Protection des boucles à la compilation contre les attaques par fautes

Julien PROY[1], Karine HEYDEMANN[2], Alexandre BERZATI[1], Albert COHEN[3]

[1]Invia Secure Semiconductor Meyreuil, France
[2]Sorbonne Université, UPMC Univ Paris 06, CNRS, Lip6, Paris, France
[3]INRIA and DI, Ecole Normale Supérieure, Paris, France

23 Mai | JAIF 2019 Grenoble

- **Introduction & contexte**
  - Attaques physiques
  - Contre-mesures
- Schéma de protection des boucles
  - Principe
  - Implémentation dans LLVM
- Résultats expérimentaux
- Conclusion

Systèmes embarqués & IoT

- 2000’s: Composants sécurisés conçus pour être resistant aux attaques

Systèmes embarqués & IoT

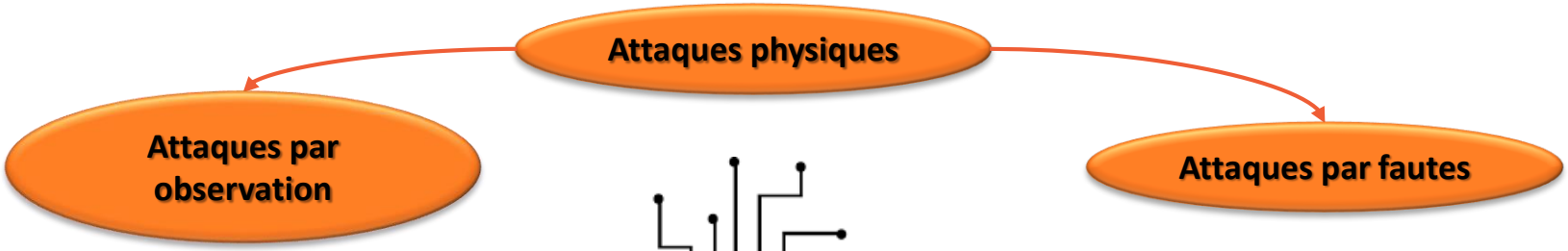
- 2000’s: Composants sécurisés conçus pour être resistant aux attaques



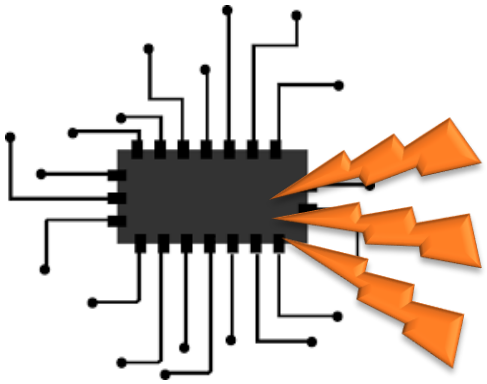
- 2010’s: Multitude de composants contenant des données sensibles




Attaques physiques



- Temps
- Consommation
- Température



- Glitch d'horloge
- Laser
- Impulsion électromagnétique

But commun: Récupérer des données sensibles / contourner des protections

- Besoin de se protéger contre ces attaques
  - Protections **matérielles** pas suffisantes, besoin de protections **logicielles**
  - Coût en **performances** et empreinte mémoire
  - Coût de development élevé car appliqué manuellement par des experts en sécurité
- Besoin d'approches génériques et d'outils pour automatiser

Codes attaqués: boucles

- Plusieurs attaques visent le nombre d'itération
  - Sur codes cryptographiques: **AES** [Demirci et al. 2008][Dehbaoui et al. 2013]
  - Sur codes système: **Buffer overflow** [Nashimoto et al. 2016], **PIN verification** [Dureuil et al. 2016]

- Propriétés de sécurité à préserver
  - Assurer l'execution du bon **nombre d'iterations**
  - Assurer que la **bonne sortie** de boucle est prise

```
int diff = 0;
for (int i=0; i<n; i++) {
    if (A[i] != B[i]) {
        diff = 1;
        break;
    }
}
...
```

- Contre quels effets?

Modèle de faute

- La conception de contre-mesure requiert un modèle précis
- **Caractérisation** des effets des fautes

- Modèles de fautes classique visant des smart card
  - Mise à 0/1 ou inversion de bit simple ou multiple
  - Corruption aléatoire de registre
  - Remplacement ou saut d'instruction
  - Altération des transferts mémoire/CPU

Modèle de faute

- La conception de contre-mesure requiert un modèle précis
- **Caractérisation** des effets des fautes

- Modèles de fautes classique visant des smart card
  - Mise à 0/1 ou inversion de bit simple ou multiple
  - Corruption aléatoire de registre
  - Remplacement ou saut d'instruction
  - Altération des transferts mémoire/CPU

- Modèles de fautes sélectionnés
  - **Saut d'instruction**
  - **Corruption** aléatoire de **registres** globaux

Contre-mesures logicielles

Contre-mesures logicielles

[2014] *Lalande et al.*

Niveau code source

- Plus compréhensible
- Optimisations du compilateur compromettent la sécurité
- Vérification manuelle de l'assembleur
- Pas de correspondence des modèles de faute

```
int i = 0, j = 0;
while (i != n) {
    dst[i] = src[i];
    i++;
    j++;
}
```

Contre-mesures logicielles

[2014] *Lalande et al.*

Niveau code source

- Plus compréhensible
- Optimisations du compilateur compromettent la sécurité
- Vérification manuelle de l'assembleur
- Pas de correspondence des modèles de faute

```
int i = 0, j = 0;
while (i != n) {
    dst[i] = src[i];
    i++;
    j++;
}
```

Contre-mesures logicielles

[2014] *Lalande et al.*

Niveau code source

- ✚ Plus compréhensible
- ▬ Optimisations du compilateur compromettent la sécurité
- ▬ Vérification manuelle de l'assembleur
- ▬ Pas de correspondence des modèles de faute

```
int i = 0, j = 0;
while (i != n) {
    dst[i] = src[i];
    i++;
    j++;
}
```

[2016] *De Keulenaer et al.*

Niveau code binaire

- ✚ Modèle de faute plus réaliste
- ✚ Code source non requis
- ▬ Manque d'information sémantique
- ▬ Comment obtenir des registres disponibles

…
1001010010010101100 1
1100101001010001010 1
1100101001010001010 1
1100101010111100 1010
…

Contre-mesures logicielles

Compilation

[2014] *Lalande et al.*

Niveau code source

- ✚ Plus compréhensible
- ▬ Optimisations du compilateur compromettent la sécurité
- ▬ Vérification manuelle de l'assembleur
- ▬ Pas de correspondence des modèles de faute

```
int i = 0, j = 0;
while (i != n) {
    dst[i] = src[i];
    i++;
    j++;
}
```

[2016] *Barry et al.*

Niveau compilateur

- ✚ Automatique
- ✚ Analyses du compilateur
- ✚ Surcout limité
- ✚ De plus en plus étudié
- ▬ Optimisations en aval

