

Sécurisation à la compilation de logiciels contre les attaques en fautes

Thierno Barry

▶ **To cite this version:**

Thierno Barry. Sécurisation à la compilation de logiciels contre les attaques en fautes. Autre. Université de Lyon, 2017. Français. NNT : 2017LYSEM037 . tel-01783995v2

HAL Id: tel-01783995
https://cea.hal.science/tel-01783995v2

Submitted on 24 Nov 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre NNT : 2017LYSEM037

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
Opérée au sein de
L'École des Mines de Saint-Etienne

Ecole Doctorale N° 488
Sciences, Ingénierie, Santé

Spécialité de doctorat : Microélectronique

Soutenue publiquement le 24/11/2017, par :
Thierno Barry

Sécurisation à la compilation de logiciels contre les attaques en fautes

---

Devant le jury composé de :

M. GUILLEY Sylvain	Professeur	Telecom Paris Tech	Rapporteur
M. ROHOU Erven	Directeur de recherche	INRIA	Rapporteur
Mm. POTET Marie-Laure	Professeure	Université de Grenoble	Examinatrice
M. DE GRANDMAISON Arnaud	Ingénieur	ARM / LLVM Foundation	Examinateur
M. COHEN Albert	Professeur	INRIA	Examinateur
M^{me}. HEYDEMANN Karine	Maître de conférences	UPMC	Examinatrice
M. ROBISSON Bruno	Ingénieur chercheur	CEA DPACA	Directeur de thèse
M. COUROUSSÉ Damien	Ingénieur chercheur	CEA-LIST	Encadrant de thèse

Spécialités doctorales	Responsables :	Spécialités doctorales	Responsables
SCIENCES ET GENIE DES MATERIAUX	K. Wolski Directeur de recherche	MATHEMATIQUES APPLIQUEES	O. Roustant, Maître-assistant
MECANIQUE ET INGENIERIE	S. Drapier, professeur	INFORMATIQUE	O. Boissier, Professeur
GENIE DES PROCEDES	F. Gruy, Maître de recherche	SCIENCES DES IMAGES ET DES FORMES	JC. Pinoli, Professeur
SCIENCES DE LA TERRE	B. Guy, Directeur de recherche	GENIE INDUSTRIEL	X. Delorme, Maître assistant
SCIENCES ET GENIE DE L'ENVIRONNEMENT	D. Graillot, Directeur de recherche	MICROELECTRONIQUE	Ph. Lalevée, Professeur

EMSE : Enseignants-chercheurs et chercheurs autorisés à diriger des thèses de doctorat (titulaires d'un doctorat d'État ou d'une HDR)

ABSI	Nabil	CR	Génie industriel	CMP
AUGUSTO	Vincent	CR	Image, Vision, Signal	CIS
AVRIL	Stéphane	PR2	Mécanique et ingénierie	CIS
BADEL	Pierre	MA(MDC)	Mécanique et ingénierie	CIS
BALBO	Flavien	PR2	Informatique	FAYOL
BASSEREAU	Jean-François	PR	Sciences et génie des matériaux	SMS
BATTON-HUBERT	Mireille	PR2	Sciences et génie de l'environnement	FAYOL
BEIGBEDER	Michel	MA(MDC)	Informatique	FAYOL
BLAYAC	Sylvain	MA(MDC)	Microélectronique	CMP
BOISSIER	Olivier	PR1	Informatique	FAYOL
BONNEFOY	Olivier	MA(MDC)	Génie des Procédés	SPIN
BORBELY	Andras	MR(DR2)	Sciences et génie des matériaux	SMS
BOUCHER	Xavier	PR2	Génie Industriel	FAYOL
BRODHAG	Christian	DR	Sciences et génie de l'environnement	FAYOL
BRUCHON	Julien	MA(MDC)	Mécanique et ingénierie	SMS
CAMEIRAO	Ana	MA(MDC)	Génie des Procédés	SPIN
CHRISTIEN	Frédéric	PR	Science et génie des matériaux	SMS
DAUZERE-PERES	Stéphane	PR1	Génie Industriel	CMP
DEBAYLE	Johan	MR	Sciences des Images et des Formes	SPIN
DEGEORGE	Jean-Michel	MA(MDC)	Génie industriel	Fayol
DELAFOSSE	David	PR0	Sciences et génie des matériaux	SMS
DELORME	Xavier	MA(MDC)	Génie industriel	FAYOL
DESRAYAUD	Christophe	PR1	Mécanique et ingénierie	SMS
DJENIZIAN	Thierry	PR	Science et génie des matériaux	CMP
DOUCE	Sandrine	PR2	Sciences de gestion	FAYOL
DRAPIER	Sylvain	PR1	Mécanique et ingénierie	SMS
FAUCHEU	Jenny	MA(MDC)	Sciences et génie des matériaux	SMS
FAVERGEON	Loïc	CR	Génie des Procédés	SPIN
FEILLET	Dominique	PR1	Génie Industriel	CMP
FOREST	Valérie	MA(MDC)	Génie des Procédés	CIS
FRACZKIEWICZ	Anna	DR	Sciences et génie des matériaux	SMS
GARCIA	Daniel	MR(DR2)	Sciences de la Terre	SPIN
GAVET	Yann	MA(MDC)	Sciences des Images et des Formes	SPIN
GERINGER	Jean	MA(MDC)	Sciences et génie des matériaux	CIS
GOEURIOT	Dominique	DR	Sciences et génie des matériaux	SMS
GONDRAN	Natacha	MA(MDC)	Sciences et génie de l'environnement	FAYOL
GONZALEZ FELIU	Jesus	MA(MDC)	Sciences économiques	FAYOL
GRAILLOT	Didier	DR	Sciences et génie de l'environnement	SPIN
GROSSEAU	Philippe	DR	Génie des Procédés	SPIN
GRUY	Frédéric	PR1	Génie des Procédés	SPIN
GUY	Bernard	DR	Sciences de la Terre	SPIN
HAN	Woo-Suck	MR	Mécanique et ingénierie	SMS
HERRI	Jean Michel	PR1	Génie des Procédés	SPIN
KERMOUCHE	Guillaume	PR2	Mécanique et Ingénierie	SMS
KLOCKER	Helmut	DR	Sciences et génie des matériaux	SMS
LAFOREST	Valérie	MR(DR2)	Sciences et génie de l'environnement	FAYOL
LERICHE	Rodolphe	CR	Mécanique et ingénierie	FAYOL
MALLIARAS	Georges	PR1	Microélectronique	CMP
MOLIMARD	Jérôme	PR2	Mécanique et ingénierie	CIS
MOUTTE	Jacques	CR	Génie des Procédés	SPIN
NEUBERT	Gilles			FAYOL
NIKOLOVSKI	Jean-Pierre	Ingénieur de recherche	Mécanique et ingénierie	CMP
NORTIER	Patrice	PR1	Génie des Procédés	SPIN
O CONNOR	Rodney Philip	MA(MDC)	Microélectronique	CMP
OWENS	Rosin	MA(MDC)	Microélectronique	CMP
PERES	Véronique	MR	Génie des Procédés	SPIN
PICARD	Gauthier	MA(MDC)	Informatique	FAYOL
PIJOLAT	Christophe	PR0	Génie des Procédés	SPIN
PINOLI	Jean Charles	PR0	Sciences des Images et des Formes	SPIN
POURCHEZ	Jérémy	MR	Génie des Procédés	CIS
ROUSSY	Agnès	MA(MDC)	Microélectronique	CMP
ROUSTANT	Olivier	MA(MDC)	Mathématiques appliquées	FAYOL
SANAUR	Sébastien	MA(MDC)	Microélectronique	CMP
STOLARZ	Jacques	CR	Sciences et génie des matériaux	SMS
TRIA	Assia	Ingénieur de recherche	Microélectronique	CMP
VALDIVIESO	François	PR2	Sciences et génie des matériaux	SMS
VIRICELLE	Jean Paul	DR	Génie des Procédés	SPIN
WOLSKI	Krzystof	DR	Sciences et génie des matériaux	SMS
XIE	Xiaolan	PR1	Génie industriel	CIS
YUGMA	Gallian	CR	Génie industriel	CMP

Mise à jour : 03/02/2017

Remerciements

C'est avec un immense plaisir que j'aborde l'écriture de ces quelques traditionnels remerciements. Qu'ils me permettent de témoigner enfin la reconnaissance et la gratitude que j'éprouve envers ces nombreuses personnes qui m'ont accompagné tout au long de cette aventure.

Merci à Bruno Robisson, mon directeur de thèse, pour son engagement, sa disponibilité et pour ses remarques toujours pertinentes. Merci également à Damien Couroussé, mon encadrant, pour son investissement et son encadrement rigoureux qui m'ont poussé à donner le meilleur de moi-même.

Je tiens à exprimer ma profonde gratitude aux membres de mon jury, en commençant par monsieur Sylvain Guilley et monsieur Erven Rohou, pour avoir accepté de rapporter cette thèse. Merci à madame Marie-Laure Potet, à monsieur Arnaud De Grandmaison et à monsieur Albert Cohen pour avoir accepté d'examiner mes travaux. Un remerciement particulier à madame Karine Heydemann, examinatrice de cette thèse, pour ses précieux conseils et pour son implication active dans l'aboutissement de ces travaux de thèse.

Je souhaiterais également remercier tout le personnel du laboratoire LIALP, en particulier, Vincent Olive, *le chef*, pour son sens de l'écoute et ses conseils. Merci à Suzanne Lesecq et à Henri-Pierre Charles pour avoir toujours pris soin de nous les doctorants (*les pioupious*). Merci à Nicolas Belleville, notre jeune et brillant doctorant, pour ses relectures de mon manuscrit. Merci à Abderrhamane Seria, mon collègue de bureau, pour ses conseils, ses encouragements et les bons moments passés ensemble. Merci à Olivier A., mon fidèle compagnon de Midi Minatec.

J'aimerais aussi remercier Nicolas Moro pour son assistance au tout début de cette thèse, et Louis Dureuil pour m'avoir permis d'utiliser le simulateur de fautes CELTIC. Merci également à Ivan Llopard et Victor Lomüller pour leurs précieuses aides dans la prise en main de LLVM, et à Fernando Endo pour ses réponses à mes questions sur l'architecture ARM.

Je tiens aussi à remercier mes camarades (*le groupe de 13h*) : Adja, Isabel, Roxana, Geneviève, Sanaa, Mauricio, Julie et Fayçal.

Enfin, je voudrais adresser un sincère remerciement à ma famille pour m'avoir toujours soutenu dans mes différents projets et pour avoir su, à maintes reprises durant ces trois années de thèse, me remonter le moral et me redonner la motivation d'aller de l'avant. Une pensée particulière à ma grande sœur, Docteur Fatoumata Barry, que j'admire profondément et que j'apprécie énormément, qui m'a toujours soutenu et qui a été un modèle de réussite pour toute la famille.

Table des matières

I Introduction générale **1**

1 Introduction **3**
1.1 Contexte et motivations 4
1.2 Problématiques 5
1.3 Contributions 6
1.4 Plan du manuscrit 7

II État de l'art **9**

2 Attaques physiques **11**
2.1 Attaques par canaux auxiliaires 12
2.2 Attaques par injection de fautes 14
2.3 Schéma de contre-mesures contre les attaques en fautes 20
2.4 Exemples d'implémentations logicielles 22
2.5 Conclusion 26

3 Compilation **27**
3.1 Principes généraux 28
3.2 Le compilateur LLVM 32
3.3 Compilation pour la sécurité 39
3.4 Conclusion 44

III Contributions **45**

4 Simulateur de fautes **47**
4.1 Fonctionnement 48
4.2 Modèles de fautes supportés 50
4.3 Architecture interne 52
4.4 Utilisation 54

5 Compilation d'un schéma de tolérance aux fautes **57**
5.1 Modèle de fautes 58
5.2 Mise en œuvre du schéma de protection 58
5.3 Expérimentations 69
5.4 Conclusion 72

6 Généralisation du schéma de tolérance au saut d'instructions 75
6.1 Modèles de fautes . . . 76
6.2 Description du schéma de protections . . . 77
6.3 Mise en œuvre . . . 79
6.4 Expérimentations . . . 80
6.5 Conclusion . . . 83

7 Combinaison du schéma de tolérance avec du CFI 85
7.1 Modèles de fautes . . . 86
7.2 Notations . . . 86
7.3 Description du schéma CFI . . . 87
7.4 Mise en œuvre . . . 91
7.5 Illustrations . . . 95
7.6 Expérimentations . . . 100
7.7 Conclusion . . . 101

8 Conception et mise en œuvre du schéma CCFI 103
8.1 Modèles de fautes . . . 104
8.2 Principe de fonctionnement du schéma CCFI . . . 105
8.3 Mise en œuvre . . . 105
8.4 Expérimentations . . . 115
8.5 Conclusion . . . 117

9 Conclusion et perspectives 119
9.1 Conclusion . . . 119
9.2 Perspectives . . . 121

A Description des passes de compilation du schéma CCFI 123
A.1 Détails d'implémentation . . . 123

Bibliographie personnelle 135

Bibliographie 137

Table des figures 147

Liste des tableaux 150

Résumé 154

Première partie

Introduction générale

Chapitre 1

Introduction

Sommaire

1.1 Contexte et motivations 4
1.2 Problématiques 5
1.2.1 Automatisation 5
1.2.2 Optimisation 5
1.2.3 Combinaison 6
1.3 Contributions 6
1.3.1 Automatisation 6
1.3.2 Optimisation 6
1.3.3 Combinaison 7
1.4 Plan du manuscrit 7

1.1 Contexte et motivations

Un système embarqué est un système électronique souvent dédié à une ou quelques tâches spécifiques, soumis à des contraintes de temps d'exécution, de consommation énergétique, d'espace mémoire et de sécurité. Grâce aux progrès de la miniaturisation et aux faibles coûts des composants électroniques, les systèmes embarqués deviennent de plus en plus accessibles au grand public avec des fonctionnalités de plus en plus complexes. Ces systèmes, tels que les cartes de crédit, les cartes SIM ou encore les passeports biométriques, s'imposent aujourd'hui comme des objets incontournables de notre quotidien. Compte tenu de la nature des données que ces systèmes sont amenés à stocker et à manipuler (données personnelles, confidentielles, critiques, etc.), la sécurité de ces systèmes est devenue un enjeu et une préoccupation majeure pour les industriels, les organisations étatiques et le grand public. Ces systèmes font face à des attaques qui visent leur :

- *Confidentialité* : accéder frauduleusement à des informations sensibles
- *Intégrité* : altérer des données sensibles
- *Disponibilité* : rendre le système non disponible pour les utilisateurs légitimes

La confidentialité et l'intégrité des données sont aujourd'hui largement traitées par la cryptographie moderne. Il existe, en effet, plusieurs algorithmes et protocoles cryptographiques standardisés et considérés comme sûrs d'un point de vue mathématique, par exemple l'algorithme AES (*Advance Encryption Standard*) [97] pour le chiffrement, et DSA (*Digital Signature Algorithm*) [79] pour la signature numériques. La sécurité des algorithmes cryptographiques repose fondamentalement sur le constat que certains problèmes mathématiques sont difficiles à résoudre dans un temps raisonnable. C'est le cas, par exemple, de l'algorithme de chiffrement RSA basé sur le problème de factorisation de grands entiers, et pour lequel, le meilleur algorithme de factorisation connu en 2017 est d'une complexité sous-exponentielle par rapport à la taille de l'entier à factoriser. Cependant, la sécurité de ces algorithmes peut être compromise en exploitant des failles liées au matériel sur lequel ils sont exécutés. Ce type d'attaques est connu sous le nom d'*attaques physiques*. On dénombre deux grandes classes d'attaques physiques :

1. Les attaques par observation de canaux auxiliaires : elles consistent à collecter des informations sur le composant en cours d'exécution, puis à exploiter ces informations pour remonter aux données manipulées par le composant.
2. Les attaques par perturbations : elles consistent à perturber le fonctionnement normal du composant pendant son exécution, et à tirer parti de cette perturbation pour compromettre le système.

Le secteur le plus concerné par ces attaques est celui de la carte à puce avec ses nombreuses déclinaisons. Eurosmart prévoit une vente de 9 à 10 milliards de cartes à puces en 2017 [47]. Toutefois, l'efficacité de ces attaques a été démontrée dans d'autres secteurs tels que le débridage de produits grand public. Un des exemples les

plus notables est le déverrouillage (*jailbreak*) de certaines fonctionnalités restreintes de l'iPhone 3G réalisé par George Hotz [102].

De nombreuses contre-mesures contre les attaques physiques ont déjà été proposées ; certaines visent à détecter une attaque et à réagir en conséquence, d'autres à rendre l'attaque plus difficile à réaliser. Les contre-mesures implémentées en matériel ont l'avantage d'être plus efficaces en temps d'exécution, mais sont susceptibles d'augmenter la taille du composant électronique et ne peuvent pas être mises à jour. Elles sont généralement utilisées dans des composants dédiés à la sécurité comme la carte à puce ou les éléments sécurisés (*secure element*). En revanche, les contre-mesures implémentées en logiciel, bien que plus coûteuses en performance, sont plus faciles à déployer et à mettre à jour. Elles sont principalement utilisées pour ajouter de la sécurité sur des composants non sécurisés. Dans la plupart des cas, des contre-mesures matérielles et logicielles sont combinées pour plus d'efficacité.

1.2 Problématiques

Nous déclinons les problématiques de cette thèse en trois points détaillés ci-après : automatisation, optimisation et combinaison de contre-mesures logicielles contre les attaques physiques.

1.2.1 Automatisation

Aujourd'hui, les contre-mesures contre les attaques physiques sont intégrées manuellement et au cas par cas. Ce qui, en plus d'être une tâche fastidieuse et coûteuse, peut être source d'erreurs dans sa mise en œuvre. D'un point de vue industriel, l'intégration manuelle allonge le temps de développement nécessaire pour mettre au point un produit sécurisé (*time to market*). Avec le raccourcissement des cycles de vie des produits, le time to market est aujourd'hui pour une entreprise, un facteur stratégique à prendre en compte pour améliorer sa rentabilité. Automatiser l'intégration d'une contre-mesure est néanmoins un problème difficile, car cela nécessite de prendre en compte toutes les spécificités qu'un programme peut avoir, mais aussi de garantir que la fonctionnalité du programme n'est pas altérée.

1.2.2 Optimisation

La littérature scientifique fait état de nombreux schémas de contre-mesures contre les attaques physiques. Nous pouvons constater que pour un schéma de contre-mesure donné, l'intérêt principal de la communauté scientifique est plutôt porté sur sa validité vis-à-vis des attaques qu'il protège. Très peu d'intérêt est porté sur son l'aspect réaliste ou sa possible utilisation dans des produits grand public, alors que ces systèmes sont soumis à des contraintes en termes d'empreinte mémoire ou de temps d'exécution. Par exemple pour les cartes à puce de paiement, le standard EMV définit un temps maximal de réponse d'une carte, au-delà duquel la transaction en cours échoue.

1.2.3 Combinaison

Dans l'état de l'art, un schéma de contre-mesure est conçu pour protéger contre un type d'attaque particulier. Or, en pratique, un attaquant n'est pas tenu à se limiter à un seul type d'attaque. Pire encore, une contre-mesure contre un type d'attaque peut rendre un code plus vulnérable à d'autre type d'attaques [87]. Il est donc impératif de savoir prendre en compte dans un seul schéma plusieurs types d'attaques. La solution qui consisterait à superposer *naïvement* plusieurs contre-mesures pose deux principaux problèmes : (1) une contre-mesure pourrait annuler l'effet d'une autre [86], (2) le coût (en taille du code et temps d'exécution) final serait la somme des coûts de chaque contre-mesure, ce qui rend cette solution irréaliste pour les systèmes embarqués. Savoir efficacement articuler plusieurs contre-mesures de natures différentes est aujourd'hui un problème non résolu.

1.3 Contributions

Cette thèse propose d'intégrer les contre-mesures logicielles au cœur du processus de compilation. Nous nous sommes focalisé particulièrement sur les contre-mesures contre les attaques en faute. Les principales contributions de cette thèse sont organisées autour des trois problématiques soulevées précédemment.

1.3.1 Automatisation

Nous proposons un compilateur basé sur LLVM permettant l'application automatisée de plusieurs schémas de contre-mesures au moment de la compilation. L'utilisateur fournit à l'entrée du compilateur un code source et une description des contre-mesures à appliquer. Le compilateur produit automatiquement un code binaire fonctionnel et sécurisé conformément aux descriptions de l'utilisateur.

1.3.2 Optimisation

Le compilateur modifié que nous proposons permet de faire coexister les propriétés de sécurité des contre-mesures et les optimisations de code. Nous avons mis en œuvre des mécanismes permettant de garantir que les optimisations n'altèrent pas les propriétés de sécurité de contre-mesures, et inversement, que les propriétés des contre-mesures n'empêchent pas les optimisations. Puisqu'il peut y avoir des cas où seulement une partie du programme nécessite d'être sécurisée, nous offrons la possibilité à l'utilisateur d'annoter le code source fourni en entrée pour spécifier les sections à sécuriser et la nature des contre-mesures à appliquer pour chaque section.

```
1 @begin_secure("param1", "param2", ...)
2 code à sécuriser
3 @end_secure("param1", "param2", ...)
```

1.3.3 Combinaison

Nous avons intégré, dans le compilateur, plusieurs schémas de contre-mesures contre les attaques en fautes. Parmi lesquels : (1) des schémas destinés à protéger contre un seul modèle d'attaque, (2) des schémas conçus pour protéger contre plusieurs modèles d'attaques et (3) des schémas combinés pour protéger contre plusieurs modèles d'attaques. Ces schémas sont introduits dans les sections suivantes.

