

  /********** Comparaison **********/
  for (i = 0; i < size ; i++)
      if (userPin[i] != cardPin[i])
         return FALSE;

  return TRUE;
}
```

Exemple classique amélioré

```
#define FALSE 0
#define TRUE  1

int verifyPIN(char *cardPin, char *userPin, unsigned size){
  unsigned i;
  unsigned diff = 0;
  /********** Comparaison **********/
  for (i = 0; i < size ; i++)
      diff |= userPin[i] ^ cardPin[i]); // temps constant

  if (diff == 0)      // codes identiques
      return TRUE;

  return FALSE;       // codes différents
}
```

Exemple classique amélioré

```
#define FALSE 0
#define TRUE  1

int verifyPIN(char *cardPin, char *userPin, unsigned size){
  unsigned i;
  unsigned diff = 0;
  /********** Comparaison **********/
  for (i = 0; i < size ; i++)
      diff |= userPin[i] ^ cardPin[i]); // temps constant

  if (diff == 0)      // codes identiques
      return TRUE;

  return FALSE;       // codes différents
}
```

Objectif d'un attaquant : s'authentifier malgré un **`userPin`** non valide

Exemple classique amélioré

```
#define FALSE 0
#define TRUE  1

int verifyPIN(char *cardPin, char *userPin, unsigned size){
  unsigned i;
  unsigned diff = 0;
  /********** Comparaison **********/
  for (i = 0; i < size ; i++)
      diff |= userPin[i] ^ cardPin[i]); // temps constant

  if (diff == 0)      // codes identiques
      return TRUE;

  return FALSE;       // codes différents
}
```

Objectif d'un attaquant : s'authentifier malgré un **`userPin`** non valide

Possibilités :

- Corruption de la boucle (0 itération)
- Corruption de la vérification
- Corruption de la valeur finale

Protection de l'exemple

```
#define FALSE 0xA5 // valeurs spécifiques pour les Booléens
#define TRUE  0x5A

int verifyPIN(char *cardPin, char *userPin, unsigned size){
  unsigned i;
  unsigned diff = 0;
  /********** Comparaison **********/
  for (i = 0; i < size ; i++)
      diff |= userPin[i] ^ cardPin[i]); // temps constant

  if (diff == 0)      // codes identiques
      return TRUE;

  return FALSE;       // codes différents
}
```

Objectif d'un attaquant : s'authentifier malgré un **`userPin`** non valide

Possibilités :

- Corruption de la boucle (0 itération)
- Corruption de la vérification
- Corruption de la valeur finale

Protection de l'exemple

```
#define FALSE 0xA5 // valeurs spécifiques pour les Booléens
#define TRUE  0x5A

int verifyPIN(char *cardPin, char *userPin, unsigned size){
  unsigned i, j = 0; // variable d'itération redondante
  unsigned diff = 0;
  /********** Comparaison **********/
  for (i = 0; i < size ; i++, j++)
      diff |= userPin[i] ^ cardPin[i]); // temps constant

  if (j < size) error()

  if (diff == 0)      // codes identiques
      return TRUE;

  return FALSE;       // codes différents
}
```

Objectif d'un attaquant : s'authentifier malgré un **`userPin`** non valide

Possibilités :

- Corruption de la boucle (0 itération)
- Corruption de la vérification
- Corruption de la valeur finale

Protection de l'exemple

```
#define FALSE 0xA5 // valeurs spécifiques pour les Booléens
#define TRUE  0x5A

int verifyPIN(char *cardPin, char *userPin, unsigned size){
  unsigned i, j = 0; // variable d'itération redondante
  unsigned diff = 0;
  /********** Comparaison **********/
  for (i = 0; i < size ; i++, j++)
      diff |= userPin[i] ^ cardPin[i]); // temps constant

  if (j < size) error();

  if (diff == 0)      // codes identiques
  {
      if (diff != 0) // test redondant
          error();
      return TRUE;
  }
  return FALSE;       // codes différents
}
```

Objectif d'un attaquant : s'authentifier malgré un **`userPin`** non valide

Possibilités :

- Corruption de la boucle (0 itération)
- Corruption de la vérification
- Corruption de la valeur finale

Problématique liée à la compilation

Code source

Code binaire

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

Problématique liée à la compilation

Code source

Code binaire

```
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();
```

Problématique liée à la compilation

Code source

```
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
```

Code binaire

Problématique liée à la compilation

Code source

```
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();
```

Code binaire

Problématique liée à la compilation


```
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();
```

Problématique liée à la compilation

Compilation/optimisation

Code source

```
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();
```

Code binaire

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

Problématique liée à la compilation

Compilation/optimisation

Code source

Code binaire

```
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);
}
```

Problématique liée à la compilation

Compilation/optimisation

Code source

