Université Pierre et Marie Curie
Master Informatique Spécialité SESI
2013-2014

Rapport de stage

Vérification formelle de contre-mesures logicielles contre des fautes transitoires

Présenté par
Lucien Goubet

Encadrants
Emmanuelle Encrenaz
Karine Heydemann

Laboratoire d'accueil
LIP6

Équipe
ALSOC

19 juin 2014

Table des matières

1 Contexte/Problématique **4**
1.1 Contexte 4
1.2 Problématique 6

2 Modélisation SAT/SMT **7**
2.1 Transposition du schéma de preuve 7
2.1.1 Contre-mesures existantes 8
2.1.2 Représentation en automates 9
2.1.3 Transposition 10
2.2 SAT 11
2.3 SMT 13

3 Automatisation **13**
3.1 Principe 13
3.2 Architecture logicielle 14
3.3 Outils 15

4 Langage MAE **16**
4.1 Description des automates 16
4.2 Visualisation des automates et de leur trace d'exécution 19
4.3 Autres possibilités de preuve 19

5 De la description MAE au problème SAT **19**
5.1 SAT 20
5.2 Abstraction 21
5.2.1 Principe de l'abstraction 22
5.2.2 Avantages et Inconvénients 22

6 De la description MAE au problème SMT **25**
6.1 SMT 25
6.2 Instructions implémentées 26
6.3 Modélisation de la mémoire 26
6.3.1 Problématique 26
6.3.2 Solution 28
6.4 Optimisation de la vérification 30

7 Résultats et analyse détaillés des expérimentations et des tests de validation 32
7.1 Vérification de la correction des modélisations . . . 33
7.2 Mesure et analyse du temps de vérification . . . 34
7.2.1 Modélisations . . . 34
7.2.2 Conditions d'expérimentation . . . 35
7.2.3 Différentes approches de preuves MC BDD, SAT, SMT . . . 35
7.3 Mesure et analyse du coût de l'abstraction . . . 37
7.4 Comparaison des approches SAT avec abstraction et SMT sans abstraction . . . 38

8 Conclusion 39

9 Annexes 42
9.1 Liste des différentes instructions assembleur reconnues . . . 42
9.1.1 Récapitulatif des différentes instructions . . . 42
9.1.2 Récapitulatif des différentes formats d'instructions . . . 42

1 Contexte/Problématique

1.1 Contexte

Les attaques physiques ont été introduites vers la fin des années 90 comme un nouveau moyen de casser les systèmes cryptographiques [13]. De manière générale, les attaques physiques peuvent être utilisées pour casser un système contenant des informations sensibles, comme des cartes à puce bancaires, d'authentification, de métro... Le but d'une attaque physique active est de perturber l'exécution d'un programme pour en tirer des informations. Il existe différents types d'attaques physiques. La figure 1 représente un micro-contrôleur décapsulé sur lequel une fibre optique est positionnée. L'idée est d'introduire une lumière laser forte afin d'interférer avec le bon fonctionnement du processeur [15]. La figure 2 représente un autre micro-contrôleur et un allumeur de briquet [15]. Ce dernier est utilisé pour créer des perturbations électromagnétiques. Il existe d'autres types d'attaques physiques actives comme l'introduction d'une anomalie dans la fréquence ou la température de fonctionnement de la puce [11] [16], et la diminution de la tension d'alimentation [4].

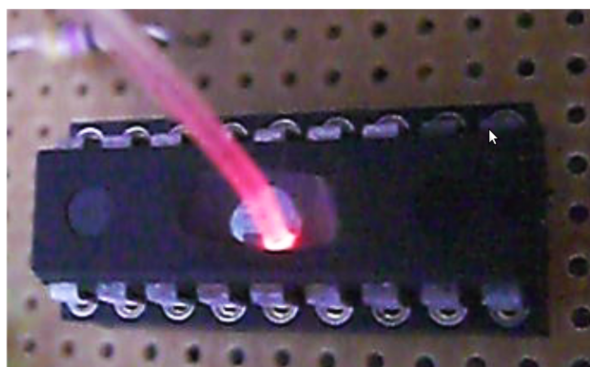

Figure 1 – Attaque au laser

Figure 2 – Attaque à impulsions électromagnétique

Ces attaques ont un effet au niveau physique sur une puce, et permettent par exemple la permutation d'un ou plusieurs bits sur le bus lors d'un transfert depuis la mémoire [11] [14] (la faute est alors transitoire) ou la permutation d'un bit ou plusieurs bits en mémoire (la faute est alors rémanente). Il existe différents types de fautes induites par un moyen physique d'attaque, certains effets visibles sur l'exécution du code peuvent être modélisés au niveau assembleur. Soit le code assembleur listing 4. Un modèle de faute possible est le remplacement d'une instruction assembleur. Il existe deux possibilités.

- L'instruction peut être remplacée par une instruction ayant un effet sur les registres utilisés dans la suite du programme. Dans ce cas, l'instruction sera remplacée par une nouvelle instruction. C'est le cas de l'exemple de code donné dans le listing 3, l'instruction *add r2, r2, 1* est remplacée par l'instruction *xor r2, r2, 1*.
- L'instruction peut être remplacée par une instruction n'ayant aucun effet sur les registres utilisés par le programme. Dans ce cas l'instruction de remplacement est assimilée à un NOP [1]. C'est le cas de l'exemple de code du listing 2, l'instruction *add r2, r2 1* à été remplacée par un NOP. Remplacer une instruction par un NOP peut être modélisé comme un "saut d'une instruction assembleur".

Listing 1 –

```
loop:
mov r3, #12
add r1, r2, r3
add r2, r2, #1
bne r2 loop
```

Listing 2 –

```
loop:
mov r3, #12
add r1, r2, r3
nop
bne r2 loop
```

Listing 3 –

```
loop:
mov r3, #12
add r1, r2, r3
xor r2, r2, #1
bne r2 loop
```

En résumé, les attaques physiques permettent d'introduire une faute lors de l'exécution d'un programme. Ces fautes sont modélisées à différents niveaux par des modèles de faute. Étant donné un modèle de faute, le challenge est alors : comment se protéger contre ces attaques ? Il existe deux types de protections : les protections matérielles et les protections logicielles. Un dispositif matériel de protection peut, par exemple, permettre de détecter une surchauffe de la puce et réagir. Au niveau logiciel, il s'agit souvent d'ajouter du code pour détecter et/ou tolérer une faute. Une protection logicielle est plus flexible qu'une protection matérielle car elle ne nécessite pas la modification du matériel. Elle permet par exemple la mise-à-jour d'une protection obsolète à moindre coût. Une protection logicielle au niveau assembleur permet une granularité d'étude plus fine comparé à un langage de haut niveau comme le C. Par exemple, proposer des contre-mesures en C pour le modèle de faute "saut d'une instruction assembleur" serait délicat. En effet une ligne de code en C correspond à une ou plusieurs instructions assembleur ; le nombre dépend de la ligne de code C, du compilateur et des options utilisées lors de la compilation. Il est difficile de prévoir le code assembleur correspondant à une ligne C et donc de prévoir l'effet d'une faute modélisée au niveau assembleur sur le code C. Il est donc difficile de proposer des contre-mesures

1. Une instruction NOP est une instruction qui ne fait aucune action utile dans le processeur. NOP = No OPeration.

C adaptées à un modèle de faute au niveau assembleur. Pour résoudre ce problème une option est alors d'approximer le modèle de faute en augmentant la granularité, en utilisant par exemple le modèle de faute "saut d'une ligne de code C". Dans ce cas, les protections seront moins fines et moins proches de l'effet réelle d'une attaque. C'est pourquoi nous choisissons de travailler au niveau assembleur : plus de fautes peuvent être prises en compte et plus fine est la protection du code.

Il existe un besoin croissant de certification des produits logiciels sur le marché, notamment ceux destinées à fonctionner sur des systèmes critiques tel que les transport en commun (avions, train, métro, ...) mais aussi tout système devant respecter une exigence de sécurité. Par ailleurs, le plus haut niveau de certification requiert des preuves formelles. Il existe donc un besoin de prouver l'efficacité des contre-mesures proposées destinées à protéger des systèmes contenant des informations sensibles. Il est donc intéressant de déterminer un moyen de preuve formelle de contre-mesure au niveau assembleur : c'est le sujet de mon stage.

Dans la suite nous approfondirons la problématique de mon stage, la solution et la méthode de validation proposées ainsi que la mise en œuvre de la solution et les résultats obtenus.

1.2 Problématique

Mon stage se base essentiellement sur un article publié par Nicolas Moro et al. [13]. Dans celui-ci, il propose pour presque toutes les instructions du jeu assembleur Thumb2 de ARM [6] une séquence de remplacement tolérante au type de faute "saut d'une instruction assembleur". Une seule faute est supposée survenir lors de l'exécution d'une séquence de remplacement : il est difficile de provoquer deux fautes dans un intervalle de temps inférieur à l'exécution de quelques instructions [13]. Ces séquences de remplacement ont été formellement prouvées correctes/tolérantes aux fautes. La figure 3 représente le schéma de preuve utilisé. L'idée est de modéliser l'effet de chaque instruction et sa contre-mesure en présence ou non de faute avec des automates. Les deux automates sont alors initialisés avec les mêmes entrées (état initial) et le principe de la preuve d'efficacité de la contre-mesure est de comparer les états finaux des deux automates. Si ceux-ci sont égaux, cela signifie que les effets de l'instruction et sa contre-mesure avec ou sans faute sont équivalentes. C'est le principe de l'équivalence-checking.

Dans [13], la vérification formelle a été réalisée par model-checking à base de BDD [2]. Malheureusement, cette méthode se heurte à une explosion combinatoire avec le nombre d'instructions et la taille des registres en nombre de bits. Dans certains cas, le temps de vérification à été supérieur à 24 heures ne permettant pas d'utiliser cette approche pour des vérifications sur des codes plus complexes/gros.

2. Binary Decision Diagram

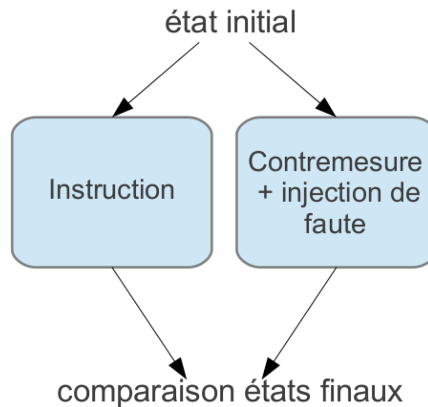

FIGURE 3 – Schema de preuve

Heureusement, il existe d'autre méthodes de vérification formelle utilisant des représentations SAT et SMT. Ces dernières appartiennent à la catégorie du "bounded model-checking", alors que la méthode utilisée appartient au "unbounded model-checking". La grande différence est que dans le cas des automates, le "unbounded model-checking" cherchera à prouver la validité d'une propriété dans tous les scénarios possibles non bornées dans le temps. Alors que le "bounded model-checking" cherchera à le prouver en un temps borné : l'exécution de l'automate est dépliée d'un nombre fixe de pas.

Le but de mon stage est de proposer une modélisation SAT et SMT pour vérifier l'efficacité des contre-mesures proposées. L'objectif à terme est de pouvoir utiliser cette modélisation sur des problèmes plus complexes [9]. De plus, ce stage vise aussi une automatisation du schéma de preuve.

2 Modélisation SAT/SMT

Cette partie présente d'abord les contre-mesures de l'article [13] et le schéma de preuve. La solution proposée pour transposer le schéma de preuve (l'approche envisagée) est présentée avant d'expliquer les problèmes SAT et SMT ainsi que leurs différences.

2.1 Transposition du schéma de preuve

Dans un premier temps les contre-mesures sont détaillées puis le schéma de preuve est expliqué.

2.1.1 Contre-mesures existantes

Dans [13], les instructions du jeu d'instructions Thumb2 sont regroupées en trois classes, chacune de ces trois classes ayant un schéma de séquence de remplacement associé. Le tableau 1 donne un exemple d'instruction de chaque classe avec la contre-mesure correspondante.

classe	instruction	action	contre-mesure
idempotente	*add rs, ra, rb*	$rs \leftarrow ra + rb$	*add rs, ra, rb* *add rs, ra, rb*
séparable	*add rs, rs, rb*	$rs \leftarrow rs + rb$	*mov rx, rs* *mov rx, rs* *add rs, rx, rb* *add rs, rx, rb*
spécifique	*adcs rs, ra, rb*	$rs \leftarrow ra + rb + flags.carry$ *flags update*	*mrs rx, apsr* *mrs rx, apsr* *adcs rs, ra, rb* *msr apsr, rx* *msr apsr, rx* *adcs rs, ra, rb*

Table 1 – Tableau récapitulatif des classes d'instruction

Instructions idempotentes Les instructions idempotentes possèdent le même effet sur les registres qu'elles soient exécutées une ou plusieurs fois. La séquence de remplacement d'une instruction idempotente est simplement sa duplication.