1.4 Plan du manuscrit

La suite de ce manuscrit est organisée en deux parties : Partie II et Partie III

Partie II : État de l'art

chapitre 2 Attaques physiques

Ce chapitre présente l'état de l'art des attaques physiques, en se focalisant sur les attaques en fautes et ses contre-mesures logicielles.

chapitre 3 Compilation

Ce chapitre commence par présenter les principes généraux de la compilation avec un focus sur le compilateur LLVM, aborde les problématiques de la compilation pour la sécurité, et enfin, présente un état de l'art de la compilation pour la sécurité.

Partie III : Contributions

chapitre 4 Simulateur de fautes

Dans ce chapitre, nous présentons le simulateur de fautes que nous avons développé pour valider les schémas de contre-mesures vis-à-vis des modèles de fautes considérés.

chapitre 5 Compilation d'un schéma de tolérance aux fautes

Nous commençons par intégrer dans le compilateur, le schéma de tolérance au saut d'instructions, formellement vérifié par Moro et al. [76]. Le schéma consiste à dupliquer toutes les instructions d'un programme pour protéger contre une faute provocant le saut d'une instruction. Cette première mise en œuvre a pour but de valider expérimentalement l'avantage, en termes de temps d'exécution et de taille de code, de l'approche *compilation* sur l'approche code *binaire* utilisée par Moro et al. [76].

chapitre 6 Généralisation du schéma de tolérance au saut d'instructions

Dans ce chapitre, nous proposons une généralisation du schéma de tolérance pour protéger contre les fautes multiples et les fautes conduisant au saut de plusieurs instructions consécutives.

chapitre 7 Combinaison du schéma de tolérance avec du CFI

Nous intégrons dans le compilateur un schéma CFI (*Control Flow Integrity*) inspiré de celui proposé par Lalande et al. [58]. Ces deux schémas visent à garantir que le chemin d'exécution d'un programme est conforme au chemin initialement prévu. Le schéma proposé est basé sur l'utilisation d'une *routine de vérification* pour garantir la validité de chaque transition d'un bloc du programme à une autre. Cependant, ce schéma CFI s'avère vulnérable lorsqu'une faute de type saut d'instructions est injectée sur la routine de vérification. Pour corriger cette vulnérabilité, nous proposons une combinaison du schéma CFI avec le schéma de tolérance au saut d'instructions.

chapitre 8 Conception et mise en œuvre du schéma CCFI

De façon générale, un schéma CFI permet de garantir la validité du chemin d'exécution suivi mais ne garantit pas que toutes les instructions le long de ce chemin ont été exécutées. Dans ce chapitre, nous proposons un nouveau schéma de contremesure appelé CCFI (*Code and Control-Flow Integrity*) apportant des garanties sur la validité du chemin d'exécution et l'intégrité des instructions exécutées.

chapitre 9 Conclusion et perspectives

Ce chapitre fait un bilan des contributions présentées dans ce manuscrit, puis présente des pistes d'évolutions de ces travaux.

Deuxième partie

État de l'art

Chapitre 2

Attaques physiques

Sommaire

2.1 Attaques par canaux auxiliaires 12
2.1.1 Contre-mesures 12
2.2 Attaques par injection de fautes 14
2.2.1 Moyens d'injection de fautes 14
2.2.2 Modèles de fautes 16
2.2.3 Niveau instruction 16
2.2.4 Types de fautes 18
2.2.5 Exemples de scénarios d'attaques 19
2.3 Schéma de contre-mesures contre les attaques en fautes . . 20
2.3.1 Schémas de détection de fautes 20
2.3.2 Schémas de tolérance aux fautes 21
2.4 Exemples d'implémentations logicielles 22
2.4.1 Détection de fautes 22
2.4.2 Tolérance aux fautes 22
2.4.3 Intégrité du flot de contrôle (CFI) 24
2.4.4 Limitations 24
2.5 Conclusion 26

Dans ce chapitre, nous présentons l'état de l'art des attaques physiques. La section 2.1 présente les attaques par canaux auxiliaires en se limitant aux principes de base et aux principaux schémas de contre-mesures existantes. Dans la section 2.2, nous présentons les attaques par injection de fautes, avec un focus sur les schémas de contre-mesures existants et leurs mises en œuvre logicielles.

2.1 Attaques par canaux auxiliaires

Les attaques par canaux auxiliaires ou attaques par canaux cachés consistent à collecter des informations sur le composant (cible) en cours de fonctionnement, puis à exploiter ces informations pour révéler les données manipulées par ce composant. Ces informations, appelées *grandeurs physiques mesurables* ou des *observables*, peuvent être la consommation électrique [57], le temps d'exécution [56], le rayonnement électromagnétique [85] ou le signal acoustique [95]. Lorsque l'information sur une donnée manipulée par le programme est obtenue à partir des observables du circuit, on dit que cette donnée *fuit*. Ces attaques fonctionnent parce qu'il existe une corrélation entre les opérations réalisées à l'intérieur d'un circuit électronique et les observables.

2.1.1 Contre-mesures

L'objectif des contre-mesures est de rompre ou de rendre inexploitable la corrélation entre les données manipulées par un composant et ses observables. Plusieurs schémas de contre-mesures ont été proposés, ces schémas peuvent être regroupés en deux concepts : concept de dissimulation et concept de masquage.

2.1.1.1 Concept de dissimulation (*Hiding*)

Il s'agit de dissimuler les données sensibles d'un programme dans les observables du circuit [68]. Parmi les techniques de dissimulation, on peut citer :

Ajout du bruit : Cette technique consiste à diminuer le rapport signal sur bruit (SNR) dans les observables du circuit. Cela peut se faire en insérant des opérations *factices* qui ne modifient pas la fonctionnalité du programme. Cette technique est illustrée par la figure 2.1. Les deux fonctions `add_ten` des figures 2.1a et 2.1b retournent le même résultat, mais ne produisent pas les mêmes observables. En effet, l'exécution des instructions des lignes 2, 3 et 4 de la figure 2.1b n'interagit pas avec la fonctionnalité du programme.

Ordonnancement aléatoire des exécutions : L'ordonnancement aléatoire ou *shuffling* consiste à changer aléatoirement l'ordre d'exécution des opérations. Cette technique fonctionne mieux lorsque le programme comporte des opérations indépendantes [16, 70]. Cette technique vise à faire varier l'instant de fuite des données sensibles dans les observables.

```
1 int add_ten(int x){
2     return x+10;
3 }
```

(a) Code initial

```
1 int add_ten(int x){
2     int y = x;
3     for (int i=0; i<10; i++)
4         y = y + 1;
5     return x+10;
6 }
```

(b) Code avec des instructions factices

FIGURE 2.1 – Illustration de l'ajout du bruit dans un programme pour modifier les observables du circuit qui l'exécute

Codes polymorphes : Un code polymorphe est un code capable de changer une partie de son implémentation tout en conservant sa fonctionnalité [2, 31]. Parmi les techniques de polymorphisme il y a la substitution qui consiste à substituer une opération par une autre, différente mais sémantiquement équivalente à la première. Par exemple : l'opération $a \oplus b$ peut être substituée par $(a \vee b) \oplus (a \wedge b)$.

Équilibrage de la consommation : Cette technique vise à rendre constante la consommation électrique d'une opération quelles que soient les données manipulées. Une des méthodes utilisées consiste à calculer le complément de chacune des variables. Ce principe a été décliné à tous les niveaux d'abstraction, du niveau logiciel au niveau transistor. La logique double rail (*dual-rail*), couramment utilisée dans la littérature scientifique, consiste à utiliser deux fils pour encoder une valeur binaire. Un exemple de codage est qu'un des deux fils prenne la valeur 1 pour encoder une donnée à 1 et l'autre la valeur 1 pour encoder la donnée à 0. Plusieurs exemples de la logique double rail sont présentés par Guilley et al. [52, 51] et Danger et al. [36]. Une implémentation logicielle est proposée par Hoogvorst et al. [54].

2.1.1.2 Concept de masquage (*Masking*)

Le masquage vise à rendre les informations observables du circuit statistiquement indépendantes des données manipulées [20, 90]. L'une des techniques utilisées consiste à appliquer un *masque* aléatoire sur les données sensibles avant de les manipuler. Il existe plusieurs types de masquage, parmi lesquels le masquage booléen qui se définit comme une fonction f tel que : $f(x) = x \oplus r$, où x est la valeur secrète et le maque r est une valeur aléatoire [29].

Pour illustrer ce concept, supposons que nous souhaitons calculer l'opération $c = m \oplus k$, où k est une donnée sensible et m peut être connu de l'attaquant. Pour éviter que l'information sur k ne fuite lors du calcul de c, on masque au préalable k en $k' = k \oplus r$, puis on calcule $c' = m \oplus k'$ à la place. Ainsi, la fuite sera corrélée avec k' et non k. Pour retrouver c, on effectue $c = c' \oplus r$.

2.2 Attaques par injection de fautes

Les fautes sur les circuits électroniques ont été étudiées dès les années 70, suite à l'observation de l'effet d'une particule radioactive sur un circuit électronique [10]. Il a été observé que lorsqu'une particule radioactive entre en contact avec la surface d'un circuit électronique, tel qu'une mémoire volatile, elle génère une charge électrique qui peut modifier la valeur d'un ou de plusieurs bits. Les attaques par injection de fautes consistent à perturber de manière volontaire le fonctionnement normal du circuit dans le but d'extraire des données sensibles, par exemple une clé de chiffrement. L'utilisation des fautes pour attaquer un système a été introduite pour la première fois par Dan Boneh, Richard DeMillo et Richard Lipton en 1997 [21]. Dans cet article, les auteurs ont présenté une attaque par injection de fautes permettant de retrouver la clé de chiffrement sur une implémentation de l'algorithme RSA.

2.2.1 Moyens d'injection de fautes

Il existe plusieurs techniques pouvant être utilisées pour injecter une faute dans un circuit. Celles qui ont été les plus décrites dans l'état de l'art sont les suivantes.

2.2.1.1 Perturbation de la tension d'alimentation

Cette technique, communément appelée *Power Glitch*, consiste à faire varier pendant un court instant la tension d'alimentation d'un circuit. Une telle perturbation va avoir un effet sur la vitesse de transfert des données et des instructions entre le processeur et la mémoire principale. Elle peut notamment induire le processeur à mal interpréter des données ou à sauter des instructions [8, 108, 25].

2.2.1.2 Augmentation de la fréquence d'horloge

De manière analogue au Power Glitch, la perturbation du signal d'horloge ou *Clock Glitch* consiste à augmenter pendant un court instant la fréquence d'horloge [3]. Or, la fréquence maximale d'un circuit synchrone étant définie en fonction de la durée de transfert des données entre ses blocs logiques, pousser cette fréquence au-delà de cette limite engendre des dysfonctionnements dans les opérations du circuit.

2.2.1.3 Augmentation de la température

Les fabricants de circuits électroniques définissent, pour chaque circuit, un intervalle de température au-delà duquel le fonctionnement normal du circuit n'est pas garanti. L'objectif est de faire varier la température du circuit au-delà de cet intervalle pour provoquer une faute. Cette perturbation affecte particulièrement la capacité de lecture et d'écriture du circuit. En modifiant la température, S. Skorobogatov[96] a par exemple réussi à injecter des fautes dans des mémoires Flash

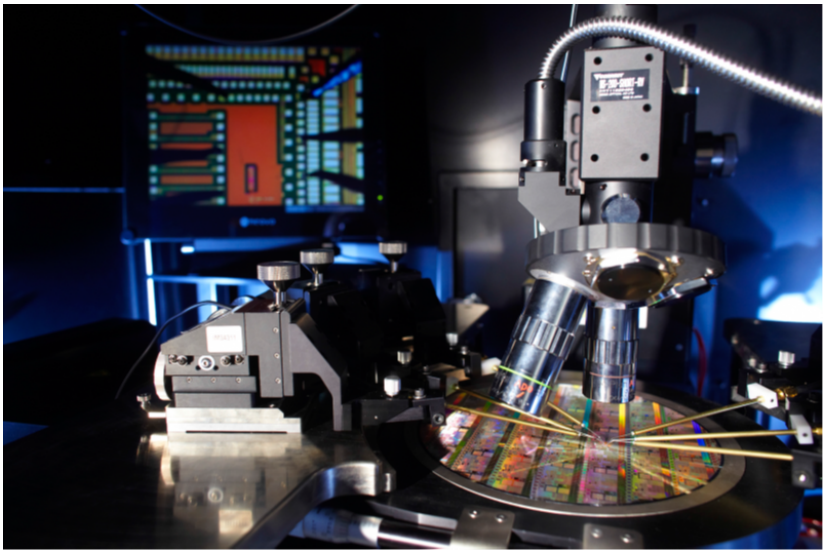


FIGURE 2.2 – Équipement d'injection laser du la plate-forme Micro Packs du centre microélectronique de Provence

et EEPROM. Hutter et al. [55] sont, quant à eux, parvenus à attaquer une implémentation de l'algorithme RSA en jouant sur les températures de fonctionnement du circuit.

2.2.1.4 Lumière focalisée

Une lumière focalisée peut être utilisée pour injecter une faute dans un circuit électronique, car l'énergie transportée par les photons émis par la source lumineuse est absorbée par les électrons du silicium. Lorsque la longueur d'onde de la lumière correspond à un niveau d'énergie supérieur au seuil permettant à des électrons de passer dans la bande de conduction du silicium, des paires électrons-trous sont créées le long du faisceau lumineux [42]. Ce phénomène entraîne l'apparition d'un courant photoélectrique susceptible de modifier l'état des portes logiques. La figure 2.2 montre un des bancs d'injection laser de la plate-forme MicroPacks du Centre Microélectronique de Provence [1].

2.2.1.5 Injection électromagnétique

L'injection électromagnétique consiste à créer un champ électromagnétique intense à la surface d'un circuit électronique. Quisquater et al. [85] ont présenté la première utilisation de l'injection électromagnétique pour attaquer des algorithmes cryptographiques. Depuis, plusieurs attaques utilisant différents types de générateurs d'impulsions électromagnétiques ont été présentées [84, 39, 93, 80, 24].

1. www.mines-stetienne.fr/recherche/5-centres-de-formation-et-de-recherche/centre-microelectronique-de-provence

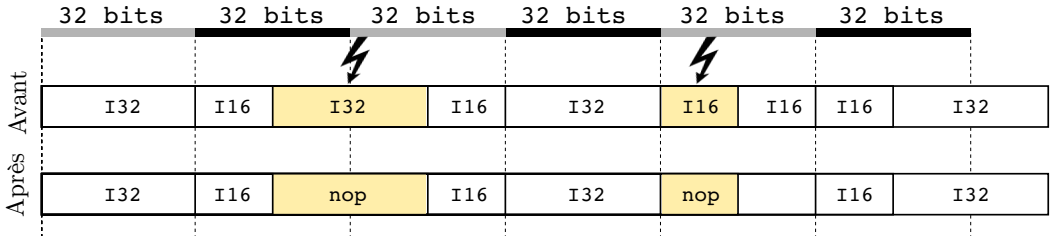


FIGURE 2.3 – Illustration du modèle de fautes saut d'instructions `skip{1, 2}`

2.2.2 Modèles de fautes

Nous venons de voir que plusieurs phénomènes physiques peuvent être à l'origine d'une faute sur un circuit électronique. Un modèle de fautes est une abstraction utilisée pour modéliser l'effet de la faute sur un circuit. Cette modélisation permet d'expliquer ou de prédire l'impact d'une faute sur le fonctionnement global du circuit.

L'effet d'une faute peut être décrit par plusieurs modèles selon le niveau d'abstraction choisi. Verbauwhede et al. [101] présentent une classification de différents modèles en fonction du niveau d'abstraction. Barenghi et al. [12] font une revue bibliographique des attaques par injection de fautes et les modèles de fautes couramment associés. Par la suite, nous proposons de décrire l'effet d'une faute au niveau *instruction*.

2.2.3 Niveau instruction

Les modèles au niveau instruction décrivent les effets perceptibles d'une faute sur une instruction du programme. Ces modèles ont beaucoup été utilisés dans l'état de l'art [100, 9, 22, 18, 12]. Une instruction impactée par une faute peut ne plus correspondre à une instruction valide. Cependant, dans le cas d'un processeur à jeu d'instructions réduit (*Reduced Instruction Set Computer* ou RISC) réservant très peu de bits pour encoder une instruction, il est fréquent que la modification (ne serait-ce qu'un bit) d'une instruction corresponde à une autre instruction valide. C'est le modèle *remplacement d'instructions*. On utilisera la notation `repl{instr}` (qui sera détaillé dans la section 4.2) pour désigner le remplacement d'une instruction du programme par l'instruction `instr`.

Selon la nature de l'instruction de remplacement `instr` ou l'effet que celle-ci aura sur le programme, on peut distinguer plusieurs modèles dérivés du modèle remplacement d'instructions, notamment les modèles *saut d'instructions*, *détournement de flot de contrôle* et *saut d'un bloc logique* présentés ci-dessous.

2.2.3.1 Modèle saut d'instructions

Ce modèle correspond au cas où l'instruction de remplacement `instr` est une instruction qui n'a aucun effet fonctionnel sur le programme. Par exemple, sur les

architectures ARM, les instructions suivantes ne modifient pas la fonctionnalité du programme :

— `mov rn, rn` : qui copie le contenu du registre `rn` dans lui-même.

— `pld [rn]` : qui précharge dans le cache, la valeur se trouvant à l'adresse mémoire contenue dans le registre `rn`

— `nop` : qui ne réalise aucune opération.

— `isb` : qui est une instruction de synchronisation dont le rôle est d'attendre que l'exécution de toutes les instructions se trouvant dans le pipeline du processeur soit terminée avant de poursuivre.

On peut ajouter à cette liste les instructions qui manipulent des registres non utilisés dans le programme.

Par la suite, on modélisera le saut d'instructions par le remplacement d'une instruction par un `nop`. On utilisera la notation `skip{N, M}` pour désigner le saut de `M` séquences de `N` instructions consécutives. La figure 2.3 illustre le modèle `skip{1, 2}`. Les notations `I16` et `I32` représentent respectivement des instructions de taille 16 bits et 32 bits.

2.2.3.2 Modèle détournement de flot de contrôle

Ce modèle correspond au cas où l'instruction de remplacement conduit à faire dévier le programme de son chemin d'exécution initial. Par exemple :

— le remplacement d'une instruction par une instruction de transfert de flot de contrôle, c'est-à-dire une instruction de la famille *jump* pour les architectures x86, et *branch* pour les architectures ARM [18].

— saut d'une instruction de transfert de flot de contrôle [100].

Il existe d'autres types de perturbations qui ne sont pas au niveau instruction mais qui ont un effet sur le flot de contrôle, comme la modification de l'adresse de retour d'une fonction [48], la modification du compteur de programme (registre `pc` sur ARM) et la corruption d'une instruction de transfert de flot de contrôle [22, 11].

Par la suite, on modélisera le détournement de flot de contrôle par :

— le remplacement d'une instruction du programme par une instruction de branchement (`br`) à une adresse `addr`. Ce remplacement sera noté `repl_br{addr}`.

— le saut d'une instruction, qu'on notera `skip_br`.

2.2.3.3 Modèle saut de bloc logique

Un bloc logique est un bloc d'instructions alignées sur une adresse multiple de la taille d'un mot mémoire. Sur une architecture 32 bits par exemple, le processeur charge les données/instructions (à partir d'une mémoire) par blocs multiples de 32 bits. Un exemple de bloc logique peut être les instructions présentes au même moment dans le bus de transfert d'instructions ou dans le *buffer* de préchargement d'instructions [91].

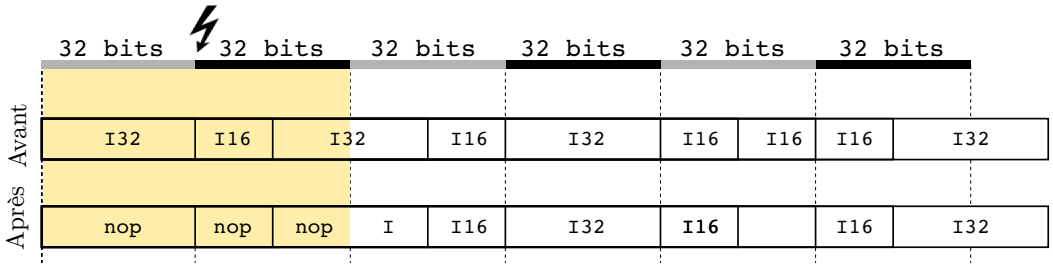


FIGURE 2.4 – Illustration du modèle de fautes `skip_block{64 bits, 1}`

Par la suite, on modélisera le saut d'un bloc logique par le remplacement de toutes les instructions de ce bloc par des `nop`. Si une instruction se trouve *à cheval* entre deux blocs, alors sa partie qui se trouve dans le bloc fauté sera remplacée par un `nop` correspondant à la taille de cette partie ; l'autre partie est laissée en l'état, de sorte que le processeur soit (1) décodera une autre instruction soit (2) lèvera une erreur de décodage. On utilisera la notation `skip_block{size, M}` pour désigner le saut de `M` blocs logiques de taille `size`. La figure 2.4 illustre le modèle `skip_block{64 bits, 1}`.

2.2.4 Types de fautes

Indépendamment de l'effet qu'une faute peut engendrer sur un programme, de nombreux autres paramètres peuvent caractériser une faute.

2.2.4.1 Persistance

Cette caractéristique définit la durée de validité d'une faute, c'est-à-dire, le temps qui sépare le moment où une valeur fautée est corrompue et le moment où la valeur d'origine est restaurée. Selon cette caractéristique, nous distinguons trois types de fautes [81] :

Transitoire : une faute est dite transitoire lorsque son effet est temporaire, après lesquels la faute disparaît et le circuit retrouve son état initial. Dans la plupart des modèles de l'état de l'art, une faute transitoire sur une donnée n'affecte que la prochaine lecture de cette donnée.

