MODÉLISATION DE FAUTES UTILISANT LA DESCRIPTION RTL DE MICROARCHITECTURES POUR L'ANALYSE DE VULNÉRABILITÉ CONJOINTE MATÉRIELLE-LOGICIELLE

SOUTENANCE DE THÈSE

Johan LAURENT

Dirigée par Vincent BEROULLE
Co-encadrée par Christophe DELEUZE et Florian PEBAY-PEYROULA

This work was funded thanks to the French national program 'programme d'Investissements d'Avenir, IRT Nanoelec' ANR-10-AIRT-05

19 NOVEMBRE 2020

I. CONTEXTE ET PROBLÉMATIQUE

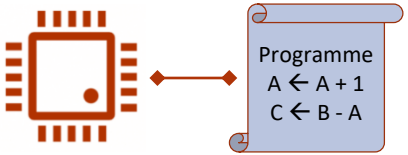
Qu’est-ce que l’injection de fautes ?

Quel est l’état de l’art ?

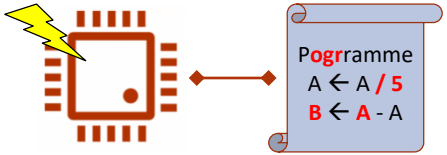
Quel problème cherche-t-on à résoudre ?

INTRODUCTION À L'INJECTION DE FAUTES MATÉRIELLES

- **Principe : Créer une perturber physique dans le processeur pour faire dévier l'exécution du programme.**



Exécution normale du programme par le processeur



Injection de faute dans le processeur ; le comportement du programme s'en trouve modifié

- **Moyens : Perturbation de la tension d'alimentation ou du signal d'horloge, injection électromagnétique, injection laser…**

- **Objectifs de l'attaquant : passer outre des mesures de sécurité (authentification…) ou corrompre le résultat d'un calcul (pour retrouver une clé de chiffrement par exemple)**

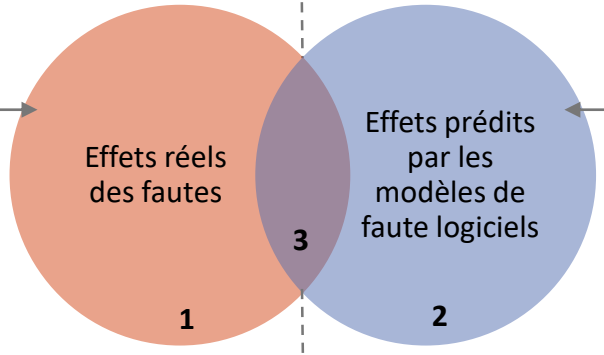
ANALYSE DE SÉCURITÉ ET CONTREMESURES

- **Analyse de vulnérabilité**

Niveau matériel :
précis mais peu rapide

Niveau logiciel :
peu précis mais **rapide**

Injection physique de fautes



Injection basée sur des modèles de faute logiciels

- Corruption de registre
- Corruption de mémoire
- Saut d'instruction
- …

- Conséquence : les contremesures peuvent être sur-dimensionnées, ou, plus grave, sous-dimensionnées.
- Besoin de construire des modèles de faute logiciels plus proches de la réalité matérielle pour aboutir à une analyse à la fois **précise** et **rapide**.

POSITIONNEMENT DE LA THÈSE

- **<u>Vision proche de l'attaquant :</u> inférer des modèles de faute en réalisant des injections physiques**
  - Comportements fautifs réels
  - Mais propagation des fautes dans « une boite noire »

- **<u>Vision proche du concepteur :</u> la description RTL du processeur est disponible pour analyser la propagation de fautes**
  - Plus tôt dans le flot de développement
  - Besoin d'un modèle de faute RTL : fautes non permanentes, bit-flip sur les bascules, simples ou multiples
  - Meilleure contrôlabilité et observabilité des fautes et de leur propagation

OBJECTIFS DE LA THÈSE

- **Proposer une approche multi-niveaux (HW/SW) qui permette d'automatiser une modélisation précise des fautes au niveau logiciel et l'évaluation de contremesures**

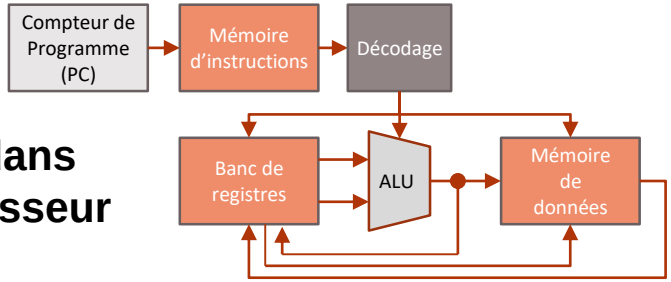
- **Utiliser les nouveaux modèles de faute logiciels pour mener des analyses de vulnérabilité de programmes avec des outils d'analyse avancés (analyse statique)**

II. MICROARCHITECTURE ET INJECTION DE FAUTES

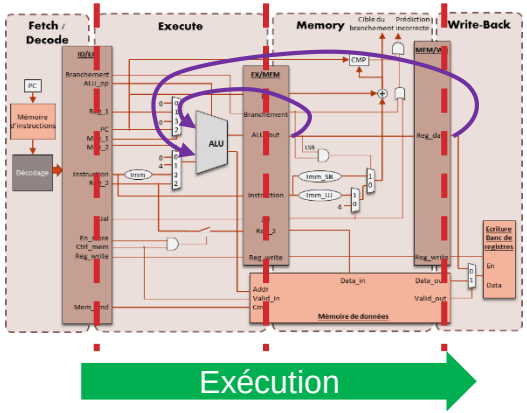
Comment l'architecture d'un processeur complexifie-t-elle la compréhension des fautes au niveau logiciel ?

Quels effets peut-on obtenir en exploitant des structures spécifiques du processeur ?

ORIGINE DES IMPRÉCISIONS



- **Pas de prise en compte de la microarchitecture dans les modèles typiques : vision théorique du processeur**
- **Complexité et diversité des microarchitectures**
- **De nombreuses optimisations matérielles transparentes du côté logiciel**
  - Pipeline, Forwarding, Exécution spéculative, Exécution dans le désordre…



ADD **a2** = a0 + a1
ADD a3 = **a2** + a0

PREMIÈRE ÉTUDE EN SIMULATION

- **Etude du processeur et simulation de diverses injections**
  - Processeur LowRISC (architecture RISC-V)
  - Segments de code assembleur conçus pour faire apparaître des nouveaux comportements

- **On observe une grande diversité des comportements fautifs :**
  - Signaux de contrôle des instructions : opérations mathématiques qui font des sauts, sauts qui modifient un registre…
  - Forçage/annulation du forwarding
  - Exploitation des états fantômes issus d'une mauvaise spéculation
  - …

- **Ces comportements remettent en cause la sécurité de diverses contremesures logicielles typiques et d'une application réelle (VerifyPIN)**