Instructions séparables Les instructions séparables ne sont pas idempotentes, mais elles peuvent être décomposées/séparées en différentes instructions idempotentes. La séquence de remplacement d'une instruction séparable est donc la duplication de chacune des instructions idempotentes de sa décomposition.

Instructions spécifiques Les instructions spécifiques ne sont ni idempotentes ni séparables, elles ne possèdent pas de séquence de remplacement évidente. Chaque instruction de cette classe a sa propre contre-mesure spécifique. Certaines n'en ont pas.

2.1.2 Représentation en automates

Une modélisation possible pour vérifier une propriété à la fin d'une suite d'instructions est de modéliser la suite d'instructions par un automate. Les instructions et les contre-mesures correspondantes sont ainsi modélisées. Cette partie explique comment représenter une suite d'instructions avec des automates interprétés.

États Chaque état de l'automate représente un ensemble de configurations du système réel. Dans notre cas, une configuration du système est l'association d'une valeur à chaque registre et case mémoire manipulée, ainsi qu'au PC [3] et aux drapeaux (flags) [4]. Chaque instruction agit comme une fonction prenant en entrée une configuration et produisant en sortie une nouvelle configuration. Un suite d'instructions assembleur est alors vu comme une succession de fonctions. Nous cherchons à vérifier l'équivalence entre une instruction et sa contre-mesure. Pour ce faire, ces deux dernières seront représentées avec des automates, initialisés dans le même état. Et une propriété est vérifiée sur les configurations possibles des états finaux des deux automates.

Transitions Chaque transition de l'automate représente l'effet de l'instruction pointée par le PC sur l'ensemble des registres. À partir d'un état source il existe potentiellement plusieurs états destinations, c'est le cas de la représentation d'un branchement conditionnel. Une transition peut alors avoir une condition de franchissement, celle correspondant au branchement.

Une faute de type "saut d'une instruction assembleur" est représentée par une transition n'ayant aucun effets sur les registres, les flags, ou la mémoire. Le PC est par-contre mis-à-jour, et pointe vers l'instruction suivante dans le code.

Propriété à vérifier pour prouver l'efficacité Nous cherchons à savoir si les deux automates produisent des traces menant dans des états finaux identiques. La propriété que nous cherchons à vérifier est la suivante : dans tous les cas, si les deux automates sont arrivés dans leurs état finaux, alors les valeurs contenues dans les registres sont identiques. Chercher à savoir que cette propriété est vraie dans tous les cas, équivaut à chercher un cas dans lequel cette propriété est fausse, et donc un contre-exemple. S'il n'existe pas de contre-exemple, la propriété est toujours vraie.

3. PC = Program Counter, adresse de la prochaine instruction à exécuter.
4. Ensemble de bits contenant des informations sur l'état du processeur.

2.1.3 Transposition

Dans la suite, la solution proposée pour transposer le schéma de preuve d'équivalence-checking en utilisant une modélisation SAT et SMT est développée. Afin de mieux comprendre, la représentation sous forme d'automate de l'instruction *add* idempotente et de sa contre-mesure est prise comme exemple. Ces automates sont illustrés dans la figure 4.

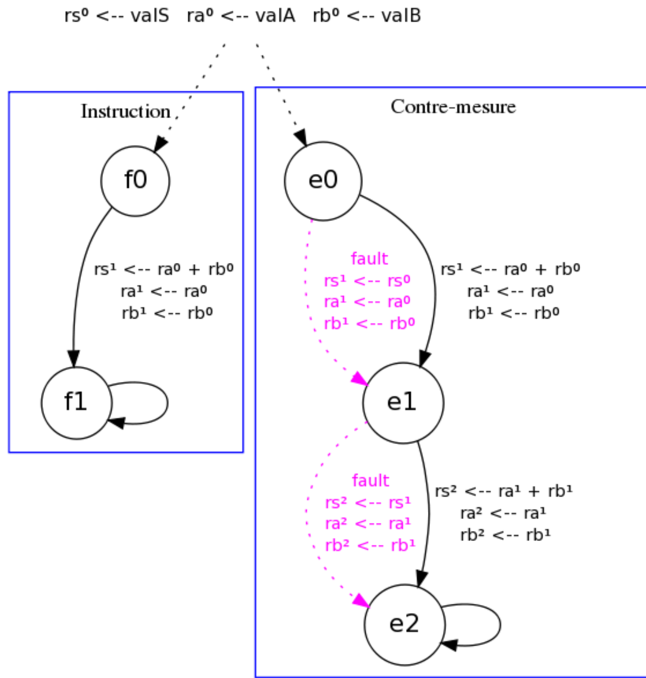

FIGURE 4 – Modélisation de l'instruction *add* idempotente et de sa contre-mesure potentiellement fautée

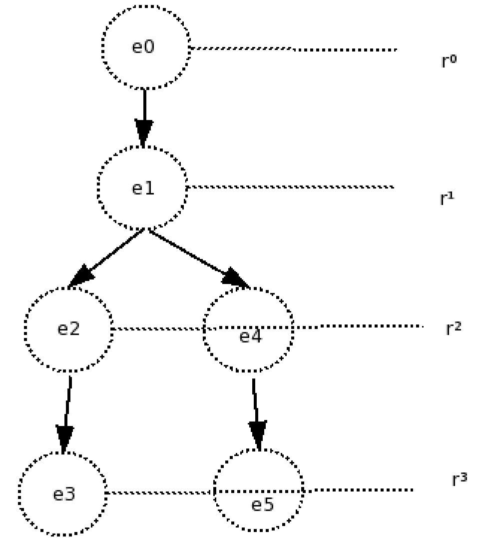

FIGURE 5 – Numérotation des variables représentant un registre

États Un état représente une configuration de l'ensemble des registres du processeur. Un registre est représenté par un ensemble de variables, chacune de ces variables correspond au registre à un pas donné de la trace d'exécution. Le nombre de pas étant le nombre de transitions franchies depuis l'état initial. Ainsi les variables rs^0, rs^1, rs^2 représentent le registre rs au bout de zéro, une, deux transitions. Ce principe est illustré dans la figure 5. Les deux automates sont initialisés avec le même état initial, les variables représentant les registres à l'état initial sont alors affectées avec des variables libres : $rs^0 \leftarrow valS$ $ra^0 \leftarrow valA$ $rb^0 \leftarrow valB$. Ici $valA$, $valB$, $valC$ sont des variables libres. Une variable libre peut avoir n'importe quelle valeur dans son domaine de définition.

Transitions Chaque transition représente l'exécution d'une instruction. Par exemple la transition $e0 \longrightarrow e1$ de l'automate *instruction* effectue une addition des variables représentant *ra* et *rb* à l'état initial et l'affecte à la variable représentant *rs* au pas suivant : $rs^1 \leftarrow ra^0 + rb^0$ Les variables ra^1 et rb^1 subissent une simple affectation des variables ra^0 et rb^0, autrement dit les registres *ra* et *rb* ne subissent aucune modification de la part de cette instruction : $ra^1 \leftarrow ra^0$ $rb^1 \leftarrow rb^0$ Ces actions représentent donc l'instruction *add rs, ra, rb* au premier pas d'exécution. Toutes les transitions de l'automate *contre-mesure* sont accompagnées d'une transition représentant une faute. Le modèle de faute considéré est "saut d'une instruction assembleur". Une telle faute se traduit dans l'automate par une absence d'actions sur les variables représentant les registres. En d'autres termes, la valeur d'une variable à un pas est simplement transmise à la même variable du pas suivant : $rs^1 \leftarrow rs^0$ $ra^1 \leftarrow ra^0$ $rb^1 \leftarrow rb^0$

Propriété à vérifier pour prouver l'efficacité Nous cherchons à savoir si les deux automates produisent des traces menant dans des états finaux identiques modulo le PC. Dans l'exemple, nous cherchons à savoir si dans tous les cas, lorsque l'automate *instruction* atteint l'état *f1* et l'automate *contre-mesure* l'état *e2*, les valeurs des variables rs^1 ra^1 rb^1 de *instruction* sont identiques aux valeurs des variables rs^2 ra^2 rb^2 de *contre-mesure.*

Représentation finale sous forme logique La modélisation avec des automates est la représentation choisie pour vérifier la propriété d'équivalence entre une instruction et sa contre-mesure. Mais la représentation finale doit être sous forme d'une formule logique afin qu'un outil de résolution de problèmes puisse déterminer s'il existe un contre-exemple de la propriété. La formule peut être exprimée avec la logique propositionnelle (formulation SAT) ou avec la logique avec des prédicats (formulation SMT). Ensuite un outil de résolution de problèmes cherchera alors une configuration des variables de la formule logique la rendant vraie. S'il existe une telle configuration alors il existe au moins un contre-exemple. Sinon il n'en existe pas.

2.2 SAT

Un problème SAT a pour objectif de déterminer si une formule possède une solution, s'il existe une affectation de variables (booléennes) qui rend la formule vraie. Un SAT-solver est un outil permettant de résoudre ce type de problème exprimé avec une formule en logique propositionnelle avec des variables booléennes. Plus précisément, un problème SAT se traduit par une formule en logique propositionnelle avec des variables booléennes. Une formule logique peut être perçue comme un ensemble de contraintes que les variables doivent respecter. Voici deux exemples pour mieux comprendre :

Soit l'ensemble de contraintes suivantes constituées par les variables $a,\ b,\ c,\ d$: $a = b$, $c = d$, $a \neq c$
Nous voulons que les contraintes $(a = b)$ et $(c = d)$ et $(a \neq c)$ soient respectées. La formule en logique propositionnelle finale obtenue est donc : $(a = b) \wedge (c = d) \wedge (a \neq c)$

Cette formule propositionnelle finale est donnée à un SAT-solver. Ce dernier cherche alors une configuration de $a,\ b,\ c,\ d$ telle que la formule propositionnelle finale soit logiquement vraie (l'ensemble des contraintes imposées soient respectées). Voici une solution de cette formule : $a = 1$, $b = 1$, $c = 0$, $d = 0$

Un SAT-solver répondra donc *SATISFIABLE* car il existe (au moins) une solution à cet ensemble de formules logiques.

Voici un autre exemple dans lequel il n'existe pas de solution : $a = b$, $c = d$, $b = d$, $a \neq c$
Un SAT-solver répondra alors *UNSATISFIABLE* car il n'existe pas de solution à cet ensemble de formules logiques (l'ensemble des contraintes imposées ne peuvent pas être respectées).

Malheureusement les algorithmes de résolution des formules dans les SAT-solvers travaillent à partir d'une représentation particulière de la formule propositionnelle : une conjonction de clauses disjonctives (CNF) [5]. Voici ci-dessous la forme CNF du premier exemple donné ci-dessus :
$Exemple1$: $(\overline{a} \vee b) \wedge (\overline{b} \vee a) \wedge (\overline{c} \vee d) \wedge (\overline{d} \vee c) \wedge (a \vee c) \wedge (a \vee \overline{a}) \wedge (\overline{c} \vee c) \wedge (\overline{c} \vee \overline{a})$

L'exemple donné est simple à écrire avec la logique propositionnelle. Pourtant la formule booléenne sous forme CNF correspondante est longue et moins intuitive. Pour cette raison, représenter un problème SAT peut s'avérer très lourd, d'autant plus si le problème cherche à représenter des opérateurs complexes tels que l'addition, la multiplication, la division, etc, qui correspondent aux manipulations effectuées par les instructions assembleur. Toutefois il existe la possibilité d'optimiser (nombre de variables utilisées, taille des clauses, nombre de clauses) finement les expressions car nous manipulons directement la formule finale.

5. Conjunctive Normal Form, exemple : $\underbrace{(a \vee b)}_{\text{clause}} \wedge \underbrace{(c \vee d \vee e)}_{\text{clause}} \wedge ...$

2.3 SMT

L'objectif des problèmes SMT est le même que celui des problèmes SAT : un problème SMT a pour objectif de savoir si un ensemble de formules logiques possède une solution. Un SMT-solver est un outil permettant de résoudre ce problème. Du point de vue utilisateur, la grande différence entre les problèmes SAT et SMT est la façon dont les formules logiques sont exprimées. Les SMT-solver se basent sur des théories décidables pour analyser et résoudre les formules, mais les variables peuvent avoir des domaines plus larges que celui des booléen (entiers, tableaux, listes, ...) et il est possible de manipuler des prédicats (des fonctions prenant en entrée des variables et renvoyant une valeur booléenne) [7] [12]. Certaines théories permettent l'utilisation de vecteurs de bits et d'opérateurs complexes ce qui rend les problèmes SMT plus faciles à décrire que les problèmes SAT. En contrepartie, la possibilité d'exprimer directement les clauses est perdue car nous n'avons plus le contrôle sur la transposition des formules sous forme CNF (la plupart des SMT-solver font appel à des SAT-solver). Nous pouvons comparer les problèmes SAT et SMT à l'assembleur et le langage C. L'expression des problèmes SAT est plus "bas niveau" que l'expression des problèmes SMT, et possède donc les avantages et inconvénients du "bas niveau". Voici deux exemples d'un problème SMT :
$Exemple1$: $(a + b = 10) \wedge (a = 2)$
$Exemple2$: $(a + b = 10) \wedge (a = 2) \wedge (b \leqslant 0)$

La seule solution pour la première formule est $b = 8$, un SMT-solver renverra *SATISFIABLE* car il existe (au moins) une solution. Par contre il n'existe pas de solution pour la deuxième formule, un SMT-solver renverra *UNSATISFIABLE*.