Permanente : ce type de faute maintient la valeur fautée jusqu'à la prochaine réinitialisation du circuit.

Destructive : ce type de fautes détruit la structure physique du circuit de manière irréversible. La faute reste donc présente même après réinitialisation du circuit.

Il faut noter que les fautes permanentes et les fautes transitoires peuvent avoir le même effet lorsque la variable impactée n'est lue qu'une seule fois au cours de l'exécution du programme.

2.2.5 Exemples de scénarios d'attaques

Plusieurs scénarios d'attaques sur des circuits ont été publiés dans la littérature scientifique, dont voici quelques exemples.

2.2.5.1 Attaque par analyse différentielle de fautes (DFA)

L'analyse différentielle de fautes ou *Differential Fault Analysis* (DFA) sur un algorithme cryptographique consiste à retrouver la clé de chiffrement à partir de la différence entre un chiffré correct et un chiffré fauté pour le même texte clair. Depuis son introduction par Boneh et al [21], plusieurs attaques par DFA sur différents algorithmes cryptographiques ont été publiées. Parmi lesquelles, l'attaque *BellCore* sur RSA [93] et Biham et al. [19] sur DES. Sur AES on peut citer Piret et al. [82], Giraud et al. [49], Moradi et al. [74] et Lashermes et al. [59].

2.2.5.2 Attaque par analyse *Safe error*

Une attaque *Safe error* sur un algorithme de chiffrement consiste à exploiter le fait que le chiffré produit par l'algorithme après une injection de faute est fauté ou non [104, 92]. Ce type d'attaque est particulièrement efficace sur des implémentations protégées contre les attaques par DFA. Une variante de cette attaque est l'analyse de faute sans effet (*Inefective Fault Analysis* (IFA)) dont le principe est de trouver une entrée pour laquelle les chiffrés normal et fauté sont identiques [26].

2.2.5.3 Attaque par saut d'instructions

Trichina et al. [100] ont réussi à attaquer une implémentation protégée de CRT-RSA sur un microcontrôleur ARM Cortex M3, en sautant une instruction par injection laser. Breier et al. [23] ont présenté une attaque réussie d'un AES implémenté sur un microcontrôleur ATmega328P. Les auteurs ont utilisé un laser pour sauter une instruction dans la fonction *AddRoundKey* de la dernière ronde. Rivière et al. [91] ont mis en œuvre une attaque permettant de sauter plusieurs instructions sur un microcontrôleur ARM Cortex M4. Leur attaque consiste à empêcher le préchargement des instructions dans le buffer *prefetch* (de taille 128 bits). Ce qui provoque la réexécution des instructions déjà présentes dans le buffer et ensuite le préchargement des instructions situées 128 bits plus loin dans la mémoire programme.

2.2.5.4 Attaque par dépassement de tampon

Le phénomène de dépassement de tampon ou débordement de tampon (*buffer overflow*) se produit lorsqu'un programme écrivant dans un tampon, dépasse l'espace alloué à ce tampon. Une attaque par dépassement de tampon exploite ce phénomène. Ces attaques sont très répandues dans le domaine de la sécurité logicielle [32]. L'objectif de ces attaques est l'exécution d'un code arbitraire. Fouque et al. [48] sont parvenus à utiliser une injection de faute pour provoquer un dépassement de tampon.

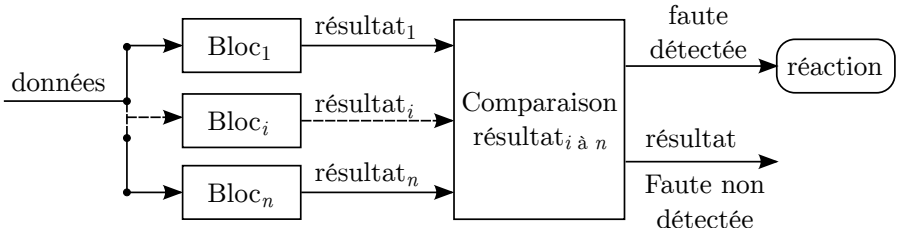


FIGURE 2.5 – Redondance spatiale d'exécution

2.3 Schéma de contre-mesures contre les attaques en fautes

2.3.1 Schémas de détection de fautes

La détection de faute peut se faire en constatant des incohérences dans les résultats des opérations effectuées, ou à travers de détecteurs physiques sensibles aux effets provoqués par certains moyens d'injection de fautes.

2.3.1.1 Détecteurs physiques

Les détecteurs physiques visent à détecter les différentes formes de perturbations fréquemment utilisées pour injecter une faute [43, 107]. On peut notamment citer :

- *Détecteurs de tension d'alimentation* qui détectent si la tension d'alimentation est au-delà de l'intervalle de tension prévu par le fabricant pour un fonctionnement normal du circuit.
- *Détecteurs de luminosité* qui utilisent des cellules photoélectriques sensibles à la lumière environnante.
- *Détecteurs de fréquence* qui se déclenchent lorsque la fréquence d'horloge varie anormalement.

2.3.1.2 Bouclier métallique (*shield*)

Il s'agit d'une couche métallique fine couvrant la surface du circuit, à l'intérieur de laquelle un signal circule en continu. Lorsque ce signal est interrompu, le circuit détecte qu'une perturbation a eu lieu [10].

2.3.1.3 Redondance spatiale et temporelle

La technique de redondance consiste à exécuter plusieurs fois les mêmes opérations et à comparer leurs résultats. Si tous les résultats ne sont pas identiques, cela signifie qu'au moins une des exécutions a été perturbée. Cette redondance est dite spatiale lorsque chaque exécution utilise des blocs distincts du circuit. Elle est dite temporelle lorsque c'est le même bloc qui est re-exécuté plusieurs fois [10]. La redondance spatiale, illustrée par la figure 2.5, nécessite plus d'espace et la redondance temporelle, illustrée par la figure 2.6, nécessite plus de temps de calcul.

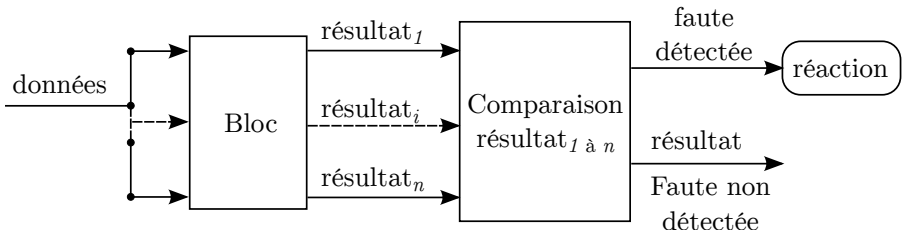


FIGURE 2.6 – Redondance temporelle d'exécution

2.3.1.4 Codes de détection d'erreurs

Cette technique est inspirée des procédés de contrôle et de correction d'erreur, tel que le *Cyclic Redundancy Check* (CRC) souvent utilisé dans les transferts de données d'un support de stockage vers un autre. Elle vise à détecter l'altération d'une entité du programme. Une entité peut être une donnée, un bloc de données, une instruction ou un bloc d'instructions [78].

Cette technique consiste à associer au préalable un code à chaque entité du programme. Le code d'une entité peut être son haché ou son bit de parité [13]. À chaque fois qu'une entité est manipulée, son code est recalculé pour vérifier son intégrité.

2.3.1.5 Intégrité du flot de contrôle

Le flot de contrôle d'un programme est l'ordre dans lequel ses blocs d'instructions sont exécutés, on parle de *chemins* d'exécution du programme. L'intégrité du flot de contrôle ou *Control Flow Integrity* (CFI), vise à garantir que le chemin d'exécution suivi est conforme à ce qui est prévu dans le flot de contrôle.

Le principe de détection consiste à déterminer à l'avance l'ensemble des chemins d'exécutions possibles que le programme peut emprunter. Puis, lors de l'exécution, il s'agit de s'assurer que l'exécution suit bien l'un de ces chemins.

Plusieurs variantes de CFI ont été proposées, dont certaines implémentations seront présentées à la section 2.4.3.

2.3.2 Schémas de tolérance aux fautes

La tolérance aux fautes est l'aptitude d'un système informatique à accomplir sa fonction malgré la présence de fautes, qu'il s'agisse de dégradations physiques du matériel, des défauts logiciels, d'attaques malveillantes ou d'erreurs d'interaction homme-machine [6].

Moro et al. [76] ont proposé un schéma de tolérance au saut d'instructions. Le schéma est basé sur le principe de redondance d'instructions et sera présenté en détail à la section 2.4.2.

```
1 ldr r4, [r7]
```

(a) Code non protégé

```
1 ldr r4, [r7]
2 ldr r6, [r7]
3 cmp r6, r4
4 bne <error>
```

(b) Code protégé

FIGURE 2.7 – Procédé de duplication d'instructions selon Barenghi et al. [13]

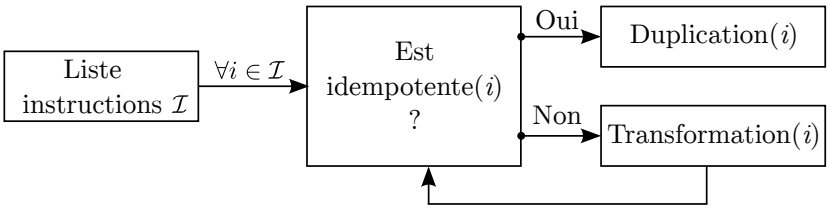


FIGURE 2.8 – Schéma de tolérance

2.4 Exemples d'implémentations logicielles

Cette section détaille quelques exemples d'implémentations logicielles de schémas de contre-mesures. Les implémentations de Moro et al. [76] (schéma de tolérance) et Lalande et al. [58] (schéma de CFI) étant les plus proches des travaux réalisés dans cette thèse, seront décrites plus en détail.

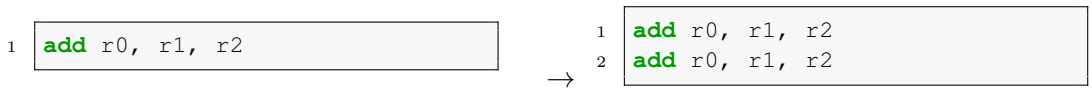
2.4.1 Détection de fautes

Barenghi et al. [13] proposent une implémentation logicielle du schéma de redondance (présenté à la section 2.3.1.3). Leur implémentation, illustrée par la figure 2.7, consiste à dupliquer toutes les instructions assembleur d'un programme, et cible l'architecture ARMv7. Les résultats des instructions dupliquées sont sauvegardés dans des registres différents. Une instruction de comparaison est insérée pour comparer ces résultats et une instruction de branchement conditionnel pour sauter vers une routine `<error>` dans le cas où les résultats ne sont pas identiques.

De Keulenaer et al. [38] proposent une solution similaire à celle de Barenghi et al. [13]. Leur mise en œuvre utilise un outil de réécriture de code binaire appelé *Diablo*.

2.4.2 Tolérance aux fautes

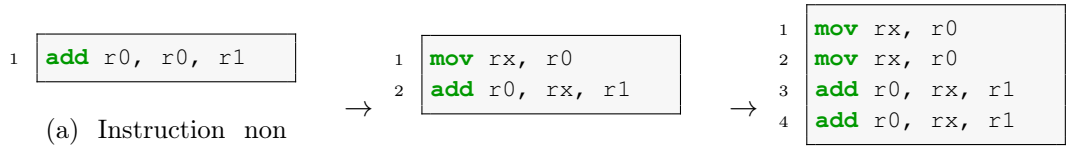
Reis et al. [89] proposent une implémentation d'un schéma de tolérance aux fautes basé sur la redondance d'instructions. Ils mettent en œuvre des techniques permettant d'exploiter les ressources non utilisées du processeur pour amortir le coût de cette redondance. Cependant, leur solution ne s'applique que sur des architectures hautes performances permettant d'exploiter le parallélisme d'instruction, et requiert l'utilisation de plusieurs registres supplémentaires.



(a) Instruction idempotente

(b) Duplication triviale

FIGURE 2.9 – Duplication d'une instruction idempotente



(a) Instruction non idempotente

(b) Transformation

(c) Duplication

FIGURE 2.10 – Duplication d'une instruction non idempotente

Moro et al. [76] ont proposé et formellement vérifié un schéma de tolérance au sauts d'instructions. Ils ont proposé une implémentation de ce schéma au niveau code binaire pour une architecture ARMv7. Le principe, illustré par la figure 2.8, consiste à dupliquer toutes les instructions après les avoir transformées en une forme idempotente. L'idempotence signifie qu'une opération a le même effet lorsqu'on l'effectue une ou plusieurs fois [103]. Un code sécurisé avec ce schéma a 2 exemplaires de la même instruction, dont l'un est la copie (ou duplicata) de l'autre. Ainsi, suite à une faute, si une instruction a été sautée, son duplicata sera exécuté, sinon les deux instructions sont exécutées, et grâce à leur propriété d'idempotence, la fonctionnalité du programme ne sera pas altérée.

Par exemple, l'instruction ARM `add r0, r1, r2`, qui réalise l'opération d'addition $r0 \leftarrow r1 + r2$, est idempotente. Dupliquer une telle instruction est trivial, il suffit de la réécrire à nouveau. La figure 2.9 montre un exemple de duplication d'une instruction idempotente.

Dans le cas des instructions non idempotentes, Moro et al. proposent des schémas pour les transformer en une suite équivalente d'instructions idempotentes. La figure 2.10 illustre la transformation de l'instruction `add`. Les instructions de la figure 2.10b sont idempotentes et équivalentes à l'instruction de la figure 2.10a. La transformation mise en œuvre ici se décline en deux étapes :

1. Trouver un registre supplémentaire (disponible), nommé `rx` sur la figure 2.10.
2. Copier dans `rx` le registre `r0` (le registre qui est à la fois source et destination).

Par conséquent, pour dupliquer l'instruction non idempotente de la figure 2.10a, il faut d'abord la remplacer par les instructions idempotentes de la figure 2.10b, qui seront ensuite dupliquées comme montré par la figure 2.10c. Sur cet exemple, la duplication d'une instruction non idempotente nécessite d'ajouter 3 instructions supplémentaires.

2.4.3 Intégrité du flot de contrôle (CFI)

Des implémentations matérielles de CFI basées sur des processeurs *watchdog* ont été proposées depuis le début des années 80 [5, 66, 67, 46]. Cependant, la première implémentation logicielle du principe de CFI fut publiée par Abadi et al. [1] en 2005. Ils proposent un schéma qui s'applique au code binaire du programme à protéger. Le modèle d'attaque considéré est le cas où un attaquant est capable de rediriger le flot d'exécution du programme vers une zone mémoire qu'il contrôle. Ce schéma de protection consiste à mettre au préalable dans une liste blanche les adresses de toutes les destinations de sauts valides du programme. À l'exécution, il s'agit de vérifier que l'adresse de destination de chaque saut pris est bien valide.

Lalande et al. [58] ont proposé un schéma de CFI permettant de détecter des détournements de flot de contrôle au sein d'un programme. Le schéma consiste à initialiser un compteur au début de chaque bloc d'instructions, puis après chaque instruction, à incrémenter et à vérifier la validité du compteur à ce niveau. La vérification se fait en comparant la valeur courante du compteur à sa valeur attendue. La valeur attendue d'un compteur se calcule, statiquement, en additionnant sa valeur d'initialisation au nombre d'incrémentations réalisées. Leur implémentation est appliquée dans le code source C du programme à protéger.

Danger et al. [37] proposent une solution logicielle/matérielle, appelée HCODE, permettant de garantir la validité du chemin d'exécution d'un programme. Leur solution consiste à calculer une signature pour chaque bloc de base et à associer à cette signature les adresses des successeurs de ce bloc. Ces informations sont stockées dans une nouvelle section du fichier binaire sécurisé (`.HCODE`). Un module matériel, placé entre le processeur et le cache d'instructions, permet de vérifier la validité du chemin d'exécution, en recalculant la signature des blocs de base exécutés et en comparant cette signature à celle pré-calculée.

D'autres implémentations de CFI seront présentées dans le chapitre suivant, en section 3.3.3.2, en raison de leur lien fort avec la compilation.

2.4.4 Limitations

Dans cette section, nous discutons des limitations des mises en œuvres logicielles présentées ci-haut.

Le schéma de détection de fautes mis en œuvre par Barenghi et al. [13] dans le code binaire du programme à sécuriser nécessite l'utilisation de plusieurs registres supplémentaires. Une des limitations de cette solution est la nécessité de disposer de registres disponibles. Les auteurs précisent que dans leur cas d'utilisation, le programme cible (AES) n'utilise pas tous les registres du processeur, leur laissant donc quelques registres disponibles et utilisables pour le schéma de détection. Cela rend cette solution applicable au cas par cas (solution *ad-hoc*). Une autre limitation de cette solution est qu'elle ne couvre pas tout le jeu d'instructions de l'architecture cible (ARMv7m). En effet, certaines instructions telles que les appels de fonction (`bl, blx`, etc.) ne peuvent pas être dupliquées de cette façon, car on ne peut pas faire

l'hypothèse qu'appeler une fonction deux fois de suite retourne le même résultat.

Le schéma de détection de fautes mis en œuvre par De Keulenaer et al. [38] dans le code binaire se confronte à la même limitation que Barenghi et al. [13] concernant la nécessité de disposer de registres disponibles. Dans le cas où il n'y aurait aucun registre disponible, les auteurs proposent d'en libérer certains, en sauvegardant leurs contenus dans la pile (mémoire) à travers l'instruction `push`, puis les restaurer après utilisation avec l'instruction `pop`. Ces accès répétés à la mémoire impactent considérablement le temps d'exécution du programme.

Le schéma de tolérance au saut d'instructions mis en œuvre par Moro et al. [76] dans le code binaire nécessite plusieurs transformations qui impactent le temps d'exécution et la taille du code sécurisé. En effet, chacune de ces transformations proposées par les auteurs nécessite l'ajout d'au moins une nouvelle instruction et l'utilisation d'au moins un registre supplémentaire. Les auteurs proposent d'utiliser le registre `r12` qui est le *scratch register*[2] sur les architectures ARM 32 bits. Cette solution n'est pas envisageable lorsque plusieurs registres supplémentaires sont nécessaires. Les auteurs rapportent aussi que la transformation de l'instruction `umlal` a nécessité l'ajout de 6 instructions, c'est-à-dire que dans le code sécurisé, l'instruction `umlal` est remplacée par 14 instructions.

Par ailleurs, Rivière et al. [91] et Yuce et al. [106] ont mis en œuvre des attaques permettant de sauter, en même temps, plusieurs instructions d'un programme. Ils ont montré que le schéma de tolérance de Moro et al. [76] n'est pas robuste face à ces attaques.

Le schéma CFI mis en œuvre par Lalande et al. [58] dans le code source du programme à sécuriser ne permet de garantir la conservation des propriétés de sécurité dans le code binaire que si les optimisations du compilateur sont désactivées. Or, la désactivation des optimisations lors de la compilation engendre un code binaire moins performant à l'exécution. Les auteurs rapportent un coût de +400% pour une implémentation de AES sur un microcontrôleur basé sur le processeur ARM Cotex-M3.

Nous proposons, dans les chapitres 5 et 6, une mise en œuvre du schéma de tolérance au saut d'instructions de Moro et al. [76] permettant de : (1) résoudre les problèmes rencontrés par l'approche code binaire, (2) automatiser son application, (3) réduire l'impact du schéma sur le temps d'exécution et la taille du code, et (4) renforcer son niveau de sécurité pour résister aux scénarios d'attaques mis en œuvre par Rivière et al. [91] et Yuce et al. [106].

Nous proposons, dans le chapitre 7, une mise en œuvre d'un schéma CFI permettant de garantir le chemin d'exécution d'un programme à l'échelle d'une instruction assembleur. Cette mise en œuvre, contrairement à celle proposée par Lalande et al. [58], offre une granularité de détection plus fine, car une instruction C peut correspondre à plusieurs instructions assembleur voir même plusieurs blocs d'instructions.

2. Un *Scratch register* est un registre qui ne fait pas partie du contexte d'un programme, et qui peut être réécrit à tout moment.

2.5 Conclusion

Dans ce chapitre nous avons présenté l'état de l'art des attaques physiques et plus particulièrement celui des attaques en fautes. Nous avons passé en revue les modèles et les moyens d'injections de fautes couramment utilisés dans l'état de l'art, ainsi que les schémas de contre-mesures associés. Nous avons présenté des exemples de mises en œuvre logicielles de schémas de contre-mesures contre les attaques en fautes ainsi que leurs limitations.

Dans le chapitre suivant nous présenterons les principes généraux de la compilation, avec un focus sur le compilateur LLVM. Nous aborderons ensuite les problématiques de la compilation pour la sécurité, puis nous terminerons avec quelques exemples de contre-mesures mises en œuvre par une approche *compilation*.

Chapitre 3

Compilation

Sommaire

3.1 Principes généraux 28
3.1.1 Partie frontale (*Front-end*) 28
3.1.2 Partie intermédiaire (*Middle-end*) 30
3.1.3 Partie arrière (*Back-end*) 30
3.2 Le compilateur LLVM 32
3.2.1 Clang 32
3.2.2 Passes de compilation LLVM 33
3.2.3 Types de représentations d'un programme 34
3.3 Compilation pour la sécurité 39
3.3.1 Compromis optimisation et sécurisation de code 39
3.3.2 Gestion du compromis 41
3.3.3 Sécurisation par compilation 42
3.4 Conclusion 44

Ce chapitre commence par une présentation des principes généraux de la compilation avec un focus particulier sur le compilateur LLVM que nous utilisons dans les travaux réalisés dans cette thèse.