EXEMPLES DE COMPORTEMENTS COMPLEXES

➢ Saut d'instruction différé

```
ADD a0 = a1 + a2
ADD t0 = a1 + a2
IF a0≠t0, goto error
  // Ici, a0==t0
```

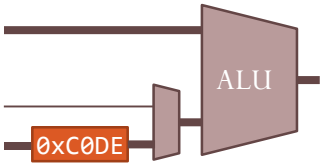


```
ADD a0 = a1 + a2
ADD t0 = a1 + a2
IF a0≠t0, goto error
  // Ici, a0 est corrompu
```

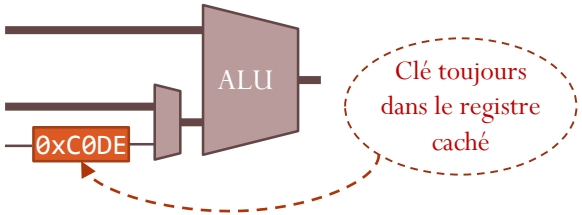
- Forwarding de la bonne valeur dans l'instruction IF puis empêchement de l'écriture du résultat de la première instruction
- Protection : ajouter des instructions NOP avant la comparaison pour lutter contre l'optimisation de forwarding

➢ Registres cachés : structures de la microarchitecture, invisibles côté logiciel, qui stockent temporairement des données pour l'exécution des instructions

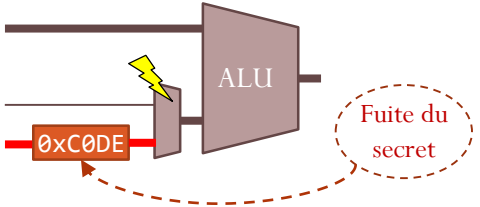
Utilisation d'une clé secrète



Instructions suivantes



Instructions suivantes - Attaque



CONCLUSIONS DE L'ÉTUDE PRÉLIMINAIRE

- **Une grande variété des comportements fautifs du processeur à prendre en compte dans les analyses au niveau logiciel**

- **Besoin d'aider le concepteur à automatiser l'analyse multi-niveaux afin d'identifier les modèles de faute logiciels les plus pertinents et développer des contremesures matérielles et/ou logicielles les plus efficaces et au plus juste coût**

- **Première étape : comment représenter les fautes au niveau logiciel ?**

III. REPRÉSENTATION DE FAUTES AU NIVEAU LOGICIEL

Comment prendre en compte les effets des fautes matérielles au niveau logiciel ?

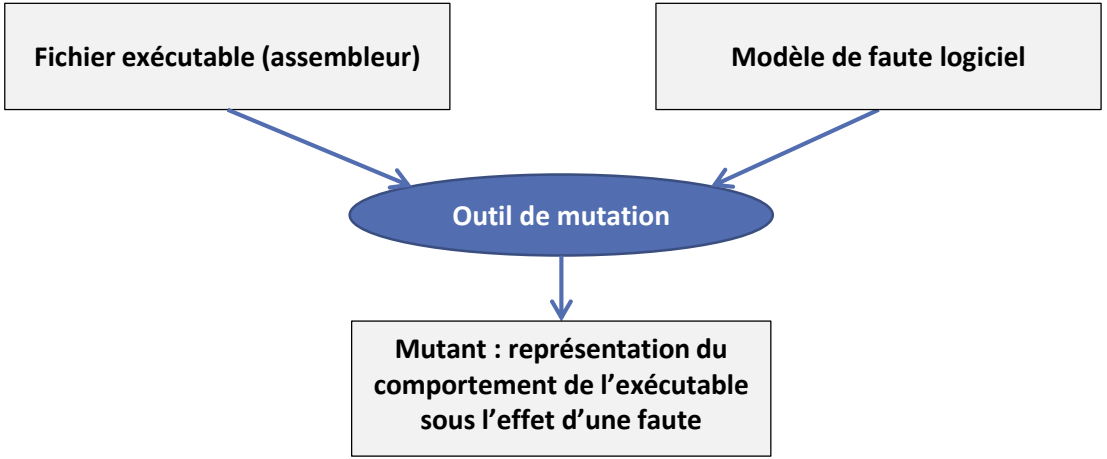
OUTILS EXISTANTS

Les méthodes existantes pour l'analyse de fautes au niveau logiciel ciblent un ou plusieurs modèles de faute logiciels particuliers.

Problème : nos modèles de fautes sont très divers.

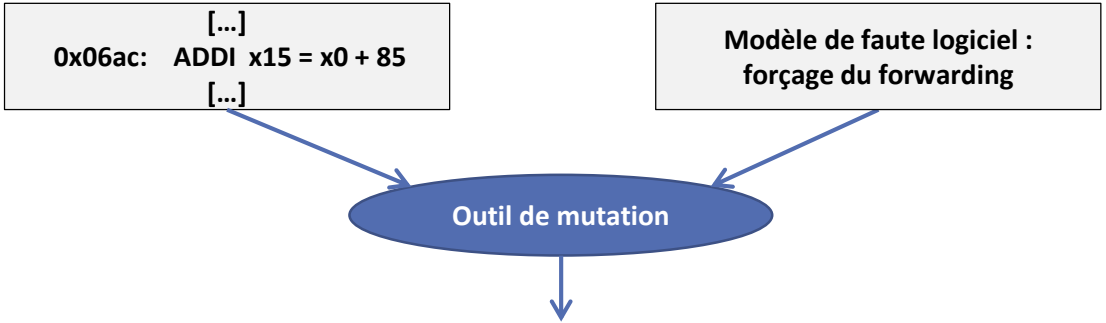
→ Développement d'un outil de mutation de programme pour prendre en compte cette diversité

SPÉCIFICATIONS DE L'OUTIL DE MUTATION

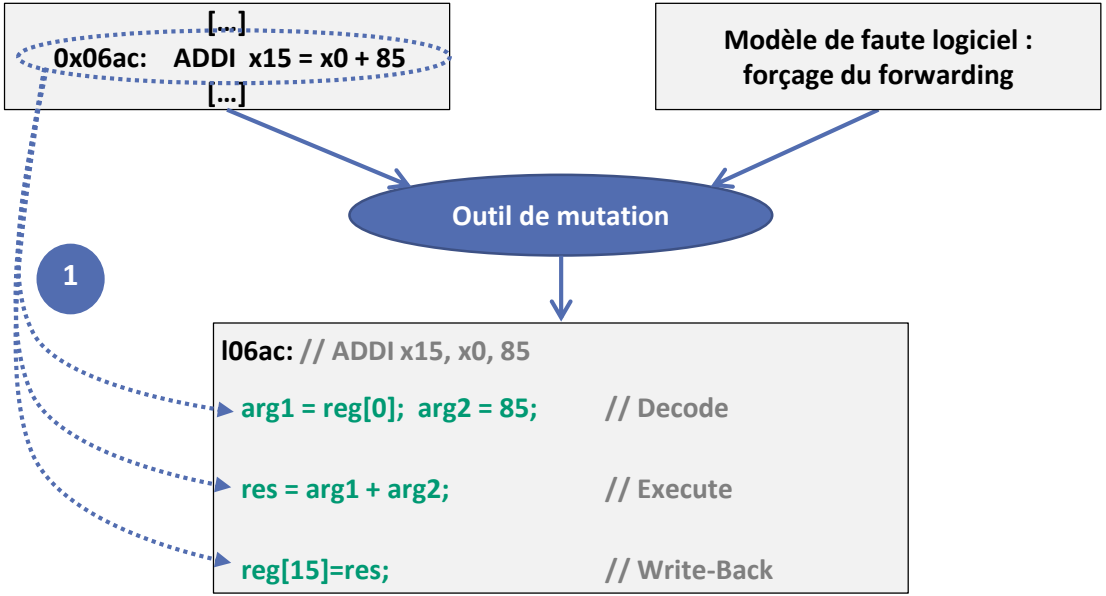


- **Flexibilité :**
  - Les comportements fautifs à représenter sont très variés
  - Les mutants doivent pouvoir être analysés par divers outils d'analyse