3 Automatisation

Nous venons de voir que les problèmes SAT sont lourds à exprimer. Même si les problèmes SMT le sont moins, afin de faciliter la preuve d'équivalence entre une instruction et sa contre-mesure il est intéressant de posséder un logiciel construisant, à partir d'un code assembleur et de sa contre-mesure, la représentation SAT ou SMT du problème d'équivalence étudié.

3.1 Principe

L'objectif est ici d'implémenter un logiciel prenant en entrée deux codes assembleur (une instruction et sa contre-mesure) et donnant en sortie des formules logiques représentant le problème d'équivalence que nous cherchons à résoudre. L'ensemble des formules est destiné à être analysé

par un SAT-solver ou un SMT-solver. Ces derniers renverront une réponse (*satisfiable* ou *unsatisfiable*) au problème d'équivalence. La figure 6 illustre ce principe.

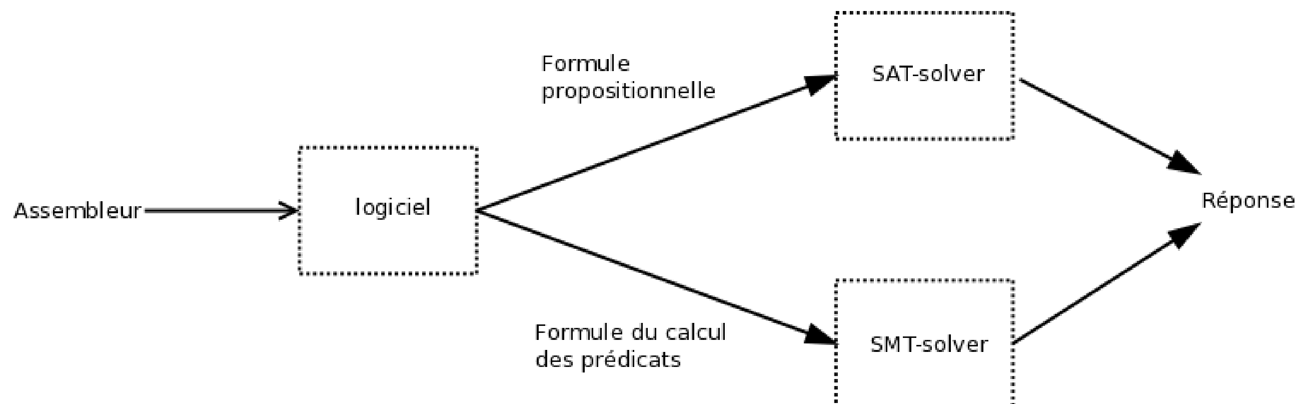

FIGURE 6 – Principe de l'automatisation

3.2 Architecture logicielle

Le découpage illustré par la figure 7 représente différentes formes d'expression du problème que nous cherchons à résoudre. Chaque brique logicielle (L1, L2, L3, L4) prend en entrée la sortie d'une autre brique logicielle. L'idée est de complexifier à chaque étape l'expression du problème jusqu'à ce qu'il soit sous la forme attendue par un SAT-solver ou un SMT-solver, qui pourra alors le résoudre. Dans la suite, les différentes formes d'expression du problème (label des flèches sur le schéma 7) sont expliquées.

MAE [6] C'est un langage décrivant simplement des automates représentant une instruction et sa contre-mesure. Ce langage est une étape intermédiaire entre le code assembleur et l'expression du problème en logique propositionnelle (*lp*) ou avec un langage d'expression des problèmes SMT (*cvc*).

cvc cvc3 est un outil permettant de résoudre des problèmes SMT. Il possède un langage d'expression des problèmes SMT qui lui est propre, nommé ici *cvc*. À cette étape les formules sont donc exprimées sous cette forme. C'est la dernière étape avant de chercher à résoudre le problème avec le SMT-solver de cvc3.

6. Machines À États

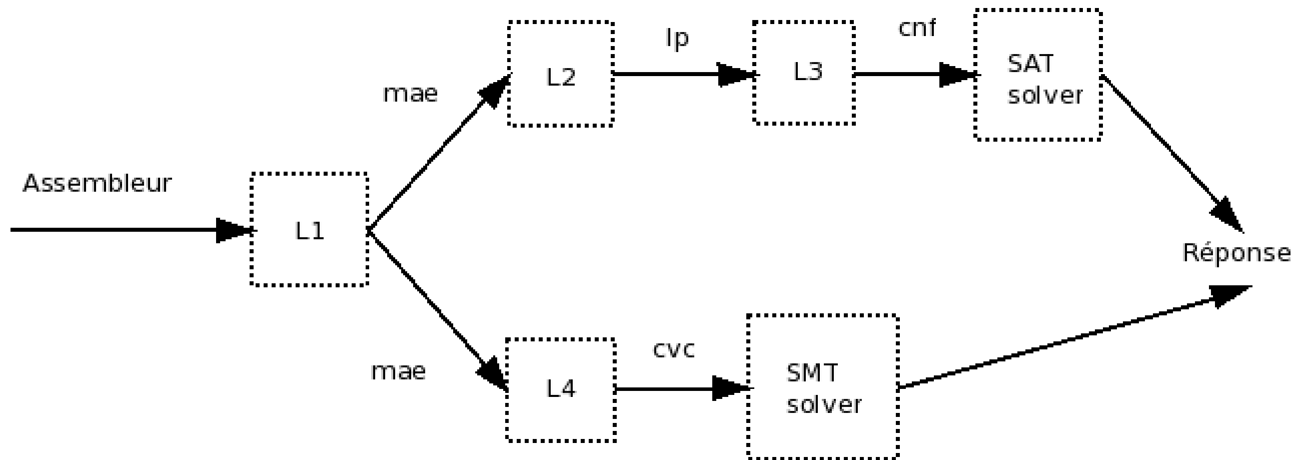

FIGURE 7 – Architecture logicielle pour l'automatisation des preuves

lp [7] A cette étape, le problème est exprimé avec une formule en logique propositionnelle plus simple à écrire que la forme CNF du même problème. L'expression des opérations doit cependant être réalisée uniquement avec des variables booléennes, elle est donc très lourde.

CNF A cette étape, le problème SAT est exprimé avec une formule booléenne sous la forme CNF forme obligatoire pour l'expression d'un problème SAT. Cette formule est l'entrée du SAT-solver. Elle peut être très longue et difficile à écrire.

3.3 Outils

Le découpage précédemment présenté nécessite l'implémentation d'un certain nombre de briques logicielles. Certaines de ces briques logicielles doivent répondre à des problèmes auxquels d'autres personnes se sont déjà confrontées et ayant proposé des solutions. Dans un but d'efficacité certaines étapes utiliserons des outils existant. Voici ci dessous la liste des outils utilisés.

- Le SAT-solver utilisé est CirCus inclus dans l'outil vis [2] .
- Le SMT-solver utilisé est cvc3 et cvc4 [3].
- PBL est un logiciel permettant de transformer une formule en logique propositionnelle en une formule booléenne sous forme CNF [1].

7. Logique Propositionnelle

4 Langage MAE

Le langage *MAE* permet de décrire des automates représentant un code assembleur. Son but est de faciliter la description du problème SAT/SMT en traduisant un fichier MAE en un fichier de description de problèmes SAT/SMT. À l'aide d'un parseur permettant de traduire les fichiers MAE, un même fichier MAE peut être traduit en un problème SAT ou en un problème SMT. Outre le fait que ce langage permet de faciliter le travail de description, il permet également de gagner du temps. Ainsi les travaux futurs (en doctorat) pourront se concentrer sur la recherche de nouvelles contre-mesures et la modélisation plutôt que sur la descriptions des problèmes à vérifier.

Dans la suite le langage MAE est présenté. Nous verrons comment décrire un automate représentant un code assembleur et les différentes traductions prises en charge par le parseur, ainsi que les différentes instructions que ce dernier peut comprendre.

4.1 Description des automates

Cette partie développe la description d'automates avec le langage MAE. Comme tout langage, le langage MAE possède un ensemble de mots clés. Nous pouvons distinguer deux types de mots clés : ceux qui sont spécifiques à la description d'automates (ils auraient pu être utilisés pour autre chose que notre problème d'équivalence-checking) et ceux qui sont spécifiques à notre problème d'equivalence-checking.

Mots clés spécifique à la description d'automates

- *graph* permet de déclarer un automate, un fichier MAE doit en contenir deux pour l'equivalence-checking.
- *state initial/ final /default* permet de déclarer un état de l'automate. Les mots clés *initial/final* indiquent si l'état en question est un état initial ou final de l'automate.
- *tran* permet de déclarer une transition de l'automate. Chaque transition possède une condition de transition (*cond*) et une instruction associée à la transition (*inst*).
- *cond* est le mot clé de déclaration de la condition associée à une transition.

Mots clés spécifique à notre problème d'equivalence-checking de code

- *register used/dead* permet de déclarer un ou plusieurs registres du processeur. Le mot clé *used* indique que les registres qui suivent doivent être vérifiés lors de la comparaison finale du contenu des registres des deux automates. Le mot clé *dead* indique que les registres qui suivent ne sont pas à considérer lors de la comparaison finale, autrement dit, la configuration de sortie des deux automates peuvent être équivalentes même si les registres *dead* ne

possèdent pas les même valeurs. Cette fonctionnalité est utilise lorsque les contre-mesures utilisent des registres de travail temporaires.

- *register size* spécifie la taille des registres (en bits).
- *flags used/dead* permet d'indiquer si les flags doivent être pris en compte dans la comparaison finale.
- *memory used/dead* permet d'indiquer si la mémoire sera prise en compte dans la comparaison finale.
- *steps* spécifie le nombre de pas d'exécution depuis l'état initial, et donc la profondeur de l'arbre représentant les exécution de l'automate déplié.
- *fault* est le modèle de faute pris en compte. Pour l'instant seul le modèle de faute "saut d'une instruction assembleur" est modélisé. Il est donc possible d'utiliser les mot clés *"fault = instruction skip"* afin d'indiquer que l'automate prend en compte ce modèle de faute, ou les mots clés *"fault = none"* afin de d'indiquer que l'automate ne prend en compte aucun modèle de faute.
- *inst* permet de déclarer l'instruction associée à une transition. Les instructions reconnues par le parseur seront développées dans la suite.

Nous cherchons à prouver l'équivalence entre deux codes assembleurs. Un fichier MAE possède alors la descriptions de deux automates. Un des automates représente l'instruction que nous cherchons à sécuriser, et l'autre la contre-mesure de l'instruction. Le code de la figue 8 représente la description en automates de l'instruction idempotente *add rs ra rb* et de sa contre-mesure. Cette instruction et sa contre-mesure sont données dans le tableau 1.

La première ligne du fichier déclare les registres utilisées, c'est à dire les registres *rs, ra, rb*. La propriété à vérifier pour l'équivalence-checking comportera une propriété d'égalité des registres *rs, ra, rb* pour les deux automates. Les deux lignes suivantes indiquent que les flags et la mémoire doivent être pris en compte dans la formule à vérifier pour la preuve d'équivalence-checking. Ensuite la taille des registres est définie, ici 16 bits. L'automate *golden* est la représentation de l'instruction *add rs ra rb*, et l'automate *faulty* celui de sa contre-mesure. Les mots clés *state* déclarent les différents états des deux automates, et donc les différentes valeurs que le PC peut prendre. L'automate *golden* en possède deux et *faulty* trois car les codes associés à ces deux automates possèdent respectivement une et deux instructions, et qu'il n'y a pas d'instructions de branchement. Le mot clé *step* indique le nombre de transitions que l'automate doit franchir, autrement dit il indique le nombre d'instructions à exécuter pour chaque automate avant de comparer les valeurs contenues dans les différents registres. Le mot clé *fault* définit le modèle de faute pris en compte dans chaque automate. Les mot clés *"fault = none"* de l'automate *golden* indiquent qu'il n'est pas fauté tandis que les mots clés *"fault = instruction skip"* de l'automate

```
register used rs, ra, rb
flags  used
memory used
register size  = 16


graph golden
     state initial e0
     state final   e1

     steps = 1
     fault = none

     tran e0 -> e1
          cond true
          inst add rs, ra, rb


graph faulty
     state initial f0
     state default f1
     state final   f2

     steps = 2
     fault = instruction skip

     tran f0 -> f1
          cond true
          inst add rs, ra, rb

     tran f1 -> f2
          cond true
          inst add rs, ra, rb
```

FIGURE 8 – Fichier de description d'automates *MAE* pour l'instruction *add* idempotente

faulty indiquent qu'il doit modéliser la faute "saut d'une instruction assembleur". Ensuite les différentes transitions sont décrites. Chaque transition possède une condition de transition et une instruction. Cette dernière modifie les registres (registres du processeur/ flags / mémoire). Chaque transition modifie l'état du PC, et les instructions associées décrites après le mot clé *inst* correspondent aux instructions des deux codes assembleurs dont nous cherchons à montrer l'équivalence avec ou sans faute.