La section 3.3 aborde les problématiques de la compilation pour la sécurité et illustre les difficultés pour faire coexister optimisation de code et propriétés de sécurité dans un compilateur à usage général (*general purpose compiler*). Nous montrons que les alternatives à l'approche compilation, c'est-à-dire les approches code source et code binaire, limitent les possibilités d'optimisations du code.

Enfin, nous présentons des exemples de l'état de l'art utilisant l'approche compilation pour l'application de protections logicielles.

3.1 Principes généraux

Un compilateur est un programme informatique qui lit un programme écrit dans un langage dit langage source, et le traduit en un programme équivalent dans un autre langage dit *langage cible* [4]. Souvent, cette traduction se fait d'un langage *haut niveau*, facilement manipulable par un développeur, vers un langage *bas niveau*, destiné à être exécuté sur une architecture matérielle. Un compilateur est considéré comme un pont reliant l'ingénierie logicielle à l'ingénierie matérielle. La plupart des compilateurs, outre leur tâche de traduction, mettent en œuvre des mécanismes supplémentaires pour optimiser le programme. Le programme qui fait la traduction inverse est appelé *décompilateur*.

La figure 3.1 illustre la structure interne de la majeure partie des compilateurs contemporains. Ils sont structurés en trois grandes parties : frontale ou *front-end*, intermédiaire ou *middle-end* et arrière ou *back-end*. Chaque partie est composée d'une ou plusieurs phases et chaque phase est constituée d'une série de modules d'analyses et de transformations appelés *passes* de compilation.

3.1.1 Partie frontale (*Front-end*)

C'est le point d'entrée du processus de compilation. Les différentes phases qui composent cette partie peuvent varier d'un compilateur à un autre. De manière générale, on retrouve les phases suivantes : analyse lexicale, analyse syntaxique, vérification de types et génération d'une représentation intermédiaire du programme.

3.1.1.1 Analyse lexicale

L'analyse lexicale consiste à lire le texte du code source caractère par caractère, pour reconnaître les symboles qui constituent le programme. Dans le jargon de la compilation, ces symboles sont appelés des *tokens* ou *lexèmes*. Un token c'est par exemple une valeur numérique, le nom d'une variable, un point-virgule ou une accolade. Cette étape transforme le texte du programme source en une liste ordonnée de tokens.

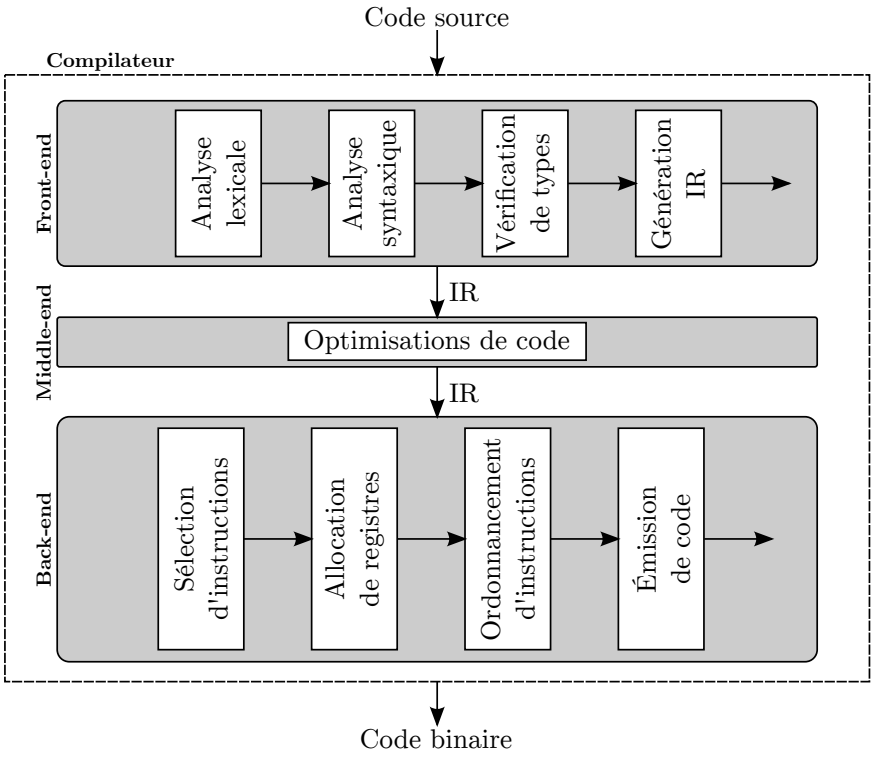


FIGURE 3.1 – Structure interne d'un compilateur

3.1.1.2 Analyse syntaxique

Cette phase construit un arbre syntaxique ou *Abstract Syntax Tree* (AST) à partir de la liste de tokens produite par l'analyse lexicale. Cette nouvelle représentation, illustrée par la figure 3.2, permet entre autres de facilement vérifier certaines propriétés sur le code source, comme par exemple qu'un opérateur binaire ($+, -, *$, ...) a bien deux opérandes.

3.1.1.3 Vérification des types

Le but de cette phase est de s'assurer que le programme ne viole pas les contraintes de typage imposées par le langage de programmation source. Par exemple, qu'une variable de type booléen ne puisse pas être utilisée en tant que pointeur, ou que la déclaration d'une variable précède bien son utilisation.

3.1.1.4 Génération d'une représentation intermédiaire

L'objectif de cette phase est de transformer le programme dans une représentation dite intermédiaire, qu'on notera IR pour *Intermediate Representation*. Cette transformation marque la fin de la partie frontale, et l'entrée dans la partie intermédiaire.

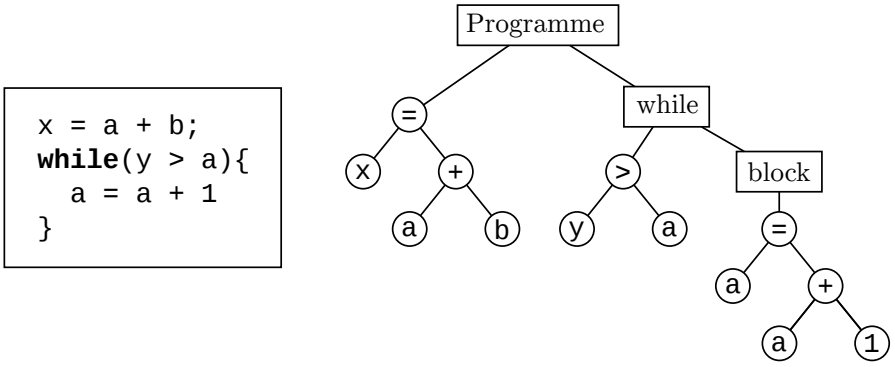


FIGURE 3.2 – Transformation d'un programme en arbre syntaxique (AST)

3.1.2 Partie intermédiaire (*Middle-end*)

Cette partie prend en entrée le programme dans sa forme IR, sur laquelle sont exécutées la majeure partie des passes d'optimisations du compilateur. L'avantage principal de la représentation IR est son indépendance vis-à-vis des langages sources et des architectures matérielles. Toutes les passes d'analyses et de transformations qui opèrent sur cette représentation sont communes à tous les langages de programmation source et toutes les architectures supportées par le compilateur. Les noms des passes d'optimisations et leurs implémentations diffèrent d'un compilateur à un autre, mais leurs objectifs restent identiques : réduire le temps d'exécution, l'empreinte mémoire et la taille du programme. Quelques exemples de passes du compilateur LLVM sont décrits en section 3.2.2.

3.1.3 Partie arrière (*Back-end*)

La partie arrière prend en entrée le programme dans sa forme IR et produit en sortie le programme dans sa forme définitive. C'est dans cette partie que se trouvent toutes les informations nécessaires pour générer et optimiser un code qui cible une architecture donnée. Cette partie se déroule en quatre phases.

3.1.3.1 Phase sélection d'instructions

L'objectif de cette phase est de sélectionner, pour chaque opération décrite dans le programme source, une manière de la réaliser sur l'architecture cible, c'est-à-dire, à l'aide d'une ou de plusieurs instructions propres à cette architecture. On appellera ces instructions des *instructions machine*. Il peut y avoir, pour une opération donnée, diverses manières de l'exprimer avec des instructions machine. Par exemple une opération de multiplication par une puissance de 2 peut être réalisée avec une instruction de multiplication `mul` ou avec une instruction de décalage de bit `shift`. Il appartient donc à cette phase de déterminer, en fonction des informations dont dispose le compilateur sur l'architecture cible, l'instruction ou la suite d'instructions machines à sélectionner. Ces informations sont par exemple : la présence d'une

unité de calcul à virgule flottante (FPU), les jeux d'instructions supportés par le processeur, la latence et la taille des instructions.

3.1.3.2 Phase d'allocation de registres

Le rôle de cette phase est d'affecter pour chaque variable du programme, un registre physique du processeur cible. La difficulté de cette tâche réside dans le fait que le nombre de variables d'un programme n'est pas borné, alors que le nombre de registres physiques d'un processeur est très limité (c'est un problème NP-complet). Une des méthodes employée pour résoudre ce problème utilise le principe de coloriage de graphe. L'idée consiste à réduire le problème d'allocation de registres à un problème de $\mathcal{K}$-coloriage, où $\mathcal{K}$ est le nombre de registres disponibles dans le processeur. Cette méthode est connue pour produire une allocation optimale mais nécessite un temps de calcul relativement élevé. Dans les compilateurs dynamiques (où le code est généré au moment de l'exécution) on privilégie d'autres méthodes moins optimales mais moins coûteuses en temps de calcul, comme la méthode *Linear Scan* [83].

Une variable est dite *vivante* en un point donné du programme lorsqu'à partir de ce point, la variable sera lue au moins une fois avant d'être réécrite ou avant la fin du programme. Lorsque le nombre de variables vivantes en un point est supérieur au nombre de registres disponibles, on libère temporairement certains registres en déplaçant leur contenu dans la mémoire. Dans le jargon de la compilation, on appelle cette opération du *register spilling*.

3.1.3.3 Phase d'ordonnancement des instructions

L'ordonnancement des instructions ou *Instruction Scheduling* vise à réorganiser l'ordre d'exécution des instructions dans le but d'améliorer le temps d'exécution global du programme tout en préservant sa sémantique. L'amélioration du temps d'exécution se fait grâce à l'exploitation optimisée du pipeline du processeur afin d'éviter, au mieux, des attentes inutiles (*Pipeline Stalls*). Le processus d'ordonnancement prend en compte les paramètres suivants :

1. Les dépendances entre les données manipulées par les instructions.
2. La latence des instructions [105].
3. Le niveau de parallélisme ou *Instruction-level parallelism* (ILP), c'est-à-dire le nombre d'instructions que le processeur est capable d'exécuter simultanément.

3.1.3.4 Phase émission de code

Cette phase commence par assembler tous les blocs du programme en un seul. Ensuite, elle génère un code dans l'un des formats suivants :

— *Objet :* qui peut être un binaire exécutable (`.exe, .elf, .dmg,` etc.), ou un code objet (`.o`) destiné à être lié à d'autres codes objets. Le code objet est

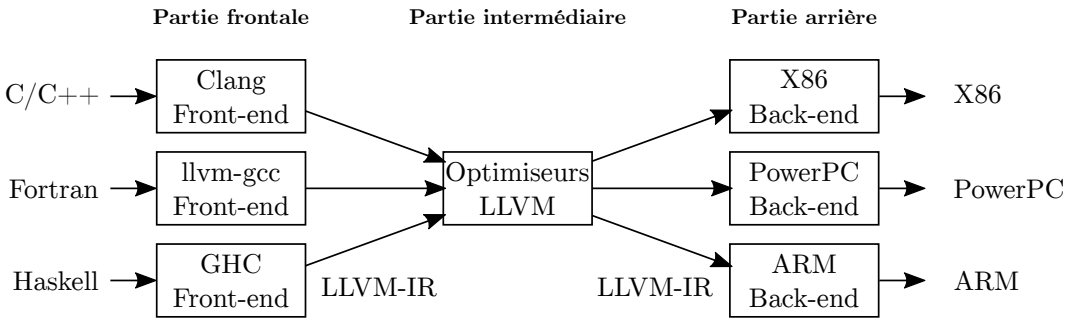


FIGURE 3.3 – Aperçu de la structure du compilateur LLVM

généré dans un format accepté par les systèmes d'exploitation cible (PE pour Windows, ELF pour la majeure partie des systèmes Unix, etc.).

— *Assembleur :* qui est une représentation textuelle des instructions machine.

3.2 Le compilateur LLVM

LLVM [60] est une infrastructure de compilation constitué d'une collection de modules réutilisables permettant de développer des compilateurs et des outils d'analyse et de génération de code. Malgré sa signification initiale de *Low Level Virtual Machine*, LLVM a aujourd'hui très peu à voir avec une machine virtuelle traditionnelle. LLVM est un projet open source écrit en C++ et dont la première version a été publiée en 2003, avec une licence dérivée de celle de BSD. Grâce à sa licence permissive permettant de développer des extensions sans être obligé de les partager, et grâce sa structure interne modulaire, LLVM est aujourd'hui le compilateur de production de plusieurs industriels, tels que : *Apple, Samsung, Sony Interactive Entertainment, Sun Microsystems* et *NVIDIA*. Il est aussi largement utilisé dans la recherche académique notamment pour le gain de temps dans le prototypage de nouveaux concepts [63].

L'une des particularités de LLVM est la séparation claire entre les parties frontale, intermédiaire et arrière. Comme illustré par la figure 3.3, il n'y a pas de lien direct entre les modules de la partie frontale (prenant en charge le programme source) et ceux de la partie arrière (chargés de générer un code exécutable pour une architecture donnée). Cette séparation facilite le support d'un nouveau langage. Par exemple, pour développer un compilateur pour le langage GO [50], il suffit de développer une partie frontale, car LLVM dispose déjà des parties restantes. De manière analogue, pour supporter une nouvelle architecture matérielle, il suffit de développer une partie arrière.

3.2.1 Clang

Clang est le front-end natif de LLVM, capable de compiler les langages de la famille C (C, C++ et Objective-C). On parle souvent du compilateur Clang/LLVM

pour insister sur le fait qu'il est basé sur l'infrastructure LLVM. C'est un compilateur open source avec la même licence que LLVM, dont le but est de proposer une alternative au *GNU Compiler Collecion* (GCC). En 2016, les principaux contributeurs de Clang étaient : *Apple, Microsoft, Google, ARM, Sony, Intel* et *AMD*. Dans certaines configurations, le temps de compilation avec Clang est meilleur que celui de GCC [71].

3.2.2 Passes de compilation LLVM

Soit R la représentation d'un programme à l'entrée d'une passe et R' sa représentation à la sortie de cette passe (les différents types de représentations d'un programme seront présentés dans la section 3.2.3). Dans LLVM, on distingue trois catégories de passes.

— *Passes d'analyse* : leur rôle est de collecter des informations sur le programme, destinées à être utilisées par d'autres passes. Cette catégorie de passe ne modifie pas le programme et R' est identique à R.

— *Passes de transformation* : elles utilisent les informations produites par les passes d'analyses pour transformer le programme. Il est parfois nécessaire entre deux passes de transformations, de ré-exécuter une passe d'analyse pour rafraîchir les informations. R' peut être différent de R.

— *Passes utilitaires* : ce sont toutes les passes qui ne rentrent pas dans les deux premières catégories. Par exemple, une passe qui affiche des informations de débogue.

LLVM dispose d'un gestionnaire de passe (*LLVM Pass Manager*) dont le rôle est d'orchestrer l'exécution de toutes les passes. Chaque passe doit déclarer sa zone d'intervention, qui peut être sur une fonction (`llvm::functionPass`) ou sur tout le programme (`llvm::modulePass`). Chaque passe doit également déclarer à quel moment de la compilation elle intervient, et enfin, si elle modifie ou non le programme. L'ensemble de ces éléments permet au gestionnaire de passes de mieux organiser leur l'ordre d'exécution, notamment afin d'éviter autant que possible de devoir ré-exécuter des passes d'analyses pour rafraîchir l'information suite à une passe de transformation. Une meilleure organisation permet de réduire le temps de compilation.

3.2.2.1 Exemples de passes d'optimisation

Cette section présente quelques exemples de passes d'optimisation de LLVM. La liste complète des passes est disponible sur la page de documentation officielle [64].

-loop-unswitch Cette passe opère sur les boucles. Elle transforme une boucle contenant des blocs conditionnels en des blocs conditionnels contenant des boucles. Une illustration du fonctionnement de cette passe est présentée par la figure 3.4.

```
for (...)
    A;
    if (cond)
        B;
    C;
```

(a) Avant optimisation

```
if (cond)
    for (...)
        A; B; C;
else
    for (...)
        A; C;
```

(b) Après optimisation

FIGURE 3.4 – illustration de la passe d'optimisation `-loop-unswitch`

```
for(i=7; i*i < 1000; ++i){
...
}
```

(a) Avant optimisation

```
for(i = 0; i != 25; ++i){
...
}
```

(b) Après optimisation

FIGURE 3.5 – illustration de la passe d'optimisation `-indvars`

-indvars Cette passe opère aussi sur les boucles. Elle vise à simplifier la variable d'induction d'une boucle (variable dont la valeur suit une progression linéaire). La figure 3.5 illustre le fonctionnement de cette passe. Après la simplification de la variable d'induction, toutes les instructions qui dépendent de cette variable sont mises à jour pour tenir compte des modifications.

-instcombine Cette passe opère sur les blocs de base. Son rôle est de simplifier et combiner les opérations arithmétiques. La figure 3.6 montre un exemple de simplification réalisée par cette passe.

3.2.3 Types de représentations d'un programme

Dans un processus de compilation, un programme prend plusieurs types de représentations avant sa représentation définitive. Chacune d'entre elles est destinée à faciliter certaines passes d'analyses ou de transformations.

3.2.3.1 Code source

Le code source est le pont entre l'utilisateur et le compilateur. Il permet à l'utilisateur de décrire son algorithme dans un langage de programmation de haut ni-

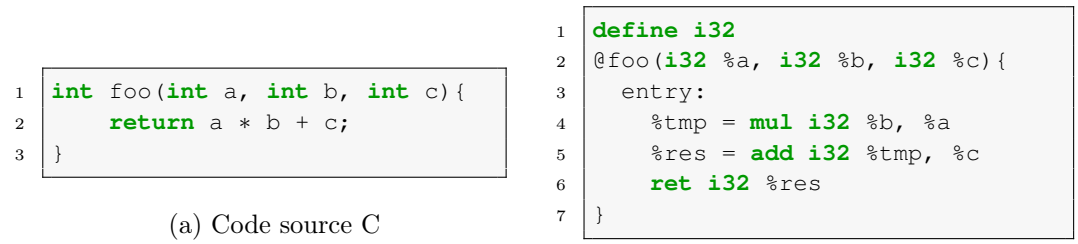
```
y = x ^ r
z = y ^ r
```

(a) Avant optimisation

```
z = x
```

(b) Après optimisation

FIGURE 3.6 – illustration de la passe d'optimisation `-instcombine`



```c
int foo(int a, int b, int c){
    return a * b + c;
}
```

(a) Code source C

```
define i32
@foo(i32 %a, i32 %b, i32 %c){
  entry:
    %tmp = mul i32 %b, %a
    %res = add i32 %tmp, %c
    ret i32 %res
}
```

(b) LLVM-IR

FIGURE 3.7 – Illustration de la représentation intermédiaire LLVM-IR

veau (proche du langage naturel), et au compilateur de saisir cette description sans ambiguïté. Cette représentation n'est manipulée que dans la partie frontale du compilateur.

3.2.3.2 Représentation intermédiaire

Dans LLVM, cette représentation est appelée LLVM byte-code ou LLVM-IR [60]. Elle est indépendante du langage source et de l'architecture matérielle cible. Elle permet au compilateur de factoriser la majeure partie de ses passes d'optimisations. La syntaxe de LLVM-IR est très proche de celle d'un assembleur sur trois adresses, à la différence qu'elle est enrichie par des informations sur la sémantique du programme, notamment les types des données. La figure 3.7 montre une représentation en LLVM-IR d'un code source C. Dans LLVM-IR, les variables sont appelées des *registres virtuels*, et elles sont représentées dans leurs formes SSA (*Static Single Assignment*) [35]. Celle-ci autorise une seule affectation par registre virtuel. En revanche le nombre de registres virtuels n'est pas borné. La forme SSA permet d'améliorer les résultats de certaines passes d'optimisations en simplifiant les propriétés des variables. Notamment, il devient trivial de faire certaines analyses telles que, identifier où commence et où se termine l'utilisation d'une variable, ou supprimer toutes les instructions qui modifient des variables qui n'ont aucune interaction avec le résultat final.

3.2.3.3 Graphe de flot de contrôle (CFG)

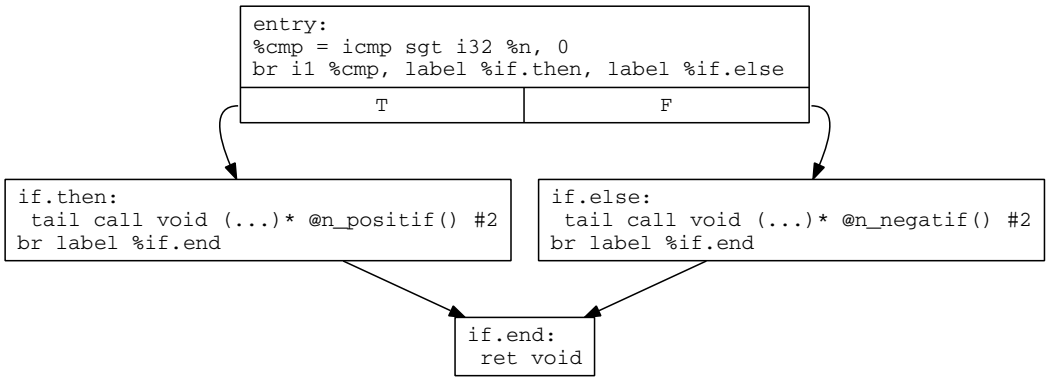
Un graphe de flot de contrôle ou *control flow graph* (CFG) est un graphe orienté utilisé par le compilateur pour représenter tous les chemins d'exécution qui peuvent être suivis par un programme. Un CFG correspond à une fonction dans le programme, les nœuds du CFG représentent les blocs de base de la fonction et les arêtes représentent les sauts dans le CFG.