[2016] *De Keulenaer et al.*

Niveau code binaire

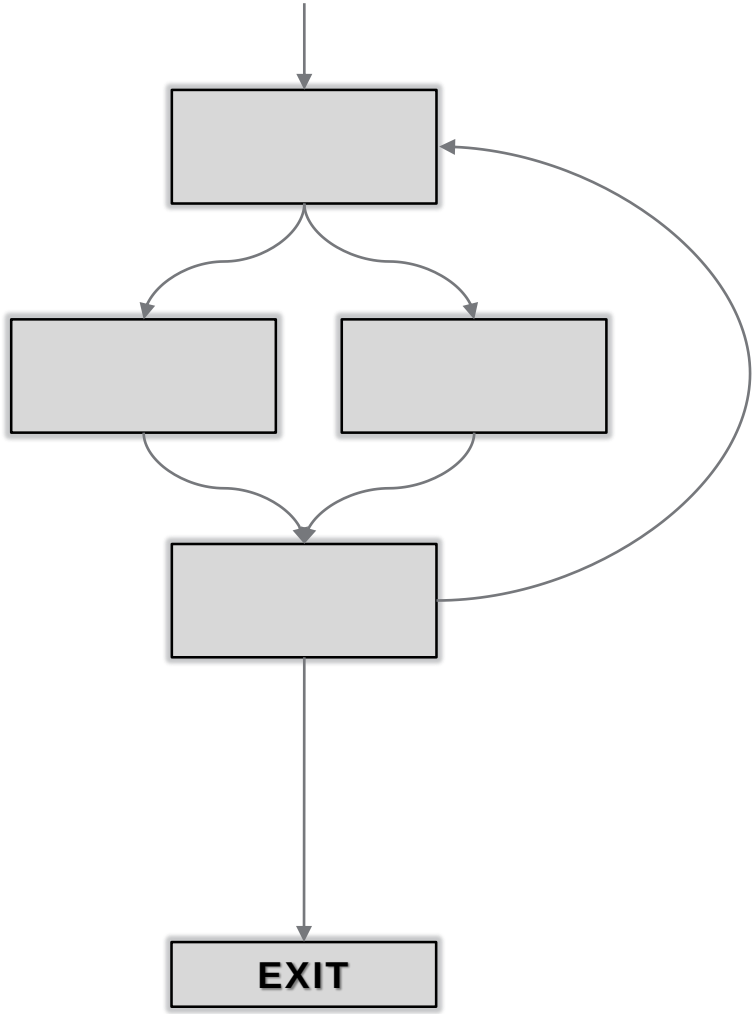
- ✚ Modèle de faute plus réaliste
- ✚ Code source non requis
- ▬ Manque d'information sémantique
- ▬ Comment obtenir des registres disponibles

…
1001010010010101100 1
1100101001010001010 1
1100101001010001010 1
1100101010111100 1010
…

- Introduction & contexte
  - Attaques physiques
  - Contre-mesures
- **Schéma de protection des boucles**
  - Principe
  - Implémentation dans LLVM
- Résultats expérimentaux
- Conclusion

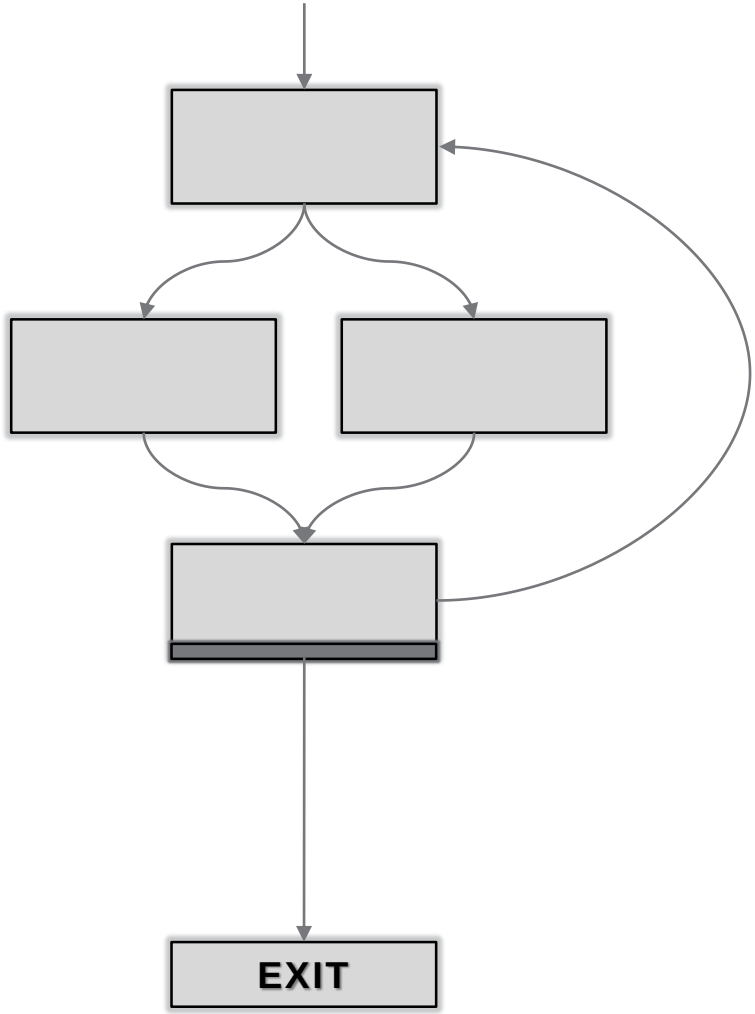
Protection des boucles: principe

- **Propriété de sécurité**
  Assurer le bon nombre d'iteration et la bonne sortie de boucle
- **Principe**
  Dupliquer les instructions impliquées dans le calcul d'une condition de sortie et ajouter des blocs de contrôle
- **Originalité**
  Coût réduit en ne ciblant que les instructions utiles