4.2 Visualisation des automates et de leur trace d'exécution

Le parseur du langage MAE permet de traduire une description de deux automates en un langage de description de problèmes SAT/SMT. Ce dernier peut ensuite être donné en entrée à un logiciel de résolution de problème SAT/SMT (SAT-solver/SMT-solver). Il se peut que le résultat de la vérification formelle ne soit pas celui attendu. Ceci peut être le résultat d'une erreur lors de la description des automates. Le parseur permet de visualiser les automates décrits dans le fichier MAE ainsi que l'arbre représentant les traces d'exécution de l'automate. Pour ce faire, le parseur traduit le fichier MAE initial en quatre fichiers *dot*. Le langage dot est un langage de description de graphe, des outils tel que *graphviz* permettent ensuite de visualiser le graphe. Les quatre fichiers générés sont la description en langage dot des deux graphes ainsi que les traces d'exécutions associées. Les deux arbres représentant les traces d'exécution des deux automates possèdent une profondeur précisée dans le fichier de description. Cette fonctionnalité permet ainsi de visualiser le code décrit dans le fichier MAE, facilitant la recherche d'erreurs.

4.3 Autres possibilités de preuve

Il est intéressent de remarquer que outre le fait que ce langage permet de comparer un code non fauté avec un code fauté il permet également que comparer deux codes non fautés ou deux codes fautés. Pour cela il suffit de déclarer *"fault = none"* ou *"fault = instruction skip"* dans les deux automates. Ceci permet de prouver l'équivalence entre deux code assembleurs par équivalence-checking, ainsi que l'équivalence entre deux différentes contre-mesures. Cela pourrait nous être utile dans la suite des travaux associés à ce stage [9].

Dans le cadre de ce sujet de recherche, le code du parseur à été structuré dans le but de pouvoir accueillir d'autres modèles de fautes, comme la faute "remplacement d'une instruction assembleur" par exemple.

5 De la description MAE au problème SAT

Cette partie développe le procédé de traduction d'un fichier de description MAE vers un problème SAT. Les difficultés de modélisation des problèmes SAT sont approfondies, notamment la représentation des opérateurs. Une abstraction permettant de simplifier la représentation des opérateurs est également détaillée.

5.1 SAT

La description d'un problème SAT est très lourde et complexe pour notre problème d'équivalence-checking. Celui-ci doit être une formule booléenne sous forme CNF. Cette formule est très difficilement compréhensible par un être humain. La recherche de bugs devient un tâche longue, même pour des problèmes de petite taille. Heureusement, nous n'avons pas à travailler directement avec la formule sous forme CNF, mais avec la formule en logique propositionnelle. Le logiciel *pbl* nous permet de traduire la formule en logique propositionnelle vers sa forme CNF. Mais le problème à exprimer reste néanmoins long et difficile, notamment parce qu'il faut exprimer des opérateurs complexes telles que l'addition, la multiplication ou encore la division.

La figure 9 illustre la transition représentant l'exécution de l'instruction idempotente *add*, et la figure 10 son implémentation en logique propositionnelle pour une taille de registre de 4 bits. La formule ci-dessus représente plus simplement la formule de la figure 10. Les indices associés aux registres représentent le pas d'exécution. Lorsqu'une instruction est exécutée le pas augmente. Seul le PC et le registre rs sont modifiés, la valeur des autres registres est conservée.

$$(pc^1 = f1) \wedge (rs^1 = ra^0 + rb^0) \wedge (ra^1 = ra^0) \wedge (rb^1 = rb^0)$$

Dans une formule en logique propositionnelle seul les variables booléennes peuvent être utilisées. C'est pour cette raison que dans la figure 10 l'opérateur d'addition est expansé et les registres sur quatre bits à un pas donné d'exécution sont représentée par quatre variables booléennes, une pour chaque bit.

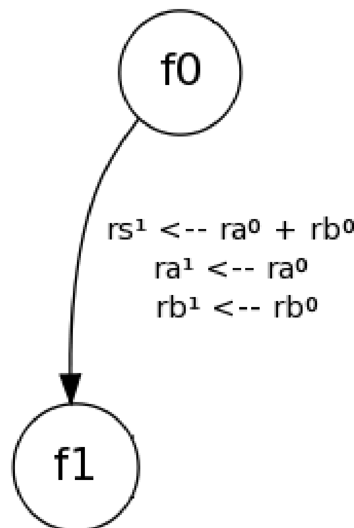

Figure 9 – Transition représentant l'exécution de l'instruction idempotente *add*

```
(faulty_step1_pc <=> f1) & (faulty_step1_carry_bit0 <=> 0)  &
(faulty_step1_carry_bit1 <=> (faulty_step0_ra_bit0 & faulty_step0_rb_bit0) |
(faulty_step1_carry_bit0 & (faulty_step0_ra_bit0 XOR faulty_step0_rb_bit0))  )
& (faulty_step1_carry_bit2 <=> (faulty_step0_ra_bit1 & faulty_step0_rb_bit1) |
(faulty_step1_carry_bit1 & (faulty_step0_ra_bit1 XOR faulty_step0_rb_bit1))  )
& (faulty_step1_carry_bit3 <=> (faulty_step0_ra_bit2 & faulty_step0_rb_bit2) |
(faulty_step1_carry_bit2 & (faulty_step0_ra_bit2 XOR faulty_step0_rb_bit2))  )
& (faulty_step1_carry_bit4 <=> (faulty_step0_ra_bit3 & faulty_step0_rb_bit3) |
(faulty_step1_carry_bit3 & (faulty_step0_ra_bit3 XOR faulty_step0_rb_bit3))  )
& (faulty_step1_rs_bit0 <=> (faulty_step0_ra_bit0 XOR faulty_step0_rb_bit0 XOR
faulty_step1_carry_bit0) )  & (faulty_step1_rs_bit1 <=> (faulty_step0_ra_bit1
XOR faulty_step0_rb_bit1 XOR faulty_step1_carry_bit1) )  &
(faulty_step1_rs_bit2 <=> (faulty_step0_ra_bit2 XOR faulty_step0_rb_bit2 XOR
faulty_step1_carry_bit2) )  & (faulty_step1_rs_bit3 <=> (faulty_step0_ra_bit3
XOR faulty_step0_rb_bit3 XOR faulty_step1_carry_bit3) )  &
(faulty_step1_ra_bit0 <=> faulty_step0_ra_bit0) & (faulty_step1_ra_bit1 <=>
faulty_step0_ra_bit1) & (faulty_step1_ra_bit2 <=> faulty_step0_ra_bit2) &
(faulty_step1_ra_bit3 <=> faulty_step0_ra_bit3)  & (faulty_step1_rb_bit0 <=>
faulty_step0_rb_bit0) & (faulty_step1_rb_bit1 <=> faulty_step0_rb_bit1) &
(faulty_step1_rb_bit2 <=> faulty_step0_rb_bit2) & (faulty_step1_rb_bit3 <=>
faulty_step0_rb_bit3)
```

FIGURE 10 – Implémentation de la transition de la figure 9 en une formule logique proportionnelle

La formule de la figure 10 fait partie d'une autre formule qui l'englobe représentant la description du problème d'équivalence-checking. L'opérateur implémenté (l'addition) est un des plus simples et la taille des bits est réduite (4 bits), il est facile d'imaginer la difficulté d'écriture et de compréhension de telles formules pour des opérateurs plus complexes (la multiplication notamment) et pour des tailles de bits plus élevés. C'est une des raisons pour laquelle une abstraction des opérations associées aux instructions à été appliquée pour les problèmes SAT. Malheureusement l'implémentation actuelle de cette abstraction possède des limitations et ne permet pas de prendre en compte certains cas, limitant ainsi le nombre de problèmes que le parseur du langage MAE peut traduire. Elle couvre néanmoins tout les cas présentés dans le jeu de test de validation. La suite développe l'abstraction des opérateurs implémentée.

5.2 Abstraction

Nous avons vu que l'expression des problèmes SAT nécessite l'expression booléenne d'opérations effectuées par les instructions ce qui peut alourdir considérablement la taille des formules. Nous cherchons à prouver à l'aide d'un SAT-solver l'équivalence entre une instruction et sa contre-mesure. Toutefois deux codes assembleur possèdent a priori des instructions effectuant autre

chose que des opérations booléennes. Dans un souci de simplicité et d'efficacité, il est alors intéressant de chercher à s'abstraire des opérations. Une abstraction a été proposée, et dans la suite son principe est développé ainsi que ses avantages et ses inconvénients.

5.2.1 Principe de l'abstraction

L'idée principale est de remplacer la valeur résultante d'une opération complexe (l'addition, la multiplication, la division, ...) par une valeur quelconque contenue dans une variable libre *val*. Les instructions effectuant la même opération avec les mêmes valeurs d'opérandes sources doivent affecter la même valeur à leur registre destination. Le tableau 2 montre un exemple. Cet exemple suppose que la valeur contenue dans le registre *rs* après la première addition est différente de la valeur contenue dans le registre *ra*. Dans ce cas, les instructions *add rs, ra, rb* et *add rs, rs, rb* possèdent des valeurs d'opérandes sources différentes. Ces deux opérations d'addition seront alors abstraites avec deux valeurs différentes : *val1* et *val2*.

	code initial	**action**	**contre-mesure**	**action**
sans abstraction	*add rs, ra, rb* *add rs, rs, rb*	$rs \leftarrow ra + rb$ $rs \leftarrow rs + rb$	*add rs, ra, rb* *add rs, ra, rb* *mov rx, rs* *mov rx, rs* *add rs, rx, rb* *add rs, rx, rb*	$rs \leftarrow ra + rb$ $rs \leftarrow ra + rb$ $rx \leftarrow rs$ $rx \leftarrow rs$ $rs \leftarrow rx + rb$ $rs \leftarrow rx + rb$
avec abstraction	*add rs, ra, rb* *add rs, rs, rb*	$rs \leftarrow val1$ $rs \leftarrow val2$	*mov rs, val1* *mov rs, val1* *mov rx, rs* *mov rx, rs* *mov rs, val2* *mov rs, val2*	$rs \leftarrow val1$ $rs \leftarrow val1$ $rx \leftarrow rs$ $rx \leftarrow rs$ $rs \leftarrow val2$ $rs \leftarrow val2$

TABLE 2 – Exemple d'abstraction

5.2.2 Avantages et Inconvénients

Sens du code Avec l'abstraction proposée les valeurs des opérations complexes sont perdues et remplacées par des valeurs quelconques. Le sens du code initial est modifié. Malgré cette modification le résultat de la vérification formelle sera correct pour le modèle de faute "saut d'une instruction assembleur" : nous ne cherchons pas à vérifier l'exactitude du résultat d'une

suite d'opérations mais nous cherchons à vérifier l'égalité de valeurs contenues dans des registres. Si dans tous les cas (pour toutes les valeurs possibles des variables libres), les valeurs comparées sont identiques alors l'instruction et sa contre-mesure sont équivalentes.

Simplification Cette abstraction simplifie l'expression d'un problème SAT. Malheureusement elle impose une analyse du code préalable afin de déterminer les variables libres à utiliser pour les résultats d'opérations. En effet, les instructions effectuant la même opération avec les mêmes valeurs d'opérandes sources doivent être remplacées avec une affectation de la même variable libre. Une méthode d'analyse automatique a été proposée. En quelques mots, la méthode consiste à enrichir la formule logique avec des contraintes sur les différentes variables libres *val* utilisées pour représenter les valeurs. Prenons en exemple le code assembleur du tableau 2. Dans ce cas toutes les instructions d'addition sont remplacées par une variable libre. Le code initial et sa contre-mesure possèdent ensemble six instructions d'addition, ces dernières sont donc abstraites par six variables libres. Il s'agit maintenant de savoir quelles sont les variables libres identiques. Dans un premier temps chaque variable libre est numérotée différemment (de 1 à 6), ensuite des contraintes sont imposées à ces variables libres afin de déterminer lesquelles doivent être égales. Si l'opération de la première variable ainsi que la valeur des registres sources sont identiques à la seconde variable alors elles sont égales, sinon si l'opération de la première variable ainsi que la valeur des registres sources sont identiques à la troisième variable alors elles sont égales, sinon si ... La méthode consiste à itérer l'algorithme précédent entre la première variable est la seconde, la troisième, la quatrième et ainsi de suite jusqu'à la dernière variable. Puis entre la seconde et la troisième, la quatrième, jusqu'à la dernière variable. Et ainsi de suite. Cet algorithme est illustré par le pseudo-code dans la figure 11. Dans cette figure, *val1, val2, val3, val4, val5, val6* sont les différentes variables utilisées dans le cas du code du tableau 2 que nous avons pris comme exemple. Dans ce pseudo-code, *val1.op* représente l'opération de *val1*, dans notre cas c'est l'opération d'addition, et *val.src* représente les différentes valeurs sources de l'opérateur.