Bloc de base (*Basic block*) : est un bloc composé d'une à plusieurs instructions ayant les propriétés suivantes :

— comporte un seul point d'entrée, qui est la première instruction du bloc.

```c
void toto(int n){
    if ( n > 0 )
        n_positif();
    else
        n_negatif();
}
```

(a) Code source C



(b) Graphe de de flot contrôle de la fonction `toto` dans la partie intermédiaire

FIGURE 3.8 – Représentation d'un programme en graphe de flot de contrôle

— comporte un seul point de sortie, qui est la dernière instruction du bloc.

— après la première instruction, toutes les instructions du bloc doivent être exécutées successivement. De manière plus formelle, chaque instruction à une position i *domine* (c'est-à-dire s'exécute avant) toute instruction à une position j tel que $(i < j)$.

Notion de dominance On dit qu'un bloc A *domine* un bloc B lorsque tous les chemins pour arriver à B passent par A. Par conséquent, le bloc d'entrée domine tous les blocs du CFG. On dit également que A *domine immédiatement* B lorsque A domine B et qu'il n'existe aucun bloc *intermédiaire* X tel que A domine X et X domine B.

Notion de précédence et de succession Un bloc A est un *prédécesseur* d'un bloc B lorsqu'il existe une arête allant de A vers B. On dit aussi que B est un *successeur* de A. Lorsque A domine immédiatement B alors A est un prédécesseur de B.

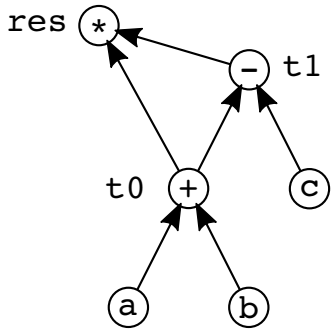
Un CFG comprend deux blocs de base spéciaux : un bloc d'entrée nommé `entry` par lequel on entre dans le CFG, et un bloc de sortie par lequel on le quitte. Ces deux blocs sont confondus pour un CFG ne possédant qu'un seul bloc de base.

```c
int foo(int a, int b, int c){
  return (a+b)*(a+b-c);
}
```

(a) Code source C

```
define i32
@foo(i32 %a, i32 %b, i32 %c){
  entry:
    %t0  = add i32 %a, %b
    %t1  = sub i32 %t0, %c
    %res = mul i32 %t0, %t1
    ret i32 %res
}
```

(b) LLVM-IR



(c) DAG représentant le bloc de base `entry`

FIGURE 3.9 – Représentation d'un programme par un DAG

La représentation par CFG est utilisée dans les parties intermédiaire et arrière du compilateur. La figure 3.8 illustre le CFG d'un programme durant la partie intermédiaire. Le nom de chaque bloc de base est inscrit sur sa première ligne. Cette représentation permet d'analyser la structure d'un programme. Par exemple, lors d'un parcours en profondeur, lorsque le successeur d'un bloc de base est visité avant lui, alors on sait qu'on a à faire à une boucle. Les propriétés d'un CFG permettent aussi de réaliser plusieurs optimisations sur le programme à l'échelle d'une fonction (optimisation intra-procédurale). Notamment, tout bloc de base différent du bloc `entry` et qui n'est pas dominé par ce dernier ne pourra pas être atteint durant l'exécution, et peut donc être considéré comme du code mort à supprimer sans risque d'altérer la fonctionnalité du programme.

3.2.3.4 Graphe orienté acyclique (DAG)

Un DAG ou *Directed Acyclic Graph* est un graphe dans lequel un nœud n'est jamais visité plus d'une fois lors d'un parcours en profondeur [28]. Le DAG est utilisé par le compilateur pour représenter les instructions dans un bloc de base. Dans un DAG, les nœuds périphériques sont ceux qui n'ont aucune flèche entrante, ils représentent les opérandes des instructions. Les nœuds internes représentent les opérateurs. La Figure 3.9 montre une représentation DAG d'un bloc de base.

Le DAG permet de suivre la propagation d'une donnée le long des instructions, et facilite certaines formes d'optimisations qui s'appliquent à l'échelle d'un bloc de

```
$ llc -print-machineinstrs cfg.bc -march=thumb -mattr=thumb2
# After Instruction Selection:
# Machine code for function foo: SSA
Function Live Ins: %R0 in %vreg0, %R1 in %vreg1, %R2 in %vreg2

BB#0: derived from LLVM BB %entry
Live Ins: %R0 %R1 %R2
%vreg2<def> = COPY %R2; rGPR:%vreg2
%vreg1<def> = COPY %R1; GPRnopc:%vreg1
%vreg0<def> = COPY %R0; rGPR:%vreg0
%vreg3<def> = t2ADDrr %vreg1, %vreg0, pred:14, pred:%noreg, opt:%noreg;↩
    rGPR:%vreg3,%vreg0 GPRnopc:%vreg1
%vreg4<def> = t2SUBrr %vreg3, %vreg2, pred:14, pred:%noreg, opt:%noreg;↩
    rGPR:%vreg4,%vreg3,%vreg2
%vreg5<def> = t2MUL %vreg4<kill>, %vreg3, pred:14, pred:%noreg; rGPR:%↩
   vreg5,%vreg4,%vreg3
%R0<def> = COPY %vreg5; rGPR:%vreg5
tBX_RET pred:14, pred:%noreg, %R0<imp-use>

# End machine code for function foo.
```

FIGURE 3.10 – Représentation de la fonction `foo` en `MachineInstr` juste après l'étape sélection d'instructions

base. Parmi ces optimisations, on peut citer le *constant folding* dont le rôle est de calculer statiquement le résultat des opérations impliquant des constantes connues au moment de la compilation. Par exemple, sur la figure 3.9, si les valeurs des variables `a` et `b` peuvent être connues au moment de la compilation, le nœud `t0` sera remplacé par le résultat de l'opération `a + b`.

Le DAG est construit dans la partie arrière du compilateur durant la phase sélection d'instructions. Le sélecteur d'instruction se base sur le DAG pour reconnaître des motifs (*patterns*) d'opérations pour lesquelles l'architecture cible dispose des instructions pour les réaliser.

3.2.3.5 Instructions machine

La forme instruction machine (`llvm::MachineInstr`) est une représentation proche de celle de l'architecture cible. Elle est générée à l'issue de la sélection d'instructions.

La figure 3.10 montre la représentation en `MachineInstr` de la fonction `foo` déjà présentée dans la figure 3.9. L'architecture cible ici est ARMv7 avec les jeux d'instructions Thumb1 (T1) et Thumb2 (T2). Les instructions possèdent déjà plusieurs caractéristiques propres à cette architecture. Le symbole `t2ADDrr` désigne la version T2 de l'instruction `add` prenant deux registres comme opérandes (`rr`). Bien que les registres n'aient pas encore été alloués, le compilateur anticipe l'allocation de certains registres (`r0, r1` et `r2`) en se conformant à l'interface binaire-programme ou *Application Binary Interface* (ABI) qui définit l'utilisation des registres `r0-r3` pour

```
1 @ BB#0:                                 @ %entry
2 add     r0, r1
3 subs    r1, r0, r2
4 muls    r0, r1, r0
5 bx      lr
```

FIGURE 3.11 – Représentation de la fonction `foo` en `assembleur`

passer les arguments d'une fonction. Dans le cas des applications ARM embarquées, l'ABI est le standard *Embedded-Application Binary Interface* (EABI) [7].

3.2.3.6 Instructions assembleur

L'assembleur est la dernière forme de représentation *lisible* d'un programme avant le code binaire. La figure 3.11 montre la représentation en assembleur de la fonction `foo` utilisée précédemment.

3.3 Compilation pour la sécurité

Les principaux objectifs d'un compilateur à usage général sont la génération d'un code (exécutable) fonctionnel et l'optimisation de ce code. La compilation pour la sécurité ajoute un troisième objectif, qui est la prise en compte de la sécurité du code généré vis-à-vis des attaques logicielles et matérielles.

Sécuriser un programme contre une attaque consiste à lui ajouter des propriétés de sécurité permettant de résister à cette attaque. Ces propriétés ne doivent pas modifier la fonctionnalité du programme. Ce sont donc des propriétés non fonctionnelles alors qu'optimiser un programme consiste à lui ôter justement toute propriété non fonctionnelle. Le challenge de la compilation pour la sécurité, détaillé ci-après, est d'arriver à faire coexister, dans un compilateur à usage général, l'optimisation de code et la sécurité.

3.3.1 Compromis optimisation et sécurisation de code

La question de savoir jusqu'à quel point faire confiance à un compilateur dans la génération d'un code exécutable fidèle aux spécifications du code source, a été très tôt soulevée par Kenneth Thompson (créateur du système Unix) [98]. Cette question n'est toujours pas tranchée et le débat persiste encore au sein de la communauté de la compilation, notamment sur les points suivants : (1) est-ce au compilateur de garantir des propriétés de sécurité présentes dans un programme ? (2) Est-ce au développeur de comprendre l'impact des optimisations du compilateur sur son code ? (3) Et enfin, est-ce que la non préservation des propriétés de sécurité d'un programme est considérée comme un bogue du compilateur ?

De graves problèmes de sécurité causés par les optimisations du compilateur ont déjà été recensés [61, 88, 27]. L'un des exemples les plus connus est décrit par le

TABLE 3.1 – Schémas de contre-mesure et les passes d'optimisations susceptibles de les altérer

Schéma de contre-mesure	**Passe d'optimisation**
Masquage (section 2.1.1.2)	Simplifications d'opérations arithmétiques `-instcombine` : *instruction combine*
Ajout de bruit (section 2.1.1.1) Redondance d'instructions (section 2.3.1.3)	Passes de suppression de code mort - `dce` : *dead code elemination* - `adce` : *aggressive dce* - `globaldce` : *global dce* - `dse` : *dead store elimination* - `gvn` : *global value numbering*
Ordonnancement aléatoire des exécutions (section 2.1.1.1)	Ordonnancement des exécutions *instruction scheduling*

Common Weakness Enumeration (CWE) [34] et est illustré par l'extrait de code ci-dessous.

```
1 crypt() {
2     key = getKey();       // reception de la clé de chiffrement
3     msg = getMessage();   // reception du message à chiffrer
4     ...
5     Encrypt(msg, key);    // chiffrer le message msg avec la clé key
6     ...
7     key = 0x00;           // effacer la clé
8 }
```

La fonction `crypt()` reçoit le la clé de chiffrement (`key`) et le message à chiffrer (`msg`), puis effectue le chiffrement (ligne 5). Pour éviter que la clé persiste dans la mémoire, la fonction efface la clé (en remplaçant sa valeur par `0x00`) avant de terminer. Lors de la compilation avec optimisation, la passe *dead store elimination* (DSE) supprime la dernière instruction (ligne 7) car elle considère inutile l'écriture d'une variable qui ne sera pas lue après. Cette passe d'optimisation ne pose aucun problème d'un point de vue sémantique, et son fonctionnement est formellement prouvé [17, 62]. Cependant, le programme final ne sera plus sécurisé comme espéré. Cet exemple illustre le décalage entre ce que le code source semble décrire et ce qui sera réellement exécuté.

Dans le cas des attaques physiques, plusieurs schémas de contre-mesures sont en contradiction directe avec certaines passes d'optimisation du compilateur. Le tableau 3.1 liste quelques schémas de contre-mesures avec les passes d'optimisations susceptibles de les altérer.

3.3.2 Gestion du compromis

La section précédente illustre le décalage entre ce qu'un code source peut sembler indiquer et ce qui sera réellement exécuté par le processeur. Les sections suivantes présentent des techniques qui permettent de maîtriser ce décalage.

3.3.2.1 Ne pas optimiser le code

Cette solution peut être mise en œuvre deux façons : la première consiste à demander au compilateur, via les options de compilations, de ne pas optimiser le code. Dans la plupart des compilateurs, l'option `-O0` permet de désactiver toutes les optimisations. Cette option entraînera la génération d'un code plus volumineux, qui s'exécutera moins vite et qui occupera plus d'espace mémoire. Pour les langages de programmation C/C++, il est possible d'utiliser des mots clés comme `volatile` [44] pour spécifier au compilateur de ne pas optimiser certaines variables, ce qui, par effet de propagation de dépendances, pourra impacter d'autres parties du code. La seconde façon qui consiste à faire en sorte que le compilateur ne puisse pas faire des optimisations. Cette solution nécessite une connaissance fine du fonctionnement du compilateur. Une façon de procéder est de rendre le code dépendant de données inconnues tout au long du processus de compilation, ce qui n'est pas toujours possible pour un développeur.

3.3.2.2 Intégrer du code assembleur dans le code source

Certains langages de programmation autorisent d'embarquer du code assembleur dans le code source, avec la garantie que le compilateur n'y touchera pas. Cette technique est généralement utilisée pour adresser au processeur des instructions qui ne sont pas possibles à exprimer avec le langage source, comme appeler des services systèmes ou interagir avec un coprocesseur. Il est donc envisageable d'utiliser cette technique pour implémenter les parties de la contre-mesure qui sont susceptibles d'être optimisées par le compilateur. Cette solution pose plusieurs problèmes : (1) ce sera au développeur de gérer manuellement le lien entre les variables du code source et les registres physiques utilisés dans l'assembleur embarqué, (2) le code devient difficilement lisible et par conséquent difficile à maintenir, (3) le code source devient non portable car l'assembleur embarqué doit être celui de l'architecture matérielle cible, (4) il faudra faire face aux limitations de l'approche code binaire présentée ci-dessous.

3.3.2.3 Approche code binaire

L'approche code binaire consiste à intégrer la contre-mesure directement dans le code assembleur de l'application. Il s'agit de compiler normalement le code source pour générer un code binaire, puis de désassembler ce code binaire pour obtenir un code assembleur dans lequel la contre-mesure sera intégrée, et enfin réassembler le code assembleur pour générer un code binaire sécurisé. D'où son appellation parfois d'approche *code assembleur*.

L'inconvénient de cette approche est qu'une bonne partie de l'information sur la sémantique du programme est perdue, car le compilateur estime qu'il n'est plus nécessaire de garder ces informations après avoir généré le code final (binaire). Par exemple, il n'est plus possible de connaître le type initial d'une donnée stockée en mémoire ou dans un registre. Un autre inconvénient de cette approche est lié au fait que pour intégrer une contre-mesure dans le code assembleur, il est souvent nécessaire de transformer certaines instructions ou d'en insérer de nouvelles, ce qui peut nécessiter l'utilisation de registres supplémentaires. Or, selon l'algorithme implémenté, il peut ne plus y avoir de registres disponibles. L'une des solutions proposée dans l'état de l'art consiste à libérer certains registres en sauvegardant leurs valeurs dans la mémoire pour les restaurer plus tard (*register spilling*). Cette solution engendre un surcoût important sur le temps d'exécution global du programme, car les accès mémoire font partie des opérations les plus coûteuses qu'un processeur soit amené à effectuer.

3.3.3 Sécurisation par compilation

La section précédente a montré qu'il était délicat d'utiliser le compilateur pour intégrer des propriétés sécuritaire dans un programme. La section suivante présente cependant, des travaux montrant que bien que délicate cette approche est particulièrement prometteuse pour contrer les attaques physiques.

3.3.3.1 Contre les attaques par observation

Moss et al. [77] ont développé un outil basé sur un compilateur permettant l'application d'un schéma de masquage booléen lors de la compilation. Leur outil prend en entrée un programme écrit dans une DSL spécifique, ce qui peut limiter son utilisation. Toutes les variables du programme sont soit de type `secret` ou `public`. L'outil applique le schéma de masquage sur toutes les variables ayant le type `secret`.

Eldib et al. [45] utilisent la partie frontale du compilateur LLVM pour synthétiser une version masquée et équivalente d'un code source C fourni à l'entrée du compilateur. Leur compilateur produit en sortie un code source masqué sur lequel les auteurs réalisent une vérification formelle du masquage. Le code source sécurisé doit ensuite être compilé sans optimisation pour éviter que les optimiseurs de code n'altèrent le masquage.

Bayrak et al. [14, 15] utilisent un compilateur pour appliquer une protection contre les attaques par analyse de consommation électrique. Les auteurs effectuent en amont une analyse de la consommation électrique de chaque instruction du programme. Ils appliquent ensuite lors de la compilation, la technique du *random prefetch* sur toutes les instructions dont la consommation dépasse un seuil fixé. Cette technique consiste à insérer des instructions factices avant et après les instructions qui dépassent le seuil, pour rendre leur consommation électrique dépendante du bruit ajouté.

Agosta et al. [2] et Couroussé et al. [31] utilisent des techniques de compilation

dynamique pour protéger un programme contre les attaques par analyse de consommation électrique. Les auteurs proposent des outils similaires permettant de faire du polymorphisme de code, lequel consiste à régénérer une version différente mais équivalente du programme à chaque exécution. Cela permet d'avoir une signature électrique différente à chaque exécution, rendant ainsi les attaques par mesure de consommation électrique difficiles à réaliser.

Crane et al. [33] proposent de générer plusieurs variantes, sémantiquement équivalentes, d'un même programme au moment de la compilation, puis de choisir aléatoirement une version à chaque exécution. Cette solution peut être vue comme la version statique de celle présentée dans le paragraphe précédent. Cependant, pour être efficace contre les attaques par mesure de consommation, plusieurs variantes doivent être générées, ce qui rend cette solution très limitée pour les microcontrôleurs à faible ressource.

3.3.3.2 Contre le détournement de flot de contrôle

Les compilateurs LLVM (depuis la version 3.7) et Microsoft Visual Studio 2015 intègrent déjà certaines variantes de CFI qui visent principalement à sécuriser les appels indirects de fonctions. Le principe consiste à identifier et à sauvegarder dans une table toutes les destinations de sauts valides au moment de la compilation. Ainsi, lors de l'exécution, avant chaque appel indirect (saut), il s'agit de vérifier que l'adresse de destination figure bien dans cette table.

Coudray et al. [30] ont implémenté dans LLVM-IR une autre variante de CFI qui consiste à instrumenter le code de l'application à sécuriser pour le rendre dépendant d'un programme externe chargé de superviser tous les sauts. Les deux programmes communiquent via un tube (*pipe*), permettant au programme superviseur de vérifier la validité de chaque saut et d'arrêter l'exécution en cas de saut invalide. Cependant, leur solution ne couvre que les appels directs de fonctions.

Tice et al. [99] ont proposé l'implémentation d'une variante de CFI, intégrée dans les compilateurs LLVM et GCC. Leur solution concerne les langages de programmation orientés objets (POO), et consiste à garantir l'intégrité des tables (*vtable*) qui contiennent les adresses des fonctions *virtuelles* d'une classe. Dans les langages de POO, une fonction (méthode) virtuelle est une fonction qui est définie dans une classe et qui est destinée à être redéfinie dans des classes dérivées. Le schéma protège contre la substitution de la table *vtable* par une table contrôlée par un attaquant, lui permettant de détourner le flot d'exécution du programme dès lorsqu'une fonction virtuelle serait appelée. Cependant, ce schéma ne couvre qu'un cas très particulier des attaques par détournement de flot de contrôle, et est propre aux langages de POO.

3.3.3.3 Autres protections

Il existe d'autres mécanismes de protection mis en œuvre de manière complémentaire par les compilateurs, les OS et les processeurs (MMU/MPU), qui rendent le détournement de flot de contrôle difficile à réaliser, parmi lesquels :

— *canaries* : insertion d'une valeur aléatoire dans la pile d'appel permettant de détecter les débordements de tampon qui peuvent être à l'origine d'une exécution de code arbitraire. Cette protection est insérée par le compilateur.

— ASLR (*Address Space Layout Randomization*) : changement aléatoire de l'emplacement des données/instructions dans la mémoire à chaque exécution.

— DEP (*Data Execution Prevention*) : empêcher l'exécution de code depuis des zones mémoire censées contenir des données. Ce dispositif peut être renforcé par le processeur, par exemple le bit NX (*No-eXecute*) sur les processeurs x86 et XN (eXecute-Never) sur les processeurs ARM.

3.4 Conclusion

Dans ce chapitre, nous avons présenté les principes de base de la compilation, nécessaires pour appréhender les contributions présentées dans les prochains chapitres. Un accent particulier a été mis sur les spécificités du compilateur LLVM. Nous avons discuté des problématiques de la compilation pour la sécurité, notamment la difficulté de faire coexister optimisation de code et propriétés de sécurité dans un compilateur à usage général. Nous avons illustré l'antagonisme qu'il peut y avoir entre certains schémas de contre-mesures contre les attaques physiques et certaines passes d'optimisation du compilateur.

Nous avons illustré sur quelques exemples que, malgré certaines difficultés, l'intégration de protections à la compilation était une approche très prometteuse.