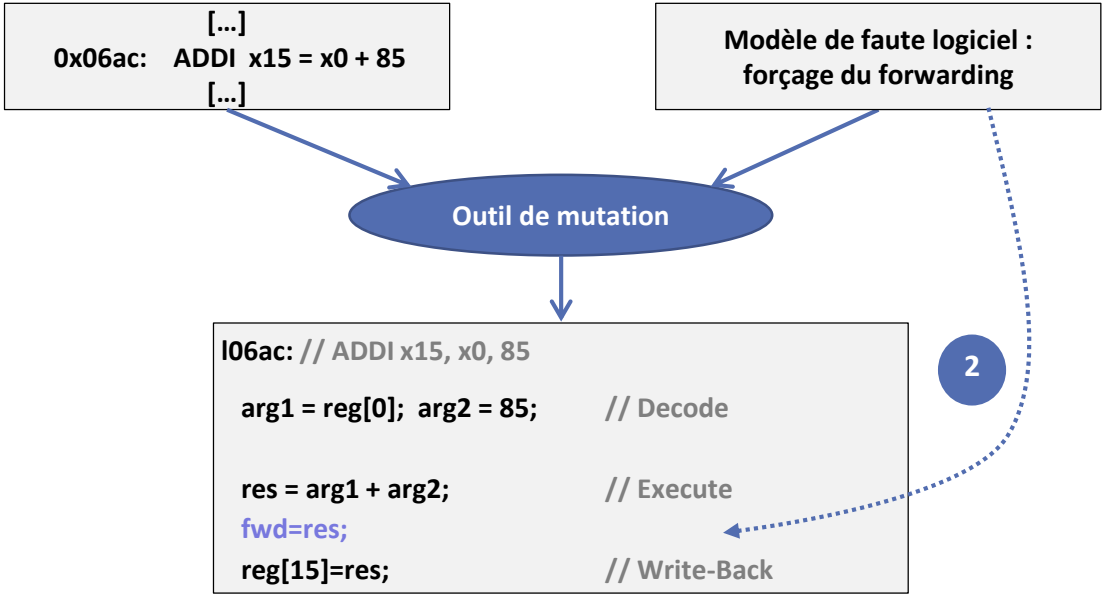
EXEMPLE DE MUTATION



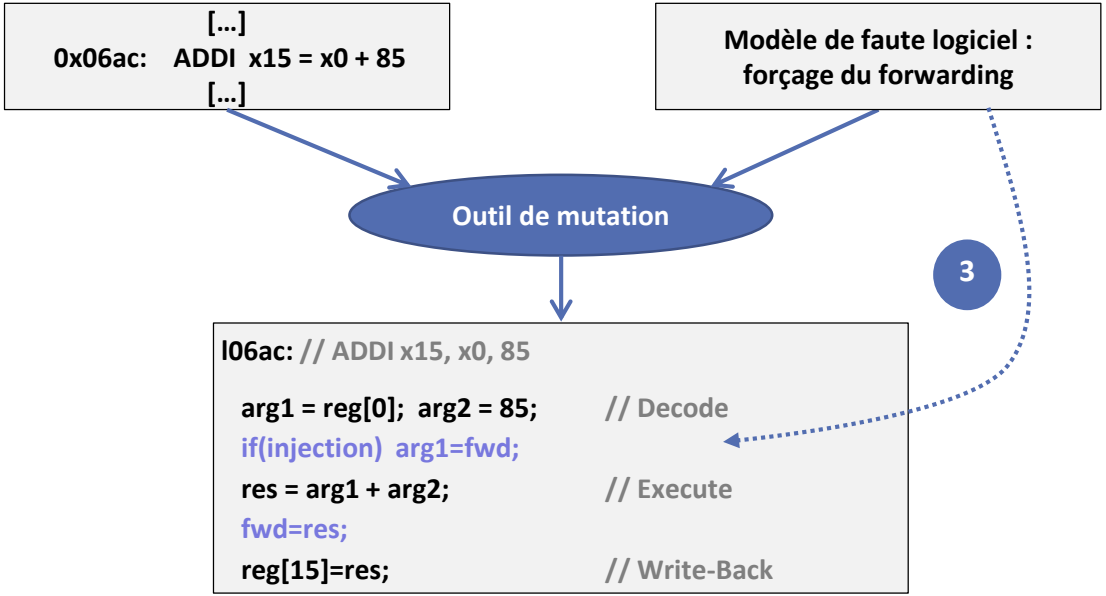
EXEMPLE DE MUTATION



EXEMPLE DE MUTATION



EXEMPLE DE MUTATION

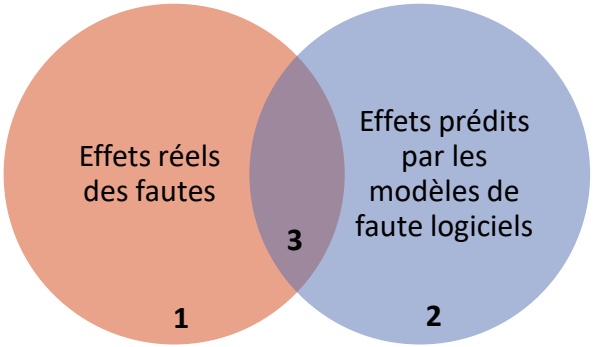


BILAN OUTIL DE MUTATION

- **Moyen pour injecter des fautes précises directement au niveau logiciel**
  - Première étape vers une automatisation du processus de modélisation

- **Comment utiliser cet outil dans un processus de modélisation ?**

  Comment évaluer la pertinence des modèles de faute logiciels ?



IV. APPROCHE DE MODÉLISATION MATÉRIEL-LOGICIEL

Comment systématiser le processus de modélisation ?