```
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();
```

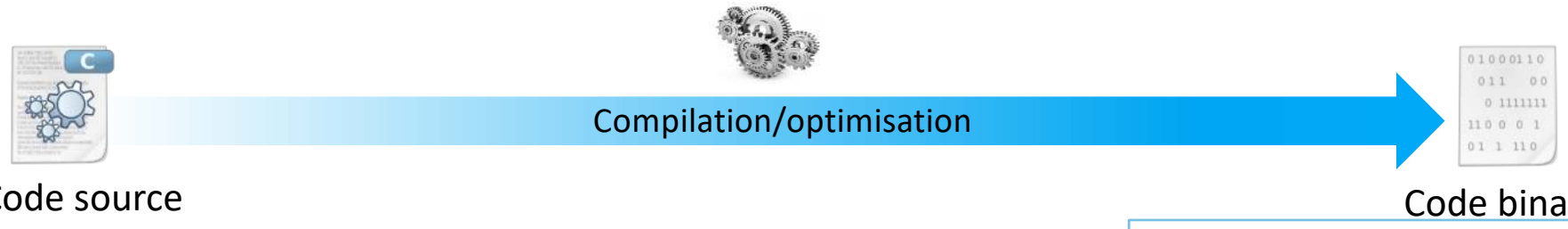
i

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

Code binaire

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

Problématique liée à l'écart code source - binaire



Code source

```
cond = *check;
if (cond)
  {
   // sensitive op
  }
else
  {
   fault_handler();
  }
```

Code binaire

```
      ldr r3, [r0] ; r3 := *check
      cmp r3, #0   ; r3 == 0 ?
      beq else     ; saut si r3 == 0
then:
      …            ; sensitive op
      j next
else:
      bl fault_handler
next:
```

Problématique liée à l'écart code source - binaire

Code source

Code binaire

```
cond = *check;
if (cond)
  {
   // sensitive op
  }
else
  {
   fault_handler();
  }
```

```
     ldr r3, [r0] ; r3 := *check
     cmp r3, #0   ; r3 == 0 ?
     beq else     ; saut si r3 == 0
then:
     …            ; sensitive op
     j next
else:
     bl fault_handler
next:
```

```
     ldr r3, [r0] ; r3 := *check
     cmp r3, #0   ; r3 == 0 ?
     b.ne then    ; saut si r3 != 0
else:
     bl fault_handler
     j next
then:
     …            ; sensitive op
next:
```

Problématique liée à l'écart code source - binaire

Code source

Code binaire

```
cond = *check;
if (cond)
  {
   // sensitive op
  }
else
  {
   fault_handler();
  }
```

- **Le code attaqué est le code binaire**
- **Sa vulnérabilité dépend**
  - sélection d'instructions
  - ordonnancement des instructions
  - allocation de registre
  - placement du code et des données
  - de la cible
  - des capacités de l'attaquant !

```
      ldr r3, [r0] ; r3 := *check
      cmp r3, #0   ; r3 == 0 ?
      beq else     ; saut si r3 == 0
then:
      …            ; sensitive op
      j next
else:
      bl fault_handler
next:
```