Troisième partie

Contributions

Chapitre 4

Simulateur de fautes

Sommaire

4.1 Fonctionnement 48
4.1.1 Entrées/Sorties 48
4.1.2 Modes de simulation 48
4.1.3 Processus de simulation 49
4.2 Modèles de fautes supportés 50
4.3 Architecture interne 52
4.3.1 Désassemblage 52
4.3.2 Cartographie 52
4.3.3 Modèles de fautes 53
4.3.4 Moteur d'injection 54
4.3.5 Émulation 54
4.3.6 Gestionnaire central 54
4.4 Utilisation 54

Dans ce chapitre, nous présentons le simulateur de faute que nous avons spécifié et implémenté pour évaluer les différents schémas de protection mis en œuvre dans cette thèse. Il est écrit dans le langage de programmation Python et fait 967 lignes de code. Le simulateur repose sur l'émulateur QEMU (`qemu-system-arm`) et utilise le désassembleur `arm-none-eabi-objdump` de la chaines d'outils *Sourcery CodeBench Lite 2013.11-24* [72].

4.1 Fonctionnement

Le simulateur réalise des fautes permanentes sur le programme en entrée en modifiant le code binaire de celui-ci avant l'exécution. La notion de fautes permanentes est décrite dans la section 2.2.4.

4.1.1 Entrées/Sorties

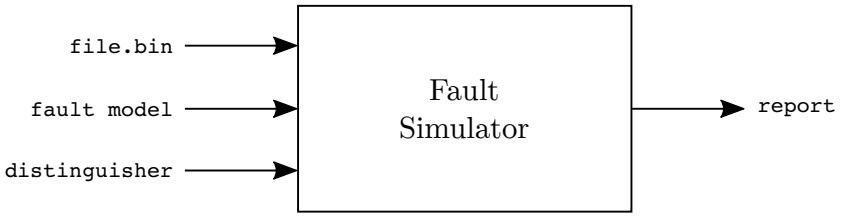


FIGURE 4.1 – Aperçu du simulateur de faute

La figure 4.1 présente un aperçu du simulateur. Il prend en entrée trois principaux éléments :

1. le fichier binaire à évaluer (`file.bin` sur la figure), qui doit être compilé pour l'architecture ARMv7m dans le format ELF.
2. le modèle de fautes à simuler (`fault model`).
3. un distingueur (`distinguisher`) pour identifier une exécution impactée par une faute, qui peut être une variable du programme, une adresse mémoire ou une plage mémoire (adresse début et taille).

Des arguments optionnels peuvent être fournis au simulateur, notamment pour spécifier les sections du code binaire à évaluer. Ces arguments seront détaillés dans la section 4.4. Un rapport détaillant les résultats des simulations effectuées est produit en sortie du simulateur.

4.1.2 Modes de simulation

Le simulateur dispose deux modes de simulation, décrits dans les sections suivantes.

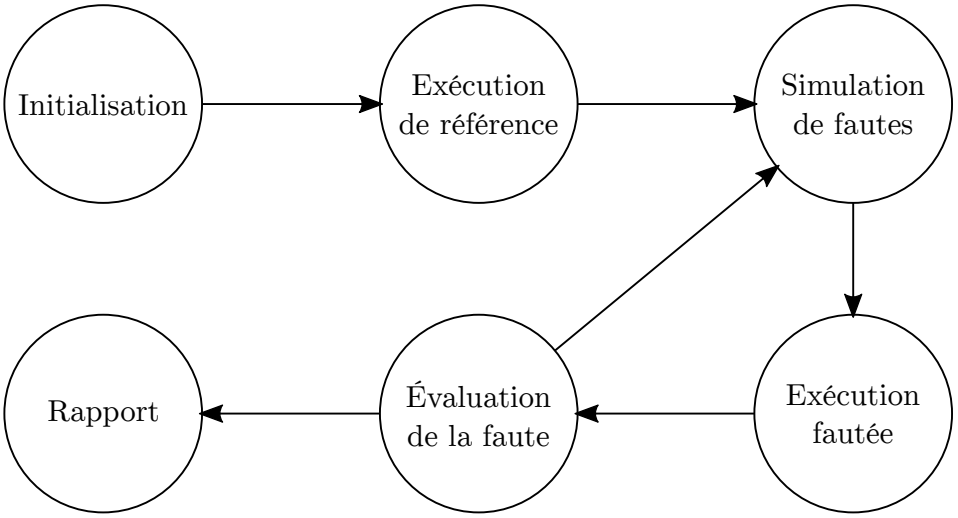


FIGURE 4.2 – Processus de simulation

4.1.2.1 Simulation de fautes unitaires

Dans ce mode de simulation, on simule une injection de faute sur une copie du programme binaire fournit en entrée, conformément au modèle de fautes. Par exemple, pour le modèle de fautes saut d'une instruction, une seule instruction est sautée dans la copie du programme binaire.

4.1.2.2 Simulation de fautes sur un programme

Dans ce mode de simulation, on automatise la *simulation de fautes unitaires* pour toutes les combinaisons de fautes possibles sur le programme, en tenant compte du modèle de fautes. Ce mode de simulation se déroule comme suit :

1. calcul du nombre de *simulations de fautes unitaires* (*nbSimu*) nécessaires pour tester tous les scénarios de fautes possibles sur le programme avec le modèle de fautes considéré.
2. création de *nbSimu* copies du programme binaire
3. réalisation d'une *simulation de fautes unitaires* sur chaque copie du programme.

4.1.3 Processus de simulation

La figure 4.2 illustre les différentes étapes du processus de simulation, ces étapes sont détaillées ci-après.

Initialisation. Cette première étape consiste à prendre en charge les paramètres de simulation : fichier binaire, modèle de fautes et distingueur.

Exécution de référence. Cette étape consiste à exécuter le code binaire fourni à l'entrée du simulateur, sans aucune modification. Après cette exécution, la valeur du distingueur et celle de chacun des registres du processeur sont sauvegardées. On appellera cette sauvegarde : *sauvegarde de référence*.

Simulation de fautes. Dans le mode *simulation de fautes unitaires*, une copie du programme binaire est créée, puis une faute est injectée dans cette copie. Dans le mode de *simulation de fautes sur un programme*, on boucle sur les scénarios de fautes possibles. Pour chaque scénario, les deux étapes suivantes sont effectuées : Exécution de faute et Évaluation. Cette boucle est illustrée par la figure 4.2.

Exécution fautée. Au cours de cette étape, le binaire fauté dans l'étape précédente est exécuté, puis la valeur du distingueur et celle de chacun des registres du processeur sont à nouveau sauvegardées. On appellera cette sauvegarde : *sauvegarde fautée*.

Évaluation. Au cours de cette étape, on compare *sauvegarde fautée* et *sauvegarde de référence*. Une faute est considérée comme *réussie* lorsque les contenus des deux sauvegardes sont différentes, ou si l'exécution du binaire fauté échoue. Dans ce cas, la nature de la faute et l'emplacement où elle a été injectée sont notifiés dans le rapport final. Dans le mode de *simulation de fautes sur un programme*, le processus reboucle à l'étape *Simulations de fautes*.

Rapport. Le simulateur fournit à la fin des simulations un rapport mentionnant pour chaque faute réussie : l'emplacement de la faute dans le code binaire, l'état des registres et du distingueur après la faute.

4.2 Modèles de fautes supportés

Cette section présente les modèles de fautes actuellement supportés par le simulateur. Pour chaque modèle de fautes, la notation en minuscules (e.g `skip{...}`) désigne une *simulation de fautes unitaires* de ce modèle de fautes ; la notation en majuscules (e.g `SKIP{...}`) désigne une *simulation de fautes sur un programme* de ce modèle de fautes.

Dans les paragraphes suivants, nous présentons pour chaque modèle de fautes supporté : l'objectif fonctionnel visé, comment il est mis en œuvre et le nombre de simulations réalisées ($nbSimu$) par une *simulation de fautes sur un programme*.

`repl{instr}` Remplacement d'une instruction par l'instruction `instr`. La taille de l'instruction remplacement doit être égale à la taille de l'instruction remplacée.

- `REPL{instr}` : $nbSimu = nbInstr$. Le modèle `repl{instr}` est appliqué à toutes les instructions du programme évalué.

`skip_br` Simule le saut d'une instruction de branchement. On la met en œuvre en appliquant le modèle `repl{nop}` sur une instruction de branchement. Ce modèle permet de simuler l'exécution d'un chemin prévu dans le CFG du programme, mais non valide fonctionnellement. Par exemple exécuter le bloc *else* au lieu du bloc *if*, ou réduire le nombre d'itérations d'une boucle.

- `SKIP_BR` : à chaque simulation une instruction `br` distincte est sautée. Soit $nbBranchInstr$ le nombre d'instructions de branchement dans le programme.

$$nbSimu = nbBranchInstr$$

`skip{N, M}` Simule le saut de `M` fautes distinctes sur des séquences de `N` instructions consécutives. Chaque faute injectée simule le saut des `N` instructions de la séquence. Ce modèle permet simuler l'injection de fautes multiples, la largeur de la faute étant paramétrable. En pratique, les `N` instructions des séquences fautées sont remplacées par `N` instructions `nop`, que l'on simule par l'application du modèle `repl{nop}` sur les `N` instructions ciblées.

- `SKIP{N, M}` : soit k le nombre de `N` instructions consécutives distinctes $k = nbInstr - (\texttt{N} - 1)$.

$$nbSimu = \frac{k!}{\texttt{M}!\,(k - \texttt{M})!}$$

`repl_br{addr}` Simule le remplacement d'une instruction du programme par une instruction de branchement à l'adresse `addr`. Ce modèle permet de simuler l'exécution d'un chemin non prévu dans le CFG du programme.

- `REPL_BR` : à chaque simulation, une instruction distincte est remplacée par un `br`. Pour chaque instruction remplacée par un `br`, on teste toutes les adresses valides du programme ($nAddrs$).

$$nbSimu = nbInstr \times nAddrs$$

`repl_br_skip{addr, N, M}` Simule l'application successive des modèles `repl_br{addr}` et `skip{N, M}` au cours d'une exécution.

- `REPL_BR_SKIP{N, M}` : à chaque simulation, une instruction distincte est remplacée par un `br`, puis `M` sauts de `N` instructions consécutives sont réalisés. Si $nAddrs$ est le nombre d'adresses valides du programme et k le nombre de `N` instructions consécutives distinctes, alors :

$$nbSimu = nbInstr \times nAddrs \times \frac{k!}{\texttt{M}!\,(k - \texttt{M})!}$$

`skip_block{size, M}` Simule le saut de `M` blocs logiques de taille `size`. La notion de bloc logique est présentée dans la section 2.2.3.3. Ce modèle permet d'émuler le saut des instructions présentes dans un *buffer* de préchargement d'instructions (*prefetch buffer*) de taille `size` ou dans le bus de transfert d'instructions de taille `size`.

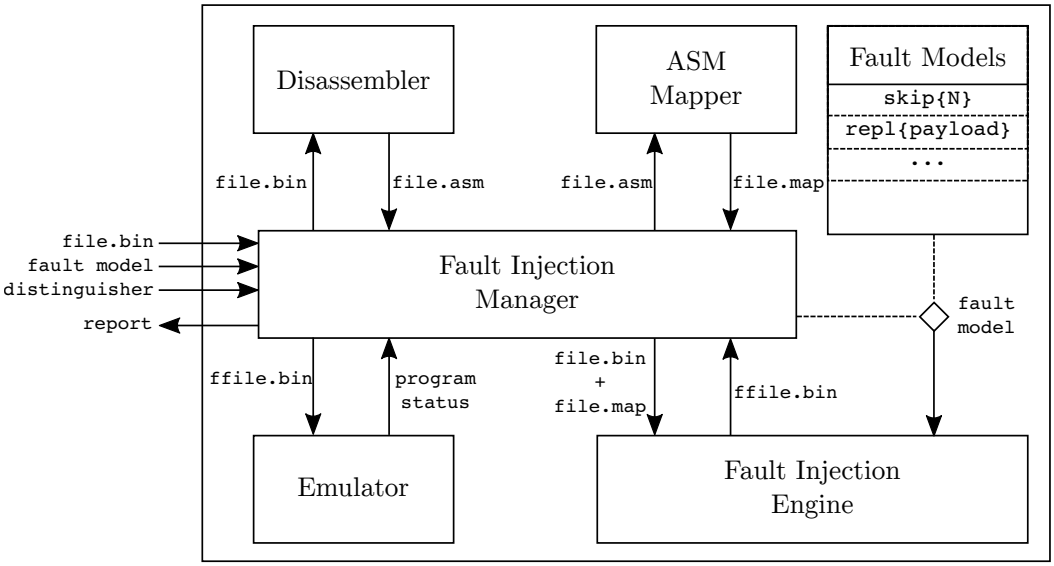


FIGURE 4.3 – Architecture interne du simulateur

- `SKIP_BLOCK{size, M}` : Soit T est la taille en bits du segment `.text` du programme, et k le nombre de blocs logiques distincts de taille `size`.
  $k = \frac{T}{\texttt{size}} - (\texttt{M} - 1)$.

$$nbSimu = \frac{k!}{\texttt{M}!\,(k - \texttt{M})!}$$

4.3 Architecture interne

La figure 4.3 présente l'architecture interne du simulateur, faisant apparaître ses différents modules qui seront décrits dans les sections suivantes.

4.3.1 Désassemblage

Ce module utilise le désassembleur `arm-none-eabi-objdump` pour obtenir le code assembleur du programme évalué. Le code assembleur est ensuite analysé pour identifier les sections concernées par l'évaluation (lorsque cela est précisé en argument). La figure 4.4 illustre un extrait du code assembleur de la fonction `AES_Encrypt` du programme AES, sécurisé avec le schéma de tolérance au saut d'instructions :

4.3.2 Cartographie

Le module `ASM Mapper` reçoit le code assembleur produit par le module de désassemblage puis effectue une cartographie des instructions afin de calculer l'adresse et la taille de toutes les instructions. Le résultat de la cartographie est un fichier `.map` (dans un format de représentation que nous avons défini). En effet, étant sur un modèle de fautes au niveau instruction, il est nécessaire d'avoir la taille de chaque instruction, notamment pour pouvoir la sauter ou la remplacer.

```
aes.bin:     file format elf32-littlearm
Disassembly of section .text:

...
00000724 <AES_Encrypt>:
724:       e90d 4ff0       stmdb   sp, {r4, r5, r6, r7, sl, fp, lr}
728:       e90d 4ff0       stmdb   sp, {r4, r5, r6, r7, sl, fp, lr}
72c:       f1ad 0c24       sub.w   ip, sp, #36     ; 0x24
730:       f1ad 0c24       sub.w   ip, sp, #36     ; 0x24
734:       46e5            mov     sp, ip
736:       46e5            mov     sp, ip
738:       f1ad 0c44       sub.w   ip, sp, #68     ; 0x44
73c:       f1ad 0c44       sub.w   ip, sp, #68     ; 0x44
...
```

FIGURE 4.4 – Extrait du code assembleur de la fonction `AES_Encrypt`. Le code est sécurisé avec le schéma de tolérance au saut d'instructions

La cartographie consiste d'abord à identifier le couple (adresse de début, adresse de fin) de chaque fonction, puis à associer ce couple à une liste contenant la taille de chaque instruction de la fonction, ordonnée à partir de l'adresse de début. La taille de chaque instruction est soit de 2 soit de 4 octets.

Pour réaliser la cartographie, il est nécessaire de connaître l'adresse de début du segment `.text` du fichier binaire. Par défaut, ce segment est placé à l'adresse `0x8000` par les éditeurs de liens de notre plateforme (`arm-none-eabi-ld`). Le simulateur utilise donc par défaut cette adresse. Néanmoins, il est possible de soumettre une nouvelle adresse au simulateur avec l'option `-T <ADDR_TEXT_SEGMENT>` ou `--addr-text-segment <ADDR_TEXT_SEGMENT>`.

Voici un extrait de la cartographie de la fonction `AES_Encrypt`, dont le code assembleur est illustré dans la section précédente :

```
[AES_Encrypt]
beginAddr = 0x0724
endAddr   = 0x1146
layout    = 4|4|4|4|2|2|4|4|...
```

Les attributs `beginAddr` et `endAddr` sont les adresses de début et de fin de la fonction et `layout` la liste de la taille des instructions.

4.3.3 Modèles de fautes

Ce module implémente, pour chaque modèle de fautes supporté, la manière d'injecter une faute correspondant à ce modèle. La logique de chaque modèle supporté par le simulateur est implémentée dans une `class` Python. Le simulateur est conçu de telle sorte que, pour ajouter le support d'un nouveau modèle de fautes, il suffit simplement de greffer une `class` implémentant ce modèle, sans aucune modification des autres modules.

4.3.4 Moteur d'injection

Le moteur d'injection est le module qui matérialise l'injection de faute dans le code binaire. Il implémente les outils nécessaires pour aller écrire un `PAYLOAD` à une adresse donnée du fichier binaire. Par exemple, pour sauter une instruction, la `class InstructionSkipper` du module *Modèles de fautes* décrit comment sauter une instruction en fournissant `PAYLOAD = nop` et le module *Gestionnaire central* (présenté plus loin) fournit l'adresse de l'instruction à sauter.

4.3.5 Émulation

Le module d'émulation utilise l'émulateur QEMU pour exécuter le binaire et GDB pour interagir avec le code en exécution. Le programme `qemu-system-arm` est lancé en mode serveur pour émuler le CPU ARM Cortex-M3. Ensuite le programme `arm-none-eabi-gdb` est utilisé pour se connecter au serveur QEMU et piloter l'exécution du binaire. Nous avons mis en œuvre un script GDB pour automatiser le pilotage du programme ainsi que la collecte des états des registres et du distingueur.

4.3.6 Gestionnaire central

Le gestionnaire central est le module chargé de coordonner et d'orchestrer l'exécution de tous les autres modules du simulateur. Le fonctionnement de ce module est décrit par l'algorithme 1. La première tâche consiste à récupérer puis à vérifier la validité des arguments fournis par l'utilisateur (lignes 2-8). Une exécution de référence est ensuite réalisée par la ligne 10. Un échec de cette exécution sera notifié dans le rapport par la ligne 12. La ligne 18 détermine toutes les adresses d'injection de fautes possibles en prenant en compte, éventuellement, les sections indexées par l'utilisateur. La boucle de la ligne 19 à 26 réalise une injection de faute à chacune de ces adresses. Chaque injection est effectuée sur une copie du binaire de référence (lignes 21,22). Cette copie fautée est ensuite exécutée par la ligne 23. La ligne 24 compare les états des registres et du distingueur de l'exécution du binaire fauté avec ceux du binaire de référence, et rend un verdict qui est notifié dans le rapport.

4.4 Utilisation

Le processus de simulation présenté dans ce chapitre est entièrement automatisé. Le simulateur est lancé à travers la commande `./simulateur.py` qui accepte plusieurs arguments obligatoires et optionnels. Une invocation de la commande `./simulateur --help` permet d'avoir le manuel d'utilisation du simulateur illustré par la figure 4.5.

Algorithm 1 Algorithme du module *Gestionnaire central*

```
1:  function FAULTINJECTIONMANAGER( )
2:      args ← GETUSERSUPPLIEDARGS( )
3:      if not VALIDATE(args) then
4:          return ERROR(BAD_ARGUMENTS)
5:      end if
6:      binFile ← GETBINARYFILE(args)
7:      faultModel ← GETFAULTMODEL(args)
8:      concernedSections ← GETCONCERNEDSECTIONS(args)
9:      report ← INITREPORT( )                          ▷ initialisation du rapport
10:     refRun ← EXECUTE(binFile)                       ▷ exécution de référence
11:     if FAILED(refRun) then
12:         LOG(report, refRun)
13:         return ERROR(INVALID_BINARY_FILE)
14:     end if
15:     asm ← DISASSEMBLE(binFile)                      ▷ désassemblage du binFile
16:     map ← MAP(asm)                                  ▷ mappage de asm
17:     INITFAULTMODEL(faultModel)                      ▷ initialisation du modèle de fautes
18:     addrStack ← GETADDRSTACK(map, concernedSections)
19:     while not EMPTY(addrStack) do
20:         addr ← POP(addrStack)
21:         copyBinFile ← BINCOPY(binFile)
22:         faultedBinFile ← INJECTFAULT(copyBinFile)
23:         run ← EXECUTE(faultedBinFile)
24:         verdict ← COMPARE(refRun, run)
25:         LOG(report, run, verdict)
26:     end while
27:     return report
28: end function
```

```
usage: simulateur.py [-h] [-T ADDR_TEXT_SEGMENT] [-d] -b BINARY
                 --functions FUNCTIONS [FUNCTIONS ...] [--sections]
                 [--from ADDR] [--to ADDR] [-s] [--number NUMBER]
                 [--consecutive] [-r] [--by [BR, PAYLOAD, BLOC_SIZE]]
                 [--dest ADDR] [--payload PAYLOAD] [--bs BLOC_SIZE]

Fault Injector program

Supplied addresses ADDR must be in hexadecimal whitout a leading '0x'

optional arguments:
  -h, --help            show this help message and exit
  -T ADDR_TEXT_SEGMENT, --addr-text-segment ADDR_TEXT_SEGMENT
                        Address of text segment of the binary file
                        the defautl address is 0x8000

General options:
  -d, --debug           Debug mode

  -b BINARY, --binary BINARY
                        Target binary file
  --functions FUNCTIONS [FUNCTIONS ...]
                        List of concerned functions
  --sections SECTIONS
  --from ADDR           Starting address of the concerned section
  --to ADDR             Ending address of the concerned section

Instruction Skip Fault Model:
  -s, --skip            Instruction skip fault model, True by default
  --number NUMBER       Number of instructions to skip, only if -s/--skip
                        option is present, defaut is NUMBER=1
  --consecutive         Whether skiped instructions are consecutive or not
                        This option makes sens only if NUMBER > 1

Instruction Replacement Fault Model:
  -r, --replacement     Instruction replacement fault model
  --by [BR, PAYLOAD, BLOC_SIZE]
                        Replacement type
  --dest ADDR           Branch destination, only if --br=BR
  --payload PAYLOAD     The replacement payload
  --bs BLOC_SIZE        Size of the bloc to replace, only if --br=BLOC_SIZE

Author: Thierno Barry <thierno.barry@cea.fr> Version 0.1
Copyright CEA-LIST 2017
```

FIGURE 4.5 – Manuel d'utilisation du simulateur

Chapitre 5

Compilation d'un schéma de tolérance aux fautes

Sommaire

5.1 Modèle de fautes 58
5.2 Mise en œuvre du schéma de protection 58
5.2.1 Génération d'instructions idempotentes 59
5.2.2 Modification de la sélection d'instructions 60
5.2.3 Modification de l'allocation de registres 64
5.2.4 Passe de transformations d'instructions 66
5.2.5 Passe de vérification pré-duplication 66
5.2.6 Passe de duplication d'instructions 67
5.2.7 Passe de vérification post-duplication 67
5.2.8 Ordonnancement 68
5.3 Expérimentations 69
5.3.1 Évaluation de performances 70
5.3.2 Évaluation de sécurité 71
5.4 Conclusion 72

Ce chapitre présente la mise en œuvre, par l'approche compilation, du schéma de tolérance au saut d'instructions qui a été formellement vérifié par Moro et al. [76] et qui est détaillé dans la section 2.4.2.