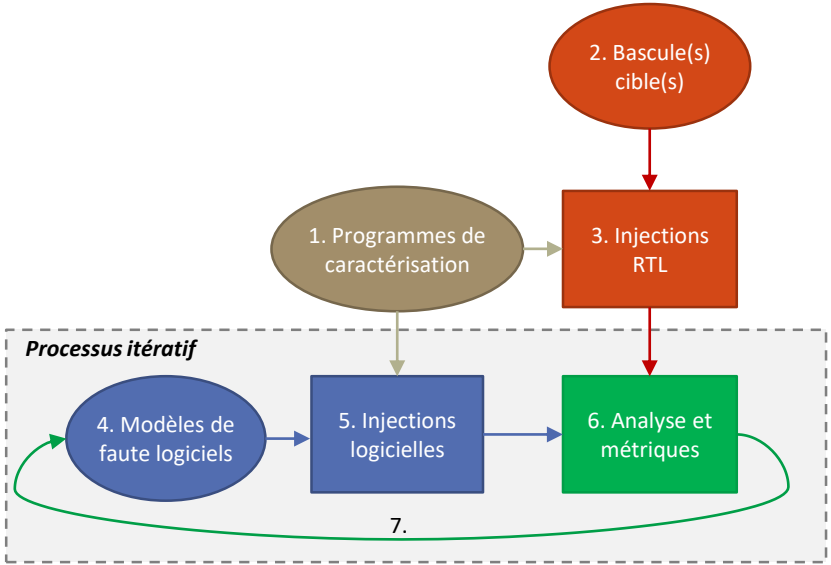
Comment évaluer la pertinence de modèles de faute logiciels ?

PRINCIPE DE L'APPROCHE

- **Objectif : donner des outils pour aider le concepteur dans le processus de modélisation et de sécurisation du système**

- **Approche basée sur la comparaison précise de résultats d'injection RTL (référence) et d'injection logicielle**
  - Injection au même point de l'exécution des programmes
  - Observations aux mêmes points de l'exécution des programmes
  - Observations du banc de registres et de la mémoire

VUE D'ENSEMBLE DE L'APPROCHE



PROGRAMMES DE CARACTÉRISATION

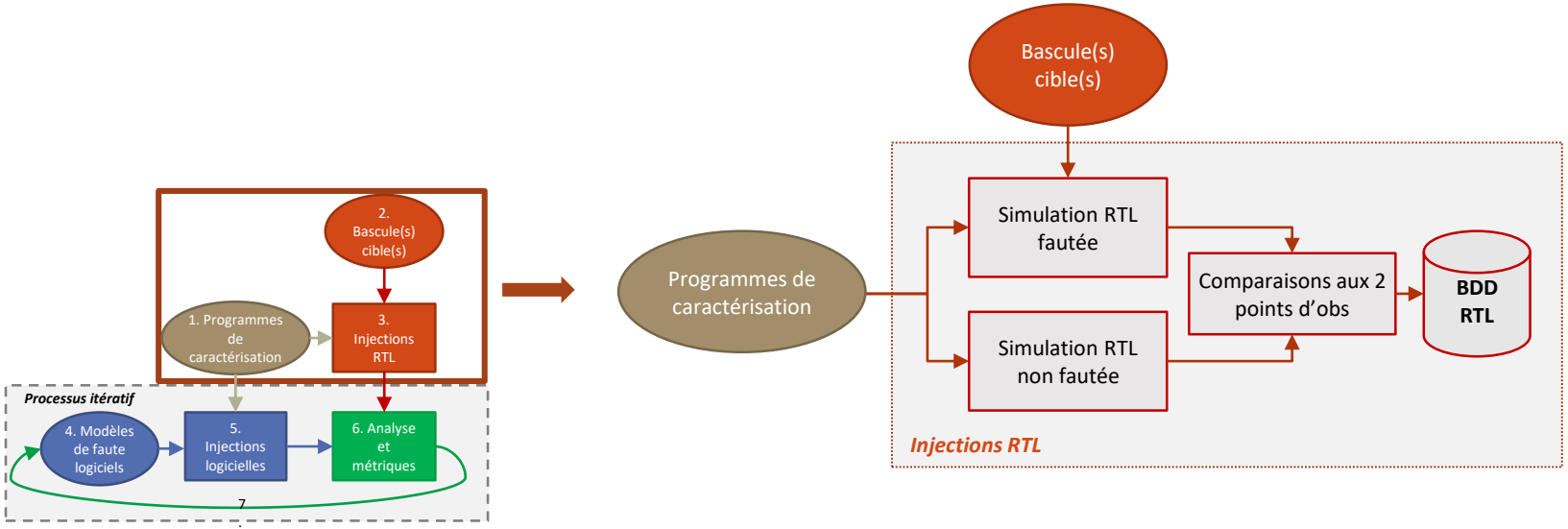
- **Séquences d'instructions assembleur avec la structure suivante :**

```
Prologue_instruction_1
Prologue_instruction_2
Instruction_cible                    // Injection + Observation effets immédiats
Epilogue_instruction_1
Epilogue_instruction_2
Epilogue_instruction_3               // Observation effets de propagation
```

