COMPILATION POUR L'APPLICATION DE CONTRE-MESURES CONTRE LES ATTAQUES PAR CANAL AUXILIAIRE

Nicolas BELLEVILLE

21 novembre 2019

Thèse dirigée par :

Henri-Pierre CHARLES, Directeur de Recherche, CEA

Karine HEYDEMANN, Maître de Conférences,
Sorbonne Université
Laboratoire d'Informatique de Paris 6

Et encadrée par :

Damien COUROUSSÉ, Ingénieur de Recherche, CEA

SOMMAIRE

1. Introduction et contexte

2. Niveaux d’application de contre-mesures

3. Application d'une contre-mesure de masquage booléen d'ordre 1 à la compilation

4. Application d'une contre-mesure de polymorphisme de code à la compilation

5. Conclusion et perspectives

SOMMAIRE

1. Introduction et contexte

2. Niveaux d’application de contre-mesures

3. Application d'une contre-mesure de masquage booléen d'ordre 1 à la compilation

4. Application d'une contre-mesure de polymorphisme de code à la compilation

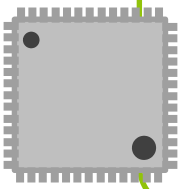
5. Conclusion et perspectives

CONTEXTE : SYSTÈMES EMBARQUÉS ET OBJETS CONNECTÉS

- **Cible de choix pour un attaquant**
  - Manipulent des données sensibles
    - Votre position
    - Des messages privés
    - Des données de santé
  - Permettent de monter des attaques de plus grande envergure
    - Attaques par déni de service
- **Sont de plus en plus répandus**
  - En 2008, 230 puces embarquées utilisées chaque jour par personne en moyenne [Sifakis, 2011]
- **Sont accessibles physiquement par un attaquant**
  - Permet à l'attaquant à recourir aux attaques par canal auxiliaire
    - Permet de retrouver des données sensibles
    - Attaques très efficaces pour retrouver des clés de chiffrement

Besoin important de sécurisation efficace contre ces attaques !

ATTAQUES PAR CANAL AUXILIAIRE

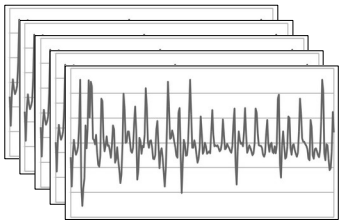


Instructions exécutées
Données manipulées

impacte

Émissions électromagnétiques
Consommation d'énergie

1. mesure



2. tente de retrouver

Attaquant

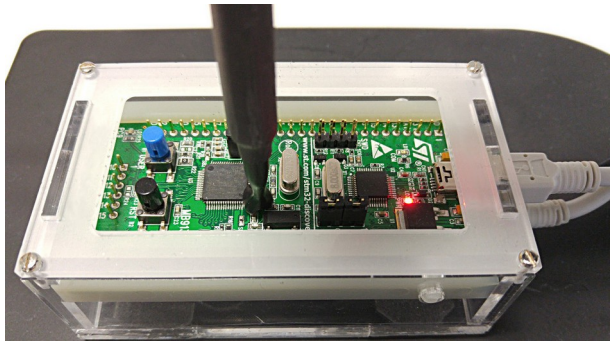


Photo by Howard Lawrence B on Unsplash

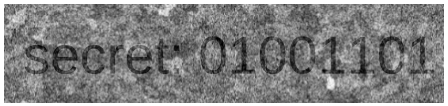
CONTRE-MESURES

But : rendre l'attaque plus difficile

Dissimulation

Rendre les mesures difficiles à exploiter (diminuer le SNR)

diminution du signal et ajout de bruit



Masquage

Décorréler les mesures des secrets (manipuler uniquement des variables indépendantes des secrets)

séparation du secret en plusieurs parties statistiquement indépendantes des secrets

share 1: 10010001
share 2: 11011100
share 1 xor share 2 = secret

PROBLÉMATIQUE

- **Attaques par canal auxiliaire**
  - classe d'attaque **la plus efficace** (avec les attaques par injection de fautes) contre les implémentations de cryptographie
  - complexité d'une attaque par cryptanalyse d'un AES > 2^{80}
  - complexité d'une attaque par canal auxiliaire d'un AES : 2^{8}

- **Plateformes ciblées variées et contraintes**
  - Peu de ressources mémoire
  - Vulnérabilité aux attaques dépendant de la plate-forme
  - Niveaux de sécurité souhaités dépendant de l'application
  - Contre-mesures très coûteuses en temps d'exécution et en mémoire
  - → Besoin de contre-mesures variées et peu coûteuses pour permettre la recherche de compromis

- **Application des contre-mesures coûteuse**
  - Requiert de l'expertise et souvent manuelle en pratique
  - → Besoin d'automatisation de l'application des contre-mesures

SOMMAIRE

1. Introduction et contexte

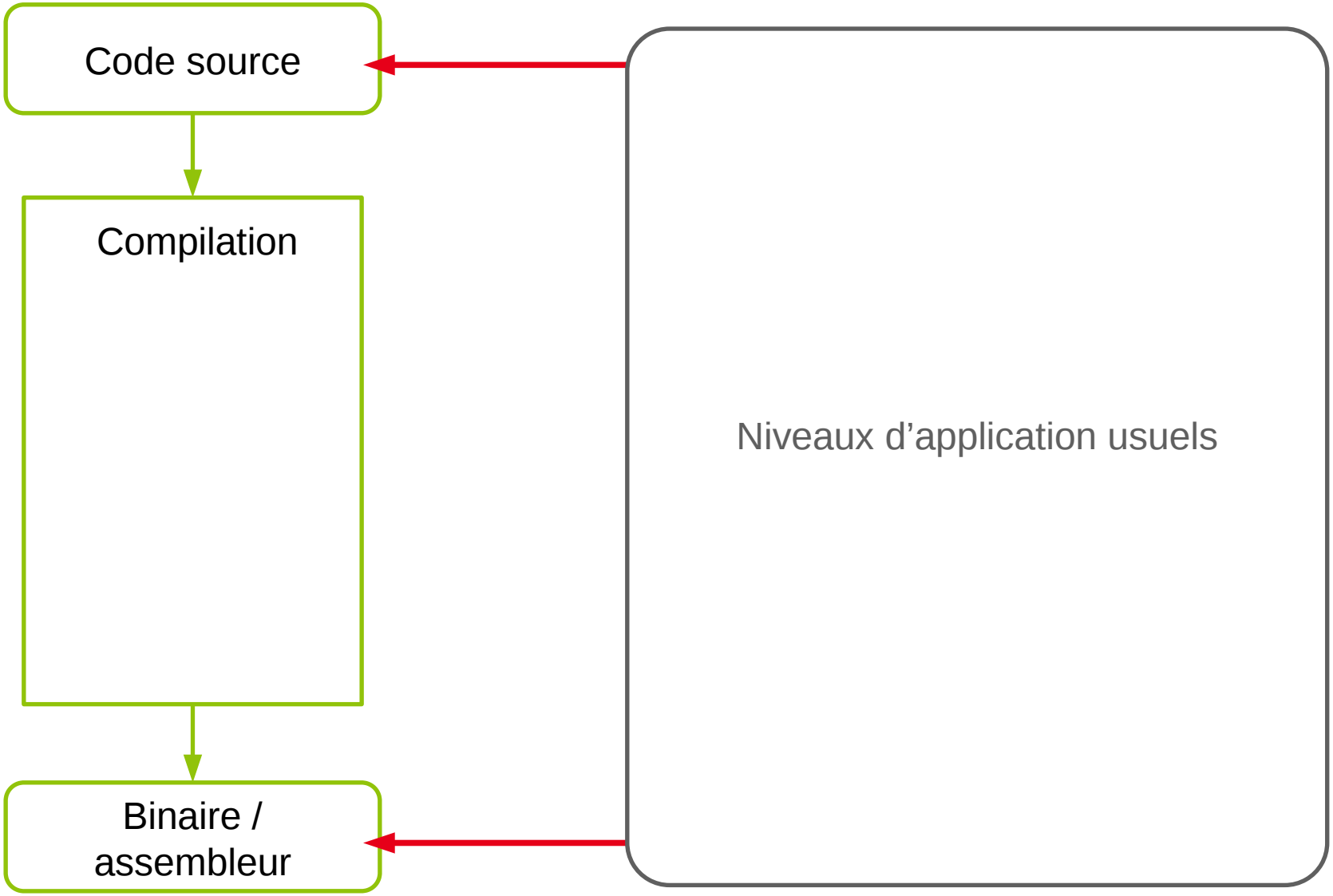
2. Niveaux d’application de contre-mesures

3. Application d'une contre-mesure de masquage booléen d'ordre 1 à la compilation

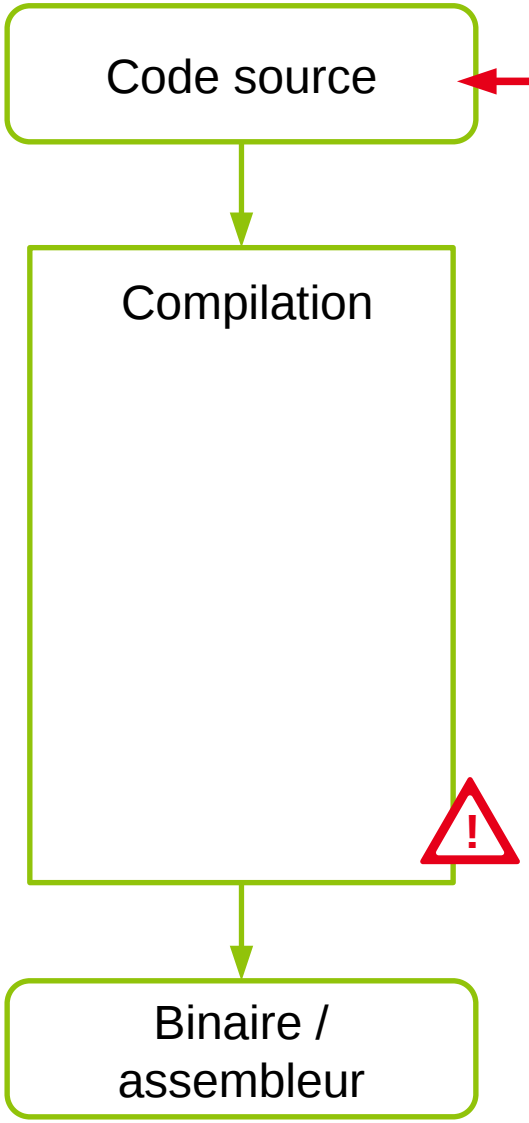
4. Application d'une contre-mesure de polymorphisme de code à la compilation

5. Conclusion et perspectives

APPLICATION DES CONTRE-MESURES



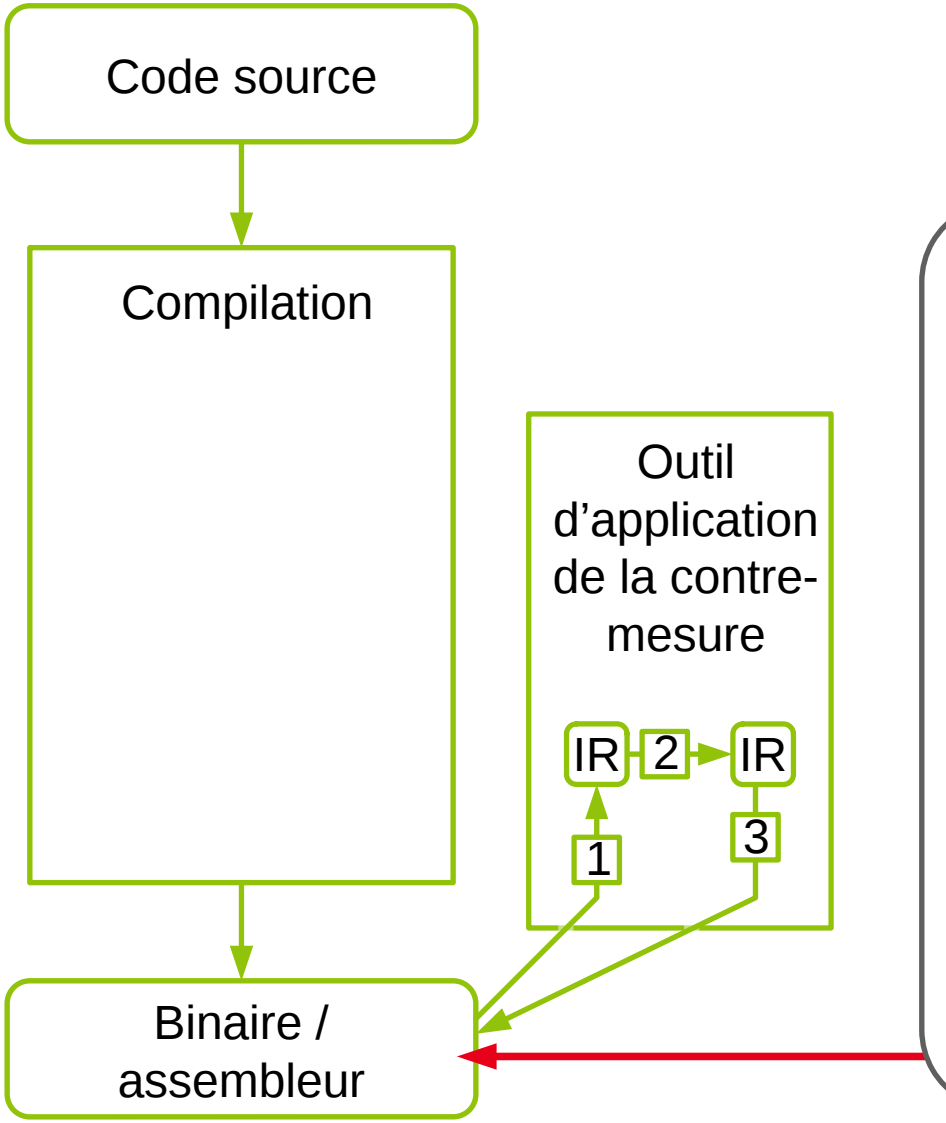
APPLICATION DES CONTRE-MESURES



- ✓ Compatible avec compilateur propriétaire
- ✓ Informations de haut niveau
- ✓ Plusieurs architectures cibles peuvent être supportées
- ✗ Niveau d'abstraction parfois trop haut par rapport aux phénomènes matériels en jeu pour ces attaques
- ✗ Nombreuses transformations de code après l'application de la contre-mesure

[Akkar et al., 2003, Lalande et al., 2014, Eldib et Wang, 2014, Luo et al., 2015, Couroussé et al., 2016]