```
      ldr r3, [r0] ; r3 := *check
      cmp r3, #0   ; r3 == 0 ?
      b.ne then    ; saut si r3 != 0
else:
      bl fault_handler
      j next
then:
      …            ; sensitive op
next:
```

Protections logicielles et compilation

Quelles solutions ?

- Ne pas optimiser

  → code lent, code plus volumineux = surface d'attaque plus grande

- Utiliser des « astuces » , déclarer `volatile` les données redondantes

  → fragile, le standard et les optimisations évoluent

- Appliquer les protections sur le code assembleur

  → sur le code final cible des attaques, mais non portable

  → besoin de retrouver les éléments à protéger

  → besoin de registres disponibles, place mémoire : défaire/refaire une partie de la compilation

- Déployer les protections à la compilation

  → ?

- Rendre le compilateur et ses optimisations « compatibles »

  → préservation et traçabilité, ?

Protections logicielles et compilation

Quelles solutions ?

- Ne pas optimiser

  → code lent, code plus volumineux = surface d'attaque plus grande

- Utiliser des « astuces » , déclarer `volatile` les données redondantes

  → fragile, le standard et les optimisations évoluent

- Appliquer les protections sur le code assembleur

  → sur le code final cible des attaques, mais non portable

  → besoin de retrouver les éléments à protéger

  → besoin de registres disponibles, place mémoire : défaire/refaire une partie de la compilation

- **Déployer les protections à la compilation**

  → ?

- Rendre le compilateur et ses optimisations « compatibles »

  → préservation et traçabilité, ?

Protection des boucles à la compilation

Motivation

- Plusieurs attaques reposent sur la **corruption du nombre d'itérations**
  - Débordement de tampon (*buffer overflow*)
  - Cryptanalyse (*round reduction)*
  - Contournement d'un contrôle d'accès (comparaison de signature, de code PIN, etc.)

Objectif

- Garantir le bon nombre d'itérations et la bonne sortie, ou détecter une erreur

Modèle d'attaquant

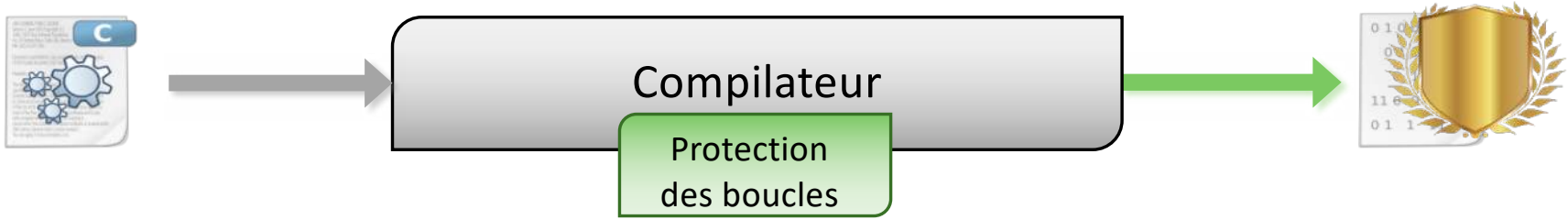
- Un « saut d'instruction » ou une « corruption de registre » lors l'exécution d'une boucle
- Raison : double fautes souvent à deux moments bien différents de l'exécution pour arriver à un « gain »

Protection des boucles à la compilation

J. Proy

PhD 2019

+annotations



Objectif : garantir le bon nombre d'itérations et la bonne sortie

Principe : dupliquer les opérations et variables* impliquées dans le calcul de la condition de sortie, ajouter des vérifications

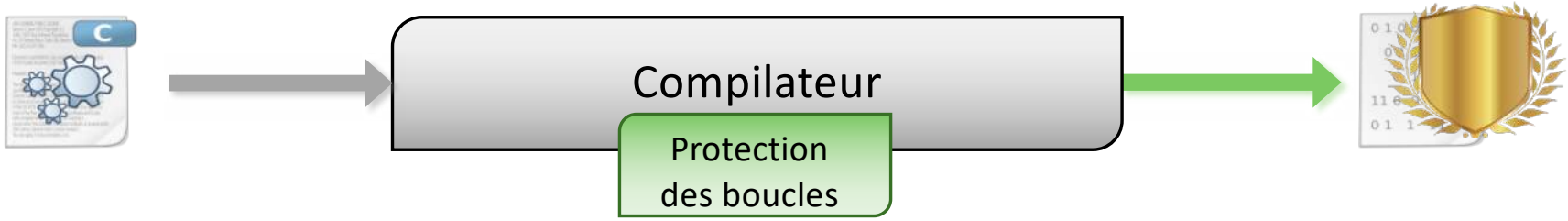
```
unsigned i;
#pragma sensitive_loop
for (i=0; i<size; ){

    foo(i);
    if (cond)
        {i+=1;}
    else
        {i+=2;}
}
```

Protection des boucles à la compilation

J. Proy

PhD 2019

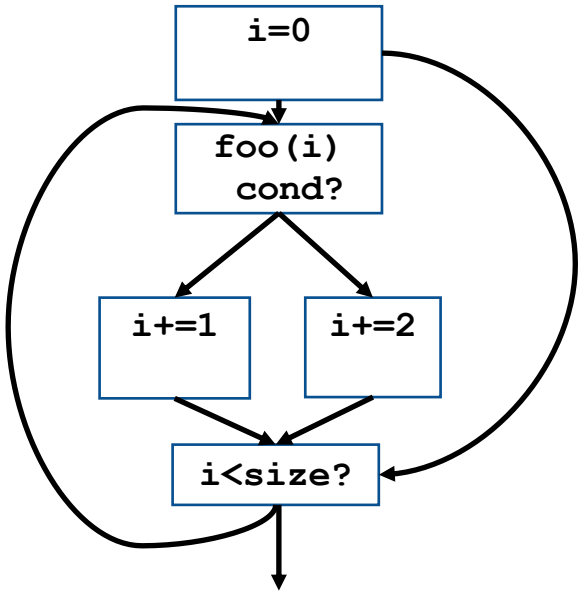
+annotations



Objectif : garantir le bon nombre d'itérations et la bonne sortie

Principe : dupliquer les opérations et variables* impliquées dans le calcul de la condition de sortie, ajouter des vérifications

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

     foo(i);
     if (cond)
          {i+=1;}
     else
          {i+=2;}
}
```



Protection des boucles à la compilation

J. Proy

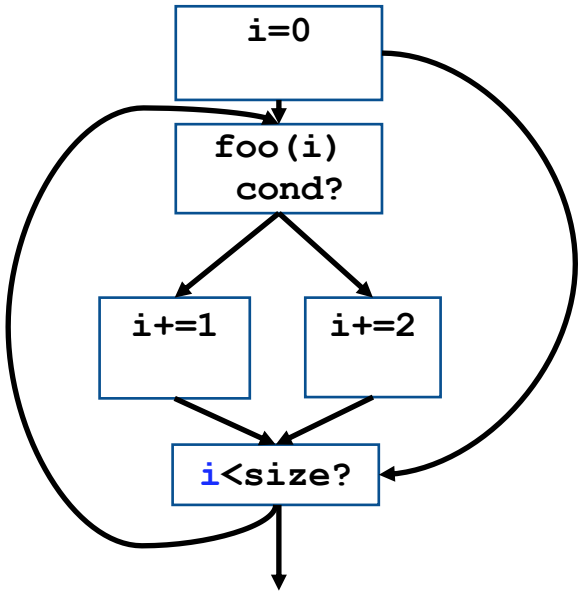
PhD 2019

+annotations

Objectif : garantir le bon nombre d'itérations et la bonne sortie

Principe : dupliquer les opérations et variables* impliquées dans le calcul de la condition de sortie, ajouter des vérifications