- **Deux points d'observation : effets instantanés et effets de propagation**

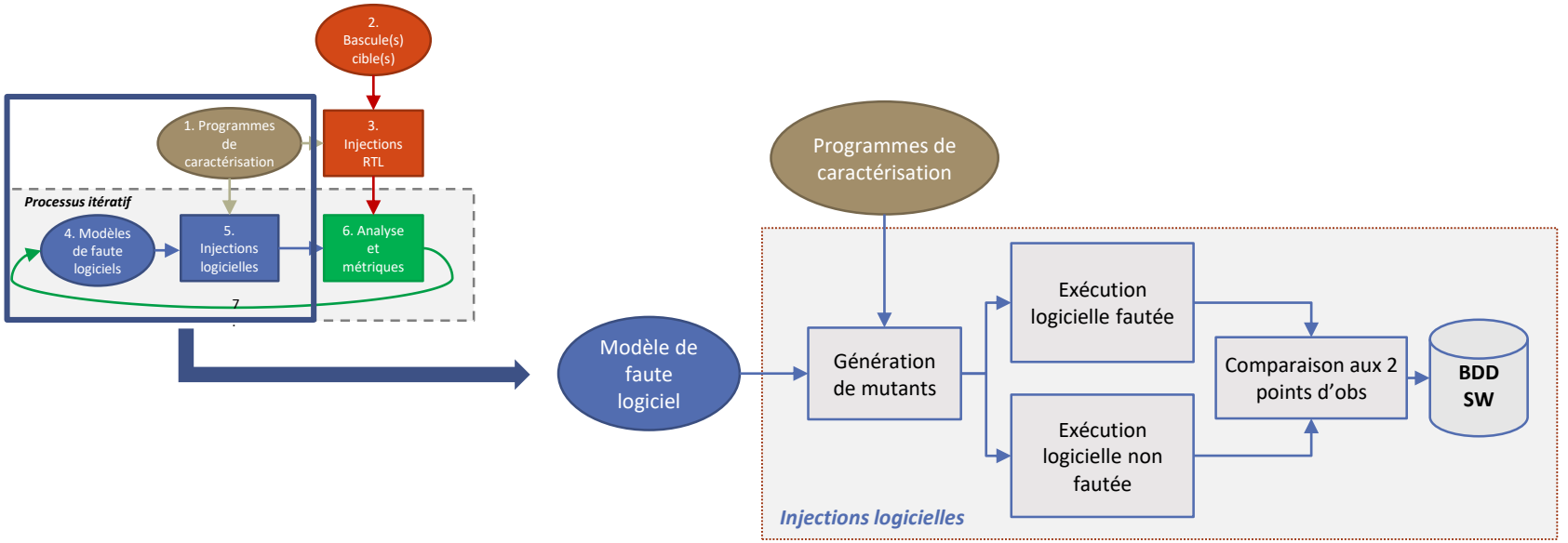
INJECTION DE FAUTE RTL

- **Injection de bit-flips grâce à des commandes du simulateur (QuestaSim)**



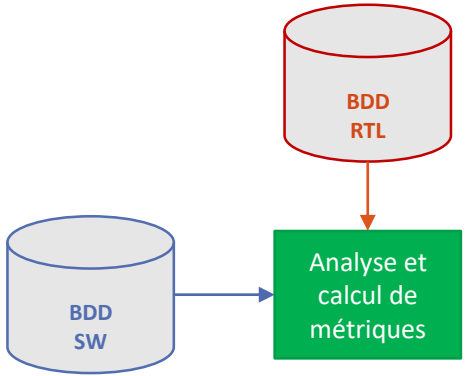
INJECTION DE FAUTE LOGICIELLE

➢ Injection avec notre outil de mutation

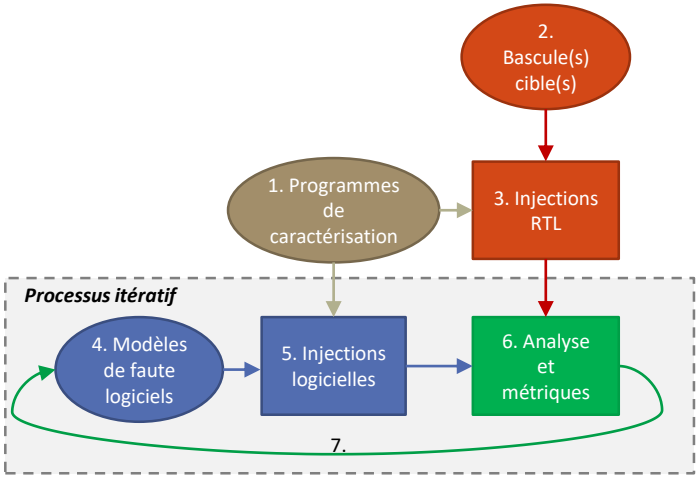


ANALYSES MATÉRIEL-LOGICIEL

- **Quels types d'analyse sont possibles en comparant les résultats d'injection contenus dans les deux bases de données ?**
  - Métrique de couverture
  - Sélection des structures RTL les mieux modélisées
  - Métrique de justesse
  - Profil de modèles



CAS D'ÉTUDE



- **Programmes de caractérisation :**
  - 105 contextes assembleur
  - VerifyPIN
  - LittleXorKey

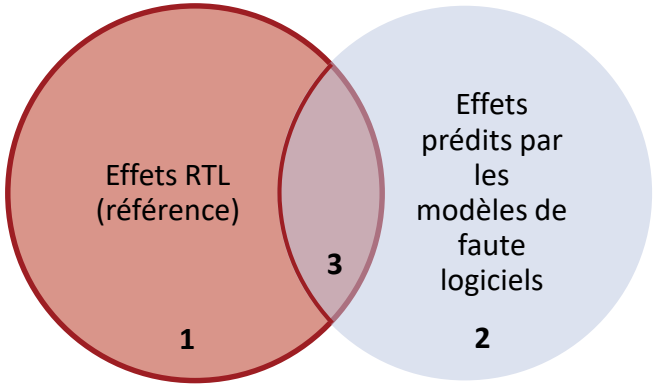
- **Bascules ciblées :**
  - Injection dans les bascules du pipeline, sauf banc de registres
  - En tout, 1308 bascules

- **Spécification de 49 modèles de faute logiciels :**
  - 32 simples bit-flips dans le résultat de l'opération
  - 6 modèles typiques (saut d'instruction, inversion de test…)
  - 11 modèles non typiques (notamment en lien avec la structure de forwarding)

MÉTRIQUE DE COUVERTURE

Quelle proportion de fautes RTL sont correctement prédites par les modèles de faute logiciels ?

$$Couverture = \frac{|aire\ 3|}{|aire\ 1| + |aire\ 3|}$$



MÉTRIQUE DE COUVERTURE - RÉSULTATS

- **Campagne d'injections RTL simple-bit exhaustive**

	Couverture
Contextes assembleur	24.0%
VerifyPIN	28.7%
LittleXorKey	29.0%

- **Campagnes d'injections RTL multi-bit statistiques sur contextes assembleur**

	Couverture
1-bit	24.04%
2-bit	~ 22.51%
3-bit	~ 20.21%
4-bit	~ 18.41%
5-bit	~ 17.68%

MÉTRIQUE DE COUVERTURE - DISCUSSION

- **Les taux de couverture paraissent assez faibles : résultats à un instant T**

- **Il est difficile d'obtenir une très bonne couverture**
  - montre la difficulté à modéliser tous les comportements fautifs au niveau logiciel

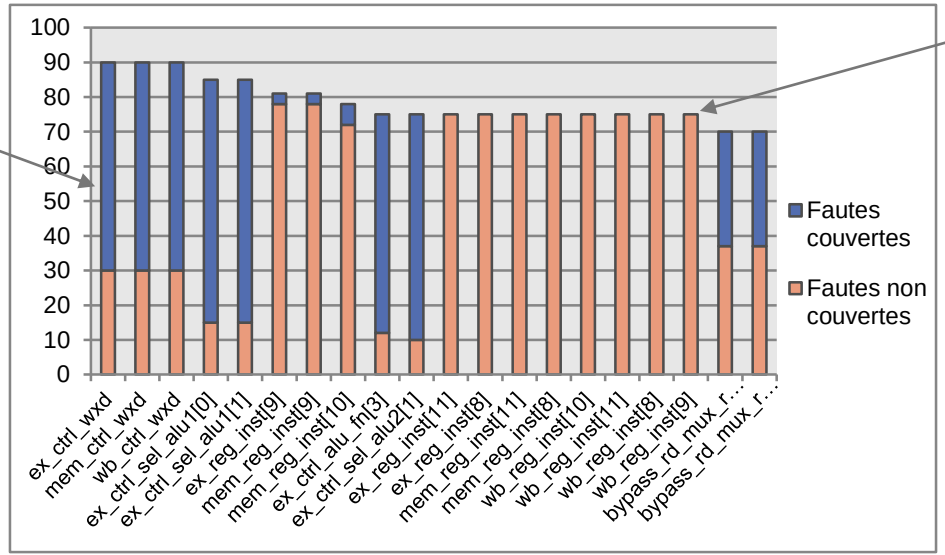
- **Notre approche donne des outils pour aider le concepteur à améliorer la pertinence des modèles**

SÉLECTION DES STRUCTURES RTL LES MIEUX MODÉLISÉES

Représentation des bascules créant le plus de comportements fautifs dans les contextes assembleur.