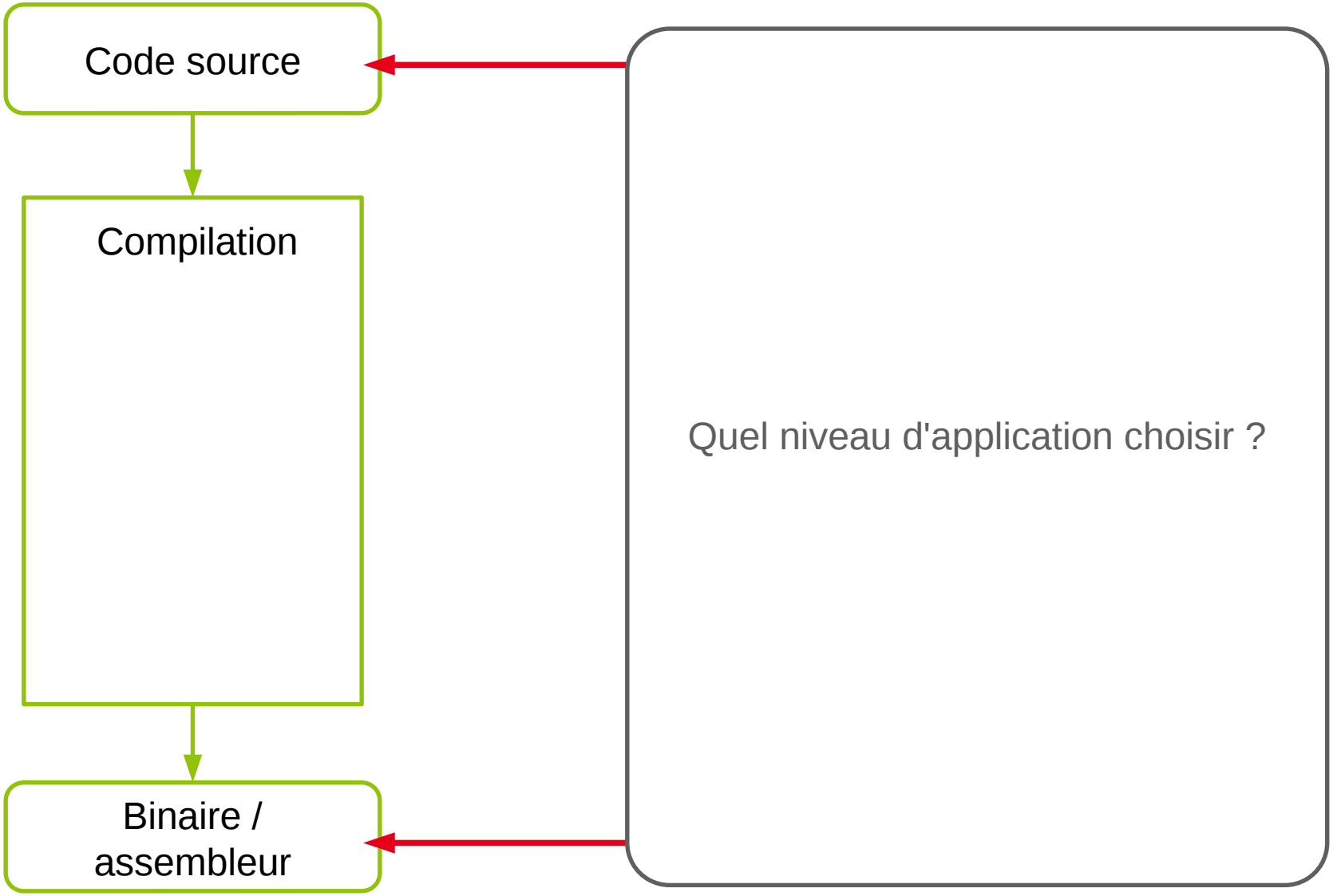
APPLICATION DES CONTRE-MESURES



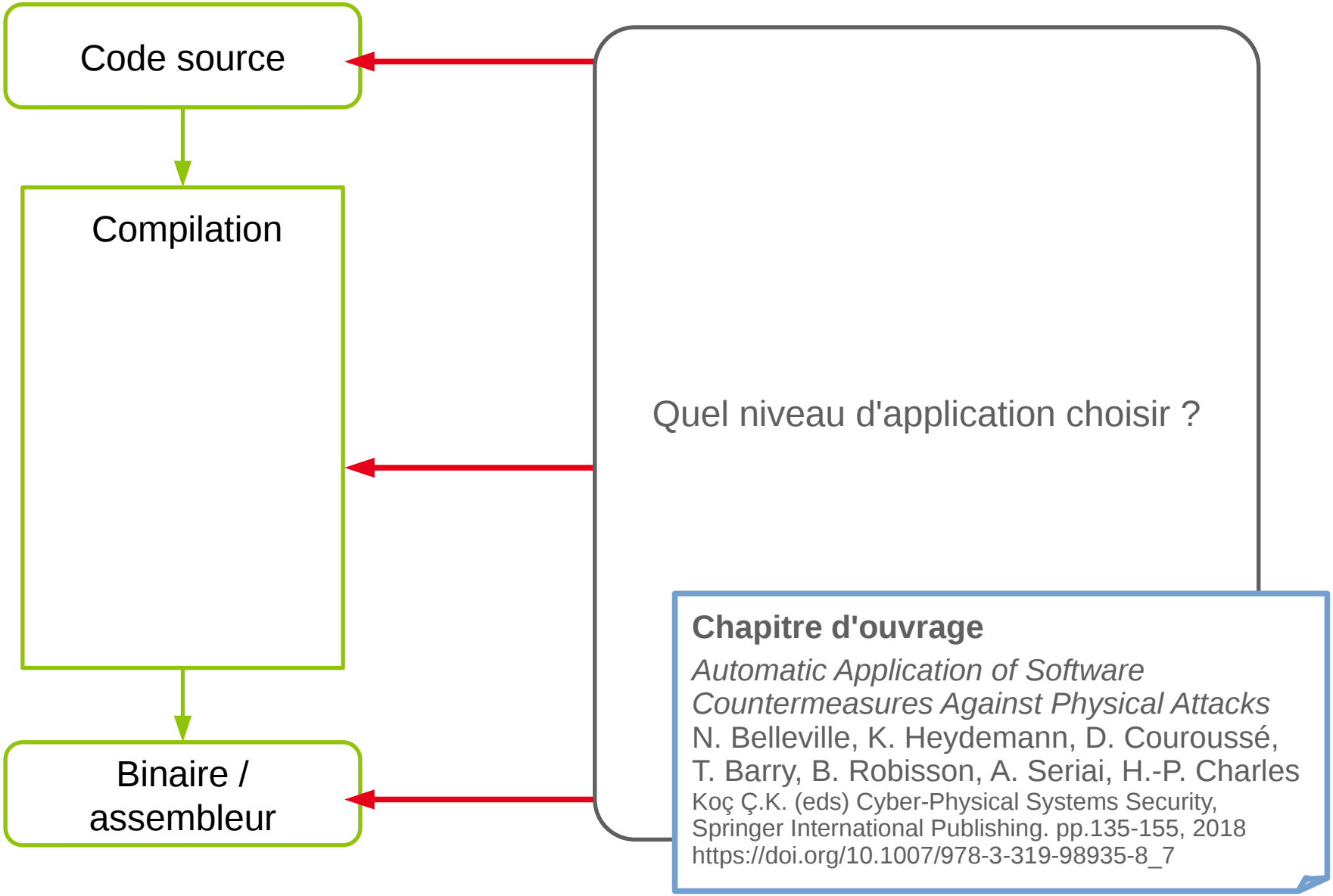
- ✓ Après toutes les transformations du compilateur
- ✓ Compatible avec compilateur propriétaire
- ✗ Manque des informations de haut niveau
- ✗ Reconstruction nécessaire d'une représentation intermédiaire (IR) (1), pour pouvoir appliquer la contre-mesure (2), puis réémettre du code assembleur (3)

[Bayrak et al., 2011, Moro et al., 2014, Rauzy et al., 2016, De Keulenaer et al., 2016]

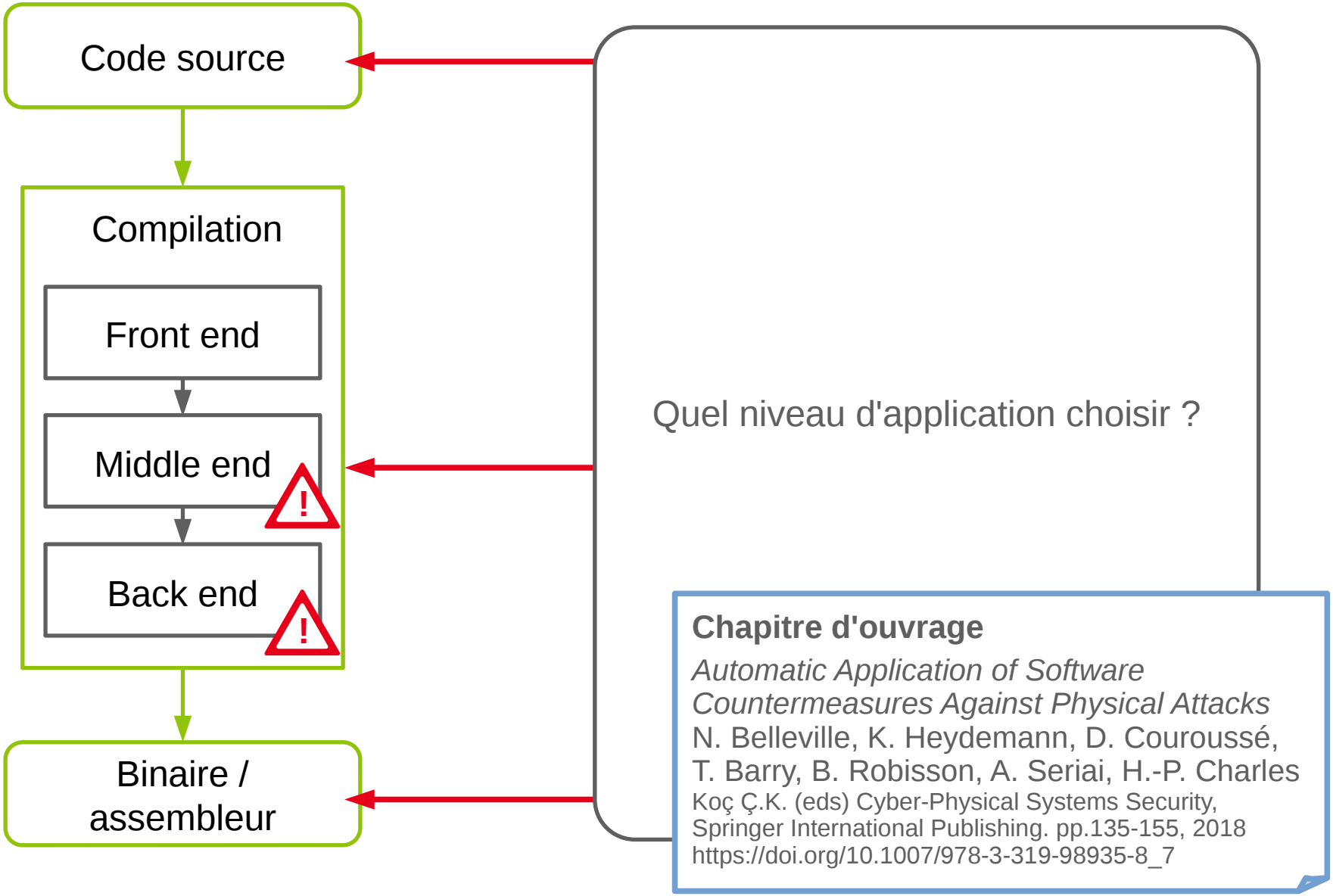
APPLICATION DES CONTRE-MESURES



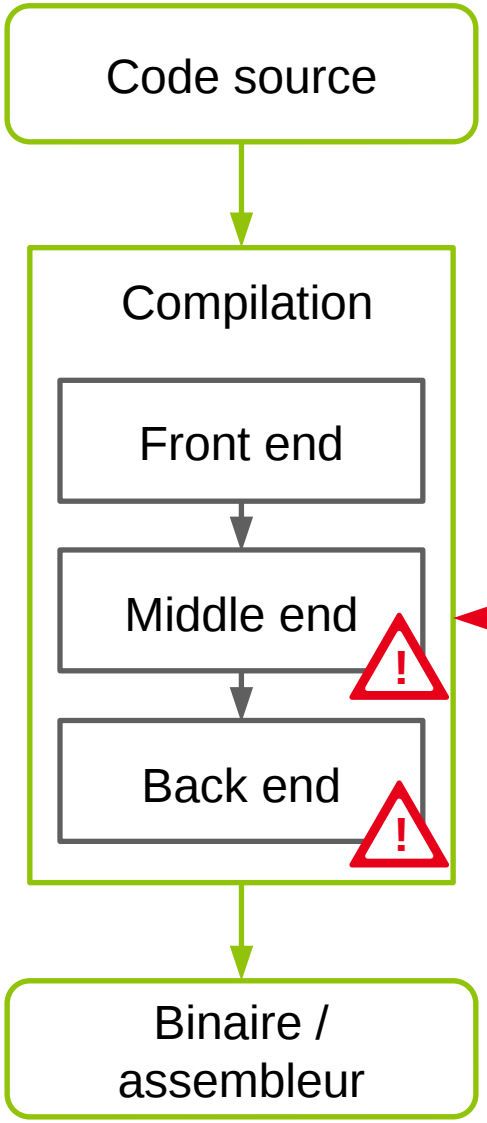
APPLICATION DES CONTRE-MESURES



APPLICATION DES CONTRE-MESURES



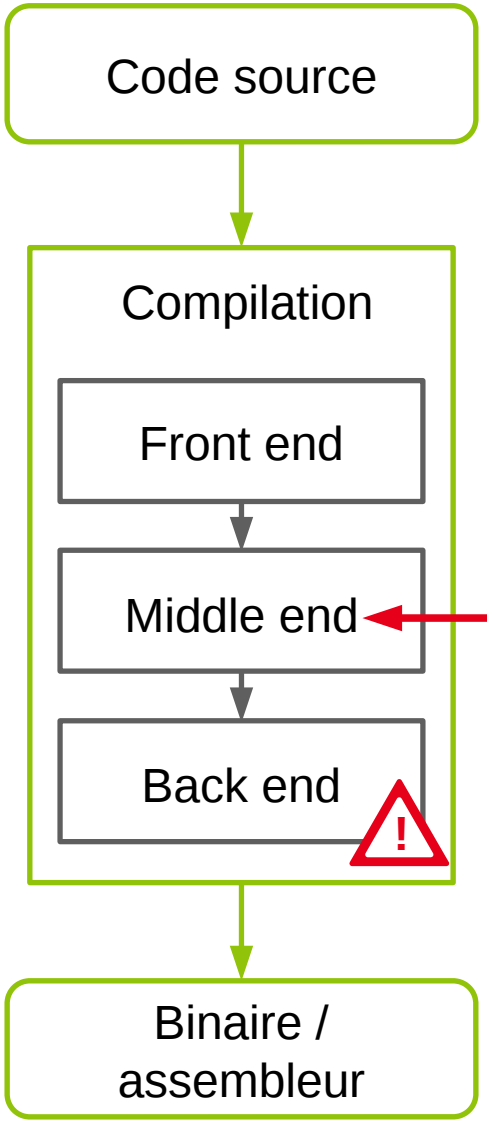
APPLICATION DES CONTRE-MESURES



- ✓ Passes d'application de la contre-mesure et d'optimisation de code entrelacées
- ✓ Optimisations de code peuvent être adaptées à la contre-mesure
- ✓ Accès à des informations de niveau varié tout au long de la compilation

[Malagón et al., 2012, Moss et al., 2012 Agosta et al., 2013a, Eldib et Wang, 2014, Chen et al., 2017, Luo et al., 2017, Barry et al., 2016, Agosta et al., 2015a, Agosta et al., 2015b, Bayrak et al., 2015b, Proy et al., 2017]

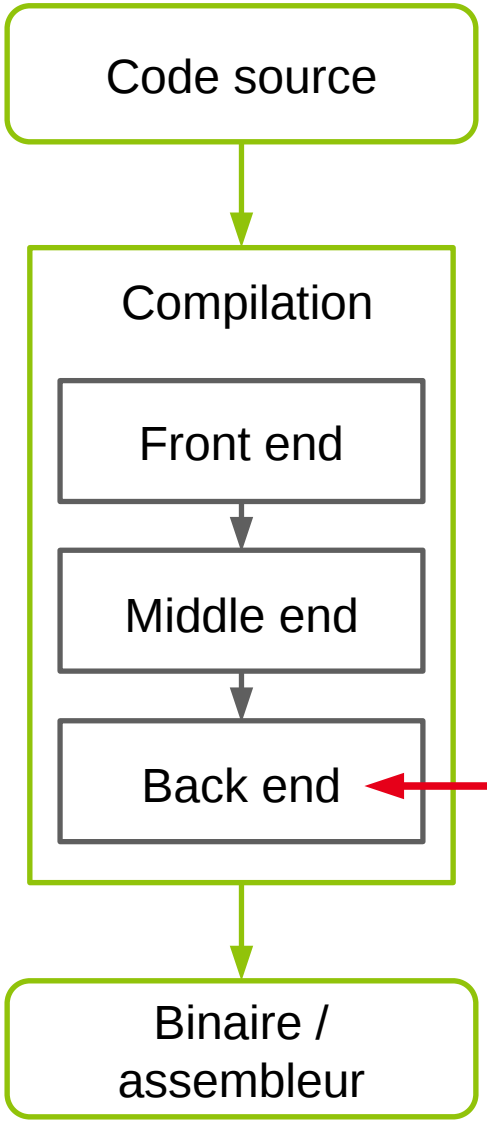
APPLICATION DES CONTRE-MESURES



- ✓ Plusieurs architectures cibles supportées
- ✓ Accès à des informations de haut niveau
- ✗ Transformations de code du back-end après l'application de la contre-mesure

[Malagón et al., 2012, Moss et al., 2012 Agosta et al., 2013, Eldib et Wang, 2014, Chen et al., 2017, Luo et al., 2017 Proy et al., 2017, Agosta et al., 2015a, Agosta et al., 2015b]

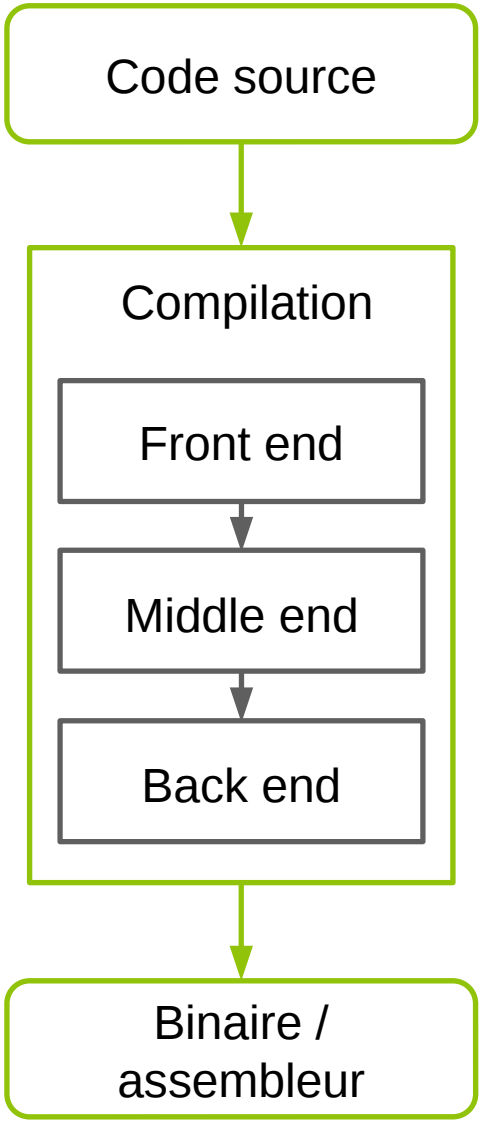
APPLICATION DES CONTRE-MESURES



- ✓ Après les optimisations du middle-end
- ✗ Perte d'informations de haut niveau
- ✗ Application spécifique à une architecture

[Barry et al., 2016, Bayrak et al., 2015b]