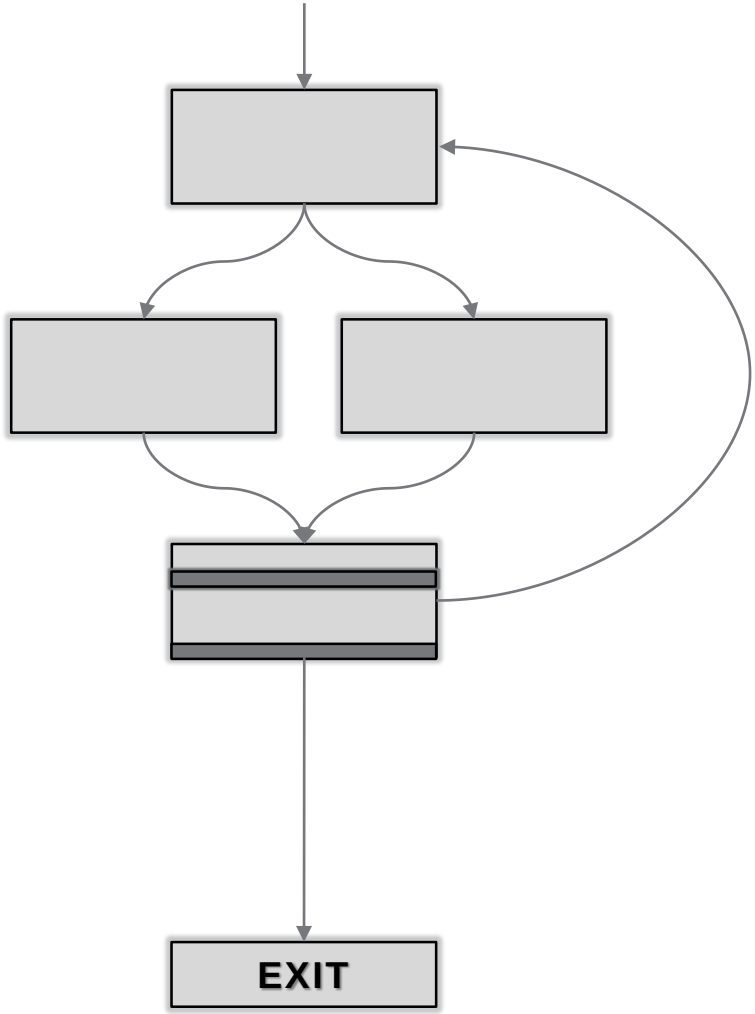
Protection des boucles: principe

- Partir d'une condition de sortie



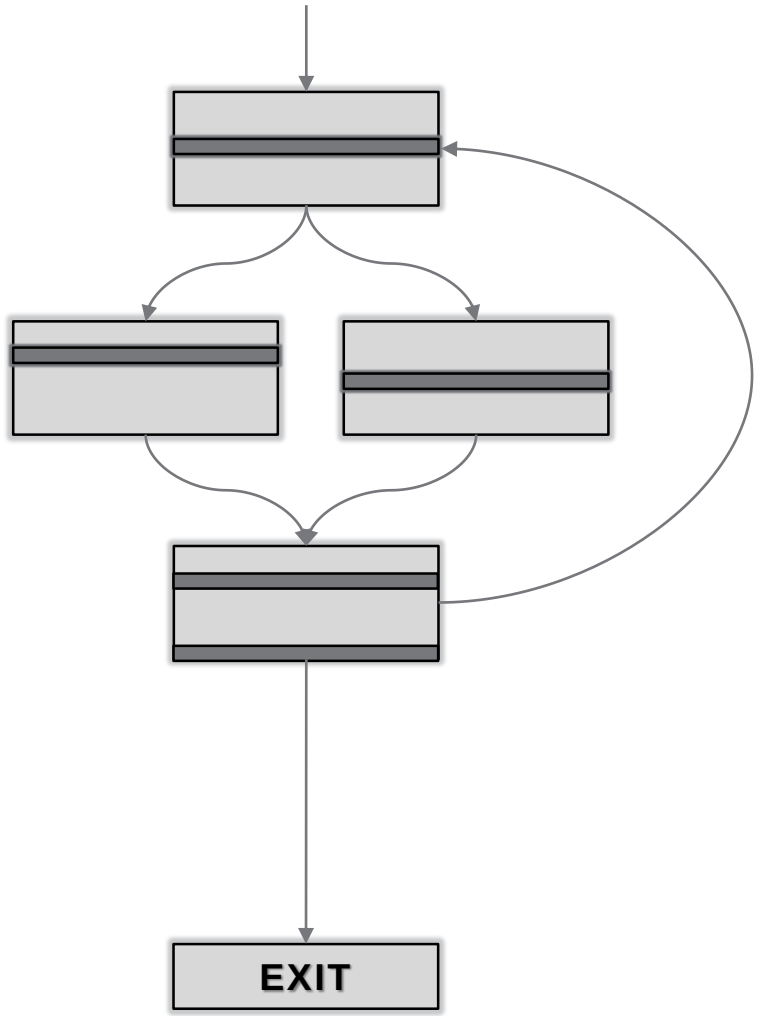
Protection des boucles: principe

- Partir d'une condition de sortie
- Analyse arrière pour determiner les instructions impliquées



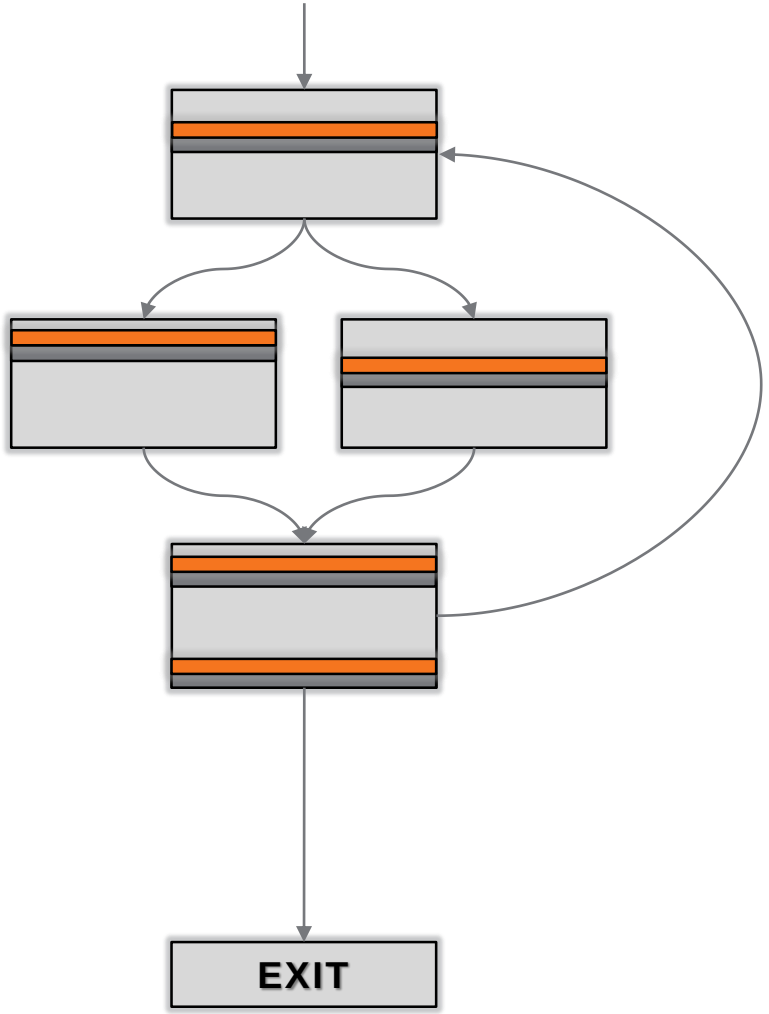
Protection des boucles: principe

- Partir d'une condition de sortie
- Analyse arrière pour determiner les instructions impliquées