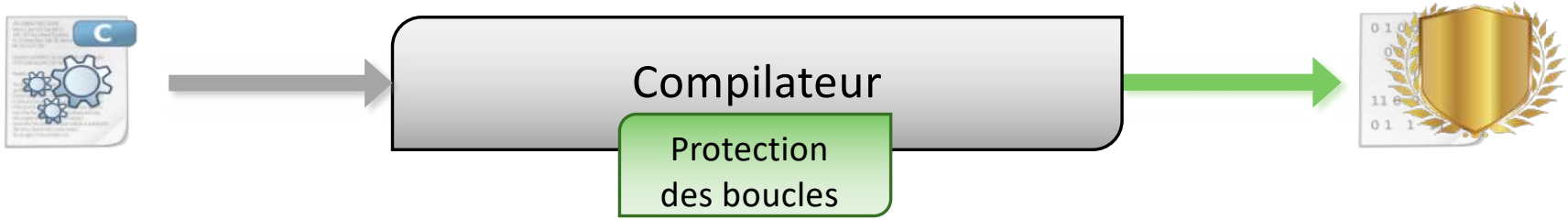
```
unsigned i;
#pragma sensitive_loop
for (i=0; i<size; ){

     foo(i);
     if (cond)
          {i+=1;}
     else
          {i+=2;}
}
```



Protection des boucles à la compilation

J. Proy
PhD 2019

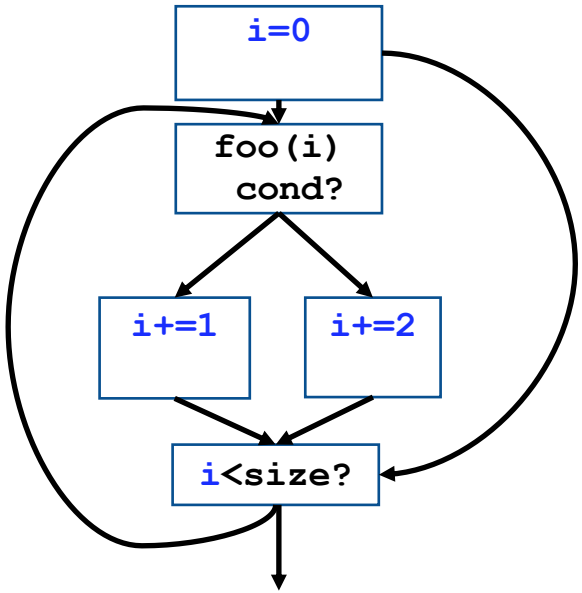
+annotations



Objectif : garantir le bon nombre d'itérations et la bonne sortie

Principe : dupliquer les opérations et variables* impliquées dans le calcul de la condition de sortie, ajouter des vérifications

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

     foo(i);
     if (cond)
          {i+=1;}
     else
          {i+=2;}
}
```

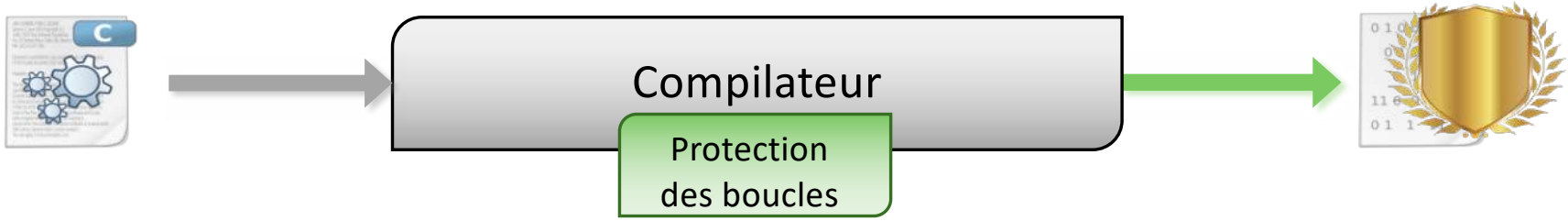


Protection des boucles à la compilation

J. Proy

PhD 2019

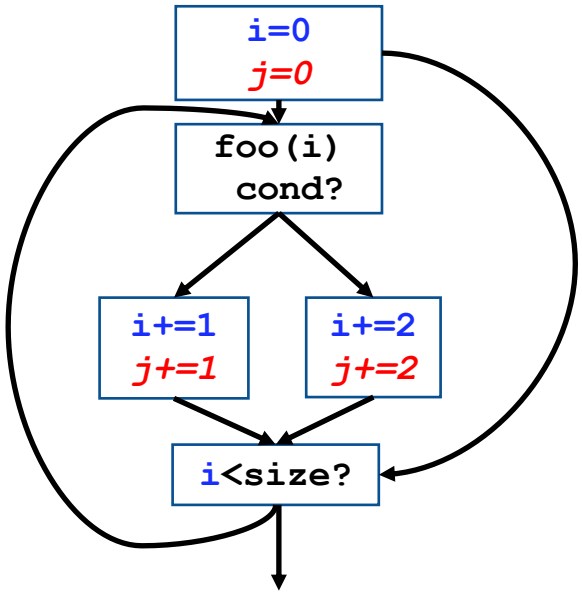
+annotations



Objectif : garantir le bon nombre d'itérations et la bonne sortie

Principe : dupliquer les opérations et variables* impliquées dans le calcul de la condition de sortie, ajouter des vérifications

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

     foo(i);
     if (cond)
          {i+=1;}
     else
          {i+=2;}
}
```