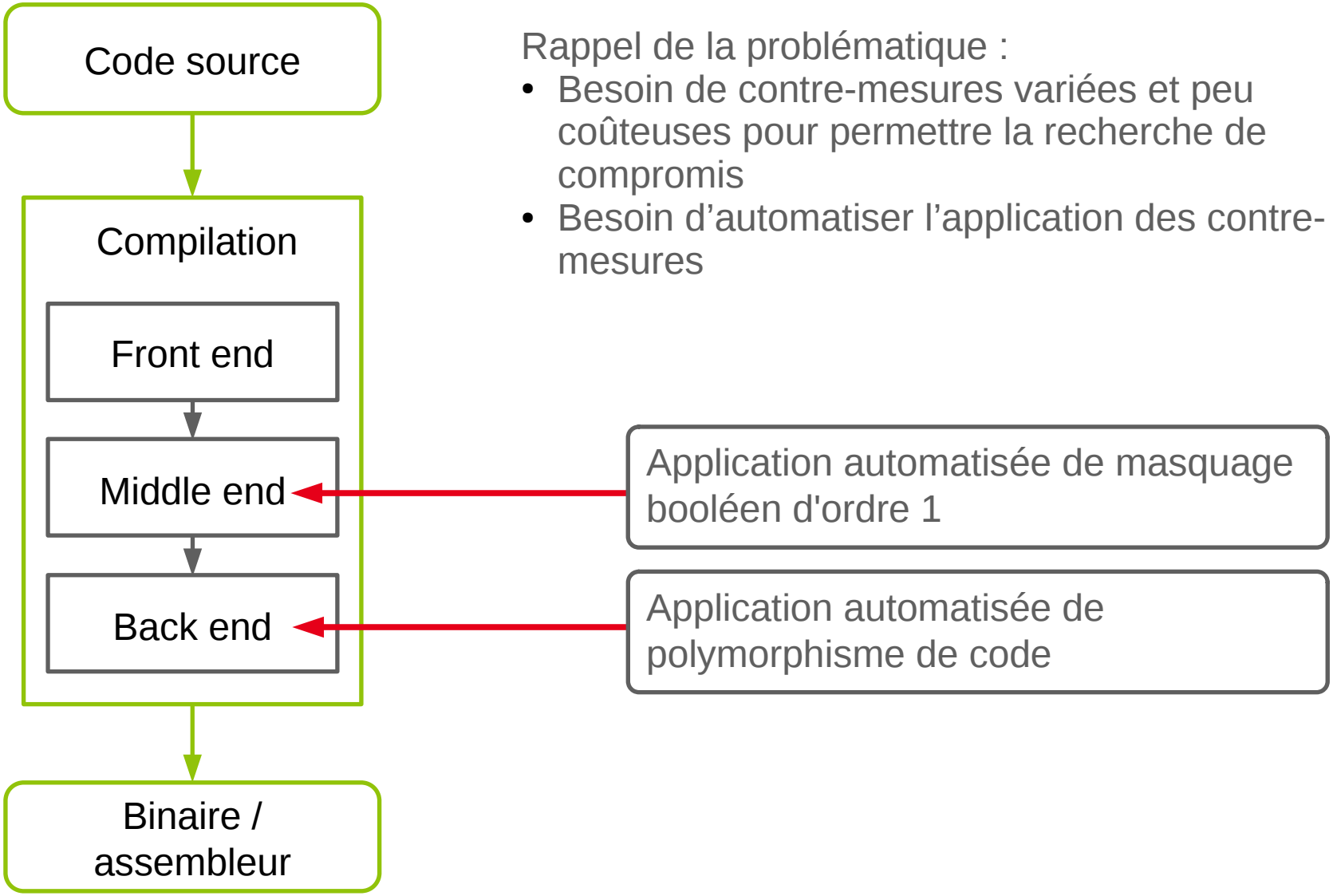
CONTRIBUTIONS



Rappel de la problématique :

- Besoin de contre-mesures variées et peu coûteuses pour permettre la recherche de compromis
- Besoin d'automatiser l'application des contre-mesures

CONTRIBUTIONS



Rappel de la problématique :

- Besoin de contre-mesures variées et peu coûteuses pour permettre la recherche de compromis
- Besoin d'automatiser l'application des contre-mesures

SOMMAIRE

1. Introduction et contexte

2. Niveaux d’application de contre-mesures

3. Application d'une contre-mesure de masquage booléen d'ordre 1 à la compilation

4. Application d'une contre-mesure de polymorphisme de code à la compilation

5. Conclusion et perspectives

MASQUAGE : PRINCIPE

[Ishai et al., 2003, Rivain et al., 2010]

- **Principe général**
  - séparation des variables secrètes en plusieurs shares
  - chaque share est **statistiquement indépendant** du secret
  - tous les calculs sont effectués avec shares séparés

- **Masquage booléen d'ordre n**
  - n+1 shares

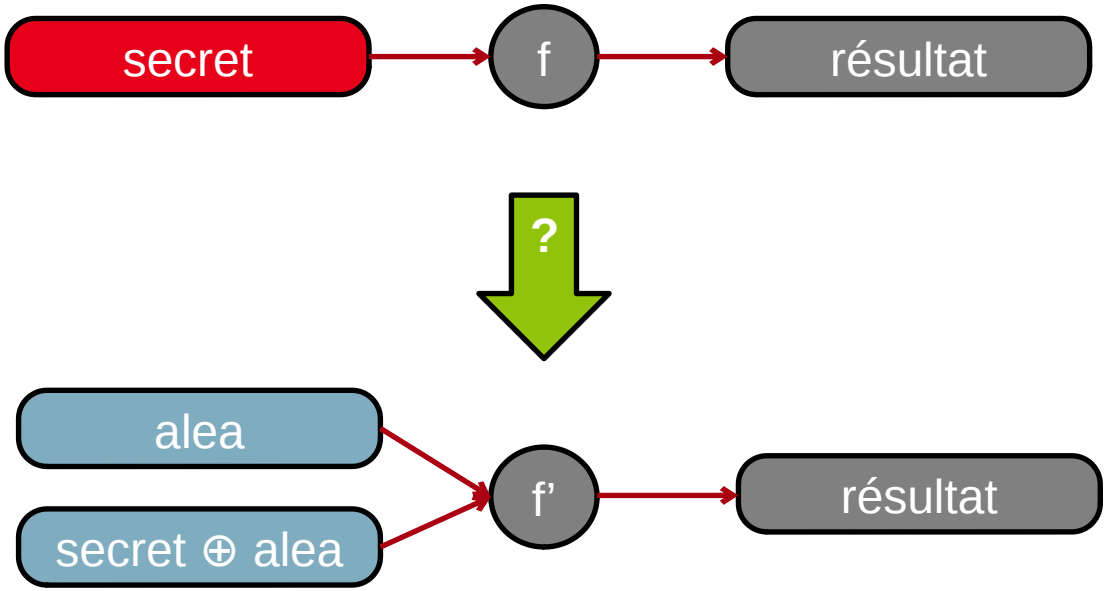
$$secret = s_0 \oplus s_1 \oplus \ldots \oplus s_n$$
$$s_0, \ldots, s_n : \text{shares}$$
$$s_1, \ldots, s_n : \text{masques, nombres aléatoires}$$
$$s_0 = secret \oplus s_1 \oplus \ldots \oplus s_n$$

- **Masquage booléen d'ordre 1**
  - 2 shares
  - séparation en shares :

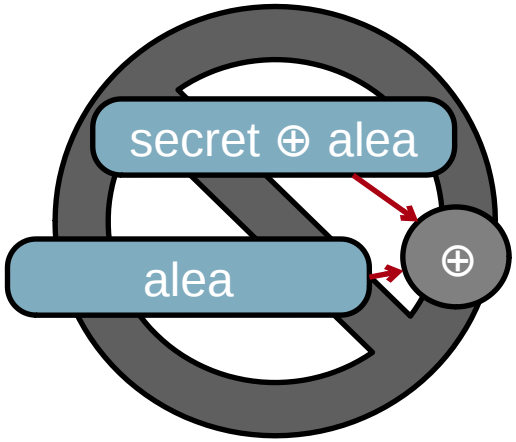
$$masque = alea()$$
$$share_0 = secret \oplus masque$$
$$share_1 = masque$$

APPLICATION DU MASQUAGE : PROBLÈME

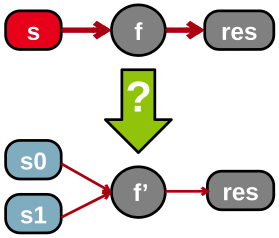
- **Transformer la fonction manipulant le secret pour s'adapter à la décomposition en shares :**



- **Ne jamais dévoiler le secret :**



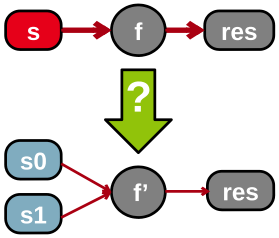
DÉTERMINATION DES INSTRUCTIONS À TRANSFORMER



- **Secret connu par l'utilisateur**
- **Tout calcul intermédiaire manipulant un secret doit être transformé :**

```
/* a et b publiques, s est secret */
bool f(bool a,b, bool s)
{
    bool c = a ^ s;
    bool d = c | s;
    bool e = d & b;
    return e;
}
```

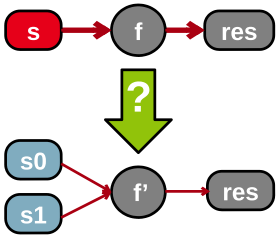
DÉTERMINATION DES INSTRUCTIONS À TRANSFORMER



- **Secret connu par l'utilisateur**
- **Tout calcul intermédiaire manipulant un secret doit être transformé :**

```
/* a et b publiques, s est secret */
bool f(bool a,b, bool s)
{
    bool c = a ^ s;
    bool d = c | s;
    bool e = d & b;
    return e;
}
```

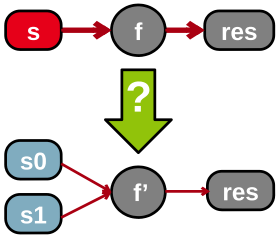
DÉTERMINATION DES INSTRUCTIONS À TRANSFORMER



- **Secret connu par l'utilisateur**
- **Tout calcul intermédiaire manipulant un secret doit être transformé :**

```
/* a et b publiques, s est secret */
bool f(bool a,b, bool s)
{
    bool c = a ^ s;
    bool d = c | s;
    bool e = d & b;
    return e;
}
```

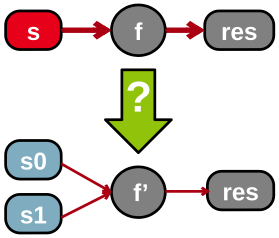
DÉTERMINATION DES INSTRUCTIONS À TRANSFORMER



- **Secret connu par l'utilisateur**
- **Tout calcul intermédiaire manipulant un secret doit être transformé :**

```
/* a et b publiques, s est secret */
bool f(bool a,b, bool s)
{
    bool c = a ^ s;
    bool d = c | s;
    bool e = d & b;
    return e;
}
```

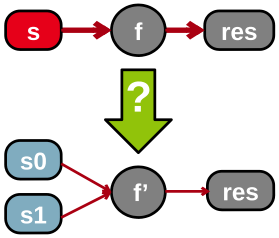
DÉTERMINATION DES INSTRUCTIONS À TRANSFORMER



- **Secret connu par l'utilisateur**
- **Tout calcul intermédiaire manipulant un secret doit être transformé :**

```
/* a et b publiques, s est secret */
bool f(bool a,b, bool s)
{
    bool c = a ^ s;
    bool d = c | s;
    bool e = d & b;
    return e;
}
```

DÉTERMINATION DES INSTRUCTIONS À TRANSFORMER

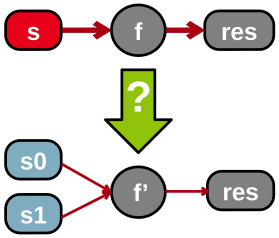


- **Secret connu par l'utilisateur**
- **Tout calcul intermédiaire manipulant un secret doit être transformé :**

```
/* a et b publiques, s est secret */
bool f(bool a,b, bool s)
{
    bool c = a ^ s;
    bool d = c | s;
    bool e = d & b;
    return e;
}
```

Peut être fait par une analyse de flot de données dans le compilateur [Moss et al., 2012, Agosta et al., 2013]

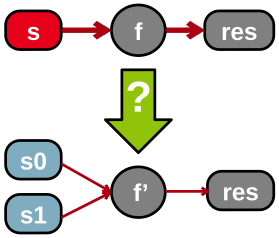
MASQUAGE D'UNE INSTRUCTION



- **Transformation dépendant de la nature de l'instruction**

- **Fonctions linéaires**
  - a op (s0 xor s1) = (a op s0) xor (a op s1)
  - Opération dupliquée :

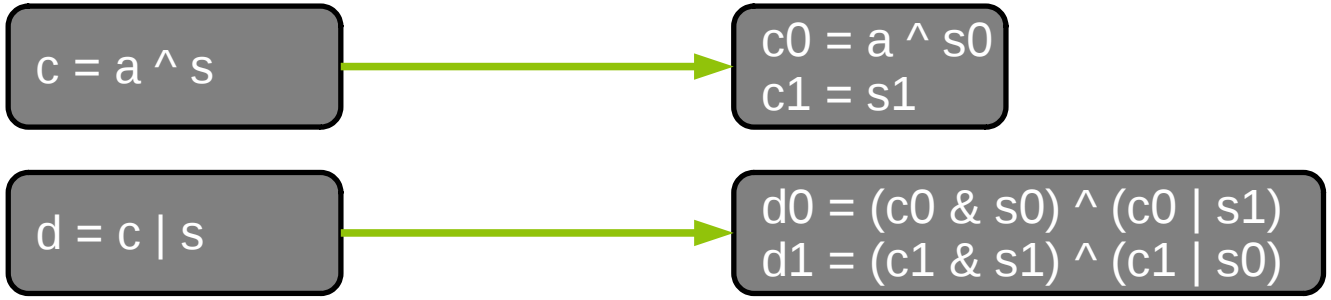
MASQUAGE D'UNE INSTRUCTION



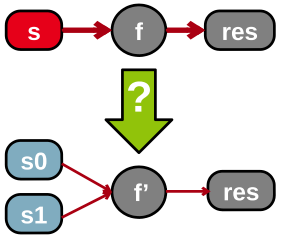
- **Transformation dépendant de la nature de l'instruction**

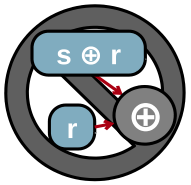
- **Fonctions linéaires**
  - a op (s0 xor s1) = (a op s0) xor (a op s1)
  - Opération dupliquée :

- **Fonctions non linéaires**
  - a op (s0 xor s1) != (a op s0) xor (a op s1)
  - Transformation définie au cas par cas



MASQUAGE D'UNE SUITE D'INSTRUCTIONS : BESOIN DE REMASQUAGE





- **Les shares et tout calcul intermédiaire doivent rester statistiquement indépendants du secret**

```
c = a & s
d = c | s
```

→

```
c0 = a & s0
c1 = a & s1
d0 = (c0 & s0) ^ (c0 | s1)
d1 = (c1 & s1) ^ (c1 | s0)
```

- **Il faut remasquer les shares si nécessaire**

```
c = a & s
d = c | s
```

→

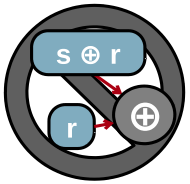
```
c0 = a & s0
c1 = a & s1
r = alea()
c0' = c0 ^ r
c1' = c1 ^ r
d0 = (c0' & s0) ^ (c0' | s1)
d1 = (c1' & s1) ^ (c1' | s0)
```

MASQUAGE D'UNE FONCTION

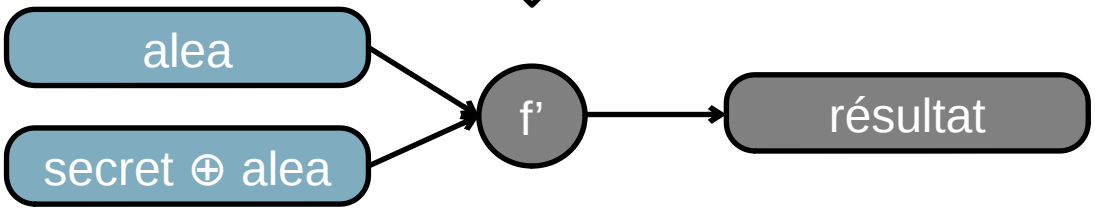
Déclaration des variables secrètes

Analyse de flot de données pour déterminer les instructions à transformer

Transformation des instructions en remasquant si nécessaire



automatisé au sein du compilateur

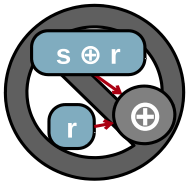


MASQUAGE D'UNE FONCTION

Déclaration des variables secrètes

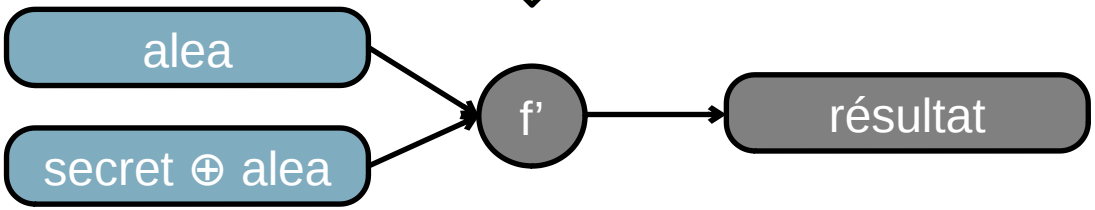
Analyse de flot de données pour déterminer les instructions à transformer