Bien modélisé au niveau logiciel

Mal modélisé au niveau logiciel

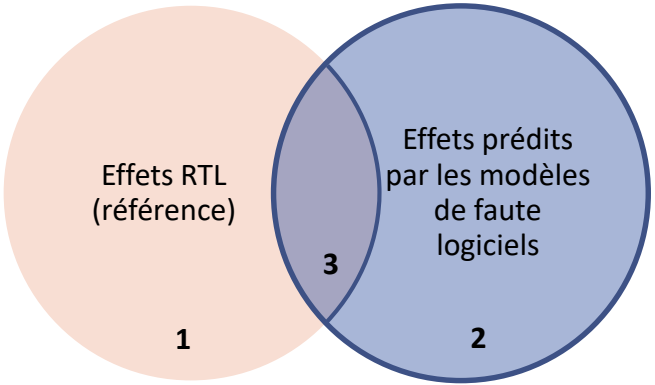


Aide à l'amélioration des modèles et l'identification des contremesures matérielles les plus efficaces

MÉTRIQUE DE JUSTESSE

Quelle proportion de fautes logicielles prédisent des effets réellement obtenus au niveau RTL ?

$$Justesse = \frac{|aire\ 3|}{|aire\ 2| + |aire\ 3|}$$



MÉTRIQUE DE JUSTESSE - RÉSULTATS

- **Globalement, la justesse de nos modèles est de 76.5%**

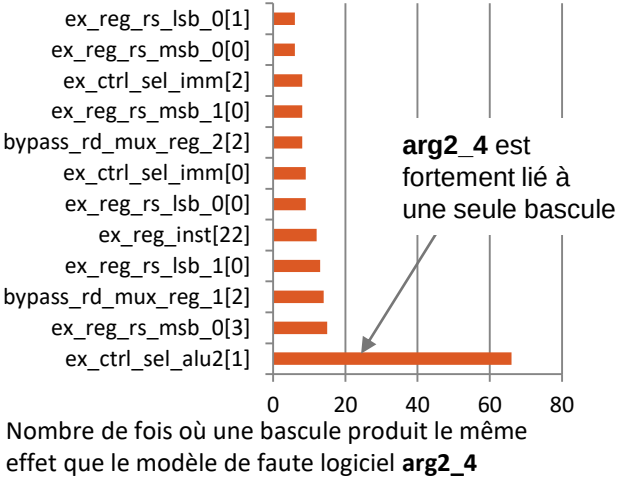
- **Des disparités entre modèles :**
  - Très bons modèles : ~100% (ex : remplacer un argument par 0)
  - Bons modèles : ~80% (ex : saut d'instruction)
  - Mauvais modèles : 60,7% et 41,4%

- **Notre approche donne des outils pour aider le concepteur à améliorer la pertinence des modèles**

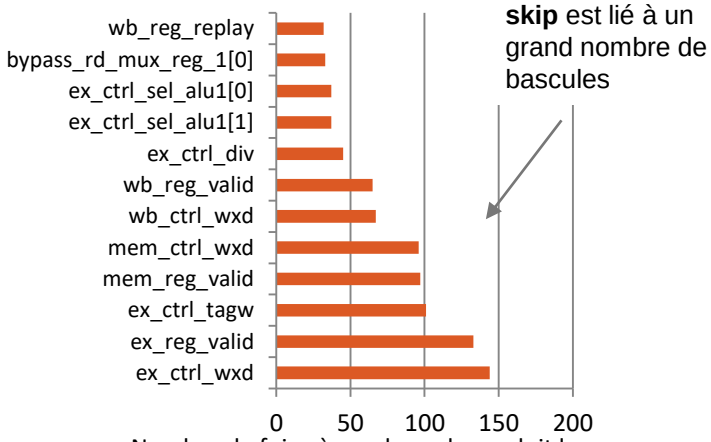
PROFIL DE MODÈLES

Comment reproduire en simulation RTL les effets obtenus par un modèle de faute logiciel ?

Les profils de modèles montrent les bascules qui permettent le plus souvent de reproduire les effets d'un modèle de faute logiciel.



Nombre de fois où une bascule produit le même effet que le modèle de faute logiciel **arg2_4**



Nombre de fois où une bascule produit le même effet que le modèle **skip**

Aide à l'amélioration des modèles et l'identification des contremesures logicielles les plus efficaces

BILAN DE L'APPROCHE

- **L'approche permet :**
  - d'évaluer la pertinence de modèles de faute logiciels
  - de guider le concepteur pour améliorer ces modèles
  - d'identifier les contremesures les plus pertinentes

- **Permet d'améliorer la sécurité en considérant conjointement matériel et logiciel**

- **Comment utiliser ces nouveaux modèles ?**

V. ANALYSE STATIQUE DE VULNÉRABILITÉ

Comment analyser la vulnérabilité de programmes face aux modèles de faute logiciels ?

Quelle flexibilité d'analyse pour la diversité de modèles de faute logiciels ?

ANALYSE DE VULNÉRABILITÉ

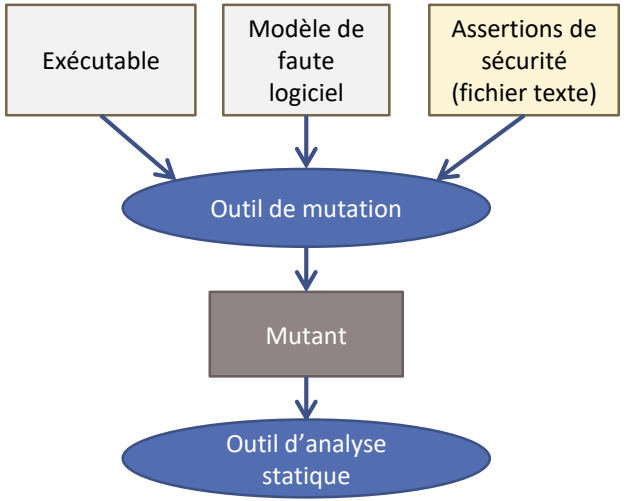
- **Analyse de vulnérabilité au niveau logiciel :**
  - Exécution du mutant : information seulement sur une exécution
  - Méthodes d'analyse statique : information plus globale sur le programme

- **Nos mutants peuvent-ils être utilisés dans des outils existants d'analyse statique ? La structure des mutants est-elle adaptée à ces méthodes ?**

- **La diversité des modèles de faute logiciels appelle une certaine flexibilité dans les méthodes d'analyse**

PROPRIÉTÉS DE SÉCURITÉ

- **On cherche à vérifier la validité de propriétés de sécurité même en présence de fautes.**
- **Exemples de propriété :**
  - Une variable *a* doit avoir une valeur positive
  - Impossible de s'authentifier avec un mauvais code