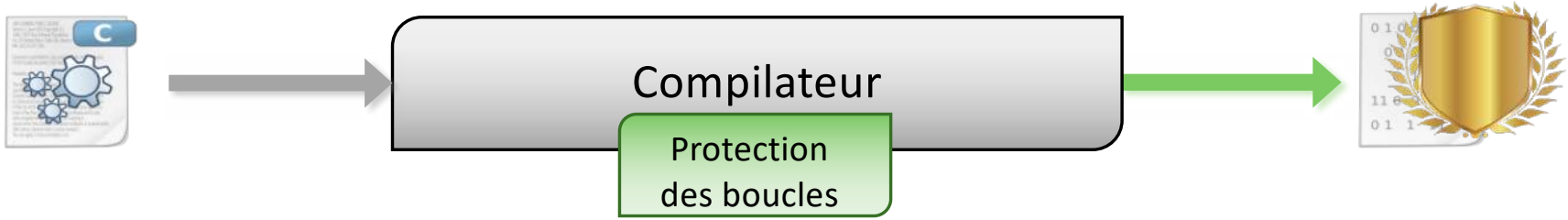


Protection des boucles à la compilation

J. Proy

PhD 2019

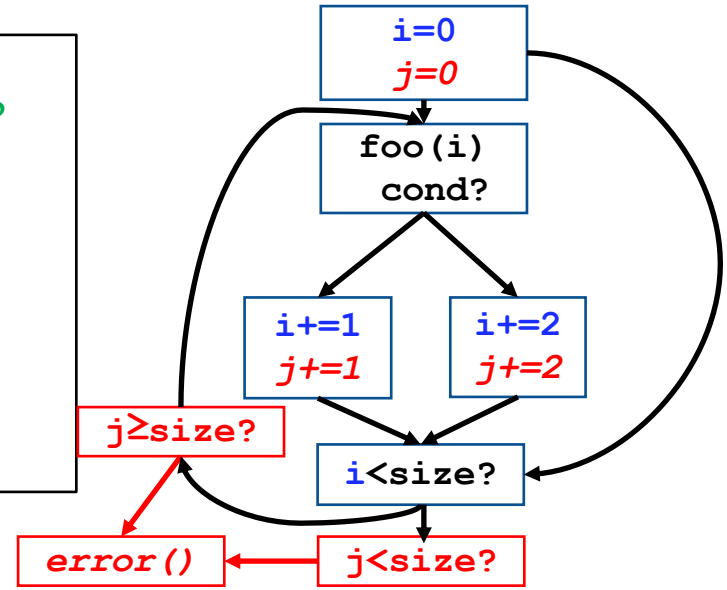
+annotations



Objectif : garantir le bon nombre d'itérations et la bonne sortie

Principe : dupliquer les opérations et variables* impliquées dans le calcul de la condition de sortie, ajouter des vérifications

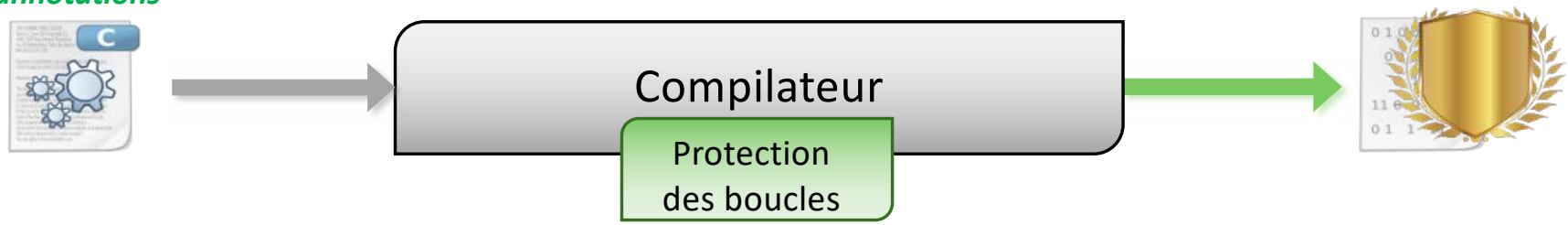
```
unsigned i;
#pragma sensitive_loop
for (i=0; i<size; ){

     foo(i);
     if (cond)
          {i+=1;}
     else
          {i+=2;}
}
```