Transformation des instructions en remasquant si nécessaire



automatisé au sein du compilateur

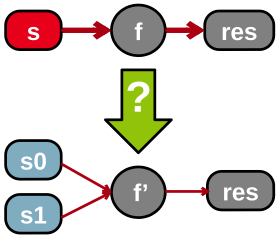
Contributions :
- Gestion des codes non déroulés
- Gestion des tables constantes



MASQUAGE : TRANSFORMATION DES TABLES CONSTANTES

- **Tables constantes couramment utilisées dans les fonctions de chiffrement**
- **Permettent de calculer efficacement une fonction quelconque sur un octet**
  - y = SBox[x]
- **Approche utilisée au sein des compilateurs jusqu'ici pour les masquer :**
  - Masquer la table elle-même
  - SBox'[x xor m] → SBox[x] xor m'
  - Nécessite de changer m et m' régulièrement pour toute la table
- **Contribution : intégration au sein du compilateur d'une approche par interpolation polynomiale pour masquer les tables constantes**
  - Recherche d'un polynôme interpolateur de la table
    - Pour tout x, P[x] = SBox[x]
  - Masquage de l'évaluation du polynôme
  - Nouvelles optimisations proposées

MASQUAGE : ÉTAPES DU MASQUAGE D'UNE TABLE CONSTANTE



- **Masquage effectué au sein du compilateur en plusieurs étapes :**

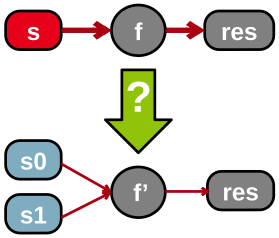
1. détection de l'accès à une table constante

2. récupération des valeurs de la table

3. recherche d'un polynôme interpolateur

4. émission de code : remplacement de l'accès à la table constante par une évaluation masquée du polynôme

MASQUAGE : ÉTAPES DU MASQUAGE D'UNE TABLE CONSTANTE

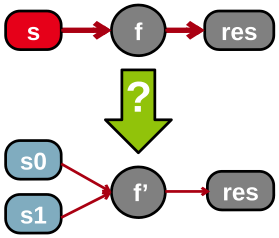


- **Masquage effectué au sein du compilateur en plusieurs étapes :**

1. détection de l'accès à une table constante
2. récupération des valeurs de la table
3. recherche d'un polynôme interpolateur
4. émission de code : remplacement de l'accès à la table constante par une évaluation masquée du polynôme

load indéxé par un secret

MASQUAGE : ÉTAPES DU MASQUAGE D'UNE TABLE CONSTANTE



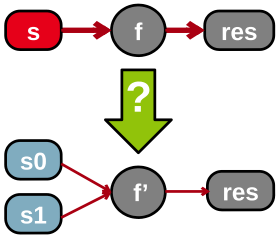
- **Masquage effectué au sein du compilateur en plusieurs étapes :**

1. détection de l'accès à une table constante
2. récupération des valeurs de la table
3. recherche d'un polynôme interpolateur
4. émission de code : remplacement de l'accès à la table constante par une évaluation masquée du polynôme

Approche de [Coron et al. 2014] :
recherche du polynôme sous la forme $\mathbf{P[x] = \Sigma(q_i[x]*p_i[x]) + p_0[x]}$
dans un corps fini binaire

But : avoir le moins possible d'opérations non linéaires par rapport au xor

MASQUAGE : ÉTAPES DU MASQUAGE D'UNE TABLE CONSTANTE

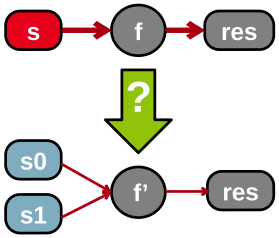


- **Masquage effectué au sein du compilateur en plusieurs étapes :**

1. détection de l'accès à une table constante
2. récupération des valeurs de la table
3. recherche d'un polynôme interpolateur
4. émission de code : remplacement de l'accès à la table constante par une évaluation masquée du polynôme

Émission d'un code masqué optimisé
pour évaluer le polynôme avec les shares

MASQUAGE : ÉTAPES DU MASQUAGE D'UNE TABLE CONSTANTE



- **Masquage effectué au sein du compilateur en plusieurs étapes :**

1. détection de l'accès à une table constante
2. récupération des valeurs de la table
3. recherche d'un polynôme interpolateur
4. émission de code : remplacement de l'accès à la table constante par une évaluation masquée du polynôme

Nouvelles optimisations permettant des gains en performance :

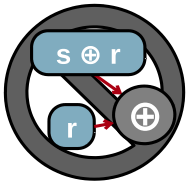
- Choix des q_i
- Optimisation de la multiplication de corps fini binaire

MASQUAGE D'UNE FONCTION

Déclaration des variables secrètes

Analyse de flot de données pour déterminer les instructions à transformer

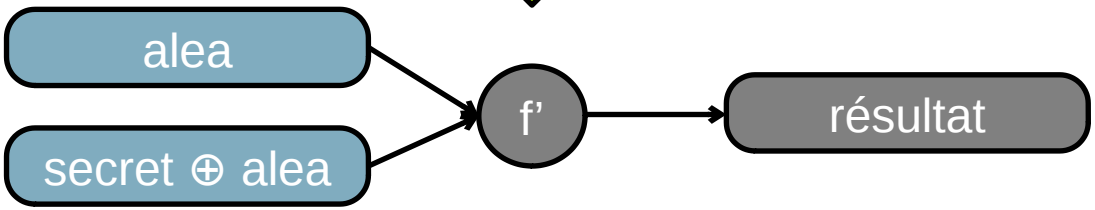
Transformation des instructions en remasquant si nécessaire



automatisé dans le middle-end de LLVM

Maskara

- Environ 4000 lignes de code
- Environ 150 commits



RÉSULTATS : PERFORMANCE

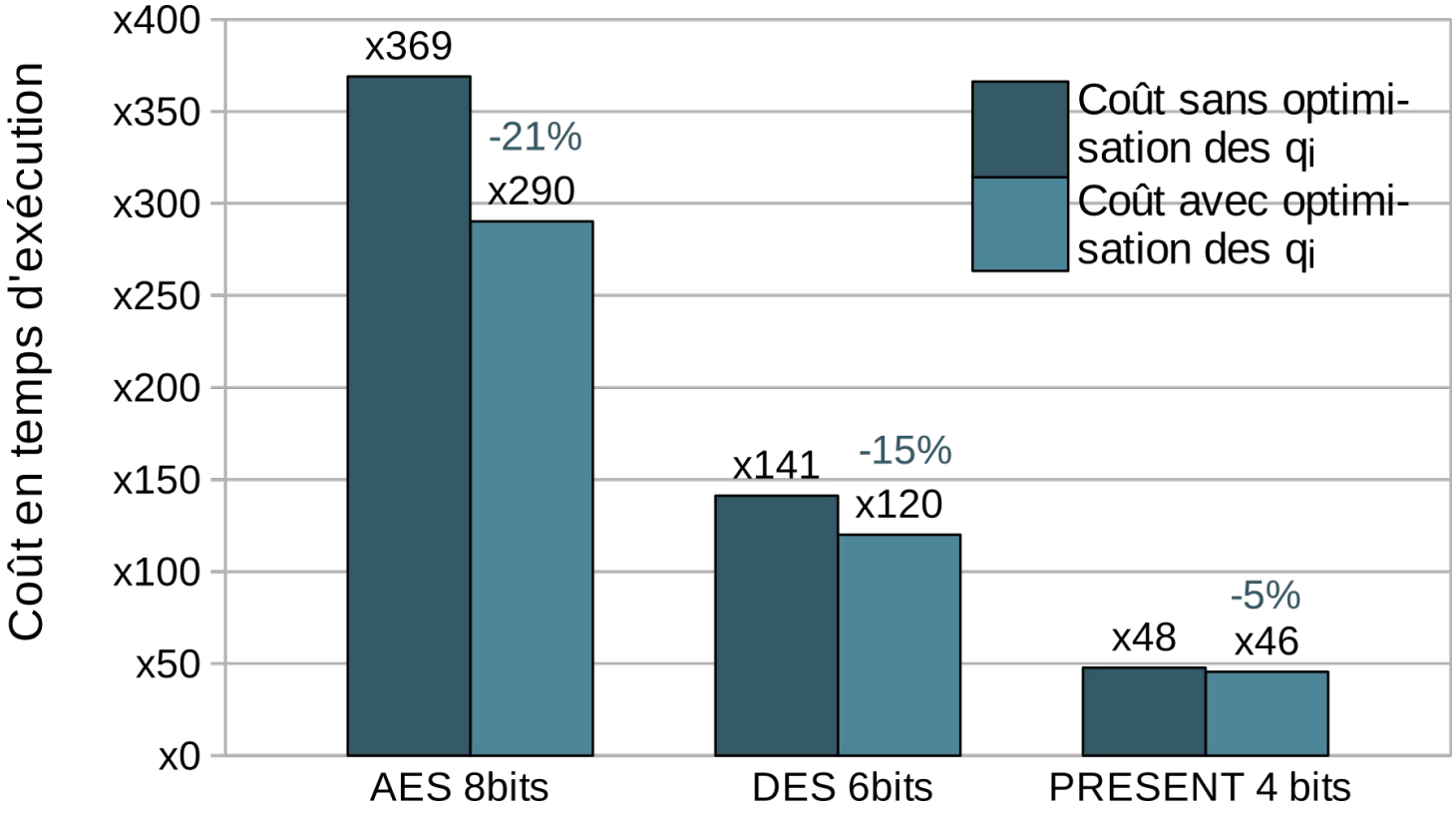
- **Besoin d'optimisation de la multiplication de corps fini**
  - Version ''naïve'' en **22 instructions** ARM-Thumb
  - Opération utilisée plusieurs fois pour l'évaluation masquée du polynôme
    - 4 fois pour la SBox de PRESENT (table à 16 éléments)
    - 7 fois pour la SBox de DES (table à 64 éléments)
    - 20 fois pour la SBox de AES (table à 256 éléments)

- **Optimisation économisant un modulo [Plank]**
  - Opération en **15 instructions** ARM-Thumb

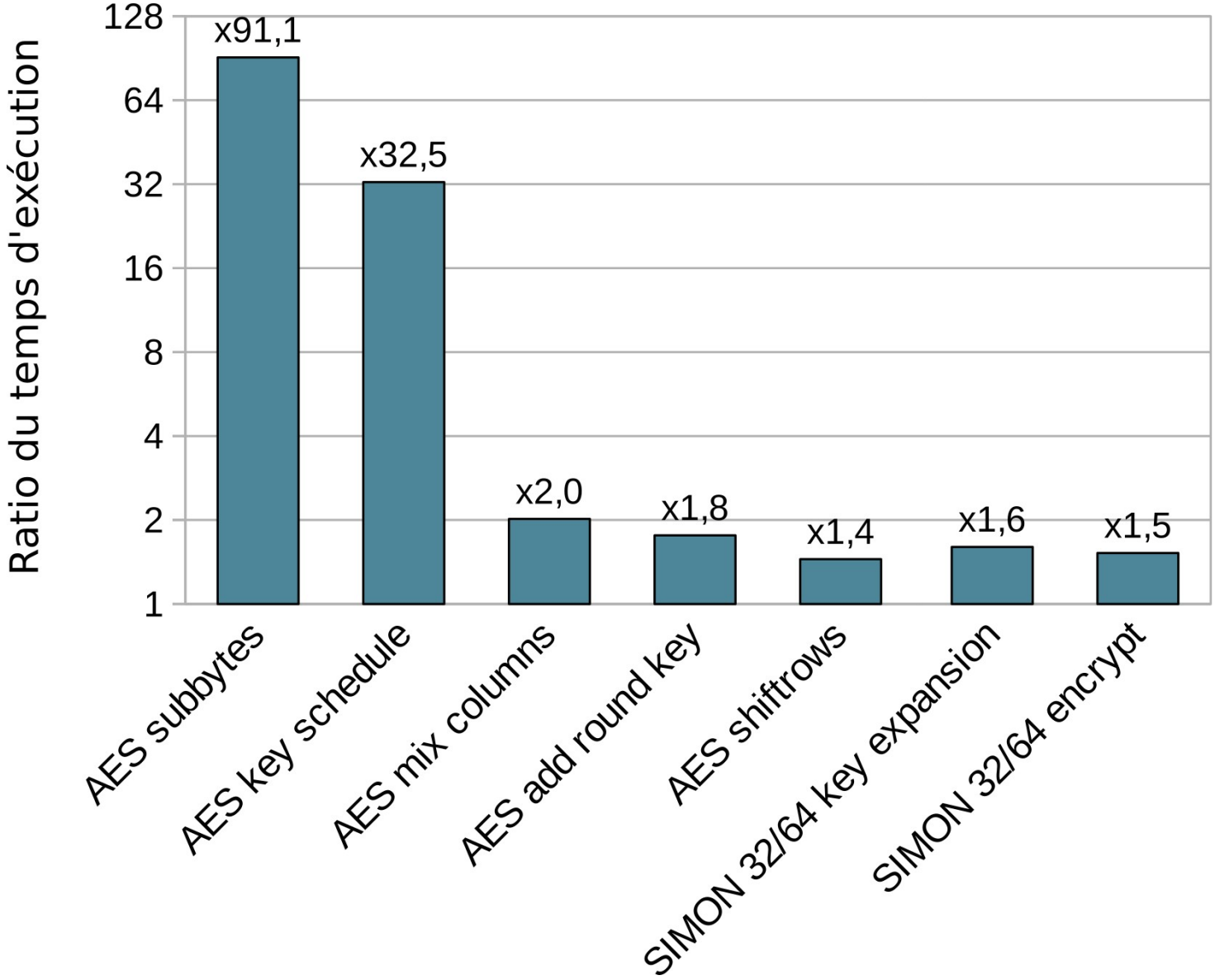
- **Nouvelle optimisation apportée par cette thèse pour éviter d'avoir à gérer le cas où un opérande est nul comme un cas particulier**
  - Opération en **8 instructions** ARM-Thumb

RÉSULTATS : PERFORMANCE

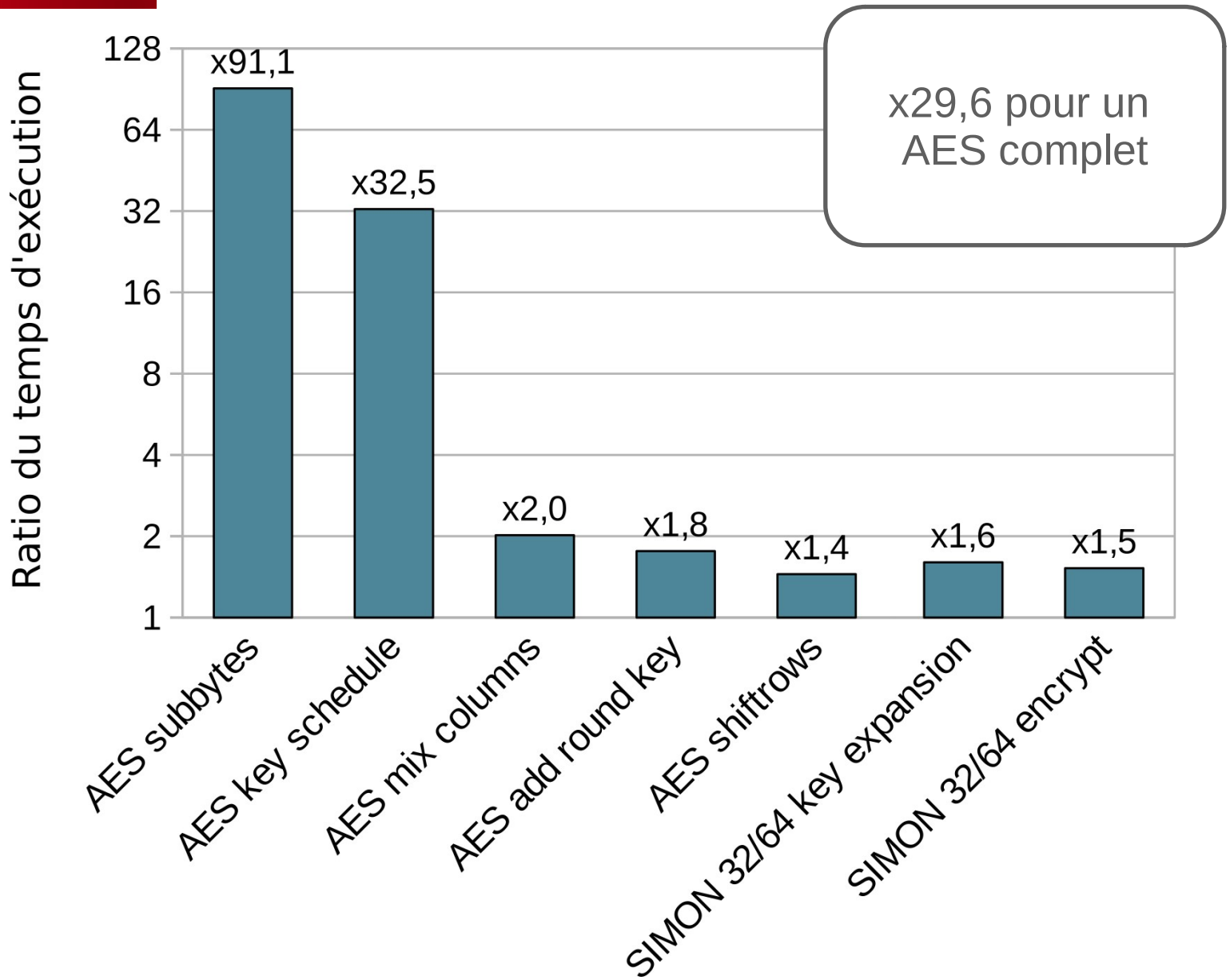
- **Recherche du polynôme sous la forme $P[x] = \Sigma(q_i[x]*p_i[x]) + p_0[x]$ dans un corps fini binaire**
  - Coefficients des q_i choisis aléatoirement
  - En forçant des coefficients à 0, on économise des calculs