La section 5.1 présente une formalisation du modèle de fautes saut d'instructions. La section 5.2 présente les détails de la mise en œuvre de ce schéma dans le compilateur. Dans la section 5.3, nous présentons les expérimentations réalisées sur le même processeur cible (ARM Cortex M3) et sur les mêmes codes sources de référence que Moro et al., permettant ainsi de comparer nos résultats aux leurs.

5.1 Modèle de fautes

Le modèle de fautes considéré dans ce chapitre est le modèle saut d'instructions, qui suppose que la faute injectée conduit à sauter une seule instruction machine. Le modèle saut d'instructions est présenté en détail dans la section 2.2.3. En utilisant les notations introduites dans la sections 4.2, ce modèle correspond au `skip{1, 1}`.

5.2 Mise en œuvre du schéma de protection

Nous avons vu dans le chapitre 3 que le processus de compilation est constitué d'une succession de passes d'analyses et de transformations, et que la représentation du programme change plusieurs fois le long de ce processus. De ce fait, implémenter une contre-mesure dans un compilateur nécessite d'identifier sur quelle représentation et dans quelle partie du compilateur l'implémenter. Car, pour la même raison qu'une représentation donnée du programme permet d'améliorer l'efficacité de certaines optimisations du compilateur, l'application d'une contre-mesure peut être plus efficace sur une représentation plutôt qu'une autre.

Dans notre cas, l'objectif est de dupliquer les instructions. Pour cela, nous identifions tout d'abord sur quelle représentation du programme et avant quelle passe d'optimisation cela est possible. Dans la partie frontale du compilateur, le programme est dans la représentation code source. Par conséquent, toute duplication d'instructions dans cette partie sera systématiquement supprimée par les optimiseurs de la partie intermédiaire. Dans la partie intermédiaire, le programme est dans la représentation IR sous sa forme SSA. La forme SSA qui stipule qu'une variable ne peut être affectée qu'une seule fois, implique qu'il n'est pas possible de dupliquer les instructions sur toutes les représentations utilisant cette forme. Sachant que la forme SSA ne disparaît que durant la phase d'allocation de registres, nous pouvons en déduire que la duplication d'instructions devra être réalisée dans la partie arrière du compilateur et plus précisément après l'allocation de registres.

D'après le schéma de tolérance, les instructions doivent être idempotentes avant d'être dupliquées, ce qui implique que les transformations pour les rendre idempotentes doivent être effectuées au plus tard pendant l'allocation de registres. Cependant, nous avons montré, dans la section 2.4.2, que le coût élevé de l'implémentation de ce schéma par l'approche code binaire était lié à la nécessité de transformer les

instructions non idempotentes en formes idempotentes. Ce problème est dû au fait que le compilateur, lors de la génération de code, fait des choix de sélection d'instructions et d'allocation de registres qui conduisent à la génération d'instructions non-idempotentes. C'est pourquoi, afin de réduire le coût du schéma, nous optons pour une démarche qui consiste à faire en sorte que le compilateur ne génère que des instructions idempotentes.

5.2.1 Génération d'instructions idempotentes

Pour pouvoir générer des instructions idempotentes, il est nécessaire d'intégrer la notion d'*idempotence* dans les passes du compilateur qui sont responsables de la génération de code. Pour cela, il est primordial de caractériser ce qu'est une instruction idempotente.

Caractéristique d'une instruction idempotente Une instruction est une étape de programmation qui indique à l'ordinateur les actions qu'il doit effectuer [40]. Elle est caractérisée par : (1) les données qu'elle prend en entrée, (2) les opérations qu'elle effectue et (3) les données qu'elle produit en sortie. Les *entrées* d'une instruction sont l'ensemble des données lues par cette instruction pour effectuer son opération, par exemple les registres sources. Les sorties sont l'ensemble des modifications, sur le contexte du programme, effectuées par l'instruction, par exemple les registres destinations.

L'entrée ou la sortie d'une instruction est dite *explicite*, lorsqu'elle fait partie des opérandes fournis explicitement à l'instruction (opérandes remplaçables) ; elle est dite *implicite* dans les autres cas. Par exemple : l'instruction `push{r0}` écrit le contenu du registre `r0` à l'adresse contenue dans le registre `sp`, puis décrémente `sp` de 4 octets. Ici le registre `r0` est une entrée explicite de l'instruction, il peut être remplacé par un autre registre. Le registre `sp` est une entrée et une sortie implicite, il n'est pas possible de la changer.

Une instruction idempotente est caractérisée par la non modification de ses entrées, c'est-à-dire qu'aucune des entrées de l'instruction n'est en même temps une de ses sorties. Les entrées d'une instruction sont modifiées dans les deux cas suivants :

cas N°1 : lorsque le mode d'opération de l'instruction est telle qu'elle met à jour implicitement l'un de ses registres sources. C'est le cas, par exemple, de l'instruction `push` décrite ci-dessus et qui modifie le registre `sp`.

cas N°2 : lorsque les registres sont alloués de telle sorte que l'un des registres sources de l'instruction est aussi un registre destination. Par exemple, l'instruction `add r0, r0, r1` qui fait la somme des contenus de `r0` et de `r1` et qui stocke le résultat dans `r0`. Le registre `r0` est ici à la fois source et destination.

Pour intégrer les caractéristiques d'une instruction idempotente dans les générateurs de code, nous avons modifié certaines passes existantes du compilateur et nous avons introduit de nouvelles, qui seront détaillées dans les sections suivantes. La figure 5.1 présente une vue simplifiée de la structure interne de notre compilateur. Les boites grises et les boites noires de la figure représentent respectivement

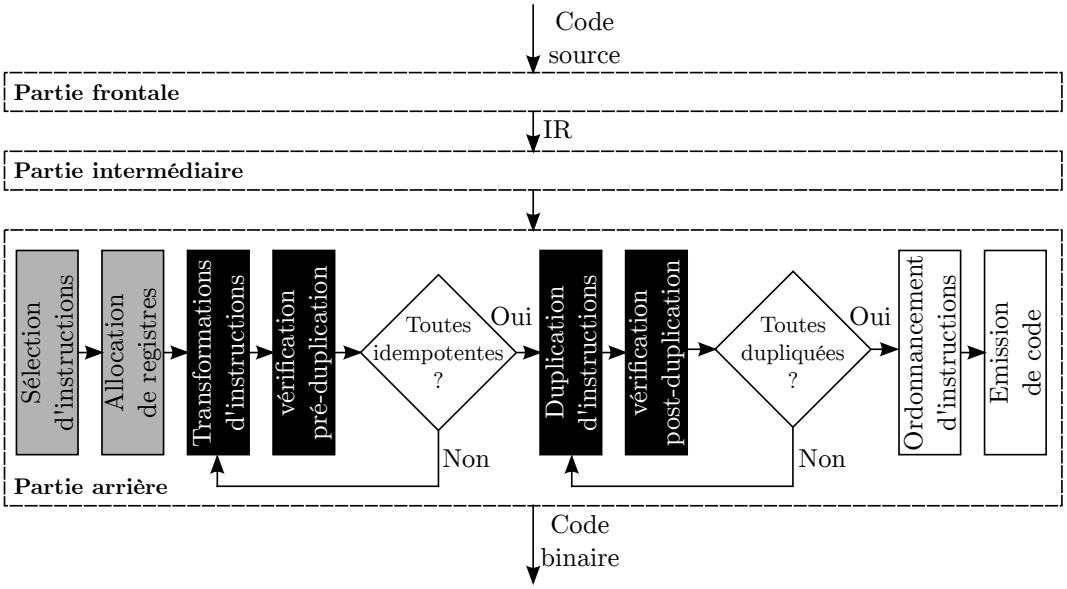


FIGURE 5.1 – Structure simplifiée du compilateur modifié. Les boites grises et les boites noires représentent respectivement les passes modifiées et celles ajoutées

les passes modifiées et celles ajoutées dans le compilateur. La sélection d'instructions, présentée en section 5.2.2, a été modifiée pour que le cas N°1 arrive le moins souvent possible, et l'allocation de registre, présentée en section 5.2.3, est modifiée pour éviter le cas N°2.

D'autres passes *utilitaires*, non représentées sur la figure 5.1 pour des raisons de simplicité, participent aussi à la mise en œuvre de ce schéma. La plus importante d'entre elles est la passe `ToleranceContext` dont le rôle est de collecter et de restituer aux autres passes du schéma les informations fournies par l'utilisateur sur le paramétrage du schéma de tolérance, notamment les sections du programme annotées. Parmi les fonctions implémentées par cette passe, celle qui nous intéressera dans la suite de cette description est la fonction `process(llvm::MachineFunction *MF)` qui prend en paramètre la référence d'une fonction du programme, puis retourne vrai ou faux selon que cette fonction est dans une section annotée ou non. La fonction `process` dispose de deux autres variantes (*fonctions surchargées*) qui sont : `process(llvm::MachineBasicBlock *MBB)` et `process(llvm::MachineInstr *MI)` qui accomplissent la même tâche mais en prenant comme paramètres respectivement un bloc de base et une instruction.

5.2.2 Modification de la sélection d'instructions

La phase de sélection d'instructions, introduite dans la section 3.1.3, est composée d'un ensemble de passes dont le rôle est de traduire les instructions LLVM-IR en instructions de l'architecture cible (instructions machine). Selon le jeu d'instructions de l'architecture cible, il peut y avoir plusieurs choix de traduction possibles. Les modifications faites dans l'étape de sélection d'instructions visent à faire en sorte

```
class Instruction{
    dag OutOperandList;
    dag InOperandList;
    string AsmString = "";
    list<dag> Pattern;
    list<Register> Uses = [];
    list<Register> Defs = [];
    list<Predicate> Predicates = [];
    bit isReturn = 0;
    bit isBranch = 0;
    ...
```

FIGURE 5.2 – Extrait de l'interface `class Instruction` proposée par LLVM pour définir une instruction machine

que les instructions idempotentes soient sélectionnées en priorité.

Pour comprendre les modifications effectuées à ce niveau, décrivons plus en détail le processus de sélection d'instruction. Au tout début de l'étape de sélection d'instructions, le programme, encore dans sa forme LLVM-IR, est représenté sous forme de DAG, appelé `SelectionDAG` dans LLVM. Ce DAG (introduit en section 3.2.3.4) est un graphe acyclique utilisé pour représenter les instructions dans un bloc de base. Chaque nœud du DAG, appelé `SDNode`, représente soit un opérande (par exemple une constante ou un registre), soit un opérateur (par exemple $+, -, *$). Une opération, appelée *DAG to DAG instruction selection*, est ensuite effectuée sur le DAG. Cette opération consiste à convertir le DAG représentant des instructions LLVM-IR en un DAG représentant des instructions machine, par un processus de reconnaissance de motifs (*pattern matching*).

5.2.2.1 Reconnaissance de motifs

Le mécanisme de reconnaissance de motifs (patterns) fait partie du module *LLVM target-independant code generator* [65] qui factorise les tâches communes réalisées par les différents back-ends (cibles ou *Target*) du compilateur. Il revient cependant à chaque back-end de définir les patterns à reconnaître sur le DAG, en fonction des instructions machine que supporte son architecture cible.

LLVM expose une interface nommée `class Instruction` que tous les back-ends doivent utiliser pour définir une instruction machine. Un extrait de cette interface est présenté par la figure 5.2. Le mot clé `dag` est un type utilisé pour désigner un nœud du DAG. Ainsi, pour une instruction machine donnée :

— `InOperandList` et `OutOperandList` désignent respectivement la liste des opérandes que l'instruction prend en entrée et ceux qu'elle produit en sortie. Par exemple, pour l'instruction `add rdest, rsrc1, rsrc2`, on a `InOperandList = [rsrc1, rsrc2]` et `OutOperandList = [rdest]`.

— `AsmString` décrit la représentation textuelle (assembleur) de l'instruction.

— `Pattern` décrit le pattern (composé d'une liste de nœuds) à reconnaître sur le DAG pour sélectionner cette instruction.

— `Defs` et `Uses` listent respectivement les registres écrits et lus par l'instruction.

— `Predicates` est un ensemble de conditions à satisfaire pour que l'instruction soit sélectionnée.

— `isBranch` et `isReturn` apportent des informations sur le comportement de l'instruction, si c'est une instruction de branchement ou de retour de fonction.

Pour chaque instruction machine, les informations demandées par cette interface sont renseignées à travers un DSL (*Domain Specific Language*) avec une syntaxe proche de celle du langage de programmation LISP. L'exemple ci-dessous présente un extrait de la définition de l'instruction `mul` du jeu d'instructions ARM Thumb :

```
def tMUL :
    Thumb1sI<(outs tGPR:$Rd), (ins tGPR:$Rn, tGPR:$Rm), AddrModeNone,
             2, IIC_iMUL32, "mul", "\t$Rd, $Rn, $Rm", "$Rm = $Rd",
             [(set tGPR:$Rd, (mul tGPR:$Rn, tGPR:$Rm))]>,
         T1DataProcessing<0b1101> {
...
}
```

Les mots clés `outs` et `ins` introduisent `InOperandList` et `OutOperandList`, les chaines de caractères correspondent à `AsmString`, et ce qu'il y a entre les crochets représente le `Pattern`. LLVM dispose d'un outil appelé *TableGen* dont le rôle est de construire une table de pattern matching à partir de la définition des instructions. C'est cette table qui est utilisée pour reconnaître les patterns sur le DAG.

L'objectif est de définir un prédicat (en faisant appel à la fonction `process`) pour empêcher la sélection d'instructions non idempotentes dans les sections annotées, et pour maintenir le comportement par défaut du sélecteur d'instructions lorsqu'on n'est pas dans une section annotée. La définition d'un prédicat pour l'instruction `INST` se fait comme suit :

```
let Predicates = [Predicate<"!ToleranceContext::process(MF)">] in {
    def INST :
        ;; définition de l'instruction
}
```

Par conséquent, toutes les instructions pour lesquelles le prédicat est faux ne seront pas sélectionnées. Cette façon de procéder permet de ne pas impacter les sections non concernées par le schéma de tolérance. Car, dans la plupart des cas, les instructions non idempotentes sont plus compactes et s'exécutent plus vite que leurs équivalentes idempotentes.

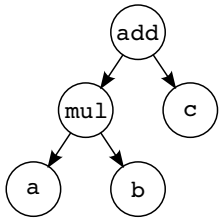
La figure 5.3 illustre le processus de sélection d'instructions pour les opérations arithmétiques $(a * b) + c$. Par défaut pour les architectures ARM 32 bits, le pattern capturé englobe tous les nœuds du DAG de la figure 5.3c, en vue de sélectionner l'instruction `mla` (Multiply-Accumulate) qui n'est pas idempotente. Nos modifications permettent de sélectionner l'instruction `mul` suivie de l'instruction `add`, lesquelles des opérations simples et idempotentes. Nous rappelons que ces deux instructions

```
1 res = a * b + c
```

(a) Code source C

```
1 %tmp = mul %a, %b
2 %res = add %tmp, %c
```

(b) LLVM-IR



(c) Selection DAG

FIGURE 5.3 – Illustration du processus de sélection d'instructions

```
1  bool Thumb2SizeReduce::ReduceTo2Addr(
2                  MachineBasicBlock &MBB, MachineInstr *MI, ... ) {
3      ...
4      if (ToleranceContext::process(MI))
5          return false;
6      if (ReduceLimit2Addr != -1 && ((int)Num2Addrs >= ReduceLimit2Addr))
7          return false;
8      ...
9      return true;
10 }
```

FIGURE 5.4 – Extrait de la passe `Thumb2SizeReduce` responsable de la réduction des opérandes d'une instruction

ne seront définitivement idempotentes que lorsque les registres seront alloués de telle sorte que les registres sources et destinations ne partagent aucun registre. C'est l'objet des modifications introduites dans l'allocateur de registres présentées plus loin (en section 5.2.3).

5.2.2.2 Sélection d'instructions à 3 adresses

Le jeu d'instructions Thumb-2 permet, pour les instructions arithmétiques et logiques de la forme `opcode dst, src1, src2`, un encodage plus compact sous la forme `opcode src1, src2`. Dans le jargon de la compilation, ce processus s'appelle la réduction d'une instruction à 3 adresses vers 2 adresses. Cet encodage permet d'utiliser moins de registres pour la même opération, en réutilisant l'un des registres sources comme registre de destination. Cependant, cette nouvelle forme n'est pas idempotente. Nous avons réécrit la passe responsable de cette réduction pour qu'elle ne soit réalisée que lorsque l'instruction à réduire ne figure pas dans une section annotée par l'utilisateur. Un extrait de cette passe est présenté par la figure 5.4. L'objectif est de faire en sorte que les instructions présentes dans les sections annotées ne soient pas réduites à 2 adresses. Pour cela, à la ligne 4 de la figure 5.4, nous faisons appel à la fonction `process(MachineInstr *MI)` pour déterminer si l'instruction en cours de réduction se trouve dans une section annotée. Si c'est le cas, alors on arrête le processus de réduction.

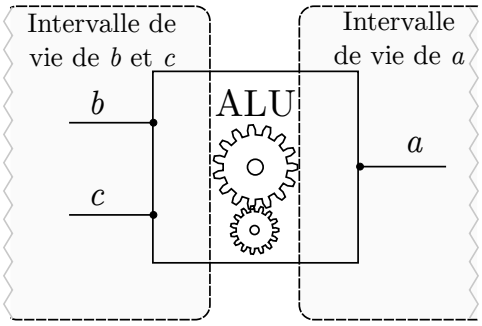


FIGURE 5.5 – Intervalles de vie des variables a, b et c à l'entrée de l'ALU, pour l'opération $a = b + c$.

5.2.3 Modification de l'allocation de registres

Le rôle de l'allocateur de registres (introduit dans la section 3.1.3) est d'allouer pour chaque variable du programme un registre physique. Le nombre de registres physiques étant très limité, la stratégie de l'allocateur consiste à réutiliser le plus possible chacun des registres pour éviter de faire appel à la mémoire (*register spilling*). Pour cela, l'allocateur se sert d'une passe d'analyse de vivacité appelée *liveness analysis*. Cette passe détermine pour chaque variable du programme, l'intervalle dans lequel la variable est considérée vivante. La vie d'une variable commence au moment de son affectation et se termine à sa dernière utilisation (dernière lecture). Le principe de base de l'allocation de registres est le suivant : si deux variables du programme ont des intervalles de vie disjoints, alors elles peuvent être allouées au même registre physique.

Pour bien comprendre la logique derrière les modifications effectuées dans l'allocateur de registres, prenons l'exemple d'une opération $a \leftarrow b + c$ et examinons le comportement par défaut de l'allocateur de registres à l'échelle de cette instruction, en supposant que les variables b et c ne sont pas utilisées par la suite. La figure 5.5 illustre les intervalles de vie de chacune des variables. Les variables b et c sont lues au même moment par l'unité arithmétique (ALU) du processeur pour procéder à l'addition. Elles sont donc vivantes au même moment. Par conséquent, l'allocateur déduit que chacune d'elles doit être assignée à un registre différent, disons `r0` pour b et `r1` pour c, représentants ainsi les registres sources de l'instruction. Leurs vies s'arrêtent dès lors que leurs valeurs sont lues par l'ALU, alors que celle de la variable a n'a pas encore commencé. L'intervalle de vie de a est donc disjoint de celui de b et c. Ainsi, pour assigner un registre à a, l'allocateur décide de réutiliser l'un des registres sources, `r0`. L'instruction machine après l'allocation sera donc `add r0, r0, r1`, laquelle a le registre `r0` qui est à la fois source et destination, et est, par conséquent, non idempotente.

Nous avons modifié l'allocateur de registres afin d'éviter le comportement décrit ci-dessus pour toutes les instructions se trouvant dans une section annotée, et qui sont de la forme suivante : `opcode dst, src1, [src2, ...]`, où :

— `opcode` peut être toute instruction qui écrit son résultat dans un registre, telle que les instructions arithmétiques, de manipulation de bits, ou de lecture mémoire.

— `dst` le registre destination

— `src1` un registre source

— `[src2, ...]` des registres sources optionnels

La modification se traduit par l'introduction d'une contrainte qui interdit de partager le registre destination avec un registre source.

Cette contrainte, appliquée à l'exemple précédent, permettrait de générer l'instruction `add r2, r0, r1` au lieu de `add r0, r0, r1`. La passe qui ajoute cette contrainte à l'allocateur de registre est décrite par l'algorithme 2.