Protection des boucles à la compilation

J. Proy
PhD 2019

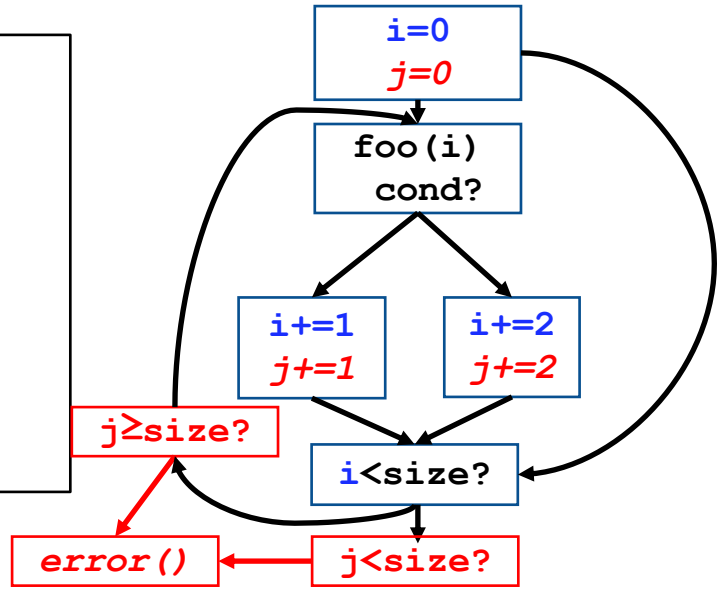
+annotations



Objectif : garantir le bon nombre d'itérations et la bonne sortie

Principe : dupliquer les opérations et variables* impliquées dans le calcul de la condition de sortie, ajouter des vérifications

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

     foo(i);
     if (cond)
          {i+=1;}
     else
          {i+=2;}
}
```




```
unsigned i, j=0;

for (i=0; i<size; ){
     if (j>=size) error();
     foo(i);
     if (cond)
          {i+=1; j+=1;}
     else
          {i+=2; j+=2;}
}
if (j<size) error();
```

Protection des boucles à la compilation

J. Proy

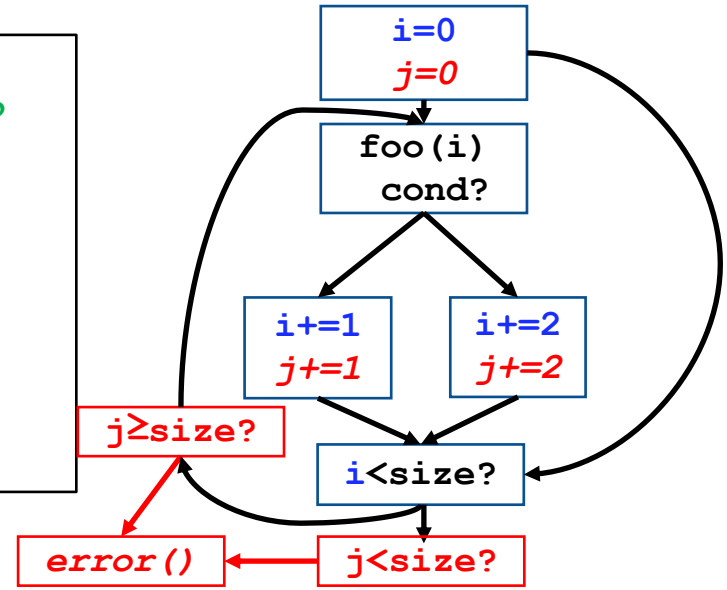
PhD 2019

+annotations

Objectif : garantir le bon nombre d'itérations et la bonne sortie

Principe : dupliquer les opérations et variables* impliquées dans le calcul de la condition de sortie, ajouter des vérifications

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

     foo(i);
     if (cond)
          {i+=1;}
     else
          {i+=2;}
}
```

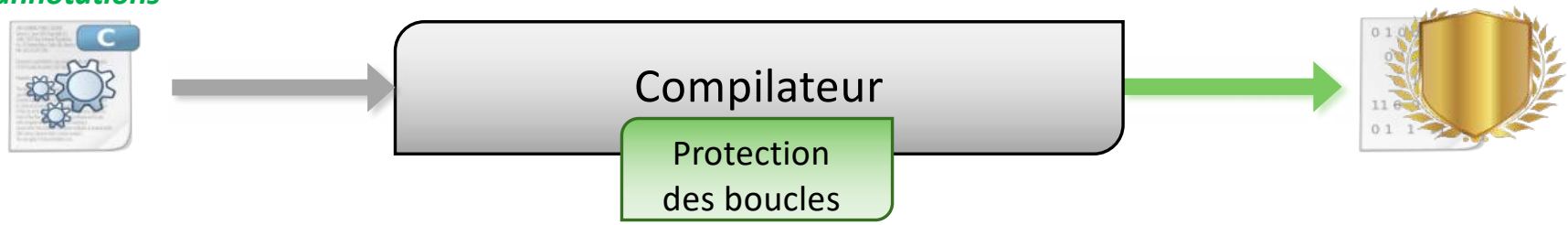



```
unsigned i, j=0;

for (i=0; i<size; ){
     if (j>=size) error();
     foo(i);
     if (cond)
          {i+=1; j+=1;}
     else
          {i+=2; j+=2;}
}
if (j<size) error();
```