RÉSULTATS : PERFORMANCE



RÉSULTATS : PERFORMANCE



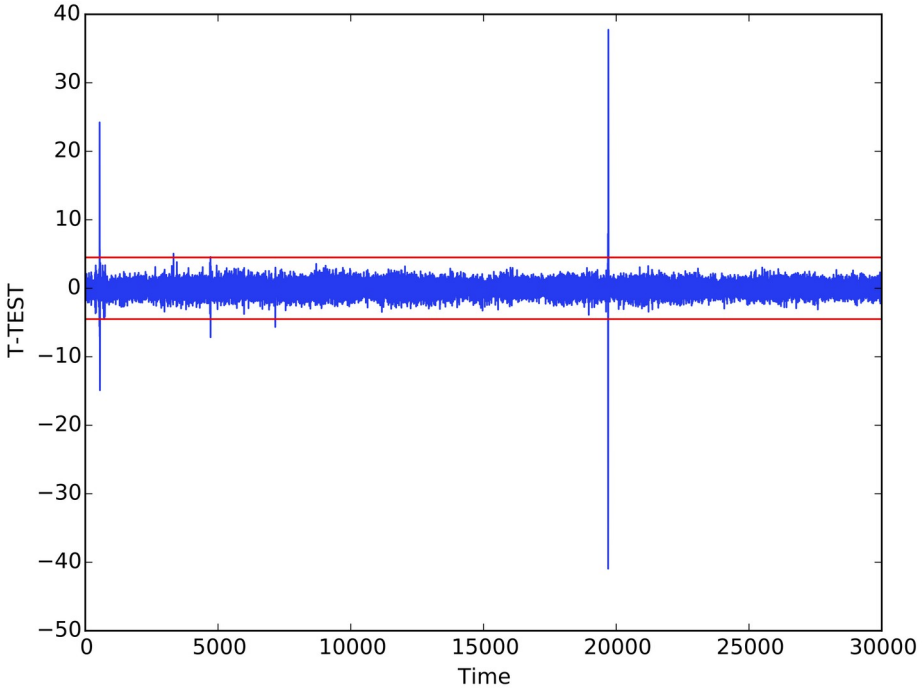
RÉSULTATS : SÉCURITÉ

- **Binaires analysés par un outil de vérification formelle [Ben El Ouahma et al., 2017]**
  - analyse symbolique au niveau binaire
  - vérification de l'application de la contre-mesure : **aucune valeur n'est statistiquement dépendante du secret**

RÉSULTATS : SÉCURITÉ

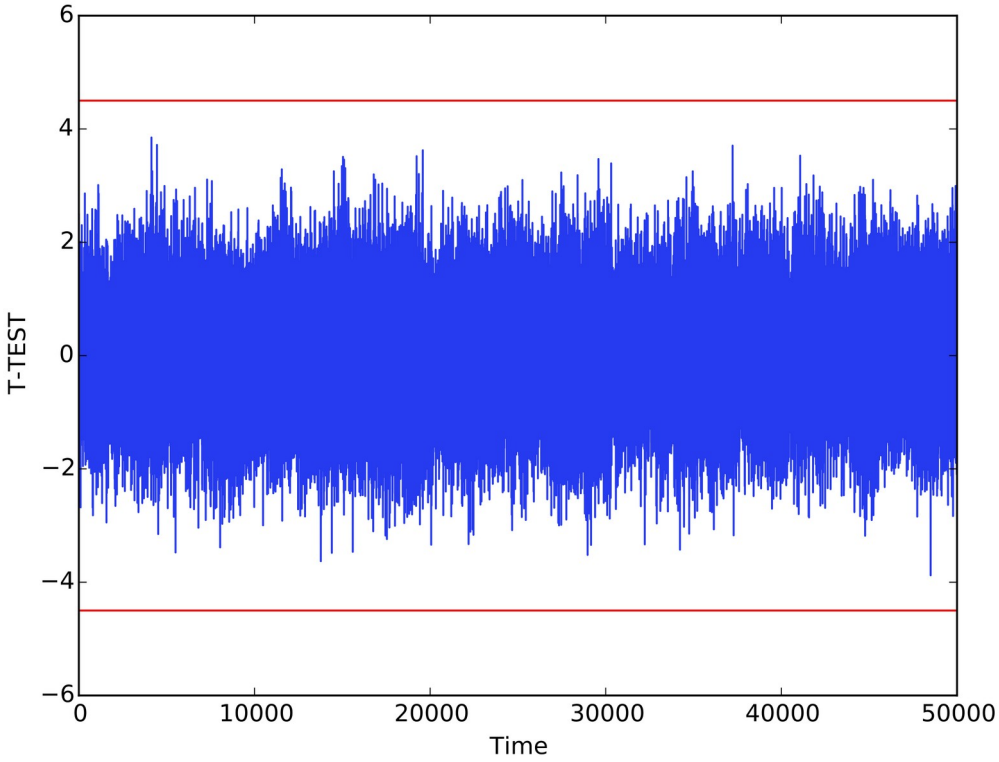
- **Binaires analysés par un outil de vérification formelle [Ben El Ouahma et al., 2017]**
  - analyse symbolique au niveau binaire
  - vérification de l'application de la contre-mesure : **aucune valeur n'est statistiquement dépendante du secret**

- **Plusieurs fuites dépendantes de données secrètes constatées en pratique**
  - Modèle de fuite en valeur pas assez fidèle



RÉSULTATS : SÉCURITÉ

- **Hypothèse : fuites en transition sur les accès mémoires et les opérations sollicitant l'ALU**
  - Ajout d'une passe dans le back-end pour valider cette hypothèse
  - T-test réussi après cet ajout



CONCLUSION MASQUAGE

- **Application du masquage automatisée au sein de LLVM**
- **Mise en œuvre de l'approche de [Coron et al. 2014] pour le masquage des tables constantes**
- **Nouvelles optimisations apportées**
- **Coûts en performance fortement dépendants du code**
- **Modèle de fuite en valeur pas assez fidèle**
  - Contre-mesure correctement appliquée relativement à ce modèle
  - Fuite d'information sur le secret constatée en pratique, sans doute liée à des fuites en transition
    → Besoin de modèles de fuite fidèles

SOMMAIRE

1. Introduction et contexte

2. Niveaux d’application de contre-mesures

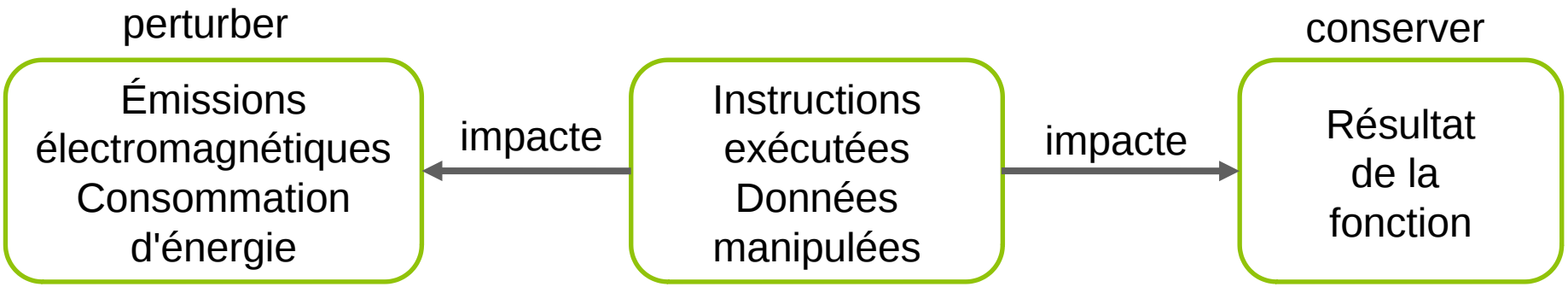
3. Application d'une contre-mesure de masquage booléen d'ordre 1 à la compilation

4. Application d'une contre-mesure de polymorphisme de code à la compilation

5. Conclusion et perspectives

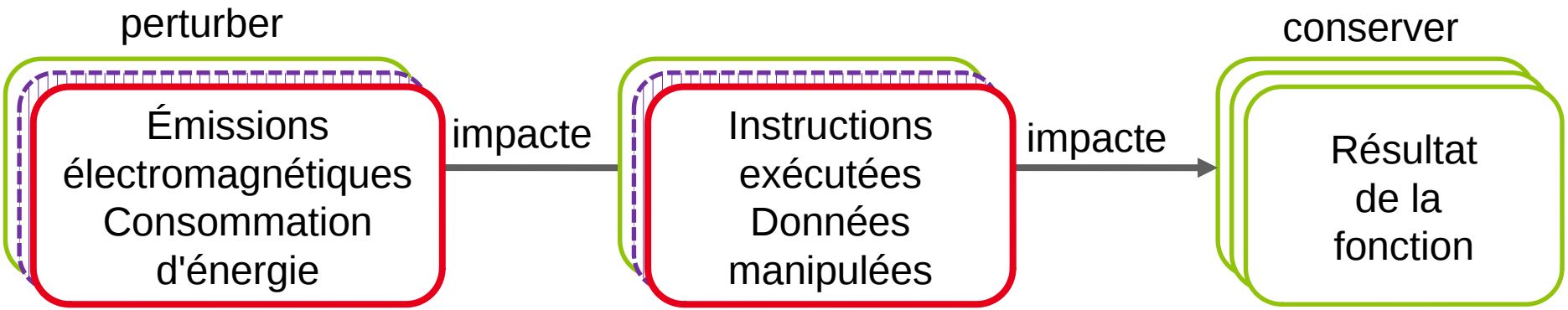
POLYMORPHISME DE CODE : PRINCIPE

- **Perturber les observations par canaux auxiliaires**
- **Conserver la sémantique du programme**



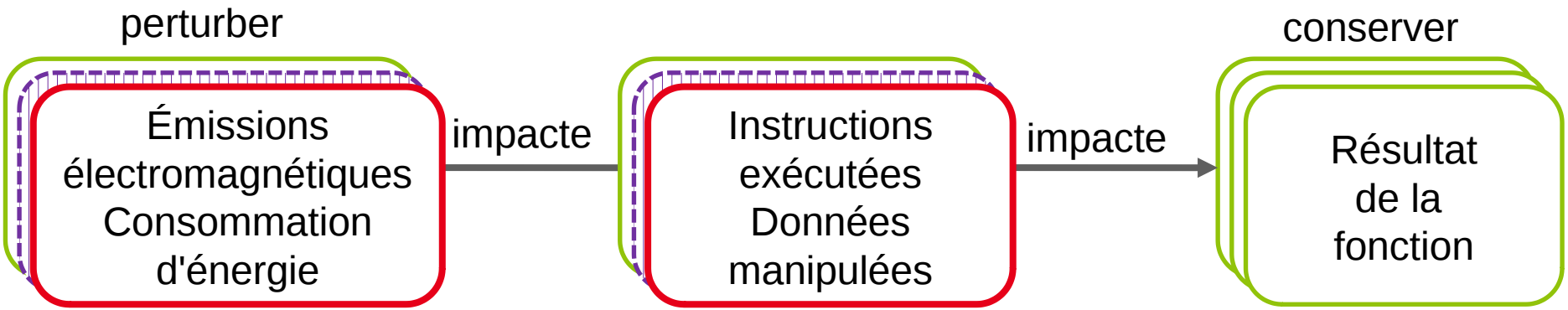
POLYMORPHISME DE CODE : PRINCIPE

- **Perturber les observations par canaux auxiliaires**
- **Conserver la sémantique du programme**



POLYMORPHISME DE CODE : PRINCIPE

- **Perturber les observations par canaux auxiliaires**
- **Conserver la sémantique du programme**



Approches statiques	Approches dynamiques
Choix aléatoire à l'exécution entre plusieurs chemin d'exécution équivalents	Regénération / modification du code pour faire varier le code exécuté
Coût mémoire important	Coût de la génération
	Permissions mémoire
	Allocation mémoire

[Agosta et al., 2012, Agosta et al., 2015b, Couroussé et al., 2016]

POLYMORPHISME DE CODE : PROBLÈMES

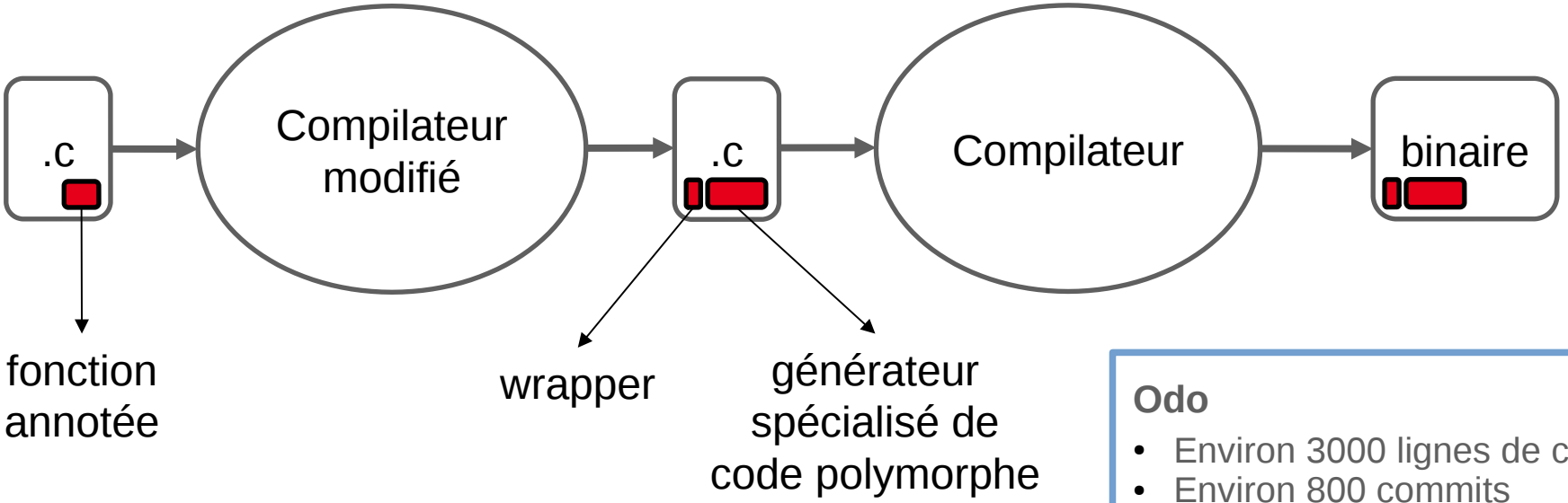
- **[Agosta et al., 2012] : approche dynamique**
  - Modification dynamique de code très coûteuse
  - Droits d'exécution et d'écriture nécessaire sur une zone mémoire
  - Problème de l'allocation d'une zone mémoire non traité
- **[Couroussé et al. 2016] : approche dynamique**
  - Droits d'exécution et d'écriture nécessaire sur une zone mémoire
  - Problème de l'allocation d'une zone mémoire non traité
  - Utilisation d'un DSL (langage spécifique)
- **[Agosta et al., 2015b] : approche statique**
  - Coût mémoire important

POLYMORPHISME DE CODE : CONTRIBUTION

- **[Agosta et al., 2012] : approche dynamique**
  - Modification dynamique de code très coûteuse
  - Droits d'exécution et d'écriture nécessaire sur une zone mémoire
  - Problème de l'allocation d'une zone mémoire non traité
- **[Couroussé et al. 2016] : approche dynamique**
  - Droits d'exécution et d'écriture nécessaire sur une zone mémoire
  - Problème de l'allocation d'une zone mémoire non traité
  - Utilisation d'un DSL (langage spécifique)
- **[Agosta et al., 2015b] : approche statique**
  - Coût mémoire important
- **Contribution : approche dynamique**
  - ✓ Modification dynamique de code efficace
  - ✓ Coût mémoire plus faible que les approches statiques
  - ✓ Solution proposée au problème des droits mémoire
  - ✓ Solution proposée au problème de l'allocation d'une zone mémoire
  - ✓ Approche automatisée