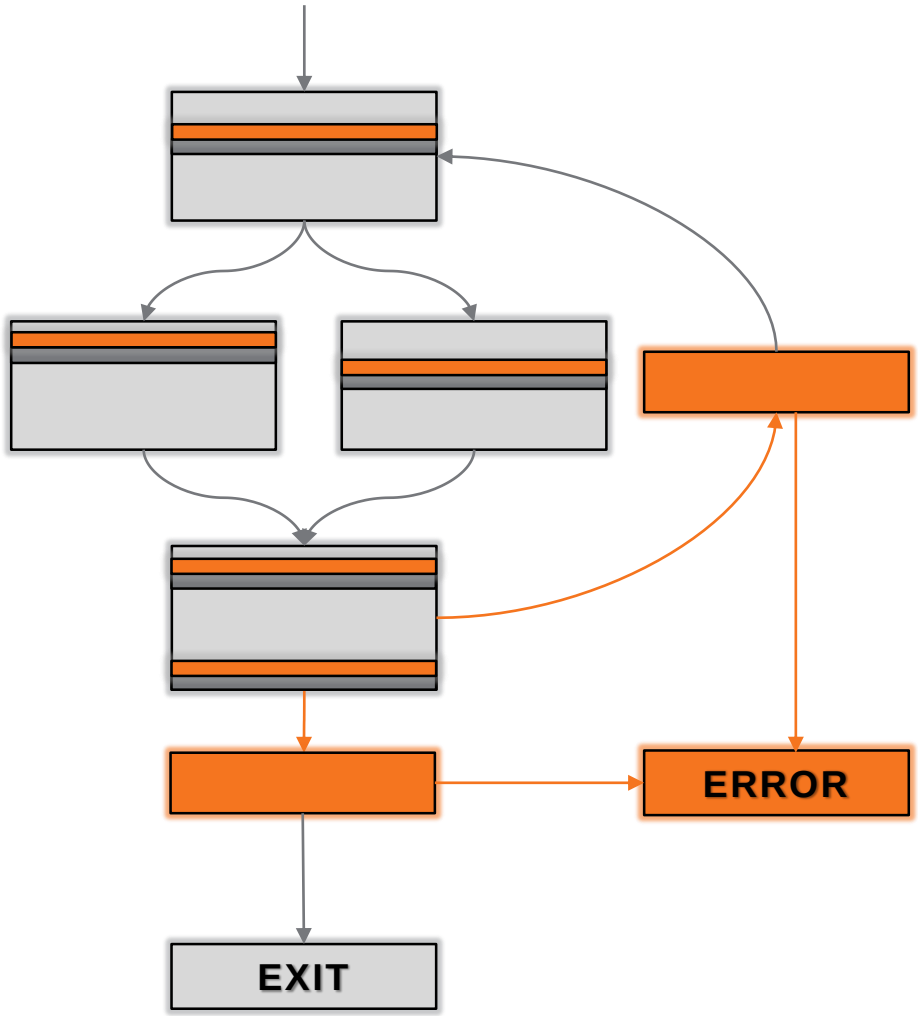
Protection des boucles: principe

- Partir d'une condition de sortie
- Analyse arrière pour determiner les instructions impliquées
- Dupliquer les instructions sélectionnées



Protection des boucles: principe

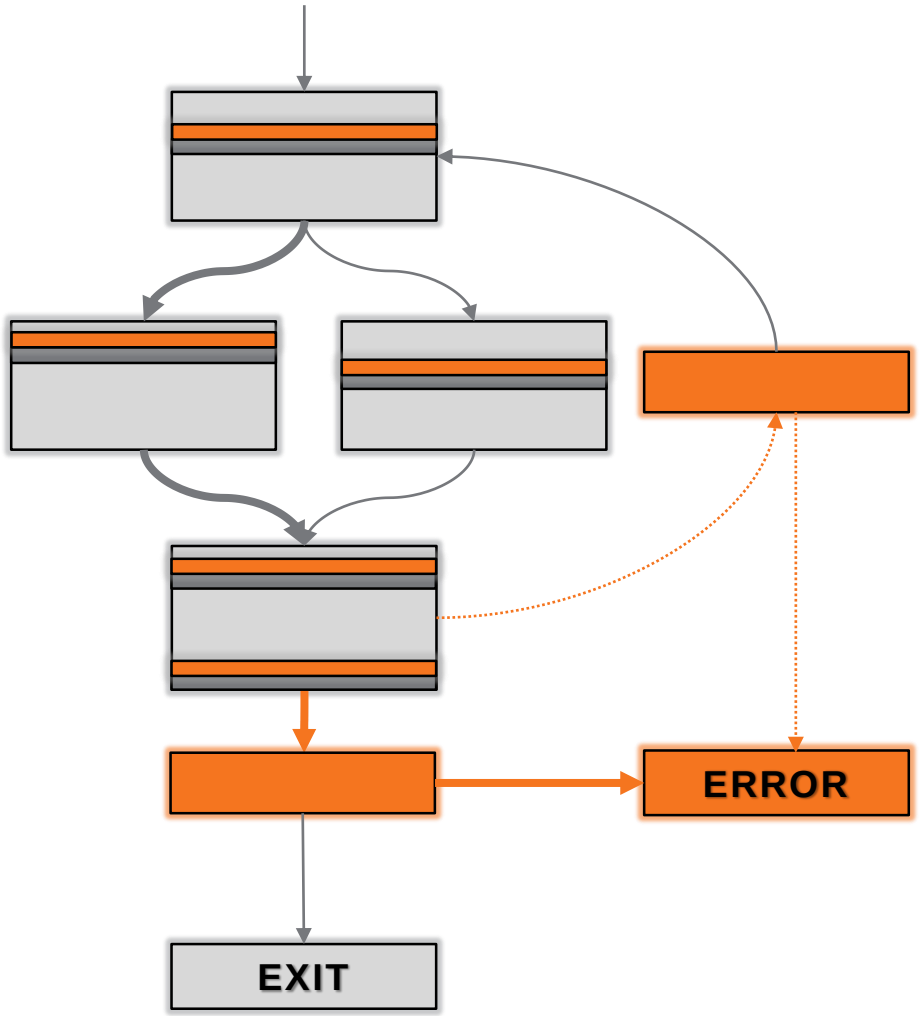
- Partir d'une condition de sortie
- Analyse arrière pour determiner les instructions impliquées
- Dupliquer les instructions sélectionnées
- Création de bloc:
  - Vérifier la condition
  - Gestion d'erreur



- Pour chaque boucle et sortie

Protection des boucles: principe

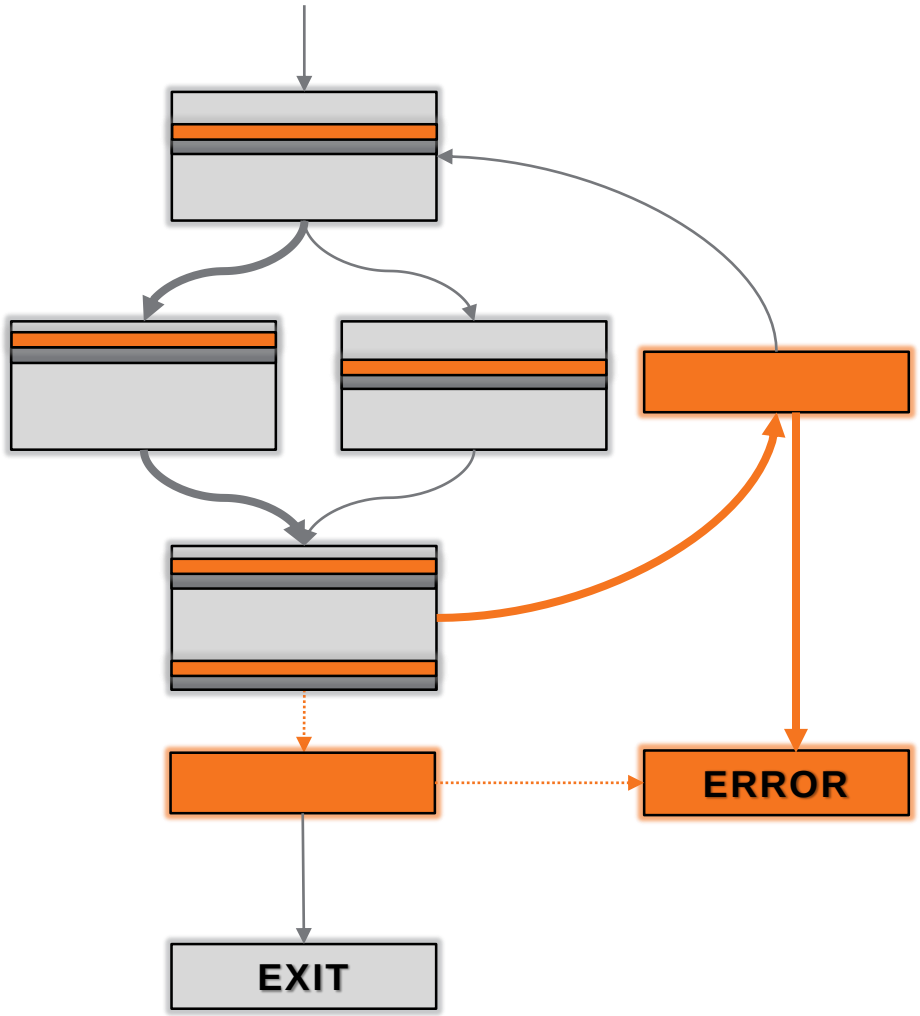
- Partir d'une condition de sortie
- Analyse arrière pour determiner les instructions impliquées
- Dupliquer les instructions sélectionnées
- Création de bloc:
  - Vérifier la condition
  - Gestion d'erreur