Protection des boucles à la compilation

J. Proy
PhD 2019

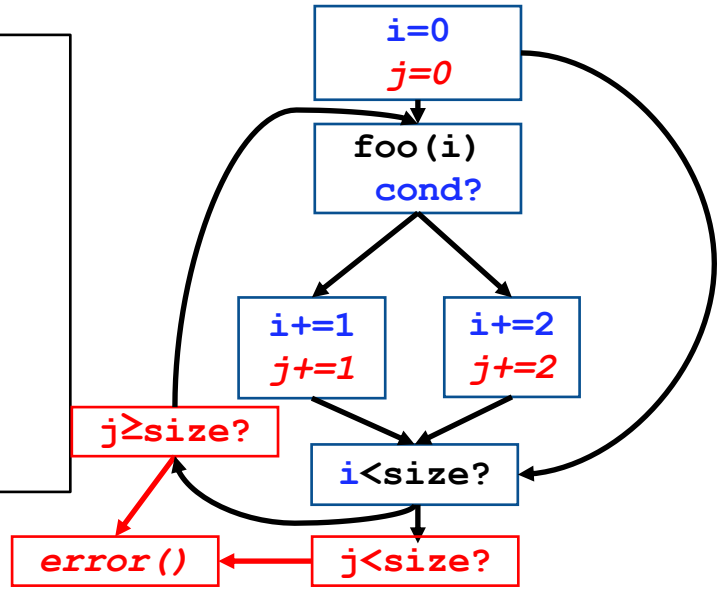
+annotations



Objectif : garantir le bon nombre d'itérations et la bonne sortie

Principe : dupliquer les opérations et variables* impliquées dans le calcul de la condition de sortie,
et dans les conditions qui influencent la condition de sortie, ajouter des vérifications

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

     foo(i);
     if (cond)
          {i+=1;}
     else
          {i+=2;}
}
```



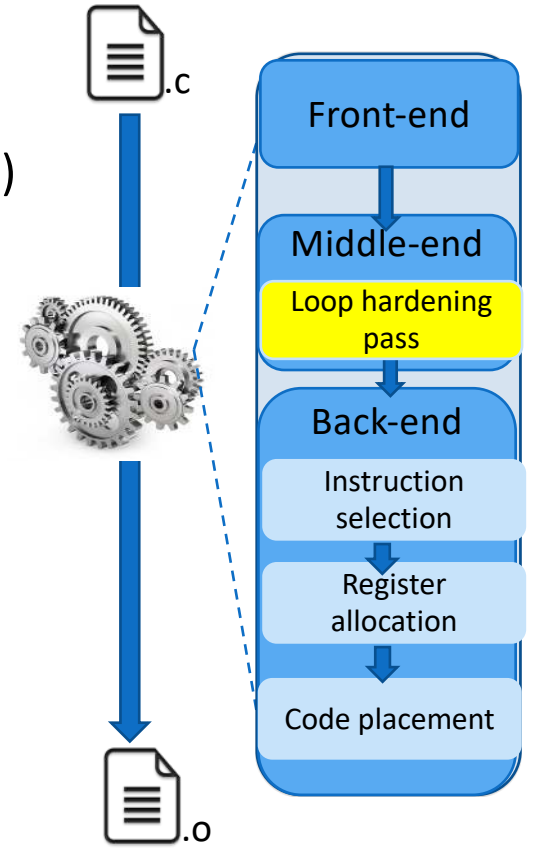

```
unsigned i, j=0;