POLYMORPHISME DE CODE : APPLICATION

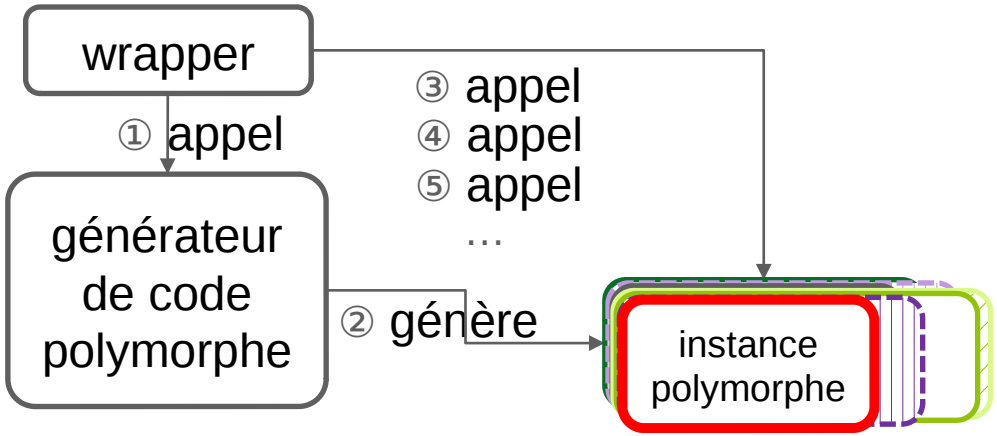
STATIQUEMENT



Odo

- Environ 3000 lignes de code
- Environ 800 commits

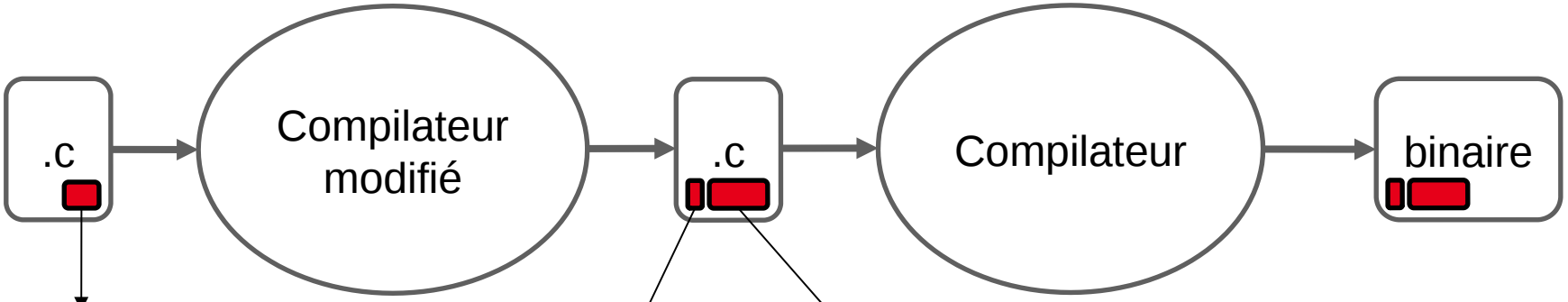
À L'EXÉCUTION



à l'exécution, une nouvelle instance polymorphe est générée régulièrement

POLYMORPHISME DE CODE : APPLICATION

STATIQUEMENT



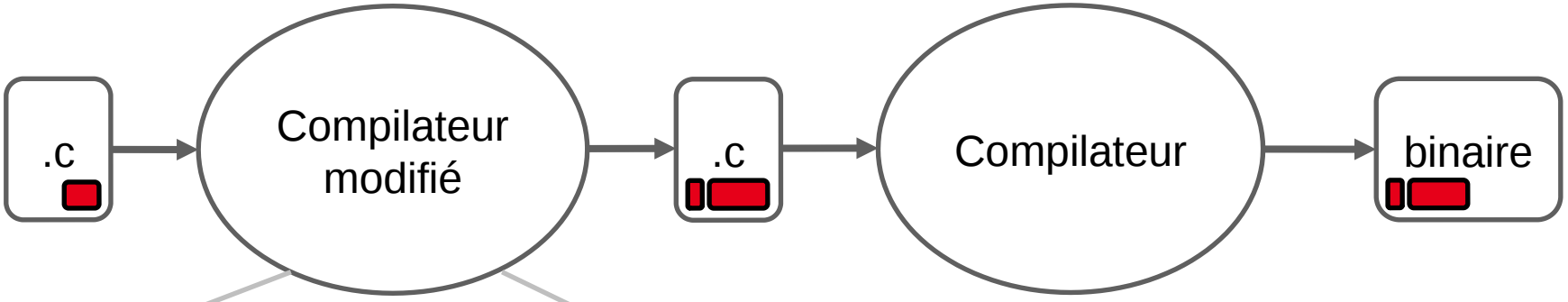

```
#pragma odo_polymorphic
int f_critical(int a, int b) {
        int c = a^b;
        a = a+b;
        a = a % c;
        return a;
}
```

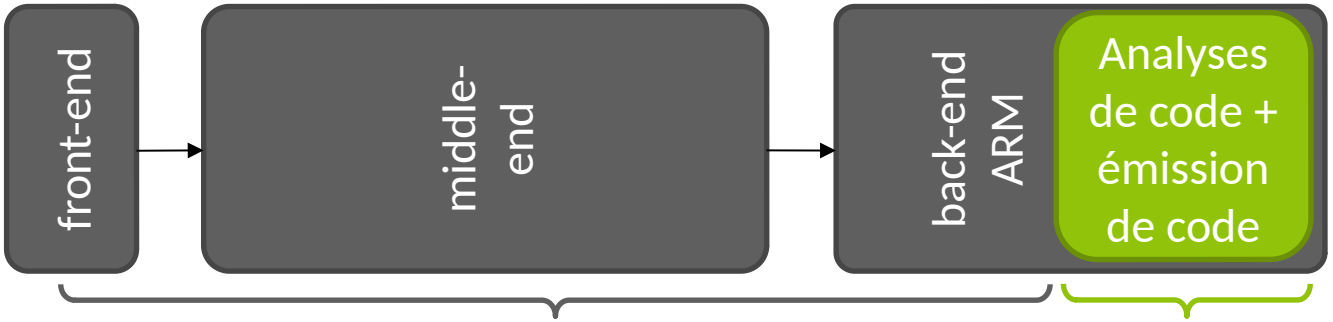
```
int f_critical(int a, int b) {
        if (SHOULD_BE_REGENERATED())
                SGPC_f_critical();
        return code_f(a, b);
}
```

```
void SGPC_f_critical() {
        raise_interrupt_rm_X_add_W(code_f);
        reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
        push_T2_callee_saved_registers();
        eor_T2(r[4], r[1], r[0]);
        add_T2(r[0], r[1], r[0]);
        sdiv_T2(r[1], r[0], r[4]);
        mls_T2(r[0], r[1], r[4], r[0]);
        pop_T2_callee_saved_registers();
        raise_interrupt_rm_W_add_X(code_f);
}
```

POLYMORPHISME DE CODE : APPLICATION

STATIQUEMENT





Code optimisé par le compilateur → le code généré à l'exécution a été optimisé statiquement

Analyses statiques pour la gestion mémoire et pour aider les transformations dynamiques

Émission de code C à partir de la suite d'instructions ARM

TRANSFORMATIONS DE CODE À L'EXÉCUTION

add r4, r4, r5
xor r6, r5, r8

Mélange de registres

permutation aléatoire des registres généralistes

r4 → r11

...

add r11, r11, r7
xor r8, r7, r5

Mélange d'instructions

émission dans un ordre aléatoire des instructions indépendantes

xor r6, r5, r8
add r4, r4, r5

Variantes sémantiques

remplacement d'une instruction par une variante sémantique choisie aléatoirement

add r4, r4, r5
xor r6, r5, #12348
xor r6, r6, r8
xor r6, r6, #12348

Instructions de bruit

insertion d'un nombre aléatoire d'instructions de bruit choisies aléatoirement

add r4, r4, r5
sub r7, r6, r2
load r3, r10, #53
xor r6, r5, r8

instructions de bruit

Bruit dynamique

insertion d'une suite d'instruction de bruits avec un saut aléatoire

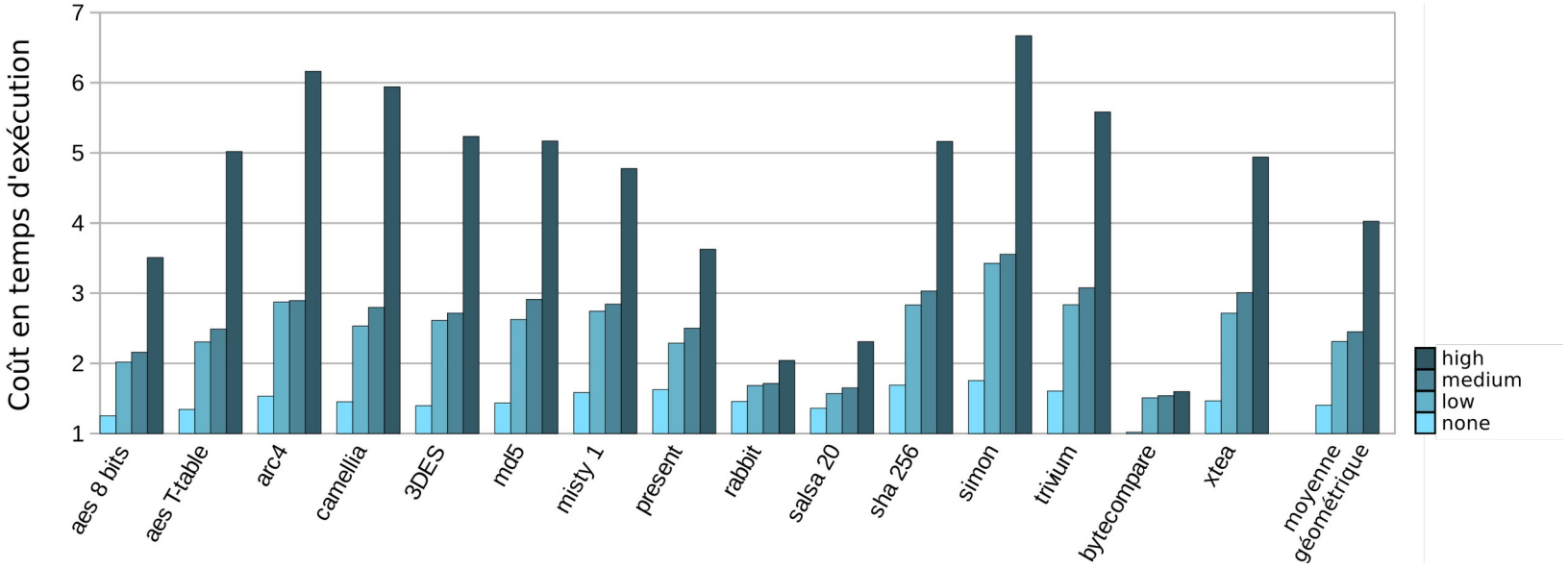
add r4, r4, r5
jump **0**, **1** or **2** instructions
sub r7, r6, r2
load r3, r10, #53
xor r6, r5, r8

CONFIGURATION DE LA CONTRE-MESURE

- **Approche fortement configurable**
  - Mélange de registres activé ou non
  - Mélange d'instructions activé ou non
  - Choix des variantes sémantiques
  - Choix de la distribution servant à choisir le nombre d'instructions de bruit insérer
  - Choix de la longueur des séquences du bruit dynamique
- **Rend la contre-mesure adaptable aux contraintes spécifiques**
  - Ressources de la plate-forme cible
  - Résistance de la plate-forme cible aux attaques
  - Niveau de sécurité souhaité
- **Pour la suite, 4 configurations retenues**
  - **none** : pas de polymorphisme
  - **low** : uniquement du bruit dynamique et du mélange de registres
  - **medium** : toutes les transformations activées
  - **high** : toutes les transformations activées, polymorphisme plus important

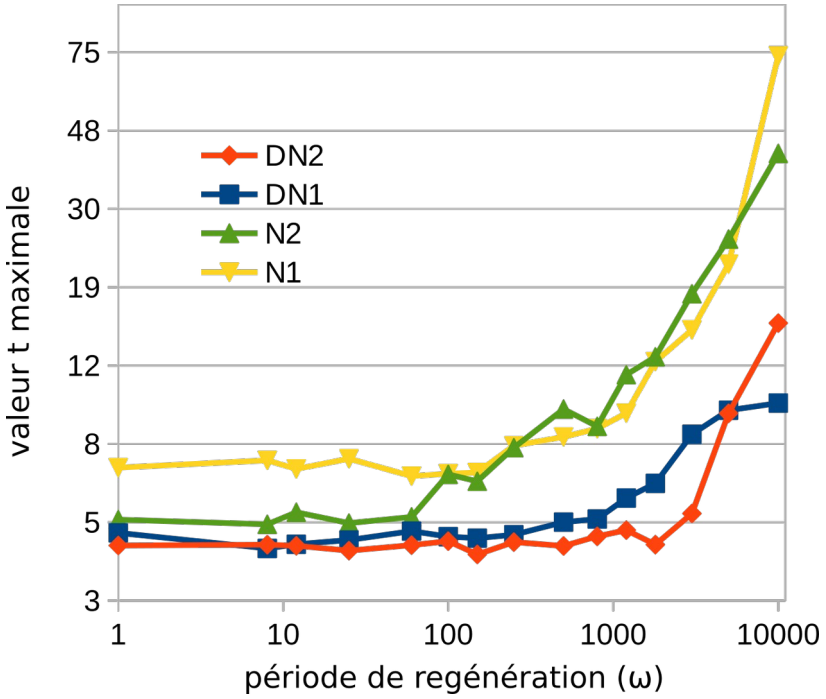
PRINCIPAUX RÉSULTATS

- **Approche complètement générique**
  - Tout types de programmes peuvent être renforcés
- **Coûts en performance moins dépendants du code source que dans le cas du masquage (x1.4-x90)**
  - ~ x1.5-x3.5 pour les configurations **low** et **medium**
  - ~ x1.5-x6.8 pour la configuration **high**



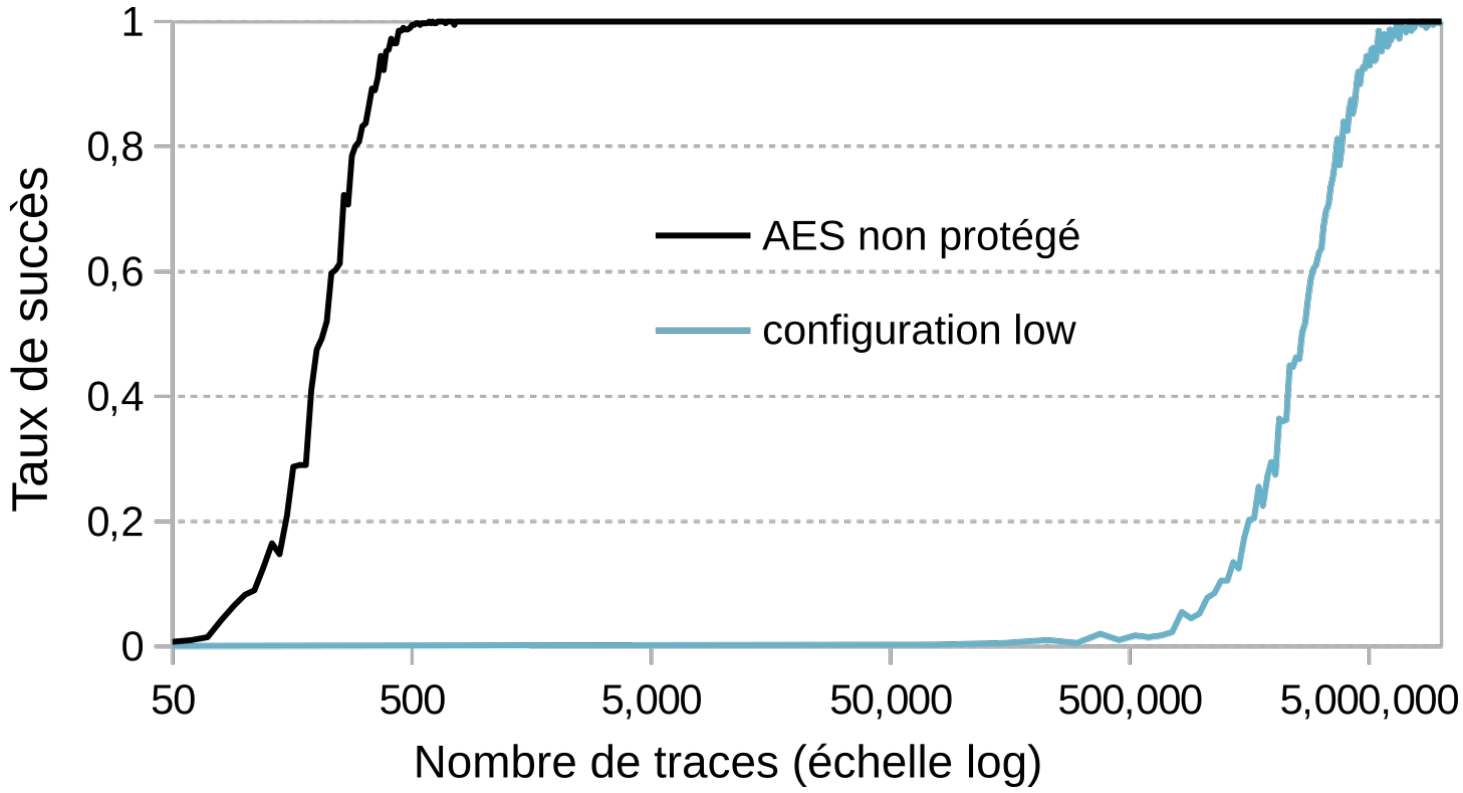
PRINCIPAUX RÉSULTATS

- **Génération dynamique de code très efficace**
  - Environ 8x plus rapide que l'état de l'art
- **Coût de la génération plus facilement amortissable grâce au bruit dynamique**
  - Permet de mieux conserver de la dissimulation entre deux regénérations, donc de régénérer le code moins souvent si nécessaire



PRINCIPAUX RÉSULTATS

- **Exemple de compromis avec la configuration low**
  - Bruit dynamique, 1 regénération toutes les 250 exécutions
  - Coût en performance : **x2,5** (temps de génération inclus)
  - Temps d'attaque en CPA **13000x** plus élevé :