- Pour chaque boucle et sortie

Protection des boucles: principe

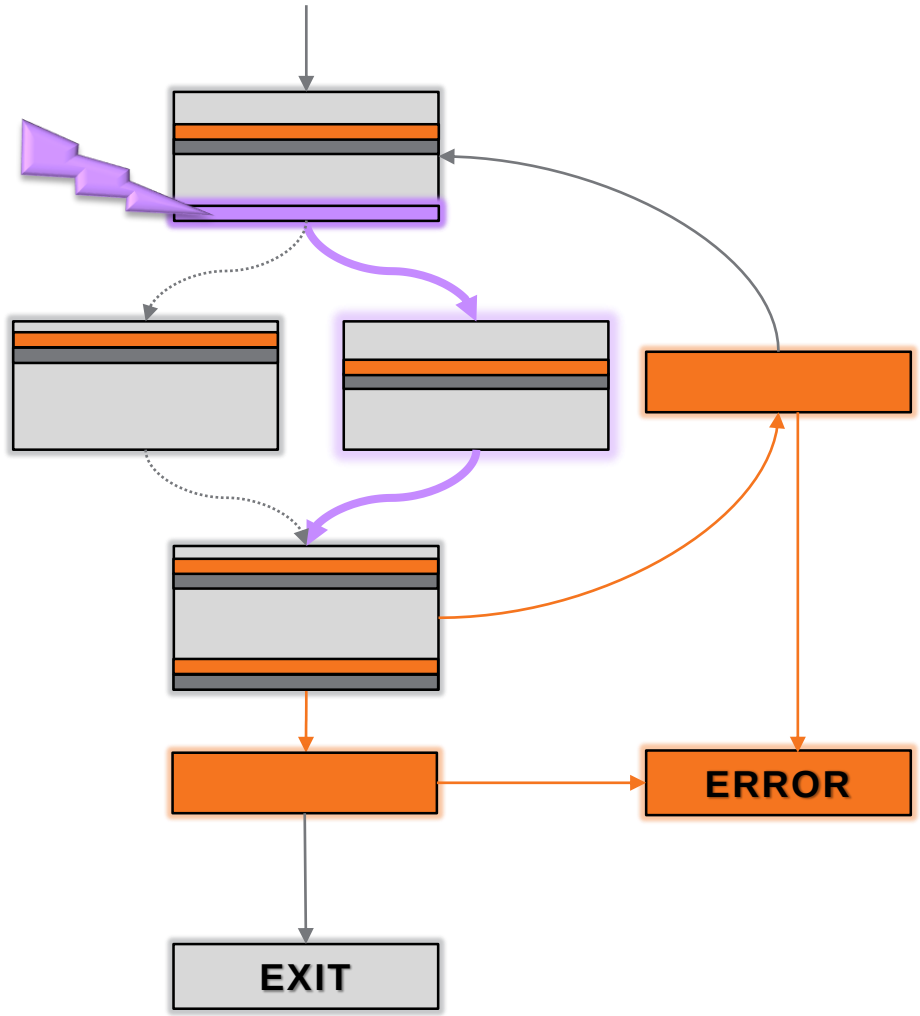
- Partir d'une condition de sortie
- Analyse arrière pour determiner les instructions impliquées
- Dupliquer les instructions sélectionnées
- Création de bloc:
  - Vérifier la condition
  - Gestion d'erreur



- Pour chaque boucle et sortie

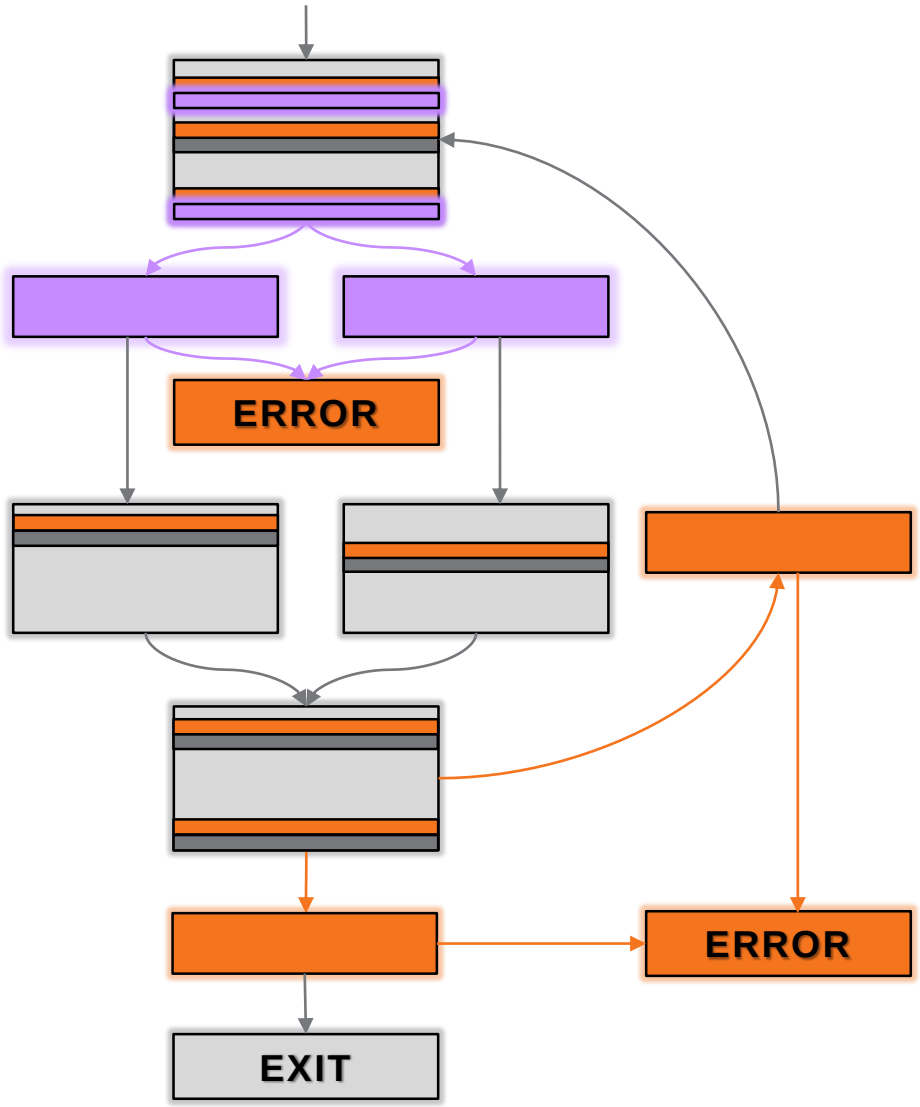
Protection des boucles: principe

- Une faute visant un **branchement interne** peut compromettre les deux flots de données



Protection des boucles: algorithme principal

- Pour chaque boucle et sortie,
  - Partir d'une condition de sortie
  - Analyse arrière pour determiner les instructions impliquées dans son calcul ou celui d'un branchement interne
  - Dupliquer les instructions sélectionnées
  - Création de bloc:
    - Vérifier la condition
    - Vérifier la condition de branchement interne
    - Gestion d'erreur



Intégration dans le flot de compilation

- Réduction des coûts => Viser le middle-end
- Algorithme conçu pour une representation intermédiaire (IR) en forme SSA