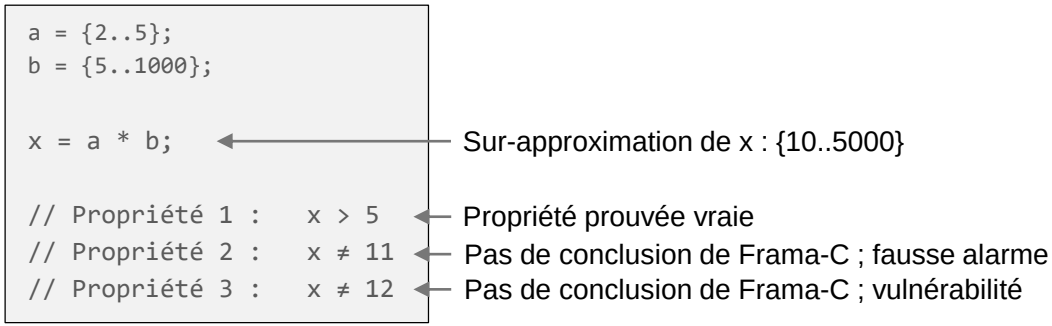


- **Deux méthodes d'analyse statique présentées ici :**
  - Interprétation abstraite
  - Exécution symbolique

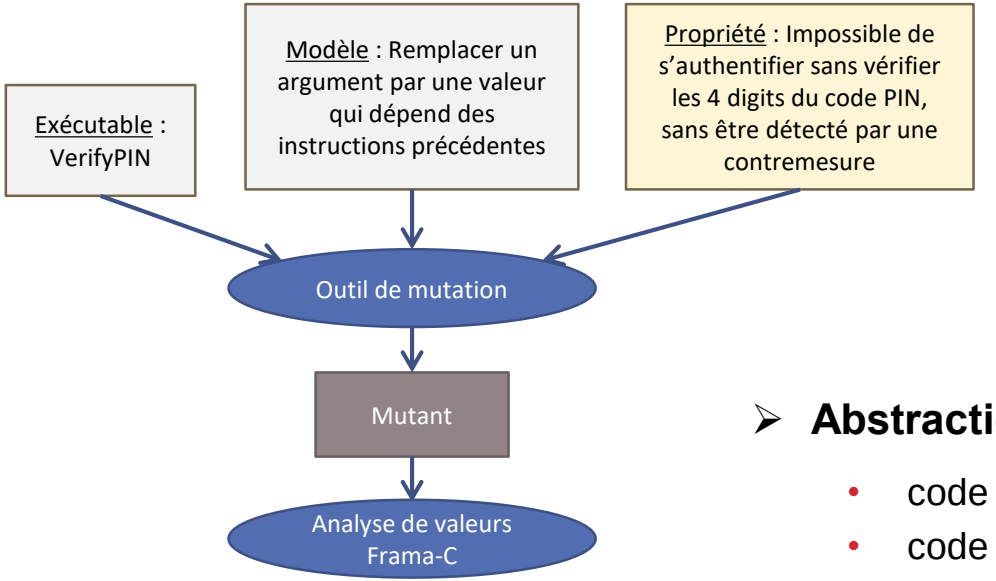
INTERPRÉTATION ABSTRAITE AVEC FRAMA-C

- **Interprétation abstraite : Abstraction des états du programme qui correspond à un sur-ensemble des états concrets de ce programme.**

- **Frama-C est un logiciel libre qui permet l'analyse de valeurs :**
  - Spécification d'intervalles de valeurs plutôt que des valeurs concrètes.



ANALYSE DE VALEURS DE VERIFYPIN



- **Abstraction de l'état du programme :**
  - code PIN secret (4 digits : 10.000 combinaisons)
  - code PIN utilisateur (4 digits : 10.000 combinaisons)
  - instant d'injection (53 instants possibles)

ANALYSE DE VALEURS DE VERIFYPIN - CONCLUSIONS

- **Résultats :**
  - Preuve de sécurité pour 47 instants d'injection
  - Pas de conclusion pour les 6 instants restants → besoin d'analyser « manuellement »

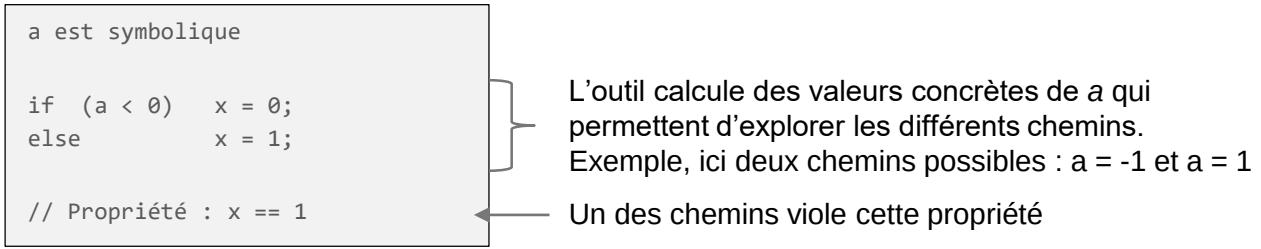
- **Vulnérabilité seulement si le premier ou le second digit secret a une valeur < 4**
  - Intérêt de l'analyse statique : découvrir ces chemins d'attaque complexes

- **Durée d'analyse : très variable**
  - Dépend de beaucoup de paramètres : modèle de faute logiciel, propriété à vérifier, états abstraits, structure du mutant…

EXÉCUTION SYMBOLIQUE

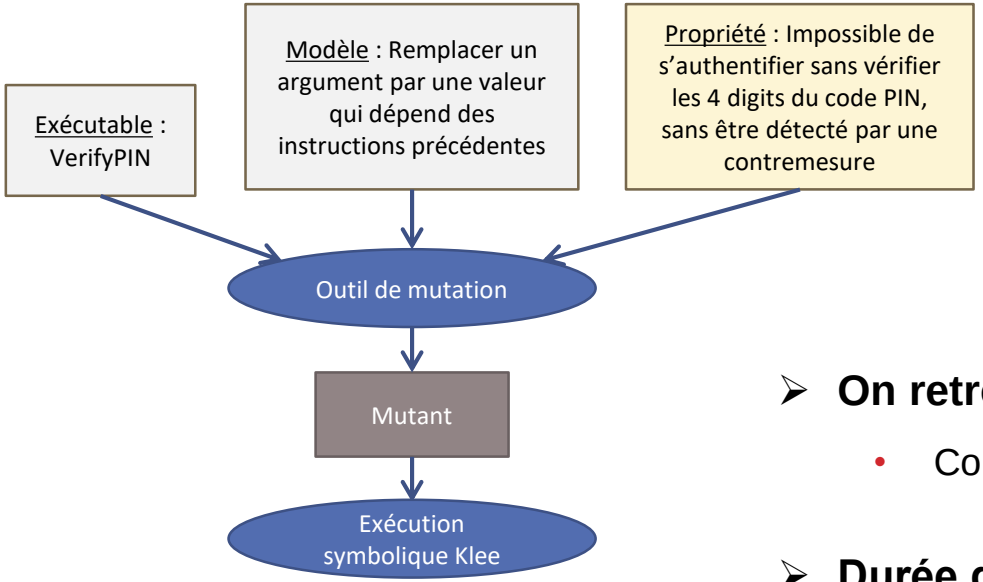
L'exécution symbolique consiste à analyser un programme en substituant aux données d'entrée des symboles représentant n'importe quelle valeur.