Le nombre de nouvelles contraintes ajoutées à la formule propositionnelle est polynomiale en fonction du nombre d'opérateurs abstraits. Malheureusement cette méthode alourdit considérablement la formule logique, ce qui va à l'encontre du but initial qui consistait à chercher à la simplifier.

Taille des registres Néanmoins, le point positif est que l'abstraction permet de limiter la taille des registres en nombre de bits. Nous cherchons à savoir si les registres de l'automate représentant une instruction possèdent toujours les mêmes valeurs que les registres de l'automate représentant sa contre-mesure. Ainsi, si l'instruction utilise trois registres *ra, rb, rc*, ces derniers seront respectivement comparées aux registres *ra, rb, rc* de sa contre-mesure. Notre but n'est pas

```
     if( (val1.op==val2.op) and (val1.src==val2.src)  val1 = val2
else if( (val1.op==val3.op) and (val1.src==val3.src)  val1 = val3
else if( (val1.op==val4.op) and (val1.src==val4.src)  val1 = val4
else if( (val1.op==val5.op) and (val1.src==val5.src)  val1 = val5
else if( (val1.op==val6.op) and (val1.src==val6.src)  val1 = val6

     if( (val2.op==val3.op) and (val2.src==val3.src)  val2 = val3
else if( (val2.op==val4.op) and (val2.src==val4.src)  val2 = val4
else if( (val2.op==val5.op) and (val2.src==val5.src)  val2 = val5
else if( (val2.op==val6.op) and (val2.src==val6.src)  val2 = val6

     if( (val3.op==val4.op) and (val3.src==val4.src)  val3 = val4
else if( (val3.op==val5.op) and (val3.src==val5.src)  val3 = val5
else if( (val3.op==val6.op) and (val3.src==val6.src)  val3 = val6

     if( (val4.op==val5.op) and (val4.src==val5.src)  val4 = val5
else if( (val4.op==val6.op) and (val4.src==val6.src)  val4 = val6

     if( (val5.op==val6.op) and (val5.src==val6.src)  val5 = val6
```

FIGURE 11 – Pseudo-code illustrant la recherche d'égalité entre les différentes variables libres

de savoir si les registres possèdent les bonnes valeurs, d'ailleurs l'abstraction présentée dans cette partie ne les conserve pas, mais de savoir si des registres sont égaux ou pas. Un bit suffit alors pour coder les valeurs des registres. La taille des registres a été une des principales limitation pour les preuves formelles à bases de BDD. Il est intéressant de comparer le temps nécessaire à la preuve de la chaîne de vérification, c'est à dire le temps mis entre l'envoi du code assembleur à un logiciel (voir la section *automatisation*) et l'obtention du résultat de la preuve, avec et sans abstraction. Ceci est fait dans la partie de validation dans la suite de ce rapport.

Limitation de l'abstraction La méthode pour trouver les différentes valeurs n'est pas complète. En effet, bien que possédant le même résultat, les opérations suivantes se verront abstraites avec des valeurs différentes (alors qu'elles ne le devraient pas) car elles possèdent des valeurs sources différentes : $s \leftarrow 2 + 8$; $s \leftarrow 5 + 5$

En conséquence, potentiellement deux codes assembleur équivalents seront considérés non-équivalents. Par contre deux codes non-équivalents ne pourront pas être considérés équivalents. Dans le cas particulier des contre-mesures que nous cherchons à prouver, ce genre de cas n'arrive

pas, ce n'est donc pas un problème. Mais le travail effectué est destiné à être réutilisé [9], et la simplicité de la méthode actuelle pour trouver les différentes valeurs peut s'avérer problématique. La méthode pourrait toutefois être améliorée en enrichissant l'analyse du code nécessaire à la détermination des valeurs à utiliser.

6 De la description MAE au problème SMT

6.1 SMT

La description d'un problème SMT est plus simple que celui d'un problème SAT, et permet ainsi d'exprimer plus facilement des cas plus complexes. Il permet notamment d'exprimer simplement des opérateurs complexes comme la multiplication. La figure 12 représente l'implémentation de la transition de la figure 9 en une formule SMT. La figure 12 et la figure 10 représentent toutes les deux la même transition, pourtant la représentation SMT (ici avec le langage de l'outil *cvc3*) de celle-ci est beaucoup plus simple à comprendre. Nous pouvons constater que pour exprimer une addition, il suffit d'utiliser le mot clé *BVPLUS* du langage SMT de l'outil *cvc3*. La taille des bits est facilement réglable en modifiant un argument lors de la déclaration des variables (vecteurs de bits) SMT utilisées.

```
(faulty.step1.pc = f1) AND
(faulty.step1.flags = faulty.step0.flags) AND
(faulty.step1.rs = BVPLUS(16, faulty.step0.ra, faulty.step0.rb)) AND
(faulty.step1.ra = faulty.step0.ra) AND
(faulty.step1.rb = faulty.step0.rb) AND TRUE;
```

FIGURE 12 – Implémentation de la transition de la figure 9 en une formule SMT

Les opérateurs complexes peuvent facilement être implémentés lors de la description de problèmes SMT. C'est pour cela que dans cette traduction l'abstraction des opérateurs n'a pas été utilisée. Le nombre considérable de variables à décrire pour la mémoire à néanmoins conduit à utiliser une abstraction de la mémoire. À noter que cette abstraction garde la valeur exacte des potentielles cases mémoire utilisées, mais ne représente pas les cases mémoires non utilisées (voir la partie *Modélisation de la mémoire*). L'implémentation de ce traducteur a nécessité une compréhension plus fine des instructions assembleur car les effets exacts sur les registres du processeur, flags, et mémoire ont été codés pour un ensemble d'instructions. Ceci a permis une modélisation plus fidèle des effets des instructions sur les registres, ouvrant ainsi la possibilité de réutilisation de ce traducteur après ce stage, notamment en doctorat.

6.2 Instructions implémentées

Afin de traduire vers le langage d'entrée du SMT-solver, le parseur reconnaît à ce jour 59 instructions assembleurs différentes du jeu d'instruction Thumb2. La plupart sont des instructions arithmétiques ou logiques avec quelques instructions d'accès à la mémoire et de branchement. Elles ont été choisies afin que le jeu d'instruction implémenté puisse modifier les registres du processeur, les flags, le PC et la mémoire. Ce qui permet de prouver formellement l'équivalence de la plupart des contre-mesures existantes dans l'article [13]. Toutes les instructions implémentées sauf celles de branchement peuvent directement être écrites après le mot clé *inst*. Afin de décrire les instructions de branchement dans un fichier MAE, il faut jouer avec les transitions (mot clé *tran*) et les conditions associées aux transitions. Ainsi l'instruction de branchement inconditionnel *"b label"* peut être traduite par une transition de condition *true* vers l'état *label* plutôt que vers l'état suivant l'instruction. Toutes les conditions du jeu d'instruction assembleur sont reconnues. Ainsi pour décrire l'instruction *"b.ne label"* il suffit que la transition vers l'état *label* possède la condition de transition *"cond ne"* et que la transition vers l'instruction suivante possède la condition de transition *"cond eq"*. Cet exemple est illustré avec la figure 13. Elle représente la description d'un code assembleur utilisant l'instruction *b.ne*. Celui-ci se trouve en commentaire dans les première lignes de l'image. À remarquer que si une instruction de branchement est mal modélisée, par exemple si les conditions utilisées pour décrire l'instruction de branchement ne sont pas complémentaires (*ne* et *eq* dans l'exemple), aucune erreur ni avertissement ne sera donné par le parseur permettant de traduire la description vers une modélisation SAT ou SMT. Ce dernier point sera amélioré.

6.3 Modélisation de la mémoire

Le parseur permettant de traduire une description des automates avec le langage MAE vers une modélisation SMT permet de conserver les effets exacts des instructions sur les différents registres du processeur utilisés, ainsi que les flags et la mémoire. Malheureusement le nombre considérable de variables à décrire pour la mémoire à conduit à utiliser une abstraction de la mémoire. Cette dernière permet de réduire le nombre de cases mémoire modélisées tout en conservant les valeurs exactes dans celles-ci. La suite développe l'abstraction de la mémoire utilisée pour la modélisation SMT sans abstraction.

6.3.1 Problématique

Nous cherchons à prouver l'équivalence entre deux automates représentant deux suite d'instructions assembleur. Pour ce faire, nous vérifions que pour un même état initial les deux

```
# e0:     b.ne fin
# e1:     add rs, ra, rb
# fin:

graph exemple
     state initial e0
     state default e1
     state final   fin

     steps = 2
     fault = none

     tran e0 -> e1
          cond eq
          inst nop

     tran e0 -> fin
          cond ne
          inst nop

     tran e1 -> fin
          cond true
          inst add rs, ra, rb
```

FIGURE 13 – Exemple de description d'une instruction de branchement conditionnel

automates possèdent le même effet sur un ensemble de registres. Autrement dit, nous voulons que dans tous les cas, pour un même état initial et lorsque les deux automates ont atteint leur état final, la valeur contenue dans les registres soient identiques. Le terme *registre* englobe les registres du processeur, certains registres spéciaux (les flags), ainsi que la mémoire. Il existe 16 registres du processeur et 4 flags différents. La mémoire peut par contre être très grande (1 milliard de cases différentes par exemple). Le temps de vérification dépend du nombre de variables utilisées dans la description finale du problème à vérifier. Le nombre important de cases mémoire représente donc un problème. Dans la suite, une solution afin de limiter le nombre de cases mémoires à représenter est développée.

6.3.2 Solution

Nombre de cases mémoire Soit un arbre d'exécution représentant les différentes possibilités d'exécution d'un automate pour un nombre de pas donné. Appelons cet arbre l'arbre d'exécution. Il y au plus autant de cases mémoires différentes accédées que d'accès mémoires dans cet arbre. Soit NA ce nombre pour un automate nommé graphA, et NB pour un automate nommé graphB. Nous cherchons à prouver l'équivalence de ces deux automates. Les adresses utilisées pour l'accès à la mémoire dans la modélisation de ces deux automates sont contenues dans des registres du processeur potentiellement représentés par des variables libres ou par des variables dont la valeur dépend de variables libres. Autrement dit, les adresses sont potentiellement autre chose que des valeurs fixes dans le code assembleur. Si c'est le cas, il n'est donc pas possible de connaître les adresses utilisées lors de la description du problème et donc le nombre de cases différentes. Néanmoins, l'arbre représentant les différentes possibilités d'exécution fournit une borne maximale $N = NA + NB$. Au plus, il y a N cases différentes accédées. La solution proposée représente une mémoire avec N cases. Au même titre que les registres du processeur et les flags, cette mémoire est représentée dans la description des deux automates. Les valeurs contenues dans les cases mémoires sont prises en compte dans la preuve d'équivalence des deux automates, et sont donc comparées dans la propriété finale.

Accès aux cases mémoires Si nous avions décidé de représenter toute la mémoire, l'accès à celle-ci serait facile (modulo la taille variables libres qui doit être suffisamment grande pour pouvoir adresser les différentes cases mémoire). Le tableau représentant la mémoire contiendrait autant de cases que la mémoire réelle, il suffirait alors de passer en paramètre au tableau représentant la mémoire une adresse afin d'obtenir le vecteur de bits contenue dans la case mémoire correspondante : *case mémoire = tableau[adresse]*
La taille imposante de la mémoire étant un problème, seul un nombre limité de cases est représenté : il y a autant de cases que d'accès mémoire dans l'arbre d'exécution. L'adresse ne peut alors plus être utilisée comme indice du tableau afin d'obtenir la case mémoire. Un problème se pose donc : comment adresser le tableau ? En sachant que celui-ci ne représente qu'un sous-ensemble de la mémoire, et que l'adresse des cases représentée n'est pas connue. Nous connaissons néanmoins le nombre maximal de cases adressées ainsi que les variables représentant les registres à un pas donné d'exécution contenant les adresses. Un début de solution consiste à faire chaque accès mémoire dans une case mémoire différente. Ainsi deux adresses différentes contenues dans deux variables différentes accéderont deux cases différentes. Malheureusement deux adresses identiques dans deux variables différentes accéderont à deux cases différentes aussi. Une solution plus complète consiste à donner à chacune de ces variables un indice du tableau et imposer

une priorité d'accès des cases de telle sorte que deux adresses identiques dans deux variables différentes accèdent la même case. Ci-dessous la fonction *memoryAccess* implémente cette idée, elle prend en entrée une adresse et renvoie en sortie la case mémoire. Plus précisément cette fonction renvoi la variable représentant la case mémoire adressée. Si nous voulons lire une case mémoire et stoker sa valeur dans un registre, il suffit d'affecter la variable représentant le registre avec la sortie de cette fonction :

$reg = MemoryAccess(address)$

Si nous voulons écrire une case mémoire avec la valeur contenue dans un registre, il suffit d'affecter la variable reenvoyée par la fonction avec la variable représentant le registre :

$MemoryAccess(address) = reg$

Listing 4 –

```
Case MemoryAccess( address )
{

        if(address==var0)   return tableau[0]
        else if(address==var1)   return tableau[1]
        else if(address==var2)   return tableau[2]
        ...
        else if(address==varN-1) return tableau[N-1]
}
```

Dans ce listing, *var0, var1, ... varN-1* correspondent aux variables contenant l'adresse lors des accès mémoire dans l'arbre d'exécution. Les différentes valeurs que peut prendre le paramètre *address* sont celles contenues dans ces variables. À chaque variable, un indice est associé, et donc une case mémoire. Ainsi l'indice K est associé à la variable *varK*, et donc la case mémoire *mem[K]* sera renvoyée lorsque cette variable sera passée en paramètre de la fonction. Soit $var1 = var2$. Avec cette fonction, lorsque *var1* sera passé en paramètre, *mem[1]* sera renvoyé. Lorsque *var2* sera passée en paramètre, *mem[1]* sera renvoyé. De cette manière, deux adresses différentes contenues dans deux variables différentes accéderont à deux cases différentes. Deux adresses identiques dans deux variables différentes accéderont à la même case.