Intégration dans le flot de compilation

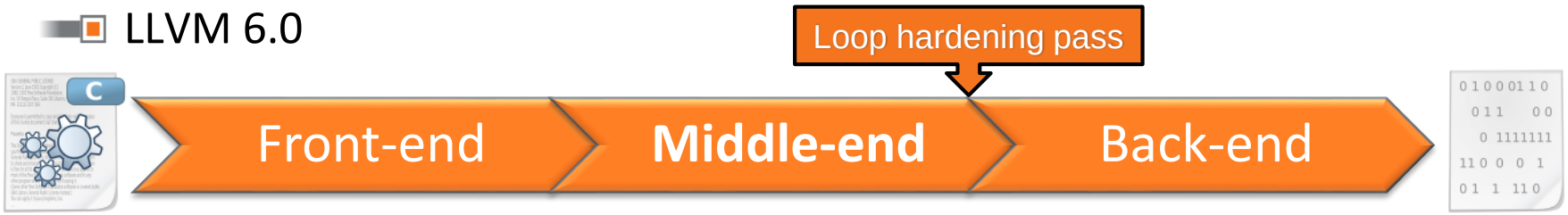
- LLVM 6.0 infrastructure de compilation

  - DCE, CSE,…
  - LICM, LSR,…

- Réduction des coûts => Viser le middle-end
- Algorithme conçu pour une representation intermédiaire (IR) en forme SSA
- Placement pour éviter les interactions avec les optimisations

Intégration dans le flot de compilation

- LLVM 6.0

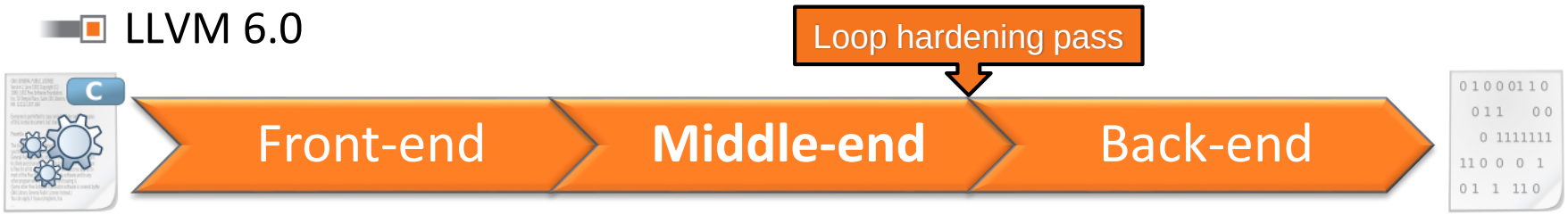


  - DCE, CSE,…
  - LICM, LSR,…

- Réduction des coûts => Viser le middle-end
- Algorithme conçu pour une representation intermédiaire (IR) en forme SSA
- Placement pour éviter les interactions avec les optimisations

Intéractions avec le backend

- LLVM 6.0



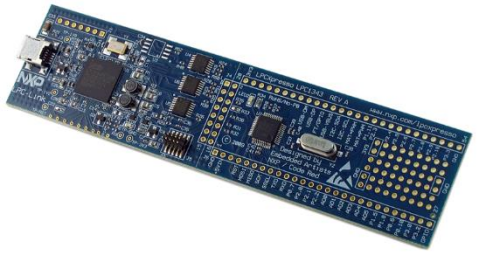
- DCE, CSE,…
- LICM, LSR,…
- Control Flow Optimizer
- ISEL / Constant Promotion
- Register Allocation / Spill

- Interactions avec le back-end?
- Comment gérer lese passes interférentes
  - **Adapter**: Instruction Selection, Register Allocation
  - Ou **Désactiver**: Control Flow Optimizer

- Introduction & contexte
  - Attaques physiques
  - Contre-mesures
- Schéma de protection des boucles
  - Principe
  - Implémentation dans LLVM
- **Résultats expérimentaux**
- Conclusion

Environnement expérimental

- Simulation
  - NXP LPCXpresso 1343 à base de ARM **Cortex M3** (ARMv7m/Thumb2)



  - Raspberry Pi3 à base de ARM **Cortex A53** (ARMv7a)

- Attaques physiques (Injection EM)
  - IMX6 RiotBoard à base de ARM **Cortex A9** (ARMv7a)

Environnement expérimental

- 18 Benchmarks couvrant differents types de codes
  - SPEC CPU2006 INT Benchmark (*reference générale*)
  - MiBench pgp & blowfish (*reference sécurité*)
  - 3 codes crypto maison (*aes, ecc, sha*)
  - gzip, gmp lib, openssl, sqlite (*boucles complexes*)

- Attaques physiques sur boucles spécifiques
  - Boucles type *memcpy*
  - Sortie du compilateur réécrite pour inclure l'instrumentation

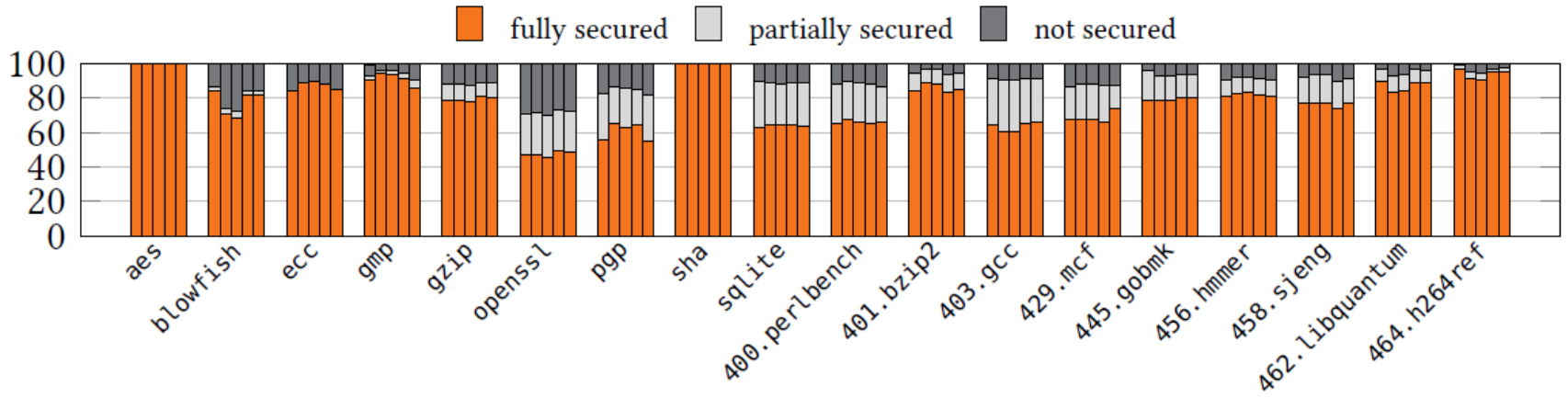
Evaluation expérimentale

- Objectifs
  - Validité fonctionnelle
  - Couverture automatique des boucles
  - Evaluation du surcout
  - Analyse sécuritaire

- Methodologie pour la validité fonctionnelle
  - Benchmarks auto-testant
  - Compilation avec plusieurs niveaux d'optimisation