CONCLUSION POLYMORPHISME DE CODE

- **Application du polymorphisme de code automatisée au sein de LLVM**
- **Approche générique et configurable**
  - configuration **low** : CPA 13000x plus longue sur un AES, pour un coût en temps d'exécution de 2,5x
- **Ajout de la transformation de bruit dynamique**
  - Permet de relâcher les contraintes sur la période de regénération
  - Meilleurs compromis dissimulation / performance

Article de revue

Automated software protection for the masses against side-channel attacks
N. Belleville, K. Heydemann, D. Couroussé, H.-P. Charles
ACM Transactions on Architecture and Code Optimization (TACO), January 2019

Brevet

Procédé d'exécution d'une fonction, par un microprocesseur, sécurisée par désynchronisation temporelle
N. Belleville, D. Couroussé
Institut National de la Propriété Industrielle, 23/07/2018, FR1856781

Vidéo de médiation scientifique

Processeurs, cuisines, et attaques par canal auxiliaire
N. Belleville
Accessit au concours ScienceInfoStream de la Société Informatique de France

SOMMAIRE

1. Introduction et contexte

2. Niveaux d’application de contre-mesures

3. Application d'une contre-mesure de masquage booléen d'ordre 1 à la compilation

4. Application d'une contre-mesure de polymorphisme de code à la compilation

5. Conclusion et perspectives

CONCLUSION GÉNÉRALE

- **Constat de départ : application des contre-mesures coûteuse, et plate-formes cibles très contraintes**
  - Application automatisée de contre-mesures à la compilation
  - Réduction des coûts des contre-mesures

- **Application du masquage**
  - Approche par interpolation polynomiale pour les tables constantes
  - Nouvelles optimisations proposées
  - Plusieurs binaires vérifiés formellement avec le modèle de fuite en valeur
  - Mise en évidence des problèmes liés à l'écart entre le modèle de fuite et la réalité

- **Application du polymorphisme de code**
  - Ajout de la transformation de bruit dynamique
  - Plusieurs apports pour la gestion de la mémoire
  - Génération dynamique de code très efficace
  - Approche très configurable, permettant d'obtenir de bons compromis entre dissimulation et performance

PERSPECTIVES

- **Masquage à l'ordre supérieur et autres schémas de masquage**

- **Combinaison des deux contre-mesures**
  - Utilisation du polymorphisme de code sur implémentation masquée
  - Exploitation les transformations de code dues au masquage pour créer de la dissimulation

- **Meilleure prise en compte de la réalité de la plate-forme**
  - Construction de modèles de fuite plus fidèles
  - Application du masquage dirigée par le modèle
  - Transformations utilisées pour le polymorphisme adaptées au modèle

- **Recherche du compromis sécurité / performance pour une contre-mesure fortement configurable par apprentissage machine**
  - Recherche d'une configuration par apprentissage machine
  - Adaptation de la contre-mesure en fonction des instructions de la fonction

MERCI DE VOTRE ATTENTION

- **Article de revue**
  - Automated software protection for the masses against side-channel attacks
  - N. Belleville, D. Couroussé, K. Heydemann, H.-P. Charles
  - ACM Transactions on Architecture and Code Optimization (TACO)
- **Chapitre d'ouvrage**
  - Automatic Application of Software Countermeasures Against Physical Attacks
  - N. Belleville, K. Heydemann, D. Couroussé, T. Barry, B. Robisson, A. Seriai, H.-P. Charles
  - Koç Ç.K. (eds) Cyber-Physical Systems Security, Springer
- **Brevet**
  - Procédé d'exécution d'une fonction, par un microprocesseur, sécurisée par désynchronisation temporelle
  - N. Belleville, D. Couroussé
  - Institut National de la Propriété Industrielle, 23/07/2018, Numéro de demande : FR1856781
- **Conférence avec acte**
  - All paths lead to Rome : Polymorphic Runtime Code Generation for Embedded Systems
  - D. Couroussé, T. Barry, B. Robisson, N. Belleville, P. Jaillon, O. Potin, H. Le Bouder, J.-L. Lanet, K. Heydemann
  - CS2-18 Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems
- **Logiciels**
  - Maskara : environ 4000 lignes de code et 150 commits dans le middle-end de LLVM
  - Odo : environ 3000 lignes de code et 800 commits dans le back-end ARM de LLVM et la librairie COGITO

RÉFÉRENCES

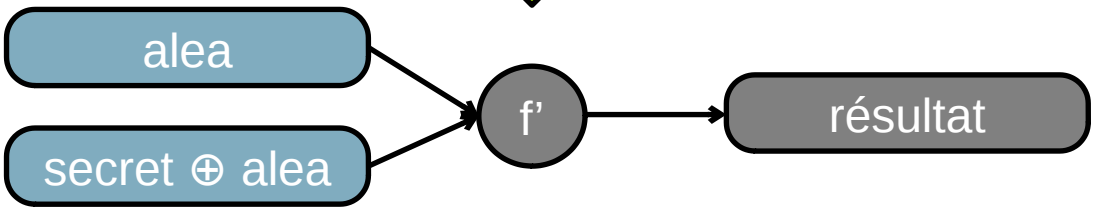
- [Malagón et al., 2012] Malagón, P., de, G., Zapater, M., Moya, J., et Banković, Z. (2012). Compiler optimizations as a countermeasure against side-channel analysis in MSP430-based devices. Sensors.
- [Moss et al., 2012] Moss, A., Oswald, E., Page, D., et Tunstall, M. (2012). Compiler assisted masking. LNCS.
- [Agosta et al., 2013] Agosta, G., Barenghi, A., Maggi, M., et Pelosi, G. (2013). Compiler-based side channel vulnerability analysis and optimized countermeasures application. In Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE
- [Eldib et Wang, 2014] Eldib, H. et Wang, C. (2014). Synthesis of Masking Countermeasures Against Side Channel Attacks. In International Conference on Computer Aided Verification. Springer.
- [Coron et al., 2014c] Coron, J.-S., Roy, A., et Vivek, S. (2014c). Fast evaluation of polynomials over binary nite elds and application to side-channel countermeasures. In International Workshop on Cryptographic Hardware and Embedded Systems. Springer.
- [Agosta et al., 2015a] Agosta, G., Barenghi, A., Pelosi, G., et Scandale, M. (2015a). Information leakage cha : feeding red herrings to side channel attackers. ACM.
- [Agosta et al., 2015b] Agosta, G., Barenghi, A., Pelosi, G., et Scandale, M. (2015b). The MEET approach : Securing cryptographic embedded software against side channel attacks. IEEE TCAD
- [Bayrak et al., 2015b] Bayrak, A., Regazzoni, F., Novo, D., Brisk, P., Standaert, F.-X., et Ienne, P. (2015). Automatic application of power analysis countermeasures. IEEE Transactions on Computers
- [Barry et al., 2016] Barry, T., Couroussé, D., et Robisson, B. (2016). Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In Proceedings of the Third Workshop on Cryptography and Security in Computing Systems. ACM.
- [Chen et al., 2017] Chen, Z., Shen, J., Nicolau, A., Veidenbaum, A., et Farhady, N. (2017). CAMFAS : A Compiler Approach to Mitigate Fault Attacks via Enhanced SIMDization. In 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). IEEE.
- [Luo et al., 2017] Luo, P., Athanasiou, K., Zhang, L., Jiang, Z. H., Fei, Y., Ding, A. A., et Wahl, T. (2017). Compiler-Assisted Threshold Implementation against Power Analysis Attacks. IEEE.
- [Proy et al., 2017] Proy, J., Heydemann, K., Berzati, A., et Cohen, A. (2017). Compiler-Assisted Loop Hardening Against Fault Attacks. ACM Trans. Archit. Code Optim.
- [Ben El Ouahma et al., 2017] Ben El Ouahma, I., Meunier, Q., Heydemann, K., et Encrenaz, E. (2017). Symbolic Approach for Side-Channel Resistance Analysis of Masked Assembly Codes. PROOFS.
- [Plank] Plank, J. Fast galois eld arithmetic library in C/C++. http://web.eecs.utk.edu/~plank/plank/papers/CS-07-593/
- [Ishai et al., 2003] Ishai, Y., Sahai, A., et Wagner, D. (2003). Private Circuits : Securing Hardware against Probing Attacks. Advances in Cryptology - CRYPTO 2003, Springer
- [Rivain et al., 2010] Rivain, M. et Prouff, E. (2010). Provably Secure Higher-Order Masking of AES. In Cryptographic Hardware and Embedded Systems, CHES 2010, Springer

MASQUAGE : EXTENSION AU MASQUAGE ARITHMÉTIQUE

Déclaration des variables secrètes → reste identique

Analyse de flot de données pour déterminer les instructions à transformer
- → ajout d'une passe de planification d'instructions
- → reste identique

Transformation des instructions en remasquant si nécessaire →
- ajout de transformations d'instructions
- ajout du support pour le passage d'un type de masquage à un autre



MASQUAGE ET OPTIMISATIONS

- **Dans quel cas une optimisation ou transformation peut casser le masquage ?**
  - Dépend du modèle de fuite
  - Par exemple :
    - allocation de registres ne pose aucun soucis avec un modèle de fuite en valeur, mais devient bien plus complexe avec un modèle de fuite en transition
    - idem pour la planification d'instructions (loads et stores)

- **Besoin de vérifier le binaire produit**

- **Besoin de propager les informations de masques**
  - Vérification après chaque passe que la contre-mesure reste valide

POLYMORPHISME DE CODE : OBSERVATION DE L'EXÉCUTION DU GÉNÉRATEUR

- **Observation de l'exécution du générateur par attaque par canal auxiliaire**
  - Le comportement du générateur de code ne dépend pas du secret
  - En déterminant le flot de contrôle emprunté par le générateur, possible de faire des attaques en 2 phases : une phase d'observation du générateur, et une phase de filtrage des observations des instances
  - Détermination des emplacements d'insertion d'instructions de bruit, des variantes sémantiques choisies, etc
  - Permettrait d'enlever une grosse partie de la désynchronisation en réalignant les traces
  - Importance du bruit dynamique : source de désynchronisation indépendante de la génération → complique le travail de réalignement
  - Polymorphisme de code : effet plus large que de la désynchronisation
    - fuites en transitions des accès mémoire modifiées par les loads aléatoires
    - fuites en transitions de l'ALU modifiées par les xors, adds, et subs aléatoires
    - fuites en valeurs modifiées par les variantes sémantiques

POLYMORPHISME DE CODE : ATTAQUE PAR INJECTION DE FAUTES

- **Injection de fautes sur le code polymorphe**
  - Simple faute : la contre-mesure peut simplifier l'attaque
  - Double faute : la contre-mesure complexifie l'attaque

- **Injection de fautes sur le générateur de nombre aléatoire**
  - Risque important pour toute contre-mesure basée sur l'aléa
  - Polymorphisme de code probablement un peu moins sensible que du masquage
  - Sécuriser le générateur de nombres aléatoire contre les fautes

- **Injection de fautes sur le générateur de code polymorphe**
  - Permettrait en théorie de :
    - éviter la regénération
    - ne pas émettre une instruction en mémoire
    - inverser l'ordre de deux instructions non indépendantes (si mélange d'instructions)
    - insérer des instructions de bruit qui utilisent des registres vivants

MASQUAGE : TRANSFORMATION DES TABLES DE SUBSTITUTIONS

- **Exemple SBox de PRESENT :**
  uint8_t **SBox[**16**] = {**0xC**,** 0x5**,** 0x6**,** 0xB**,** 0x9**,** 0x0**,** 0xA**,** 0xD**,** 0x3**,** 0xE**,** 0xF**,**
  0x8**,** 0x4**,** 0x7**,** 0x1**,** 0x2**};**

- **But : trouver un polynôme interpolateur de la table facile à masquer**
  - xors et opérations linéaires faciles à masquer
    - choix d'un corps fini où l'addition est le xor
  - multiplication non linéaire induit un surcoût important
    - privilégier les opérations linéaires
    - l'élévation au carré est linéaire par rapport au xor :
      - (a xor b) * (a xor b) = (a * a) xor ~~(a * b) xor (b * a)~~ xor (b * b)
        = (a * a) xor (b * b)
  - recherche du polynôme sous une forme particulière
    - peu d'opérations non linéaires

MASQUAGE : TRANSFORMATION DES TABLES DE SUBSTITUTIONS

- **Exemple SBox de PRESENT :**
  uint8_t **SBox[**16**] = {**0xC**,** 0x5**,** 0x6**,** 0xB**,** 0x9**,** 0x0**,** 0xA**,** 0xD**,** 0x3**,** 0xE**,** 0xF**,**
  0x8**,** 0x4**,** 0x7**,** 0x1**,** 0x2**};**

- **Exemple de polynôme interpolateur dans $\mathbb{F}_{2^4}$ muni du polynôme primitif $X^4+X^1+X^0$:**
  $P[X] = q_1(X) * p_1(X) + p_2(X)$
  - **$q_1[X] = (12X^{12} + 14X^9 + 7X^6 + 15X^3) + (5X^8 + 3X^4 + 3X^2 + 8X) + (8)$**
  - **$p_1[X] = (3X^{12} + 15X^9 + 7X^6 + 4X^3) + (14X^8 + 8X^4 + 1X^2 + 11X) + (5)$**
  - **$p_2[X] = (0X^{12} + 0X^9 + 1X^6 + 15X^3) + (9X^8 + 11X^4 + 10X^2 + 5X) + (2)$**

MASQUAGE : TRANSFORMATION DES TABLES DE SUBSTITUTIONS

- **Exemple SBox de PRESENT :**
  uint8_t **SBox[**16**] = {**0xC**,** 0x5**,** 0x6**,** 0xB**,** 0x9**,** 0x0**,** 0xA**,** 0xD**,** 0x3**,** 0xE**,** 0xF**,**
  0x8**,** 0x4**,** 0x7**,** 0x1**,** 0x2**};**

- **Exemple de polynôme interpolateur dans $\mathbb{F}_{2^4}$ muni du polynôme primitif $X^4+X^1+X^0$:**
  $P[X] = q_1(X) * p_1(X) + p_2(X)$
  - **$q_1[X] = (12X^{12} + 14X^9 + 7X^6 + 15X^3) + (5X^8 + 3X^4 + 3X^2 + 8X) + (8)$**
  - **$p_1[X] = (3X^{12} + 15X^9 + 7X^6 + 4X^3) + (14X^8 + 8X^4 + 1X^2 + 11X) + (5)$**
  - **$p_2[X] = (0X^{12} + 0X^9 + 1X^6 + 15X^3) + (9X^8 + 11X^4 + 10X^2 + 5X) + (2)$**
  - **xors, faciles à masquer**

MASQUAGE : TRANSFORMATION DES TABLES DE SUBSTITUTIONS

- **Exemple SBox de PRESENT :**
  uint8_t **SBox[**16**] = {**0xC**,** 0x5**,** 0x6**,** 0xB**,** 0x9**,** 0x0**,** 0xA**,** 0xD**,** 0x3**,** 0xE**,** 0xF**,** 0x8**,** 0x4**,** 0x7**,** 0x1**,** 0x2**};**

- **Exemple de polynôme interpolateur dans $\mathbb{F}_{2^4}$ muni du polynôme primitif $X^4+X^1+X^0$:**
  $P[X] = q_1(X) * p_1(X) + p_2(X)$
  - **$q_1[X] = (12X^{12} + 14X^9 + 7X^6 + 15X^3) + (5X^8 + 3X^4 + 3X^2 + 8X) + (8)$**
  - **$p_1[X] = (3X^{12} + 15X^9 + 7X^6 + 4X^3) + (14X^8 + 8X^4 + 1X^2 + 11X) + (5)$**
  - **$p_2[X] = (0X^{12} + 0X^9 + 1X^6 + 15X^3) + (9X^8 + 11X^4 + 10X^2 + 5X) + (2)$**
  - **élévations au carré, faciles à masquer**

MASQUAGE : TRANSFORMATION DES TABLES DE SUBSTITUTIONS

- **Exemple SBox de PRESENT :**
  uint8_t **SBox[**16**] = {**0xC**,** 0x5**,** 0x6**,** 0xB**,** 0x9**,** 0x0**,** 0xA**,** 0xD**,** 0x3**,** 0xE**,** 0xF**,**
  0x8**,** 0x4**,** 0x7**,** 0x1**,** 0x2**};**