Un solveur SMT permet ensuite de donner des valeur concrètes permettant d'explorer les différents chemins d'exécution possibles.



Nous utilisons ici le logiciel Klee pour mener ces analyses.

EXÉCUTION SYMBOLIQUE DE VERIFYPIN



- **Etats symboliques du programme :**
  - code PIN secret
  - code PIN utilisateur
  - instant d'injection

- **On retrouve la même vulnérabilité**
  - Contre-exemple : pas besoin d'analyse manuelle

- **Durée d'analyse très variable en fonction des paramètres d'analyse**

CONCLUSION ANALYSE STATIQUE DE VULNÉRABILITÉS

- **Compatibilité des mutants avec diverses méthodes d'analyse**

- **Eléments de comparaison :**

Exécution symbolique Klee	Analyse de valeurs Frama-C
+ Valeurs concrètes pour les chemins vulnérables	
+ Facilité d'utilisation	
	+ Meilleur passage à l'échelle

- **Possibilité de combiner les analyses**

VI. CONCLUSION GÉNÉRALE

Que retenir de tout ça ?

Et la suite ?

CONCLUSIONS

- **Identification de comportements fautifs complexes dus notamment à diverses optimisations dans les processeurs actuels.**

- **Développement d'un outil de mutation pour représenter ces comportements complexes au niveau logiciel.**

- **Spécification d'une approche pour aider à la modélisation et au choix de contremesures**

- **Utilisation de méthodes d'analyse statique pour détecter des chemins d'attaque complexes.**

PERSPECTIVES

- **Etude d'architectures de processeurs plus complexes (processeurs superscalaires, exécution dans le désordre…).**

- **Utilisation de modèles de faute RTL plus réalistes, voire utilisation de campagnes d'injection dans des conditions réelles (injection EM, etc).**

- **Evolution de l'outil de mutation pour être compatible à d'autres outils d'analyse.**

MERCI POUR VOTRE ATTENTION

SAUTS D'INSTRUCTION AVANCÉS - EXPLICATION

- **Vue du pipeline : au moment où l'instruction sur a0 est « annulée », les instructions a1 et a2 sont déjà passées par l'étage d'exécution.**

```
LI  a0 = 0
ADD a0 = a0 + 1    // 0
ADD a1 = a0 + 1    // 2
ADD a2 = a0 + 1    // 2
ADD a3 = a0 + 1    // 1
ADD a4 = a0 + 1    // 1
```

≡

Cycle	IF	ID	EX	MEM	WB
t0	LI a0	-	-	-	-
t1	ADD a0	LI a0	-	-	-
t2	ADD a1	ADD a0	LI a0	-	-
t3	ADD a2	ADD a1	ADD a0	LI a0	-
t4	ADD a3	ADD a2	ADD a1	ADD a0	LI a0
t5	ADD a4	ADD a3	ADD a2	ADD a1	ADD a0
t6	-	ADD a4	ADD a3	ADD a2	ADD a1

Forward

Forward

L'APPLICATION VERIFYPIN

- **Application issue de la bibliothèque FISSC**
- **Comparaison d'un PIN secret avec le PIN entré par l'utilisateur (4 digits)**
- **Contremesures :**
  - Vérification itérateur de boucle
  - Tests doublés
  - Booléens durcis
    - VRAI = 0xAA = 0b10101010
    - FAUX = 0x55 = 0b01010101

VerifyPIN

```
authentification = FALSE;
if(cnt >= 0) {
   cnt--;
   status = FALSE;  diff = FALSE;

   for(i = 0 ; i < 4 ; i++) {
      if(userPin[i] != cardPin[i])  diff = TRUE;
   }
   if(i != 4) attaque_detectee();

   if (diff == FALSE) {
      if(FALSE == diff)  status = TRUE;
      else attaque_detectee();
   } else status = FALSE;

   if(status == TRUE) {
      if(TRUE == status) {
         cnt = 3;
         authentification = TRUE;
      } else attaque_detectee();
   }
}
```

ATTAQUE DE VERIFYPIN

- **Remarques :**
  - 0x55 + 0x55 = 0xAA (en d'autres termes, FAUX + FAUX = VRAI)
  - Architecture RISC-V : l'assignation d'une valeur booléenne est une addition
    « a0 = FAUX » est équivalent à « ADDI a0 = 0 + 0x55 »

- **Comportement fautif possible : forcer l'utilisation du forwarding pour remplacer un argument par le résultat de l'instruction précédente**

- **En combinant ces remarques, une attaque est possible pour s'authentifier avec le mauvais code PIN :**

```
ADDI a0 = 0 + 0x55   // a0 = FAUX
ADDI a1 = 0 + 0x55   // a1 = FAUX
```

```
ADDI a0 = 0 + 0x55      // a0 = FAUX
ADDI a1 = 0x55 + 0x55   // a1 = VRAI
```

LIMITATIONS ET VALIDATION DE L'OUTIL

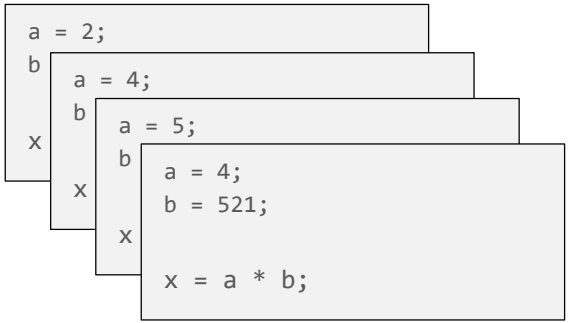
Limitations :

- Actuellement, uniquement instructions RISCV-IM 64 bits, sans les registres CSR.
- Prise en charge limitée des sauts dont l'adresse n'est pas connue à l'avance.
- Difficulté à spécifier des fautes sur le flot de contrôle.

Validation de l'outil :

- Utilisation des vecteurs de test fournis avec la suite d'outils RISC-V (~10-40 tests par instruction)

INTERPRÉTATION ABSTRAITE AVEC FRAMA-C




```
a = {2..5};
b = {5..1000};

x = a * b;

// Propriété 1 :   x > 5
// Propriété 2 :   x ≠ 11
// Propriété 3 :   x ≠ 12
```

Sur-approximation de x : {10..5000}

Propriété prouvée vraie

Pas de conclusion de Frama-C ; fausse alarme

Pas de conclusion de Frama-C ; vulnérabilité

- **Interprétation abstraite : Abstraction des états du programme qui correspond à un sur-ensemble des états concrets de ce programme.**

- **Frama-C est un logiciel libre qui permet l'analyse de valeurs:**
  - Spécification d'intervalles de valeurs plutôt que des valeurs concrètes.