Couverture automatique des boucles

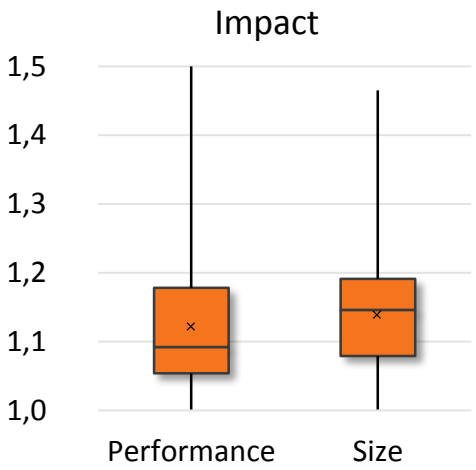
Couverture



- Résultats avec toutes les boucles considérées sensibles
- Schéma non applicable si instruction non dupliquable (ex. Appels de fonction, accès mémoire volatile…), warning émis

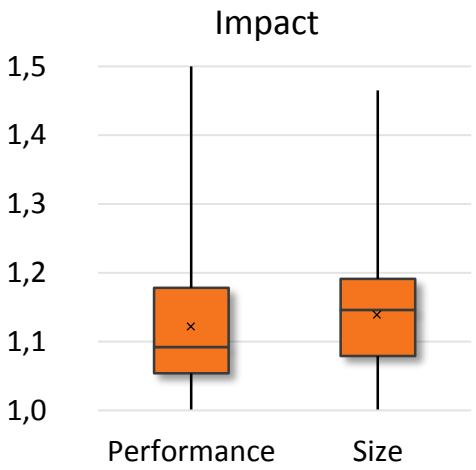
Surcout

- Impact en performances
  - De **0% à 50%** en fonction du temps passé dans les boucles
  - Moyenne **12.5%**, médiane **9,2%**
- Impact en taille de code
  - De **1% à 43%** en function de la complexité des boucles
  - Moyenne **14%**, médiane **14,5%**
  - Ecart type important dû à la diversité des benchmarks



Surcout

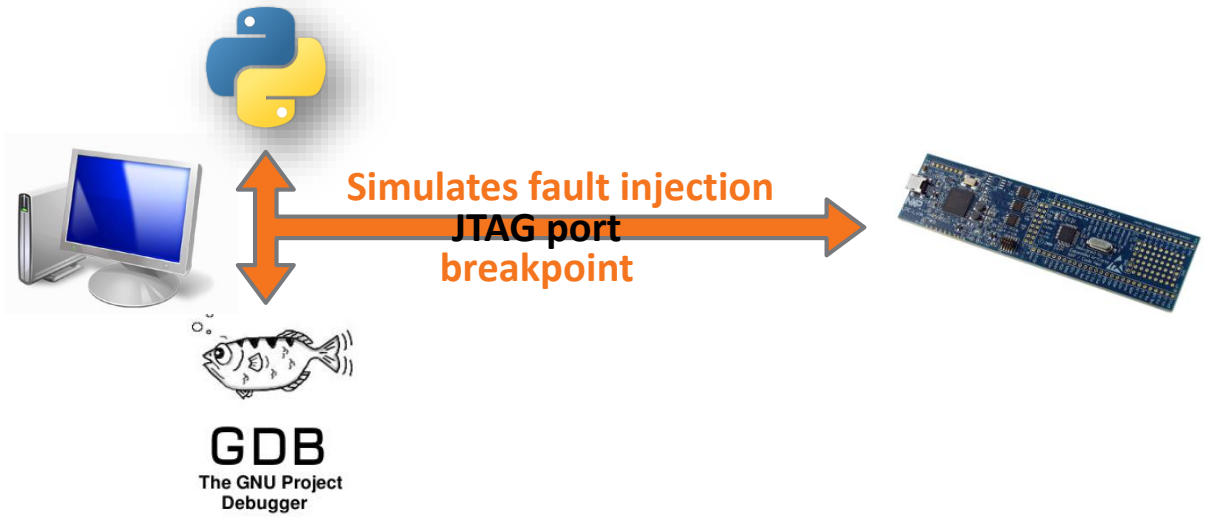
- Impact en performances
  - De **0% à 50%** en fonction du temps passé dans les boucles
  - Moyenne **12.5%**, médiane **9,2%**
- Impact en taille de code
  - De **1% à 43%** en function de la complexité des boucles
  - Moyenne **14%**, médiane **14,5%**
  - Ecart type important dû à la diversité des benchmarks



- En visant la protection du nombre d'iteration:
  - Coût limité en ciblant que les parties sensibles
  - Comparé à un schema de duplication générique (≥ x2)

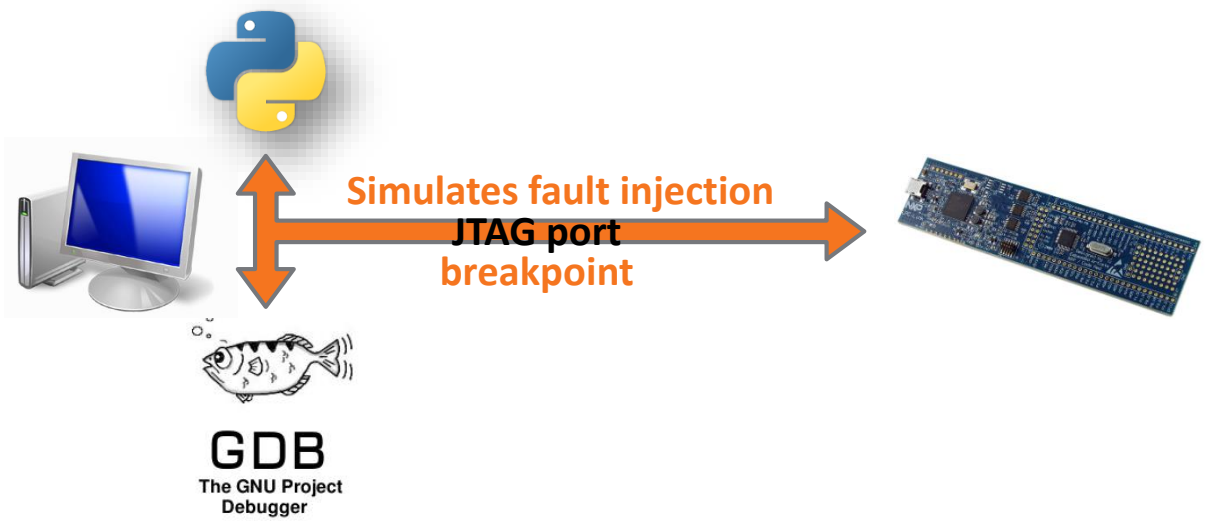
Analyse sécuritaire

- Simulation en scripts **python** interface avec **gdb**



Analyse sécuritaire

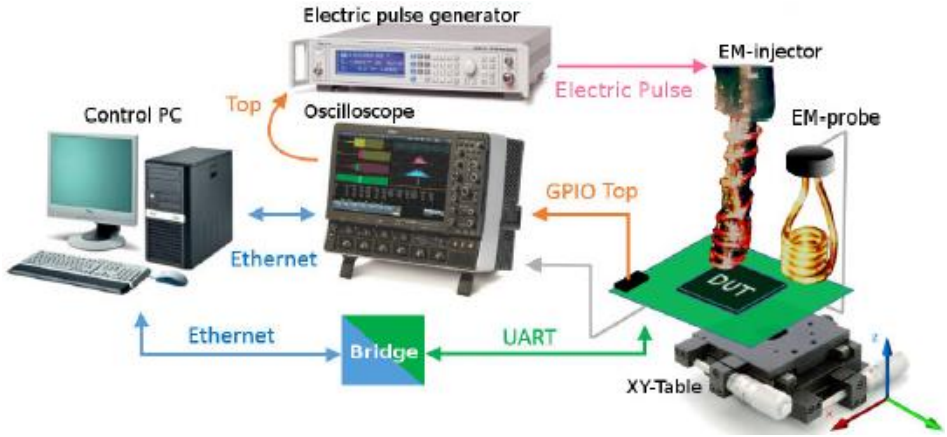
- Simulation en scripts **python** interface avec **gdb**