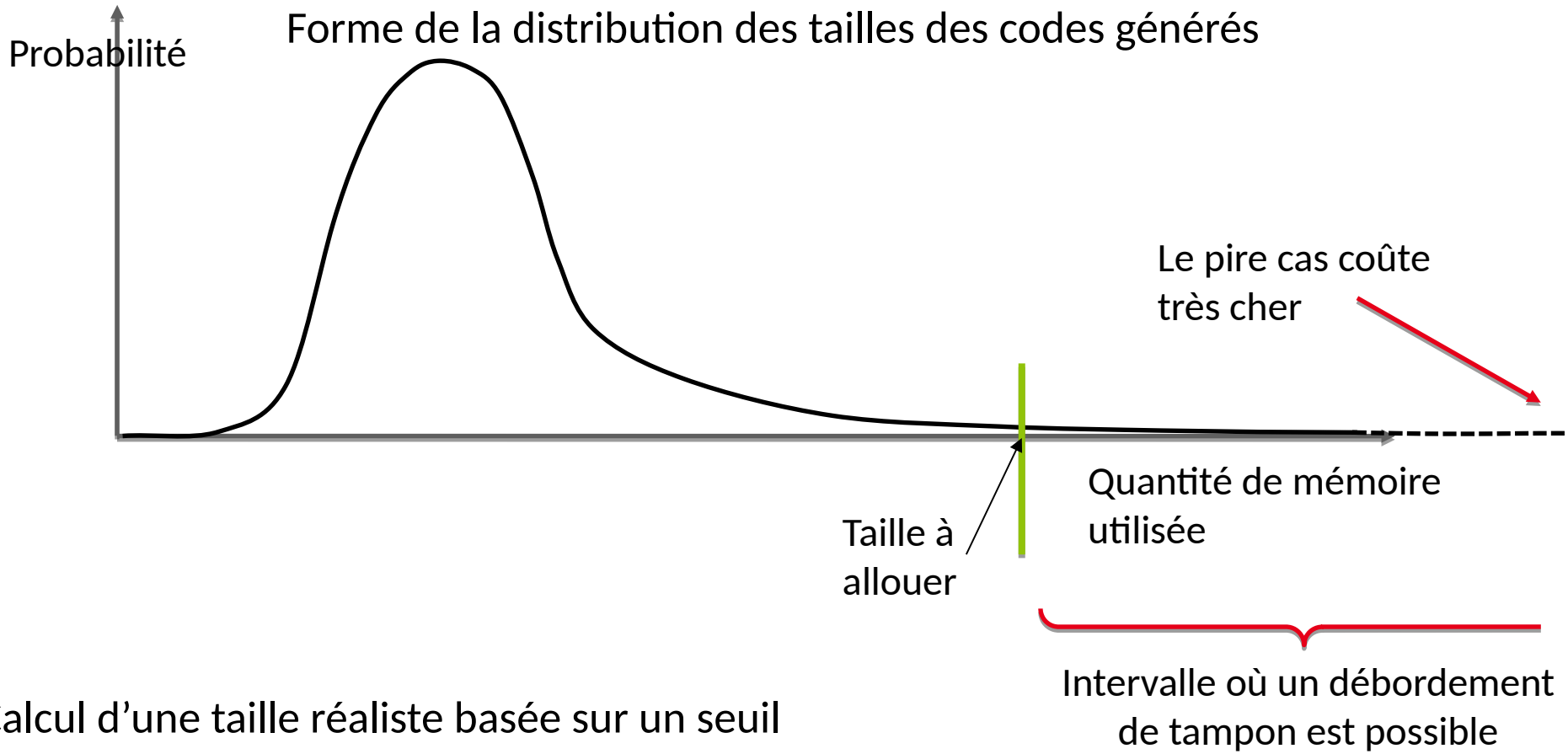
- **Exemple de polynôme interpolateur dans $\mathbb{F}_{2^4}$ muni du polynôme primitif $X^4+X^1+X^0$:**
  $P[X] = q_1(X) * p_1(X) + p_2(X)$
  - **$q_1[X] = (12X^{12} + 14X^9 + 7X^6 + 15X^3) + (5X^8 + 3X^4 + 3X^2 + 8X) + (8)$**
  - **$p_1[X] = (3X^{12} + 15X^9 + 7X^6 + 4X^3) + (14X^8 + 8X^4 + 1X^2 + 11X) + (5)$**
  - **$p_2[X] = (0X^{12} + 0X^9 + 1X^6 + 15X^3) + (9X^8 + 11X^4 + 10X^2 + 5X) + (2)$**
  - **multiplications par des constantes, faciles à masquer**

MASQUAGE : TRANSFORMATION DES TABLES DE SUBSTITUTIONS

- **Exemple SBox de PRESENT :**
  uint8_t **SBox[**16**] = {**0xC**,** 0x5**,** 0x6**,** 0xB**,** 0x9**,** 0x0**,** 0xA**,** 0xD**,** 0x3**,** 0xE**,** 0xF**,**
  0x8**,** 0x4**,** 0x7**,** 0x1**,** 0x2**};**

- **Exemple de polynôme interpolateur dans $\mathbb{F}_{2^4}$ muni du polynôme primitif $X^4+X^1+X^0$:**
  $P[X] = q_1(X) * p_1(X) + p_2(X)$
  - **$q_1[X] = (12X^{12} + 14X^9 + 7X^6 + 15X^3) + (5X^8 + 3X^4 + 3X^2 + 8X) + (8)$**
  - **$p_1[X] = (3X^{12} + 15X^9 + 7X^6 + 4X^3) + (14X^8 + 8X^4 + 1X^2 + 11X) + (5)$**
  - **$p_2[X] = (0X^{12} + 0X^9 + 1X^6 + 15X^3) + (9X^8 + 11X^4 + 10X^2 + 5X) + (2)$**
  - **multiplications non linéaires, coûteuses à masquer**
  - il y en a peu !

POLYMORPHISME : ALLOCATION STATIQUE DE LA MÉMOIRE

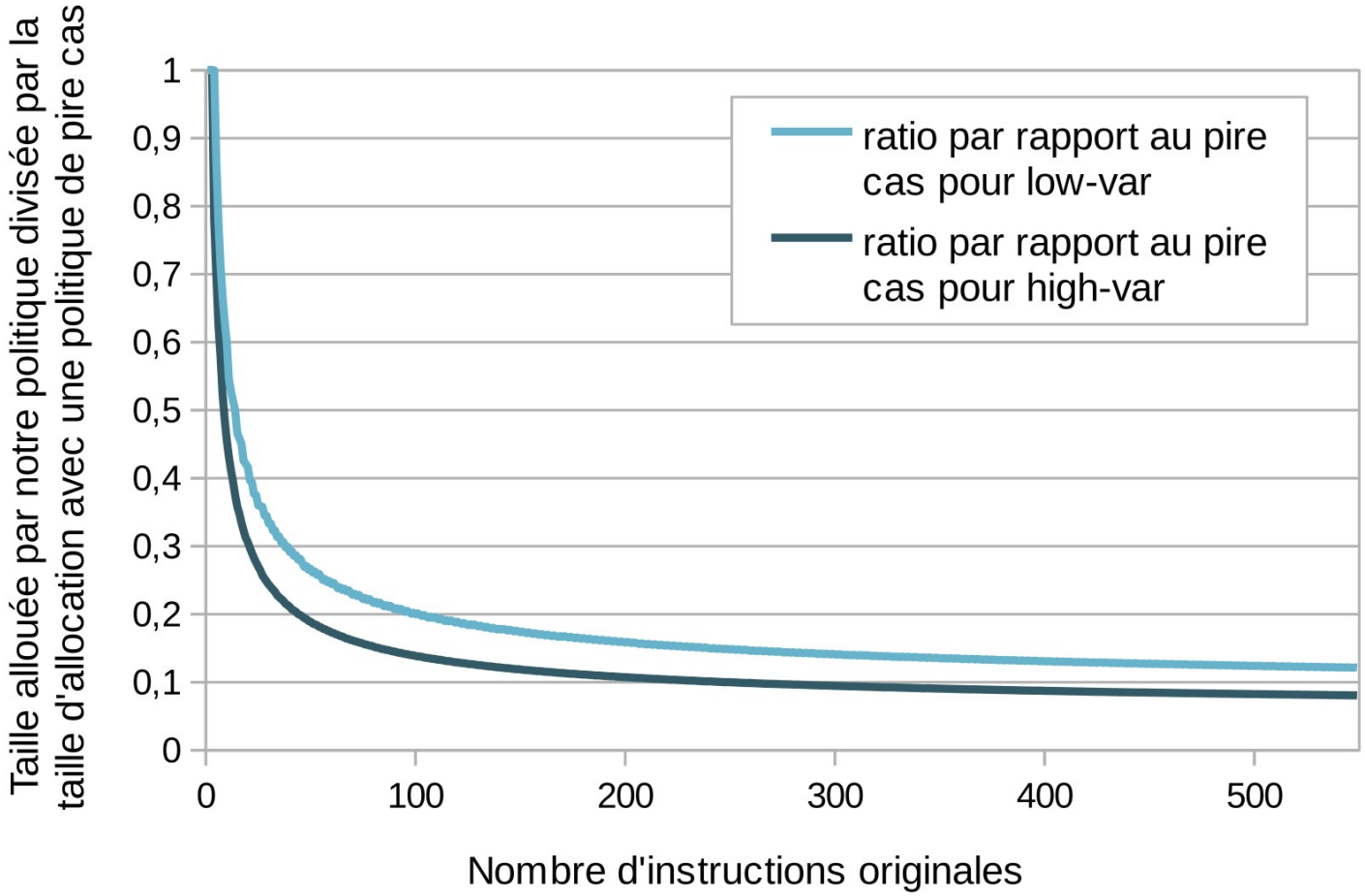


Calcul d'une taille réaliste basée sur un seuil

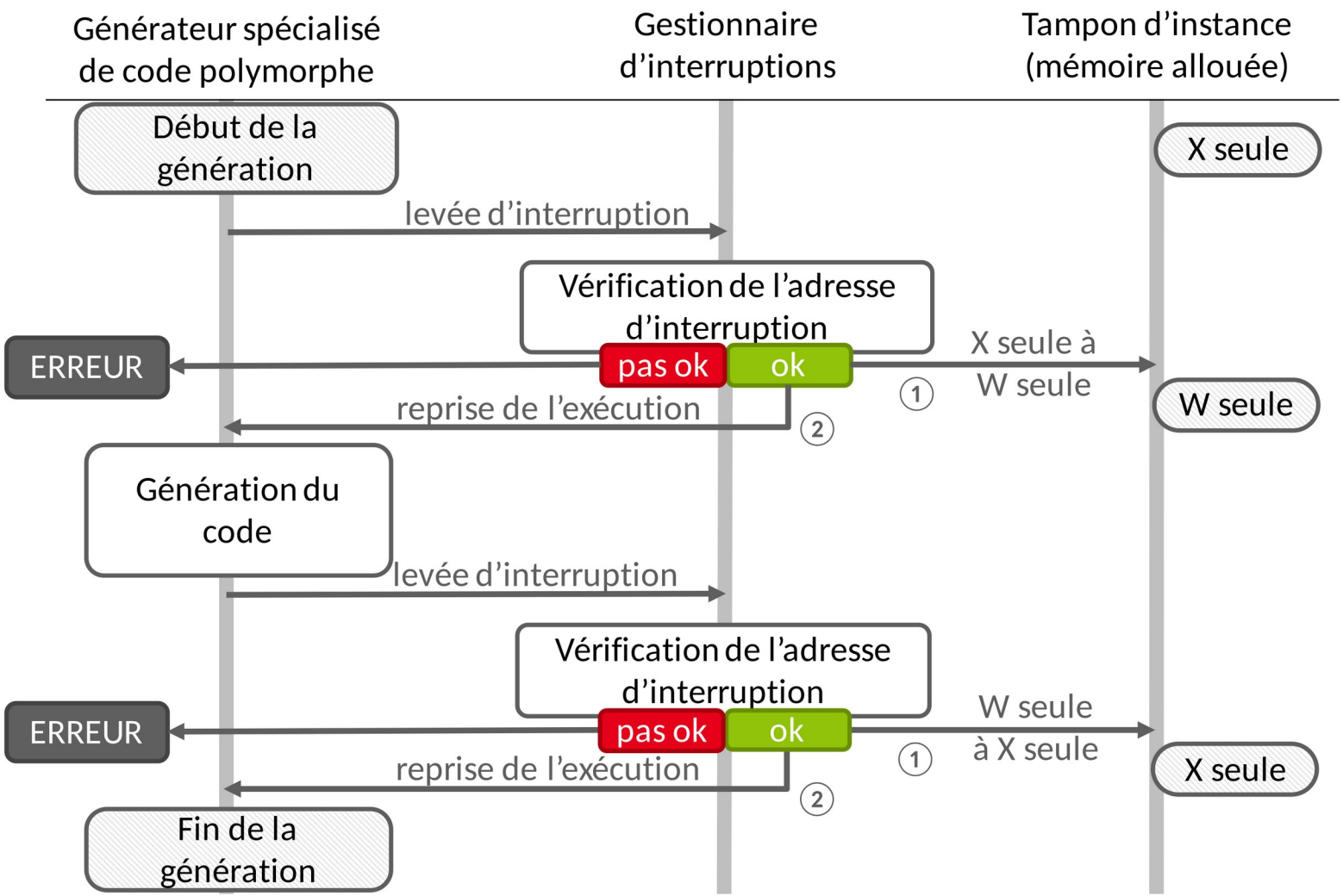
seuil : probabilité d'obtenir un débordement de tampon 10^{-6} par défaut

Pour un code de 100 instructions, taille allouée **5x** plus petite que le pire cas!
(configuration low)

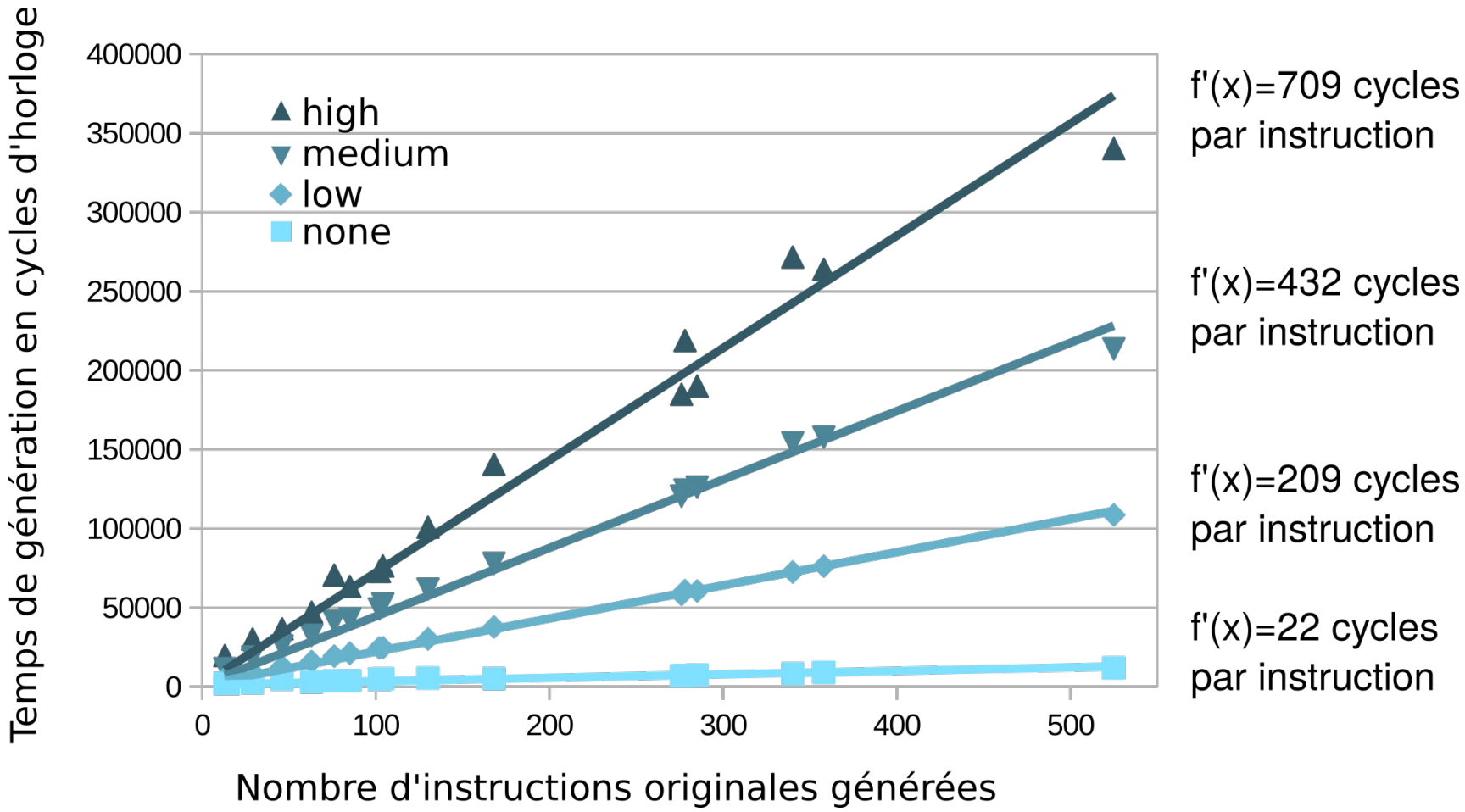
POLYMORPHISME : ALLOCATION STATIQUE DE LA MÉMOIRE



POLYMORPHISME : GESTION DES PERMISSIONS MÉMOIRE



POLYMORPHISME : TEMPS DE GÉNÉRATION



MASQUAGE : OPTIMISATION DE LA MULTIPLICATION

```
uint8_t log[16] = {0, 0, 1, 4, 2, 8, 5, 10, 3, 14, 9, 7, 6, 13, 11, 12};
uint8_t alog[15] = {1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9};

uint8_t mul(uint8_t a, uint8_t b) {
    return ((a != 0) && (b != 0)) * alog[(log[a] + log[b]) %15];
}

uint8_t log[16] = {0, 0, 1, 4, 2, 8, 5, 10, 3, 14, 9, 7, 6, 13, 11, 12};
uint8_t alog[30] = {1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9,
                    1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9};
uint8_t mul(uint8_t a, uint8_t b) {
    return ((a != 0) && (b != 0)) * alog[log[a] + log[b]];
}

uint8_t log[16] = {31, 0, 1, 4, 2, 8, 5, 10, 3, 14, 9, 7, 6, 13, 11, 12};
uint8_t alog[63] = {1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9,
                    1, 2, 4, 8, 3, 6, 12, 11, 5, 10, 7, 14, 15, 13, 9,
                    0, 0, 0, …, 0};
uint8_t mul(uint8_t a, uint8_t b) {
    return alog[log[a] + log[b]];
}
```