Avec la solution proposée le nombre de cases mémoire représentées dans la modélisation

a été limité au nombre d'accès à la mémoire dans l'arbre d'exécution, tout en conservant les valeurs exactes des cases mémoires utilisées. Toutefois, afin de ne pas modifier le sens du code, la taille des registres en nombre de bits doit être suffisante pour exprimer les N potentielles cases différentes accédées, soit être au moins $log_2(N)$ bits.

6.4 Optimisation de la vérification

L'équivalence entre deux automates peut rapidement être prouvée par un SAT/SMT solver pour des petits automates, et donc pour deux petits codes assembleurs. Plus la contre-mesure d'une instruction assembleur comporte d'instructions, plus le temps de vérification augmente. Le temps de vérification peut être alors de quelques secondes ou de plusieurs heures. Parfois, le problème est si complexe que le solver alloue une grande quantité de mémoire (supérieur à 4Go) et le système d'exploitation finit par arrêter le processus en question. Dans ce cas, la réponse au problème d'équivalence n'est jamais donnée par le solver. Cette situation a notamment été rencontrée pour la contre-mesure de l'instruction *adcs* trouvé lors de ce stage. Cette dernière possède vingt instructions et le SMT-solver utilisé est cvc3 pour une taille de registre de 4 bits dans un ordinateur avec 4Go de mémoire. Le temps de vérification est rarement un problème dans le cadre de ce stage, car les problèmes prennent rarement plus d'une heure à résoudre. Mais ceci peut s'avérer contraignant pour des contre-mesures cherchant à protéger un bloc d'instructions.

Une solution a été proposée afin de limiter le temps de vérification. Cette dernière se base sur l'observation suivante : il est plus facile de résoudre plusieurs petits problèmes qu'un seul très grand. Jusqu'à présent, le problème final est exprimée au travers d'une comparaison de tous les registres utilisés des deux automates dans un fichier. Ensuite ce fichier est donné à un solver. Au lieu de cela, l'idée consiste à comparer les registres indépendamment et produire un fichier par comparaison.

Ainsi si nous cherchons à vérifier l'égalité des valeurs de trois registres ra, rb, rc à l'état final de deux automates, nous chercherons d'abord à le vérifier pour le registre ra, puis le registre rb, puis le registre rc, au lieu de directement chercher à le vérifier pour les trois ensemble. Prouver que tous les registres à l'état final de deux automates sont toujours égaux, équivaut à le prouver individuellement pour chaque registre. Si le résultat est identique pour les trois registres alors les deux automates sont équivalents. Si un des registres ne l'est pas alors les deux automates ne sont pas équivalents. Ce principe peut être poussé encore plus loin et être appliqué pour les fautes : un fichier par instruction assembleur fautée, et un autre lorsqu'aucune instruction n'est fautée. Ainsi si nous comparons trois registres avec une contre-mesure de quatre instructions potentiellement fautées, nous obtiendrons $3 * (4 + 1) = 15$ fichiers différents. Le "+1" est parce qu'il faut prendre en compte le cas où aucune instruction est fautée. Tous les registres sont comparés dans des

fichiers différents, pour tous les cas de fautes. Chaque fichier possède la description du problème d'équivalence-checking en sa totalité, seule la propriété finale dans laquelle les registres des deux automates sont comparés ainsi que la génération de la faute sont simplifiés. Outre le potentiel gain de temps qui sera développé par la suite, cette technique permet de mieux visualiser où se trouve une potentielle erreur lors de la conception de la contre-mesure : si les deux automates ne sont pas équivalents nous saurons quels sont les registres concernés, et pour quelles instructions fautées.

La suite suppose que le solver utilisé est le SMT-solver de cvc3 sur un ordinateur moyen à ce jour. En pratique, cette astuce ne permet pas de gagner beaucoup de temps lorsque le temps de vérification est petit (moins d'une seconde). En général le temps de vérification augmente de quelques secondes. Mais ceci n'est pas grave car il n'y a pas une grande différence entre attendre 1 seconde ou 2 secondes (cas de l'instruction add idempotente et sa contre-mesure pour une taille de registres de 256 bits). Par contre il y a un gain considérable lorsque le temps de vérification est à la base très grand. Dans le cas de l'instruction séparable *umlal* et sa contre-mesure avec une taille de registre de 7 bits, elle a permis de passer de 1759 secondes (29 minutes et 19 secondes) à 538 secondes (8 minutes et 58 secondes). La vérification a été environ trois fois plus rapide dans ce cas. Dans le cas de l'instruction spécifique *adcs* et de la contre-mesure trouvée lors de ce stage pour une taille de registres de 5 bits, cette astuce a permis de résoudre le problème en 164 secondes (2 minutes et 44 secondes) alors que précédemment le SMT-solver ne donnait pas de réponse car le processus était arrêté par le système d'exploitation à cause d'une consommation beaucoup trop importante de mémoire. Malheureusement parfois, la tendance est inverse.

C'est notamment le cas de la contre-mesure de l'instruction *adcs* de l'article [13]. Pour ce dernier la vérification prend quelques secondes lorsque les registres sont comparés dans le même fichier pour une taille de registre de 32 bits, et plusieurs heures lorsque les registres sont comparés dans des fichiers indépendants.

Un solver prend en entrée une formule logique puis cherche une configuration des variables rendant la formule vrai. Si c'est le cas il répond *SATISFIABLE*. Autrement dit un solver arrête de calculer dès qu'il trouve une solution. Il est donc plus facile pour un solver de répondre *SATISFIABLE* que de répondre *UNSATISFIABLE* car dans ce dernier cas il doit continuer les calculs afin de prouver qu'il n'existe aucune configuration des variables permettant de rendre la formule vraie. La contre-mesure proposé dans l'article [13] n'est pas tolérante à la faute "saut d'une instruction assembleur" lorsque nous cherchons à vérifier l'égalité des flags à l'état final des deux automates. Le solver répond alors *SATISFIABLE* pour ce problème d'équivalence-checking. Lorsque tous les registres sont comparés dans le même fichier, le solver trouve rapidement une configuration des variables de la formule logique permettant de satisfaire l'inégalité d'au moins

un registre (dans notre cas les flags) à l'état final des deux automates. Lorsque tous les cas sont énumérés dans différents fichiers, le solver doit d'abord prouver qu'il n'existe aucune configuration des variables logiques permettant une inégalité des registres pour un certain nombre de fichiers avant de se confronter au fichier contenant la comparaison des flags avec la faute permettant de rendre la formule *SATISFIABLE*. En d'autres termes, lorsque les registres sont comparés dans des fichiers indépendants le solver doit d'abord résoudre des problèmes difficiles avant de se confronter au problème facile permettant d'arrêter les calculs, tandis que lorsque les registres sont comparés dans le même fichier il se confronte directement au problème facile. Pour cette raison la méthode consistant à découper le problème dans des fichiers indépendants n'est pas performante pour la preuve d'équivalence-checking de l'instruction *adcs* avec la contre-mesure proposée dans l'article [13]. De manière plus générale cette méthode est moins performante pour les problèmes d'équivalence-checking renvoyant *SATISFIABLE*, lorsque deux codes assembleurs ne sont pas équivalants. Mais elle permet de gagner un temps significatif pour les problèmes "difficiles", c'est à dire lorsque les deux codes assembleurs sont équivalants.

Nous avons vu que cette technique permet de mieux visualiser une potentielle erreur lors de la conception de la contre-mesure, et de gagner un temps important lors de la vérification de problèmes pour des codes conséquents étant équivalants. La méthode consistant à comparer les registres dans des fichiers indépendants ainsi que la méthode consistant à tous les comparer dans le même fichier ont toutes les deux été implémentées. Utiliser une stratégie ou l'autre est un choix de l'utilisateur qui doit passer un argument au logiciel implémenté afin de sélectionner laquelle utiliser. La méthode consistant à comparer les registres dans des fichiers indépendants est plus performante pour les problèmes *UNSATISFIABLE* et celle consistant à tous les comparer dans le même fichier pour les problèmes *SATISFIALE*. Comme nous ne savons pas à l'avance ce que le solver répondra, il est judicieux de lancer la résolution du problème avec les deux méthodes en parallèle.

7 Résultats et analyse détaillés des expérimentations et des tests de validation

Un des buts principaux de ce stage est de proposer une modélisation SAT/SMT afin d'améliorer le temps de vérification de contre-mesures logicielles. Dans cette partie, nous présentons la validité et les performances des différentes modélisations proposées afin de déduire quelle solution retenir.

7.1 Vérification de la correction des modélisations

Dans un premier temps, le résultat de l'équivalence-checking entre chaque instruction du jeu de test et les contre-mesures correspondantes est vérifié. Les contre-mesures proposées dans l'article [13] ont déjà été prouvées avec des preuves formelles à base de BDD. Les résultats de l'équivalence-checking obtenus à l'aide d'un SAT-solver ou un SMT-solver sont comparés aux résultats obtenus avec les preuves formelles à base de BDD.

La figure 14 récapitule les instructions vérifiées pour les différentes modélisations, ainsi que le nombre de bits utilisés pour coder les registres conformément au jeu de test défini dans la phase de spécification. Les cases vertes indiquent que l'instruction correspondante a été modélisée et vérifiée, les cases rouges indiquent le contraire. Le parseur du langage MAE permet de modéliser simplement des problèmes d'équivalence-checking pour les modélisations SAT avec abstraction des opérateurs et SMT sans abstraction des opérateurs. Cette automatisation a permis de tester facilement toutes les instructions décrites dans la figure 14 avec différentes tailles de registres pour ces modélisations. Seule l'instruction idempotente *add* a été vérifiée pour les modélisations SAT sans abstraction des opérateurs et SMT avec abstractions des opérateurs, comme proposé en phase de spécification car leur description en formule logique est manuelle. Pour une mesure des performances en temps exhaustive des différentes modélisations, il aurait été intéressant d'automatiser toutes les modélisations. Ceci aurait permis de rapidement décrire toutes les modélisations en passant par le langage MAE. Malheureusement le temps n'a pas permis d'implémenter les différents traducteurs permettant de transformer une description MAE vers toutes les modélisations présentées (l'automatisation de toutes les modélisations n'était pas à faire conformément à la spécification). Mais afin pouvoir comparer les performances des différentes modélisations, le problèmes SAT sans abstraction et SMT avec abstraction ont manuellement été écrits. Autrement dit, la description avec le langage MAE n'as pas été utilisée pour ces deux modélisations.

Les différentes preuves d'équivalence-checking effectuées pour les instructions de la figure 14 ont correctement donné la même réponse que les preuves d'équivalence-checking à base de BDD de l'article [13]. Le problème d'équivalence-checking a donc correctement été modélisé et implémenté pour les modélisations SAT et SMT avec ou sans abstraction des opérateurs. La suite développe les performances en temps des différentes modélisations.

			SAT	SAT avec abstraction	SMT	SMT avec abstraction
			Manuel	Automatique	Automatique	Manuel
Instructions	Idempotent	Add rs, ra, rb				
		Mul rs, ra, rb				
	Séparable	Add rs, rs, rb				
		Umlal rlo, rhi, rn, rm				
	Spécifique	Adcs rs, ra, rb				
		B <label>				
Taille des registres			4, 8, 16, 32	4, 8, 16, 32	4, 8, 16, 32	4, 8, 16, 32

FIGURE 14 – Récapitulatif des instructions vérifiées pour les différentes modélisations

7.2 Mesure et analyse du temps de vérification

7.2.1 Modélisations

L'approche utilisée dans l'article [13] est une vérification formelle à base de BDD. Malheureusement le temps de vérification avec cette méthode explose rapidement lorsque la taille des registres augmente et lorsque le nombre d'instructions assembleur augmente. Afin de prouver l'équivalence entre une instruction assembleur et sa contre-mesure il n'est pas nécessaire dans la plupart des cas d'avoir une taille de registre de 32 bits, 4 suffisent. Mais la lenteur de la vérification lorsque le nombre d'instructions augmente rend cette méthode inutilisable pour des granularités plus élevées (blocs d'instructions).