for (i=0; i<size; ){
     if (j>=size) error();
     foo(i);
     if (cond)
          {i+=1; j+=1;}
     else
          {i+=2; j+=2;}
}
if (j<size) error();
```

Protection des boucles à la compilation

J. Proy

PhD 2019

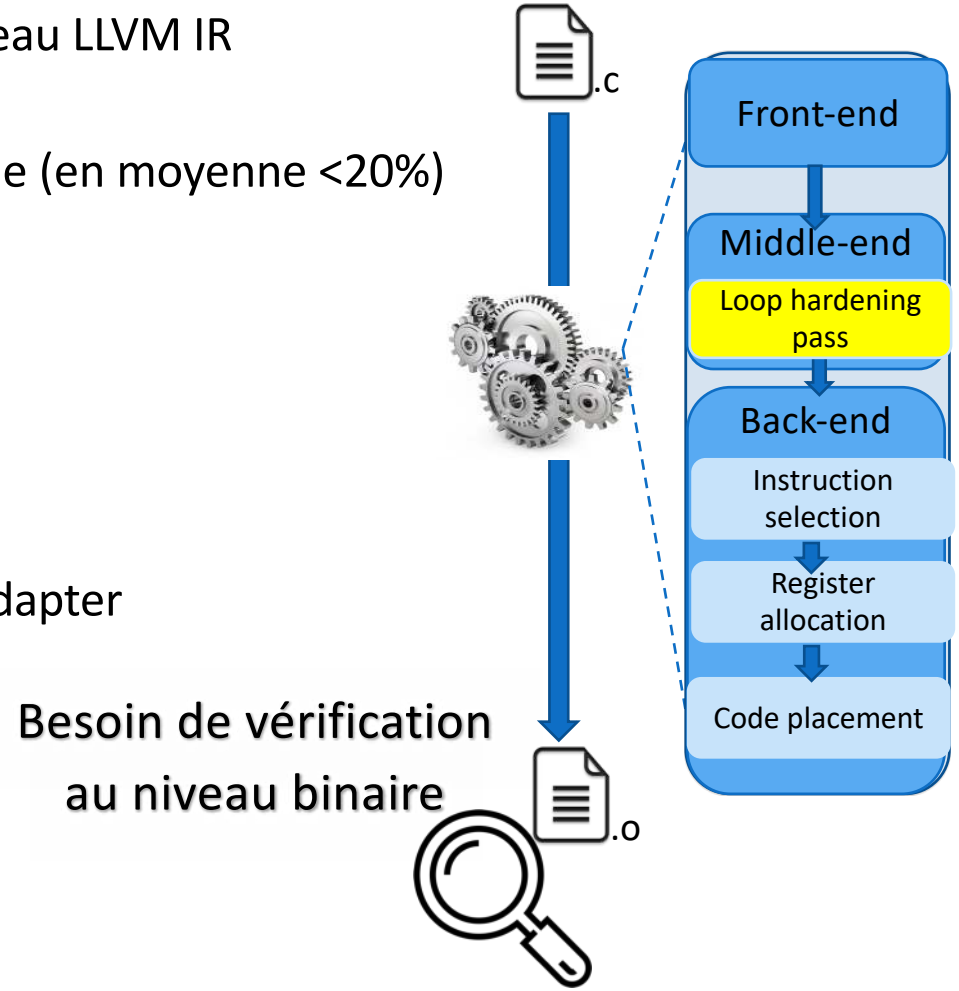
- Implémenté dans le compilateur Clang/LLVM au niveau LLVM IR
- Impact limité sur les performances et la taille de code (en moyenne <20%)
- **Simulation de faute : détection de 99% des fautes**
- **1% de faute non détectée**
  - Passe d'optimisations post-sécurisation
  - Besoin de les désactiver (si possible) ou de les adapter



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

Protection des boucles à la compilation

- Implémenté dans le compilateur Clang/LLVM au niveau LLVM IR
- Impact limité sur les performances et la taille de code (en moyenne <20%)
- **Simulation de faute : détection de 99% des fautes**
- **1% de faute non détectée**
  - Passe d'optimisations post-sécurisation
  - Besoin de les désactiver (si possible) ou de les adapter



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

Conclusion

Injection de fautes

- Puissantes et dangereuses
- Nombreuses exploitations possibles

Protections logicielles

- Des principes simples, une mise en œuvre difficile
- Problème du flot de compilation et de ses optimisations
- Renforcement de code possible à la compilation
- Problème de l'écart code source – code binaire
- Analyses du code binaire requises pour détecter les vulnérabilités restantes

Tendances actuelles et questions ouvertes

Attaques par injection de fautes

- Les moyens d'injection se démocratisent : Riscure[1], NewAE[2], [Kelly 2020]
- Les moyens d'injection se perfectionnent : de multiples fautes en une seule injection, de quelques instructions jusqu'à 100 instructions non exécutées [Dutertre 2019, Menu 2020, Claudepierre 2021]
  - « NOP-oriented programming » [Péneau 2020]
- Cibles complexes non protégées et aussi vulnérables [Proy 2019][Trouchkine 2019a] [Trouchkine 2019b]
- Fautes dans la micro-architecture mal comprises, oubliées ? [Laurent 2019]
- Vecteur d'attaques une fois les vulnérabilités logicielles éliminées : menace plus large encore à moyen termes ?

Protections

- Complexité, multiplicité et diversité des fautes à prendre en compte
- Besoin de flot de compilation appliquant les protections, préservant les protections [Vu 2020][Vu 2021]
- Approche combinée matérielle et logicielle [Chamelot 2022]
- Méthodes et outils pour la vérification de la robustesse du code binaire

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

Références bibliographiques

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

[Chamelot 2022] T. Chamelot et al. SFI-FI: Control Signal, Code and Control Flow Integrity Against Fault Injection Attacks. DATE 2022

[Claudepierre 2021] L. Claudepierre et al. TRAITOR: A Low-Cost Evaluation Platform for Multifault Injection. ASSS '21:

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

[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.

[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

Références bibliographiques

[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

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

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

[Vu 2021] S. T. Vu et al. Reconciling optimizations with secure compilation. OOPSLA 2021.