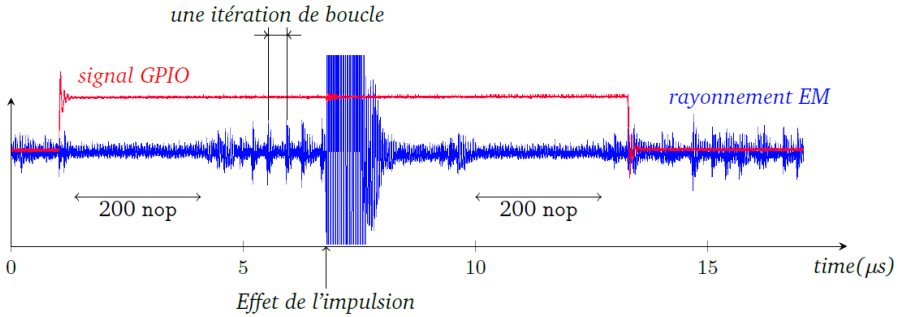
- Robustesse du schéma
  - Simulation de *sauts d'instruction* et *corruption de registre*
  - Code original: ≈600 000 injections, **≈30 000** fautes
  - Code sécurisé: ≈900 000 injections, ≈85 500 fautes, ≈85 000 détectées, i.e. **≈99.4%**
  - Jusqu'à **100%** avec des modifications du back-end

Campagnes d'attaques physique

- Plateforme d'injection EM
- 6ns, impulsion 310V,…
  (configuration existante)



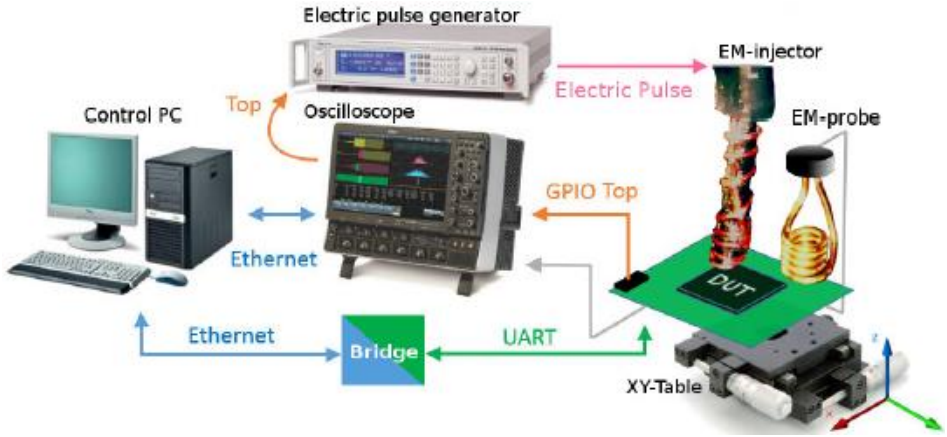
- Analyse de boucles simples
  - Code instrumenté
  - Propriété de sécurité
  - Initialisation des registres
  - Comparaison fauté/attendu



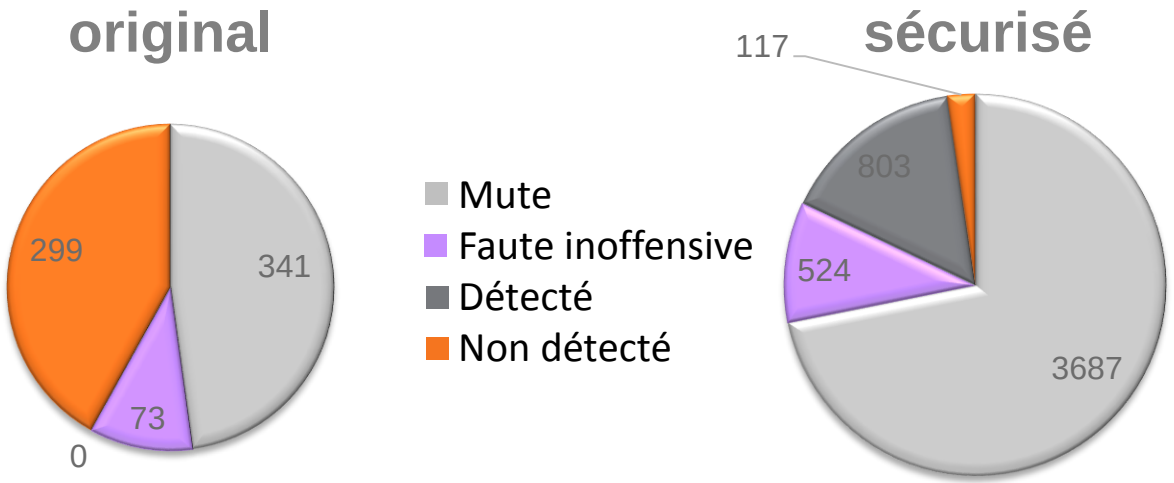
  - Caractérisation du modèle de faute par des codes simples
  - Détails: https://arxiv.org/pdf/1903.02623.pdf

Campagnes d'attaques physique

- Plateforme d'injection EM
  - 6ns, impulsion 310V,… (configuration existante)



- Résultats
  - > 95% d'injections sans faute
  - Proportion des effets



- Fautes non détectées dues à des effets complexes/mixtes

- Introduction & contexte
  - Attaques physiques
  - Contre-mesures
- Schéma de protection des boucles
  - Principe
  - Implémentation dans LLVM
- Résultats expérimentaux
- **Conclusion**

Conclusion & Perspectives

- Protection des boucles à la compilation
  - Assurer le nombre d'iteration et la bonne sortie de boucle
  - 99% des attaques simulées détectées

Conclusion & Perspectives

- Protection des boucles à la compilation
  - Assurer le nombre d'iteration et la bonne sortie de boucle
  - 99% des attaques simulées détectées
  - Réponse au besoin industriel d'automatisation
  - Beneficie des optimisations du compilateur…

Conclusion & Perspectives

- Protection des boucles à la compilation
  - Assurer le nombre d'iteration et la bonne sortie de boucle
  - 99% des attaques simulées détectées
  - Réponse au besoin industriel d'automatisation
  - Beneficie des optimisations du compilateur…
- …mais besoin de patcher le compilateur
  - Pour garantir la propriété de sécurité jusqu'au bout du flot de compilation
  - Met en avant l'incompatibiilité entre optimisations et sécurité
- Importance du modèle de faute

Conclusion & Perspectives

- Protection des boucles à la compilation
  - Assurer le nombre d'iteration et la bonne sortie de boucle
  - 99% des attaques simulées détectées
  - Réponse au besoin industriel d'automatisation
  - Beneficie des optimisations du compilateur…
- …mais besoin de patcher le compilateur
  - Pour garantir la propriété de sécurité jusqu'au bout du flot de compilation
  - Met en avant l'incompatibiilité entre optimisations et sécurité
- Importance du modèle de faute
- Questions ouvertes
  - Combinaison de protections? Comment automatiser plusieurs schémas simultanément?

Merci !

Questions ?

Soutenance de thèse le **lundi 17 juin 2019 à 14h30**

Amphithéâtre HS02, **Centre Microélectronique de Provence** Georges Charpak

880 Route de Mimet, 13120 **Gardanne**