Lors de ce stage une modélisation SAT et SMT ainsi qu'une abstraction des opérateurs ont été proposées. Chaque modélisation pouvant utiliser l'abstraction ou pas. Ce qui fait en total quatre modélisations possibles :

- SAT avec ou sans abstraction
- SMT avec ou sans abstraction

Un des grands avantages du SMT par rapport au SAT est la facilité d'expression des problèmes à vérifier. Ce dernier nous permet notamment d'exprimer des opérateurs complexes telle que l'addition ou la multiplication. Cela permet de gagner un temps considérable lors de la description du problème. Pour des contre-mesures visant à protéger des granularités plus élevées, écrire directement les formules SAT sans abstraction n'est pas envisageable. Une automatisation plus complète à travers un langage simple comme celui présenté dans ce rapport permettrait de ren-

dre l'utilisation du SAT accessible. Mais cette automatisation prendrait quelques mois de travail, plus encore que celui ayant été nécessaire à l'automatisation du SMT. Mais celle-ci s'imposerait si les performances du SAT s'avèrent bien meilleures que celles du SMT. Ceci sera discuté par la suite.

Nous avons vu que l'abstraction des opérateurs consiste à remplacer le résultat d'une opération par une variable libre. Ceci permet de simplifier l'expression des problèmes car les opérateurs ne sont plus représentés. Mais la recherche d'égalité entre les différentes variables libres impose l'ajout de nouvelles contraintes au problème, ce qui le complique. Il est alors difficile de prévoir si le temps de vérification est diminué. Le grand avantage avec l'abstraction est qu'elle permet de réduire le nombre de bits des registres utilisé à 1 bit. L'inconvénient est qu'à ce jour, l'implémentation de l'abstraction n'est pas suffisamment riche pour permettre de se baser directement la dessus pour de futurs travaux, des améliorations sont à prévoir. Elle est néanmoins suffisante pour les objectifs du stage.

7.2.2 Conditions d'expérimentation

Les expériences ont été effectuées sur un ordinateur avec un processeur *Intel Core i3-3120M CPU 2.50GHz * 4* et 4Go de mémoire RAM. La plupart des mesures ont pu être effectuées, mais certaines n'ont pas abouti. Pour les problèmes d'équivalence-checking que nous cherchons à résoudre dans le cadre de ce stage, un temps de vérification supérieur à 24h est considéré trop important. Ainsi, lorsque que le temps de vérification de la contre-mesure d'une instruction dépassait les 24h, l'expérience était abandonnée. Dans certains cas, le système d'exploitation (ici Ubuntu 12.04 LTS) arrête le processus car sa consommation de mémoire est trop importante (supérieur à 4Go).

7.2.3 Différentes approches de preuves MC BDD, SAT, SMT

Cette partie compare les performances des différentes modélisations étudiées. Dans un premiers temps seule l'instruction d'addition est utilisée pour la comparaison. Les différentes instructions du jeu de test sont utilisées dans la suite. Les modélisations n'ayant pas été automatisées ont été écrites manuellement pour cette instruction.

Les tableaux 15 et 16 suivant récapitulent les résultats des différents tests effectués pour l'instruction *add*. Les temps exprimés sont en secondes. Le symbole "< 1" indique que le temps de vérification est inférieur à une seconde, et le symbole "*memory overflow*" signifie que le test a été coupé par le système d'exploitation dû à une trop grande consommation mémoire par le processus. A noter que pour les modélisations avec abstraction des opérateurs le même fichier de description est utilisé pour les différentes tailles de registres car cette abstraction permet de

réduire le nombre de bits des registres utilisés à 1 bit.

Le tableau 15 récapitule le temps de vérification pour les différentes modélisations. Nous pouvons remarquer que la modélisation SAT et SMT mettent moins de temps pour prouver l'équivalence, pour toutes les tailles de registre testées. Ce qui n'est pas le cas pour la preuve à base de BDD : le temps de vérification reste raisonnable pour une taille de 4, 8 et 16 bits, mais le système d'exploitation coupe la vérification pour une taille des registres de 32 bits au bout de 386 secondes sur l'ordinateur testé.

	Abstraction	4 bits	8 bits	16 bits	32 bits
BDD	Sans	<1	<1	3	memory overflow
SAT	Avec	<1	<1	<1	<1
	Sans	< 1	< 1	< 1	< 1
SMT	Avec	< 1	< 1	< 1	< 1
	Sans	< 1	< 1	< 1	< 1

FIGURE 15 – temps de vérification pour l'instruction idempotente *add*

Le premier test n'est pas complet. Le temps mis par les différentes briques logicielles a été négligé. Le tableau 16 récapitule le temps total mis par chaque test afin d'avoir le résultat de la vérification, en prenant en compte le temps de traitement des briques logicielles utilisées. Plus précisément, le temps mis pour transformer un code MAE en un code SMT est négligeable. Par contre le temps varie de manière significative pour la partie SAT. En pratique le temps de vérification de la partie SAT est rapide, la partie la plus gourmande en temps est la transformation d'un code en logique propositionnelle sous sa forme CNF avec le logiciel *pbl*. Le tableau suivant récapitule le temps total mis dans les différents cas en prenant en compte le temps de traitement des différentes briques logicielles de l'automatisation, notamment le logiciel *pbl* pour la partie SAT.

Les tests effectués nous permettent déjà d'éliminer la modélisation à base de BBD et la modélisation SAT sans abstraction à cause de leur lenteur. Le temps mis par cette dernière peut néanmoins être amélioré en remplaçant le logiciel *pbl* par un logiciel plus performant. Mais pour l'instant la modélisation SMT sans abstraction nous permet d'exprimer la même chose plus facilement, tout en ayant un temps de vérification réduit.

	Abstraction	4 bits	8 bits	16 bits	32 bits
SAT	Avec	2	2	2	2
	Sans	11	44	246	1514
SMT	Avec	< 1	< 1	< 1	< 1
	Sans	< 1	< 1	< 1	< 1

FIGURE 16 – temps total utilisé par les différentes briques logicielles (solver inclus) pour l'instruction idempotente *add*

7.3 Mesure et analyse du coût de l'abstraction

Nous avons précédemment observé que le temps de vérification d'un problème SAT avec abstraction est négligeable par rapport au temps mis par la brique logicielle permettant de transformer une formule propositionnelle sous sa forme *CNF*. Ceci est une conséquence de la stratégie utilisée pour trouver les variables libres permettant d'abstraire les opérateurs. Cette dernière alourdie considérablement la formule en logique propositionnelle (voir la partie Abstraction). Autrement dit, la partie la plus gourmande en temps est la transformation d'un code en logique propositionnelle sous sa forme CNF avec le logiciel *pbl*.

Avec la méthode permettant de déterminer les variables libres à utiliser, la taille des formules propositionnelles augmente de manière polynomiale en fonction du nombre d'opérations abstraites par des variables libres. La figure 17 représente le temps mis par le logiciel *pbl* pour transformer une formule propositionnelle sous sa forme *CNF*. Les deux codes assembleur utilisés pour la preuve d'équivalence-checking de ce test sont l'instruction d'addition idempotente *add* modélisé par le premier automate, et $N - 1$ instructions d'addition idempotente *add* modélisées par le deuxième automate. Ce qui fait un total de N instructions d'additions. Dans ce cas l'opération de toutes les instructions est remplacée par une variable libre.

Les mesures ont été effectuées sur un ordinateur avec un processeur *Intel Core i3-3120M CPU 2.50GHz * 4* et 4Go de mémoire RAM. Pour un nombre limité de variables libres (et donc d'opérations), le temps mis par la transformation est faible. Par exemple *pbl* a mis un peu moins de cinq minutes pour effectuer la transformation avec dix variables libres. Malheureusement le temps de calcul augmente très rapidement. En effet, *pbl* met environs deux heures pour effectuer la transformation avec vingt variables libres. Malgré la rapide croissance du temps de calcul de la transformation d'une formule propositionnelle en sous sa forme *CNF*, cette méthode permet de vérifier des blocs d'instructions possédant une vingtaine d'instructions en un temps raisonnable (moins de deux heures).

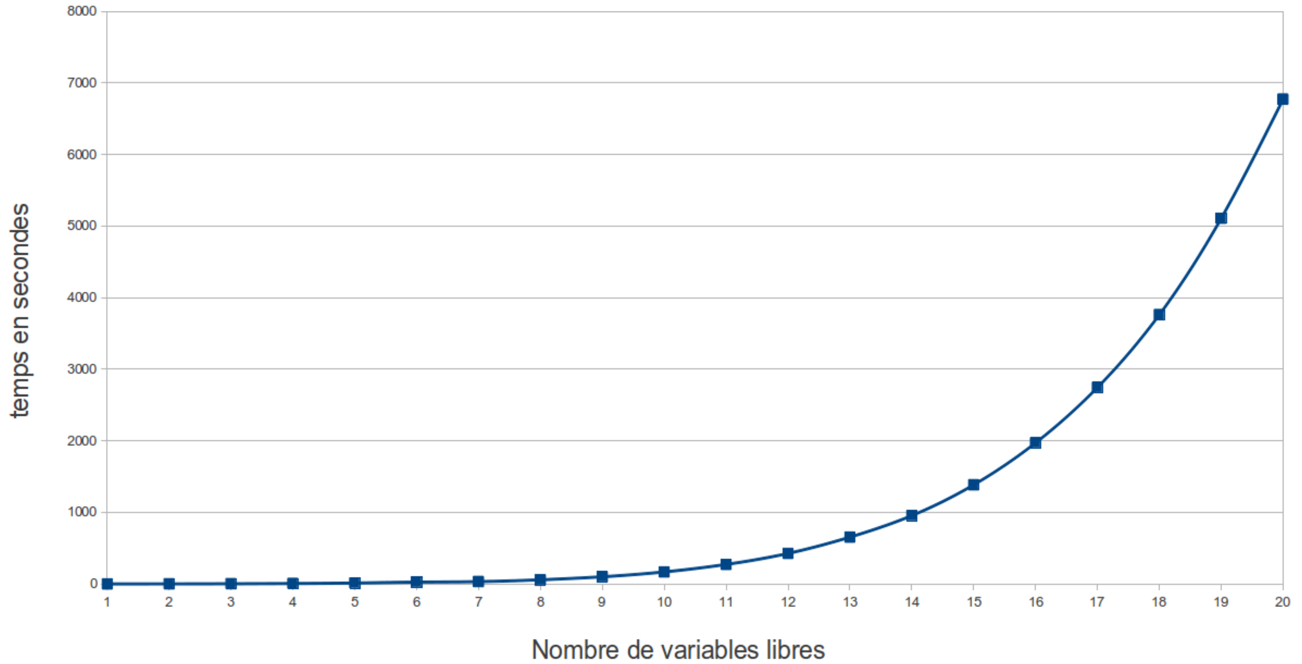

FIGURE 17 – Performances de la transformation $lp \rightarrow CNF$ avec l'outil pbl

7.4 Comparaison des approches SAT avec abstraction et SMT sans abstraction

Dans la suite la comparaison des approches SAT avec abstraction et SMT sans abstraction est développée. Le temps n'a pas permis de développer l'automatisation de la partie SMT avec abstraction et SAT sans abstraction. Mais les modélisations automatisées englobent la modélisation SAT et SMT ainsi que l'utilisation de l'abstraction des opérateurs. Le tableau 3 rappelle les différentes instructions proposées comme jeu de test lors de la partie de spécification. Afin d'englober les cas simples comme les cas complexes, deux instructions de chaque classe ont été choisies.

Le tableau 18 récapitule les performances en temps obtenus par la preuve d'équivalence-checking des différentes instructions du jeu de test rappelées dans le tableau 3. Pour toutes les instructions testées, la partie consommatrice en temps pour le SMT est la vérification formelle avec le SMT-solver, alors que pour le SAT c'est la transformation de la formule en logique propositionnelle sous sa forme CNF. Les autres briques logicielles prennent dans tous les cas moins d'une seconde pour effectuer leurs tâches. Dans le tableau 18, le temps mis par chaque brique logicielle de l'automatisation n'est pas marqué afin de ne pas surcharger la présentation des résultats. Le temps donné est le temps total, c'est à dire le temps mis pour obtenir le résultat

classe	instruction	action
idempotent	*add rs, ra, rb*	$rs \leftarrow ra + rb$
	mul rs, ra, rb	$rs \leftarrow ra * rb$
séparable	*add rs, rs, rb*	$rs \leftarrow rs + rb$
	umlal rlo, rhi, rn, rm	$rhi : rlo \leftarrow rn * rm + rhi : rlo$ [8]
spécifique	*adcs rs, ra, rb*	$rs \leftarrow ra + rb + flags.carry$ *mis à jour des flags*
	b <label>	$PC \leftarrow$ adresse représentée par le label

TABLE 3 – Instructions utilisées comme jeu de test

final (SAT/UNSAT) à partir du fichier de description d'automates MAE.

Le symbole "*time overflow*" indique que les résultats sont absents, car le temps de test à été beaucoup trop long (plus d'une journée), celui-ci à donc été abandonné. Nous pouvons observer que dans la plupart des cas testés, la modélisation SMT est plus rapide. Sauf pour l'instruction *umlal*. La contre-mesure de cette instruction est la plus longue en nombre d'instructions, et fait appel à des opérations plus complexes que les autres contre-mesures. Pour des registres de petite taille la modélisation SMT sans abstraction est la plus performante, mais elle se fait rapidement rattraper par la modélisation SAT avec abstraction. Cette instruction nous permet de voir les limitations de la modélisation sans abstraction : le temps de vérification augmente de manière significative lorsque la taille des registres augmente, notamment pour des contre-mesures complexes.

8 Conclusion

L'article [13] propose plusieurs contre-mesures tolérantes aux faute de type "saut d'une instruction assembleur". Ces contre-mesures ont formellement été prouvées tolérantes à ce type de faute à l'aide de preuve formelle à base de BDD. Malheureusement le temps de vérification de cette solution s'accroît considérablement lorsque le nombre d'instructions de la contre-mesure augmente, lorsque les opérateurs utilisés deviennent plus complexes, et que la taille des registres en nombre de bits augmente. Afin de permette d'étudier des contre-mesures plus complexes et avec un nombre d'instructions plus élevé [9], un des principaux buts de ce stage est d'améliorer les performances en temps de calculs de la vérification d'équivalence-checking. Pour ce faire, d'autre méthodes de vérification formelle ont été explorées, notamment la vérification formelle de problèmes SAT et SMT. Les problèmes SAT sont difficiles à modéliser pour notre

Classes d'instructions	Instructions		4 bits	8 bits	16 bits	32 bits
idempotent	add	SAT avec abstraction	2	2	2	2
		SMT sans abstraction	<1	<1	<1	<1
	mul	SAT avec abstraction	2	2	2	2
		SMT sans abstraction	<1	<1	<1	<1
séparable	add	SAT avec abstraction	5	5	5	5
		SMT sans abstraction	<1	<1	<1	<1
	umlal	SAT avec abstraction	56	56	56	56
		SMT sans abstraction	3	6220	time overflow	time overflow
spécifique	adcs	SAT avec abstraction	12	12	12	12
		SMT sans abstraction	<1	<1	1	3
	b	SAT avec abstraction	8	8	8	8
		SMT sans abstraction	<1	<1	<1	<1

FIGURE 18 – temps total

problème d'équivalence-checking, car les opérateurs arithmétiques comme l'addition sont lourds à exprimer. Afin de faciliter la modélisation pour ce problème une abstraction des opérateurs a été implémentée. Quatre modélisations ont donc été proposées, une modélisation SAT avec et sans abstraction des opérateurs, et une modélisation SMT avec et sans abstraction des opérateurs. Ces différentes modélisations ont été validées selon le schéma proposé lors de la phase de spécification. D'abord, la performance en temps de ces quatre modélisations a été étudiée pour le problème d'équivalence-checking de l'instruction idempotente *add*. Lors de ce test, nous avons pu constater que toutes les modélisations donnent de meilleures performances que la preuve à base de

BDD lorsque la taille des registres augmente. Parmi les modélisations proposées, la modélisation SAT sans abstraction est celle qui donne les moins bonnes performances pour cette instruction. Afin de mieux comparer les différentes modélisations, deux des quatre modélisations ont été testées avec les problèmes d'équivalence-checking des différentes instructions du jeu de test proposées lors de la phase de spécification. Les modélisations testées avec le jeu de tests sont la modélisation SAT avec abstraction et la modélisation SMT sans abstraction. La modélisation et la preuve de ces dernières ont été automatisées, les problèmes d'équivalence-checking peuvent alors facilement être décrits avec le langage MAE. Les mesures du temps total mis par les différentes briques logicielles montrent que la modélisation SMT sans abstraction est efficace et donne de meilleurs résultats que la modélisation SAT sans abstraction pour la plupart des instructions. Malheureusement elle donne de moins bon résultats lorsque la taille des registres augmente pour des contre-mesures complexes.

Pour toutes les instructions étudiées, le résultat de la vérification ne varie pas en fonction de la taille des registres. Dans ce cas, quatre bits suffisent pour modéliser nos problèmes d'équivalence-checking. La modélisation SMT sans abstraction reste efficace pour une petite taille de registres, et peut être utilisée pour résoudre le problème d'équivalence-checking de contre-mesures plus complexe lors de futurs travaux. Dans les cas où le résultat de la vérification varie en fonction de la taille des registres, la modélisation SMT sans abstraction n'est plus efficace, la modélisation SAT avec abstraction peut être utilisée. Si les contre-mesures étudiées après ce stage restent au niveau d'un bloc d'instructions et que la taille des registres est de quelques bits, la modélisation SMT suffit. Dans le cas contraire, une étude plus fine de l'abstraction est une solution afin d'améliorer les performance en temps de la vérification et ainsi permettre de résoudre en un temps raisonnable (moins de 24h) les problèmes d'équivalence-checking plus complexes.

Jusqu'à présent la plupart des instructions du jeu instructions Thumb2 de ARM possèdent une séquence de remplacement tolérante au type de faute "saut d'une instruction assembleur. D'autres perspectives sont envisageables. Dans le but de renforcer la sécurité d'un programme il serait intéressant d'étudier d'autres modèles de fautes comme le "remplacement d'une instruction assembleur". Ainsi que de changer la granularité d'étude du code assembleur à protéger, c'est à dire proposer des séquences de remplacement pour un bloc d'instructions. Et enfin intégrer les différentes contre-mesures dans une chaîne de compilation afin d'automatiser la protection d'un programme contre différents types de fautes provoquées par les attaques physiques.

9 Annexes

9.1 Liste des différentes instructions assembleur reconnues

Dans cette partie la liste des différentes instructions assembleurs reconnues par le parseur permettant de traduire une description en automates MAE vers une formule SMT est développée. Mais cette partie ne décris par les instructions. L'équivalence de chacune de ces instructions avec la contre-mesure associée a formelle été prouvée avec le SMT-solver cvc3 ou cvc4. Dans la plupart des cas, le temps de vérification était inférieur à une heure pour une taille de registres de 32 bits.

9.1.1 Récapitulatif des différentes instructions

La liste ci-dessous est un récapitulatif des 59 instructions assembleurs implémentées. Il existe différentes façons de les utiliser. Les formats d'écriture reconnus sont développés dans la partie suivante.

add, adds, adc, adcs, mul, muls, umull, umulls, umlal, umlals, sub, subs, sdiv, udiv, mov, movs, mvn, mvns, mrs, msr, lsl, lsls, lsr, lsrs, asr, asrs, ror, rors, rrx, rrxs, and, ands, eor, eors, orr, orrs, orn, orns, bic, bics, cmp, cmn, tst, teq, ldr, str, ldmia, ldmib, ldmda, ldmdb, stmia, stmib, stmda, stmdb, pop, push, nop, b, bl

Chaque instruction possède un effet différent sur les registres utilisées par le programme (registres du processeurs, pc, flags, mémoire). Cette diversité a imposé l'écriture d'une fonction en langage C spécifique à chaque instruction, ce qui a considérablement contribué à la difficulté de développement. Afin de représenter les effets exactes des ces instructions sur les différents registres, une compréhension fine des instructions du jeu d'instruction Thumb2 de ARM a été nécessaire.

9.1.2 Récapitulatif des différentes formats d'instructions

Format registre/immédiat/décalage

Soit l'instruction d'addition avec le format suivant : *add rs, ra,* ***rid***

L'opérande *rid*[9] peut être soit un registre, soit une valeur immédiate, soit une valeur immédiate avec un décalage préalable. La liste ci-dessous énumère les différentes formes que l'opérande *rid* peut prendre. L'instruction *add* est pris comme exemple.

9. rid = Registre / Immédiat / Décalage

add rs, ra, rb
add rs, ra, imm
add rs, ra, lsl, rc
add rs, ra, lsr, rc
add rs, ra, asr, rc
add rs, ra, lsl, imm
add rs, ra, lsr, imm
add rs, ra, asr, imm
add rs, ra, ror, imm
add rs, ra, rrx

Les valeurs immédiates sont représentées par le symbole *imm*, les registres par les symboles *rs, ra, rb, rc* et les instructions de décalage par les symboles *lsl, lsrn asr, lsl, lsr, asr, ror, rrx.* La liste ci-dessous énumère les instructions avec ce format d'écriture.

add, adds, adc, adcs, sub, subs, mov, movs, mvn, mvns, and, ands, eor, eors, orr, orrs, orn, orns, bic, bics, cmp, cmn, tst, teq

Format des instructions d'accès à la mémoire
Soit l'instruction de lecture de la mémoire *ldmia*. Cette instruction peut lire plusieurs cases mémoire à partir de l'adresse dans le registre *rs* et modifier la valeur contenue dans celui-ci. Voici son format d'écriture :

ldmia! rs, { *ra, rb,* ... }

Dans ce cas cette instruction lit deux cases mémoires consécutives et stocke la valeur dans les registres *ra* et *rb*. Les "..." signifient que le nombre de cases mémoires accédés peut varier. Le " !" est optionnel, s'il est présent ceci signifie que le registre *rs* sera modifié. Voici la liste des instruction respectant ce format.

ldmia, ldmib, ldmda, ldmdb, stmia, stmib, stmda, stmdb, pop, push

Format des instructions de branchement Les instructions de branchement peuvent être conditionnelles, voici les différentes conditions qu'elles peuvent supporter :

eq, ne, cs, hs, cc, lo, mi, pl, vs, vc, hi, ls, ge, lt, gt, le, true, false

Les deux instructions de branchement implémentées sont *b* et *bl*.

Autres formats Les instructions de la liste suivante possèdent un format autre que ceux précédemment présentées. La liste inclus les différents formats.

nop
mul rs, ra, rb
muls rs, ra, rb
sdiv rs, ra, rb
udiv rs, ra, rb

umull rlo, rhi, ra, rb
umulls rlo, rhi, ra, rb
umlal rlo, rhi, ra, rb
umlals rlo, rhi, ra, rb

mrs rs, apsr
msr apsr, ra
msr apsr, imm

lsl rs, ra, rb
lsls rs, ra, rb

lsl rs, ra, imm
lsls rs, ra, imm

lsr rs, ra, rb
lsrs rs, ra, rb

lsr rs, ra, imm
lsrs rs, ra, imm

asr rs, ra, rb

asrs rs, ra, rb

asr rs, ra, imm
asrs rs, ra, imm

ror rs, ra, imm
rors rs, ra, imm

rrx rs, ra
rrxs rs, ra

ldr rs, [ra]
str rs, [ra]

Le registre *apsr* utilisés avec les instructions *msr* et *mrs* est un registre spécial du processeur. Ces instructions doivent être utilisées avec ce registre.

Références

[1] http ://formal.cs.utah.edu :8080/pbl/index.php.

[2] http ://vlsi.colorado.edu/ vis/.

[3] http ://www.cs.nyu.edu/acsys/cvc3/.

[4] E. Parrinello G. Pelosi A. Barenghi, G. Bertoni. Low Voltage Fault Attacks on the RSA Cryptosystem. pages 23 – 31. IEEE, 2009.

[5] I. Koren D. Naccache A. Barenghi, L. Breveglieri. Fault Injection Attacks on Cryptographic Devices : Theory, Practice, and Countermeasures. pages 3056 – 3076. IEEE, 2012.

[6] ARM. ARM® and Thumb®-2 Instruction Set Quick Reference Card. 2007.

[7] R. Bruttomesso. Satisfiability Modulo Theories : a pragmatic introduction. 2012.

[8] N. Moro B. Robisson E. Encrenaz, K. Heydemann. Experimental evaluation of two software countermeasures against fault attacks.

[9] K. Heydemann E. Enrenaz. Proposition de thèse à l'EDITE de paris : Sécurisation de code et analyse de la robustesse d'un code contre des fautes transitoires avec vérification formelle. 2014.

[10] N. Vachharajani R. Rangan D. August G. Reis, J. Chang. SWIFT : Software Implemented Fault Tolerance. pages 243 – 254. IEEE, 2005.

[11] B. Gierlichs J. Balasch and I. Verbauwhede. An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. pages 105 – 114. IEEE, 2011.

[12] N. Bjørner L. de Moura. Satisfiability Modulo Theories : Introduction and Applications. *Communications of the ACM*, 2011.

[13] E. Encrenaz N. Moro, K. Heydemann and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. Springer Berlin Heidelberg, 2014.

[14] K. Heydemann B. Robisson E. Encrenaz N. Moro, A. Dehbaoui. Electromagnetic fault injection : towards a fault model on a 32-bit microcontroller. 2013.

[15] J. Schmidt and M. Hutter. Optical and EM Fault-Attacks on CRT-based RSA : Concrete Results. 2007.

[16] S. Skorobogatov. Local Heating Attacks on Flash Memory Devices. pages 1 – 6. IEEE, 2009.