Algorithm 2 Ajout d'une contrainte à l'allocateur de registres

```
1: function regAllocConstraint(listMachineFunctions)
2:     for all MF in listMachineFunctions do
3:         if MF in ToleranceContext then
4:             for all MBB in listMachineBasicBlocks(MF) do
5:                 for all MI in listMachineInstruction(MBB) do
6:                     if isArithmetic(MI) or isBitwise(MI) then
7:                         for all MO in listOperands(MI) do
8:                             if isRegister(MO) and isDefinition(MO) then
9:                                 setIsEarlyClobber(MO)
10:                            end if
11:                        end for
12:                    end if
13:                end for
14:            end for
15:        end if
16:    end for
17: end function
```

C'est une passe de type `llvm::MachineFunctionPass` qui parcourt le programme fonction par fonction, car l'allocation de registres se fait par fonction. La ligne 2 de l'algorithme 2 parcourt la liste des fonctions. La ligne 3 détermine si la fonction en cours de traitement se trouve dans une section annotée par l'utilisateur. La ligne 4 parcourt les blocs de base contenus dans la fonction. Pour chaque bloc de base, la ligne 5 parcourt la liste de ses instructions. La ligne 6 vérifie si l'instruction est une instruction arithmétique ou de manipulation de bits. La ligne 7 parcourt la liste des opérandes de l'instruction. À la ligne 8, nous vérifions si l'opérande en question est un registre virtuel de destination. À la ligne 9, la fonction LLVM setIsEarlyClobber définit que l'opérande de destination doit être alloué dans des registres différents de ceux des autres opérandes.

5.2.4 Passe de transformations d'instructions

Les modifications décrites dans les sections précédentes permettent de maximiser le nombre d'instructions idempotentes dans les sections annotées. Cependant, les instructions `push/pop` (sauvegarde/restauration d'une valeur de la pile) et `bl` (appel de fonction) n'ont pas de variantes idempotentes. Pour ces instructions, des passes de transformations ont été introduites pour les transformer en des séquences d'instructions idempotentes, selon les schémas de transformations proposés par Moro et al. [75].

5.2.5 Passe de vérification pré-duplication

À ce stade de la compilation, toutes les instructions contenues dans les sections annotées du programme sont supposées être idempotentes grâce aux différentes transformations décrites précédemment. Mais, avant de procéder à la duplication des instructions, nous avons inséré une passe de vérification (pré-duplication) dont le rôle est de garantir que les optimisations du compilateur n'ont pas altéré la propriété d'idempotence des instructions. Le fonctionnement de cette passe est décrite par l'algorithme 3. Elle parcourt toutes les instructions du programme ; lorsqu'une instruction non idempotente est identifiée, elle re-exécute les passes de transformations permettant de la rendre idempotente. Elle recommence le parcours tant que des instructions non idempotentes sont détectées. Cette boucle de rétroaction est illustrée sur la figure 5.1.

Algorithm 3 Passe de vérification pré-duplication

```
1: function instPreDuplication(listMachineFunctions)
2:     stop ← True
3:     for all MF in listMachineFunctions do
4:         if MF in ToleranceContext then
5:             repeat
6:                 for all MBB in listMachineBasicBlocks(MF) do
7:                     for all MI in listMachineInstruction(MBB) do
8:                         if isNotIdempotent(MI) then
9:                             transform(MI)
10:                            stop ← False
11:                        end if
12:                    end for
13:                end for
14:            until stop is True
15:        end if
16:    end for
17: end function
```

5.2.6 Passe de duplication d'instructions

Algorithm 4 Duplication des instructions

```
1:  function instDuplication(listMachineFunctions)
2:      workstack ← new stack( )                          ▷ initialisation d'une pile
3:      for all MF in listMachineFunctions do
4:          if MF in ToleranceContext then
5:              for all MBB in listMachineBasicBlocks(MF) do
6:                  for all MI in listMachineInstruction(MBB) do
7:                      push(workstack, MI)               ▷ pousser MI dans la pile
8:                  end for
9:              end for
10:         end if
11:     end for
12:     while notEmpty(workstack) do
13:         MI ← pop(workstack)                           ▷ retirer MI de la pile
14:         MI' ← duplicate(MI)                           ▷ dupliquer MI
15:         insertInstrAfter(MI', MI)                     ▷ insérer MI' après MI
16:     end while
17: end function
```

Le rôle de cette passe est de dupliquer toutes les instructions contenues dans les sections annotées du programme. Le processus de duplication, qui traite le programme fonction par fonction, est décrit par l'algorithme 4. Il consiste, pour chaque instruction MI, à créer un duplicata MI', puis à insérer MI' juste après MI. Nous commençons par pousser dans la pile *workstack* les adresses de toutes les instructions de la fonction (lignes 3 à 11), puis nous parcourons *workstack* et dupliquons toutes les instructions (ligne 12 à 16). La pile *workstack* a été utilisée pour éviter de modifier la liste des instructions pendant que nous la parcourons, car l'altération d'une structure de données en cours d'itération est une pratique de programmation fortement déconseillée [73].

5.2.7 Passe de vérification post-duplication

Le rôle de cette passe est de vérifier que toutes les instructions du programme sont dupliquées. Elle est placée après la passe de duplication et juste avant celle d'émission de code. À l'image de la passe de vérification pré-duplication (section 5.2.5), cette passe agit comme une barrière de contrôle ne permettant de passer à la suite que si toutes les instructions sont dupliquées. Elle apporte une garantie supplémentaire de l'intégrité du schéma de tolérance dans le code final (c'est-à-dire après toutes les optimisations). Lorsqu'une instruction non dupliquée est détectée, la passe de duplication est rappelée pour dupliquer cette instruction. La figure 5.1 montre l'emplacement de cette passe dans le compilateur.

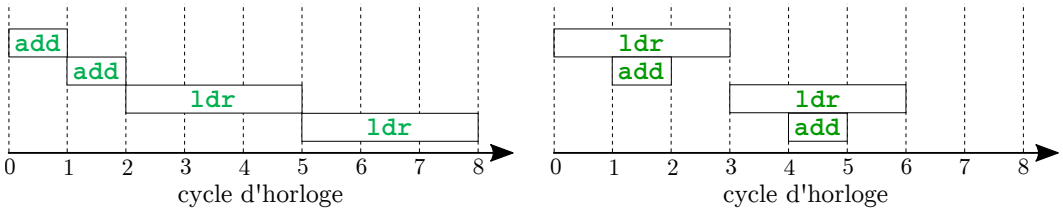
```
1 add r0, r1, r2
2 add r0, r1, r2
3 ldr r3, [r4]
4 ldr r3, [r4]
```

(a) Avant ordonnancement

```
1 ldr r3, [r4]
2 add r0, r1, r2
3 ldr r3, [r4]
4 add r0, r1, r2
```

(b) Après ordonnancement

FIGURE 5.6 – Ordonnancement des instructions dupliquées



(a) Avant ordonnancement (b) Après ordonnancement

FIGURE 5.7 – Impact du l'ordonnancement des instructions dupliquées sur le temps d'exécution

5.2.8 Ordonnancement

Nous avons fait le choix d'insérer la passe de duplication d'instructions avant la phase d'ordonnancement d'instructions (présentée dans la section 3.1.3). La figure 5.6 illustre l'ordre d'exécution des instructions dupliquées avant et après l'ordonnancement. Cette disposition présente des avantages sur le temps d'exécution du programme et sur le niveau de sécurité apporté par le schéma.

5.2.8.1 Avantage en temps d'exécution

L'ordonnanceur d'instructions (*scheduler*) vise à réduire le temps d'exécution d'un programme en établissant un ordre d'exécution des instructions qui exploite au mieux les ressources du processeur et qui évite autant que possible des arrêts inutiles du pipeline d'instructions (*pipeline stall*). Par conséquent, dupliquer les instructions avant la phase d'ordonnancement permet de bénéficier des avantages de l'ordonnanceur. La figure 5.7 montre une représentation des instructions de la figure 5.6 (avant et après l'ordonnancement) basée sur leurs temps d'exécutions en cycles d'horloge. Elle montre aussi, sur cet exemple, que le temps d'exécution des instructions après l'ordonnancement est réduit de 25 % par rapport à celui avant l'ordonnancement.

5.2.8.2 Avantage sur le niveau de sécurité

Dupliquer les instructions une fois permet de résister contre le saut d'une instruction, et N fois contre le saut de N instructions consécutives. Or, si grâce au ré-ordonnancement des instructions chaque instruction originale est séparée de son

duplicata d'une distance minimale équivalente à N instructions, alors une seule duplication suffirait pour résister à un saut de N instructions consécutives. Le problème est que l'ordonnanceur d'instructions ne déplace une instruction que si cela permet d'améliorer le temps d'exécution. Il n'y a donc aucune garantie de respecter une distance minimale entre les instructions dupliquées. Ce problème est résolu dans le chapitre 6 qui propose une généralisation de ce schéma.

5.3 Expérimentations

Nous avons mis en œuvre ce schéma de tolérance sur trois codes de références :

— l'AES 8 bits, utilisé par Moro et al. [76, 75].

— un AES 32 bits faisant partie de la suite de benchmarks de l'université de Michigan (Mibench) [53], aussi utilisé par Moro et al.

— une implémentation d'un algorithme de vérification de code PIN (*Verfiy-PIN*) tiré du projet SERTIF [94].

Nous avons ensuite réalisé une évaluation de performances (mesure des coûts en temps d'exécution et en taille de code) sur un microcontrôleur STM32 [69] embarquant un processeur ARM Cortex-M3, puis une évaluation de sécurité en utilisant un simulateur de fautes. Ces évaluations sont détaillées dans les sections suivantes.

L'évaluation de performances a été réalisée en utilisant le protocole décrit en section suivante.

Protocole de mesure

La taille du code est mesurée avec l'outil `arm-none-eabi-size` qui fait partie de la chaine d'outils *Sourcery CodeBench Lite 2013.11-24* [72]. Les mesures sont faites avant l'édition des liens (sur le fichier .o) pour mesurer uniquement le code concerné par la contre-mesure, sans ses dépendances.

Le nombre de cycles d'horloge est mesuré en utilisant l'unité *Data Watchpoint and Trace* (DWT) présent dans les processeurs Cortex-M [105]. Le DWT du processeur cortex-M3 fournit 3 registres de 32 bits, parmi lesquels, le registre `DWT_CYCCNT`, mappé à l'adresse mémoire `0xE0001004`, qui comptabilise le nombre de cycles écoulés. Pour mesurer le nombre de cycles que prend une opération, nous commençons par réinitialiser le registre `DWT_CYCCNT` à 0 (pour éviter un débordement), puis nous faisons la différence entre le contenu de `DWT_CYCCNT` avant et après l'opération. Pour avoir une mesure plus précise, nous soustrayons de la valeur obtenue le nombre de cycles nécessaires pour lire le contenu de `DWT_CYCCNT`.

Chaque programme à évaluer est compilé en faisant varier les niveaux d'optimisation, qui vont de -O0 (pas d'optimisation en vitesse d'exécution) à -O3 (optimisation maximale en vitesse d'exécution). L'option -Os vise à optimiser la taille du code (*size*).

TABLE 5.1 – Surcoût en taille code (en octets) et en temps d'exécution (en cycles d'horloge) pour chaque implémentation. Les deux dernières colonnes reportent les résultats présentés par Moro et al. [76]

	Opt.	Référence		Protégé		Surcoût		Moro et al [76]	
		Temps	Taille	Temps	Taille	Temps	Taille	Temps	Taille
AES 8 bits	-O0	17940	1736	29796	3960	×1,66	×2,28	×2,14	×3,02
	-O1	9814	1296	18922	2936	×1,92	×2,26		
	-O2	5256	1936	9934	4184	×1,89	×2,16		
	-O3	5256	1936	9934	4184	×1,89	×2,16		
	-Os	7969	1388	16084	3070	×2,02	×2,21		
AES 32 bits	-O0	1890	6140	3502	13012	×1,85	×2,12	×2,86	×2,90
	-O1	1226	3120	2172	7540	×1,77	×2,42		
	-O2	1142	3120	2111	7540	×1,85	×2,42		
	-O3	1142	3120	2111	7540	×1,85	×2,42		
	-Os	1144	3116	2111	7512	×1,85	×2,41		
Verify-PIN	-O0	212	248	350	510	×1,65	×2,05		
	-O1	101	144	180	300	×1,78	×2,08		
	-O2	42	200	77	440	×1,83	×2,20		
	-O3	42	200	77	440	×1,83	×2,20		
	-Os	81	180	155	365	×1,91	×2,02		

5.3.1 Évaluation de performances

Nous définissons le surcoût d'une contre-mesure comme le rapport entre le coût du code protégé et celui du code de référence (sans contre-mesure), en taille de code et en temps d'exécution. Moro et al. ayant utilisé les mêmes implémentations de AES sur le même processeur cible, nous allons pouvoir comparer les performances de leur mise en œuvre par approche code binaire à la nôtre. Le tableau 5.1 présente nos différentes mesures pour chaque implémentation.

Les colonnes **Temps** et **Taille** reportent respectivement le temps d'exécution en nombre de cycles d'horloge et la taille du code en octets. Les deux dernières colonnes reportent les mesures faites par Moro et al. [76].

Chaque niveau d'optimisation du compilateur correspond à un surcoût donné en taille de code et en temps d'exécution. Nous n'avons pas utilisé le même compilateur que Moro et al. [76]. Par conséquent, il est difficile de savoir lesquels de nos surcoûts sont à comparer aux leurs. Néanmoins, nous pouvons constater que dans tous les cas, nos surcoûts sont plus faibles.

Nous pouvons remarquer aussi qu'il existe une corrélation entre le niveau d'optimisation et le surcoût en temps d'exécution : plus le code est optimisé par le compilateur, plus le surcoût est élevé. Cette augmentation s'explique par le fait que dans un code non optimisé, il y a plus d'instructions dont la duplication n'impacte pas le temps d'exécution, que dans un code optimisé. C'est le cas des instructions de changement de flot de contrôle (instructions de saut) qui ne sont exécutées qu'une seule fois même si elles sont dupliquées plusieurs fois, par exemple `b, bx`. C'est aussi

le cas des instructions de retour de fonction, telle que `pop{pc}` ou `mv pc, rn`. Pour toutes ces instructions, une fois l'instruction originale exécutée, le flot d'exécution est transféré vers un autre endroit du programme, ce qui empêche l'exécution de l'instruction dupliquée et par conséquent, amortit le surcoût en temps d'exécution. D'autre part, pour améliorer le temps d'exécution, les optimiseurs de code, lorsqu'ils sont activés tendent à réduire le nombre d'instructions de saut dans le code, en fusionnant des blocs de base ou en faisant du *inlining*, par exemple.

5.3.1.1 Impact de la génération d'instructions idempotentes

On peut remarquer sur le tableau 5.1 que, malgré la duplication de toutes les instructions, le surcoût en temps d'exécution engendré par notre implémentation est toujours inférieur à $\times 2$, sauf pour l'optimisation `-Os` pour l'AES 8 bits. Ce faible coût est principalement dû au ré-ordonnancement des instructions après la duplication. Mais, de manière générale, la réduction des coûts par rapport à l'approche code binaire est due à la génération d'instructions ne nécessitant aucune transformation (instructions idempotentes). Avec l'option d'optimisation `-O2`, l'AES 8-bit de référence compte en tout 172 instructions machine, dans lesquelles 84 (49 %) ne sont pas idempotentes. En effet, avec les modifications de la sélection d'instructions et de l'allocateur de registres, ce même AES compte en tout 181 instructions dans lesquelles 175 (97%) sont déjà idempotentes avant les passes de transformations. Ce qui laisse seulement 3% des instructions à transformer, contre 51% si le schéma avait été intégré dans le code binaire.

5.3.2 Évaluation de sécurité

L'évaluation de sécurité a été réalisée avec un simulateur de fautes (CELTIC) [41] qui fonctionne sur le même principe que le simulateur présenté dans le chapitre 4.

Le programme verify-PIN prend en entrée un PIN qu'il compare au PIN valide. Si les deux PINs sont identiques, la variable `g_authenticated` est mise à *true*, sinon elle est mise à *false*. Pour évaluer ce programme, nous lui fournissons en entrée un PIN différent du PIN valide. Ainsi, on considère qu'une faute est réussie, si après l'exécution du verify-PIN la variable `g_authenticated` est à *true*.

Nous avons réalisé l'évaluation de sécurité selon deux approches : une première qui consiste à appliquer le schéma de tolérance sur l'ensemble du programme (*approche exhaustive*), et une seconde qui consiste à utiliser le rapport fourni par le simulateur pour sécuriser uniquement les sections vulnérables du programme (*approche rétroactive*). Dans les deux cas, le modèle de fautes utilisé est le saut d'une instruction une seule fois par exécution, noté `SKIP{1, 1}`

5.3.2.1 Approche exhaustive

Avec cette approche nous fournissons au simulateur un code binaire dans lequel toutes les instructions sont dupliquées. Le simulateur effectue autant d'exécutions

TABLE 5.2 – Comparaison (pour des niveaux de sécurité équivalents) des approches *exhaustive* et *rétroactive* sur les surcoûts (en taille/temps d'exécution) engendrés par la duplication d'instructions.

	Opt.	Approche exhaustive		Approche rétroactive	
		Temps	**Taille**	**Temps**	**Taille**
Verify-PIN	-O0	×1.65	×2.05	×1.04	×1.03
	-O1	×1.78	×2.08	×1.14	×1.07
	-O2	×1.83	×2.20	×1.17	×1.10
	-O3	×1.83	×2.20	×1.17	×1.10
	-Os	×1.91	×2.02	×1.23	×1.12

(*run*) que d'instructions du programme. Il s'avère que lorsque toutes les instructions sont dupliquées, le saut d'une instruction n'affecte pas la fonctionnalité du programme (`g_authenticated` est toujours à `False` après chaque exécution).

5.3.2.2 Approche rétroactive

Avec cette approche, c'est le code binaire de référence (sans contre-mesure) qui est fourni au simulateur. Pour chaque exécution, lorsqu'une vulnérabilité est détectée, l'instruction sautée lors de cette exécution est mémorisée. À la fin de la simulation, les informations sur les instructions mémorisées (instructions vulnérables) sont fournies au compilateur. Ce dernier produit un nouveau code binaire dans lequel les instructions vulnérables sont dupliquées. Le nouveau code binaire est fourni au simulateur, et le processus recommence jusqu'à ce qu'aucune faute ne réussisse.

Nous considérons que deux binaires ont des niveaux de sécurité équivalents lorsque pour le même modèle de fautes, les rapports produits par le simulateur pour ces deux binaires sont identiques. Le tableau 5.2 présente une comparaison des surcoûts, en temps d'exécution et en taille de code, de deux codes binaires du verify-PIN. L'un est sécurisé en utilisant l'approche exhaustive et l'autre en utilisant l'approche rétroactive. Les résultats présentés par ce tableau illustrent l'avantage de l'approche rétroactive en termes de réduction des coûts d'une contre-mesure.

5.4 Conclusion

Dans ce chapitre nous avons présenté une mise en œuvre par approche compilation du schéma de tolérance aux fautes formellement vérifié par Moro et al [75]. Le schéma consiste à dupliquer les instructions après les avoir transformées en forme idempotente.

Nous avons validé expérimentalement cette mise en œuvre vis-à-vis du modèle saut d'instructions et avons montré sur un exemple qu'elle permettait de réduire les coûts en temps d'exécution et en taille de code comparativement à l'approche Moro et al. [76].

Nous avons constaté que ce gain est essentiellement obtenu grâce à la modification de la passe de sélection d'instructions et à l'optimisation réalisée par l'ordonnanceur d'instructions.

Dans le chapitre suivant, nous proposons une généralisation de ce schéma de tolérance pour protéger contre le saut de plusieurs instructions consécutives ou non.

Chapitre 6

Généralisation du schéma de tolérance au saut d'instructions

Sommaire

6.1 Modèles de fautes 76
6.2 Description du schéma de protections 77
6.2.1 Modèle de protection `protect_skip{A, B}` 77
6.2.2 Modèle de protection `protect_skip_block{size, M}` 78
6.3 Mise en œuvre 79
6.4 Expérimentations 80
6.4.1 Évaluation de sécurité 82
6.4.2 Évaluation de performances 82
6.5 Conclusion 83

Le schéma de tolérance au saut d'instructions mis en œuvre dans le chapitre précédent consiste à dupliquer les instructions du programme après les avoir transformées dans des formes idempotentes. Cependant, ce schéma ne protège que contre le saut d'une seule instruction, et sa robustesse a été mise à l'épreuve dans l'état de l'art : Yuce et al. [106] ont montré qu'il est possible de sauter les deux instructions (originale et dupliquée) avec une seule faute injectée dans le pipeline du processeur. Rivière et al. [91] ont montré qu'en empêchant la mise à jour du tampon de préchargement d'instruction (*buffer prefetch*) il est possible de sauter quatre instructions consécutives, 4×32 bits étant la taille de ce tampon sur le processeur cible (ARM Cortex-M4).

Ce chapitre propose une généralisation du schéma de tolérance décrit dans le chapitre précédent pour couvrir les fautes multiples (multi-fautes) et les fautes sur plusieurs instructions consécutives. Nous avons vu dans le chapitre précédent (section 5.2.8.1) que dupliquer les instructions avant la phase d'ordonnancement permet de réduire le temps d'exécution grâce à la réorganisation de l'ordre d'exécution des instructions. La mise en œuvre que nous proposons dans ce chapitre tire parti de l'ordonnanceur d'instruction pour que chaque instruction dupliquée soit placée à une distance D de son original. En paramétrant D en fonction des caractéristiques du processeur cible, il est possible, par exemple, de faire en sorte qu'une instruction originale et son duplicata ne se retrouvent pas en même temps dans le tampon de préchargement d'instruction protégeant ainsi contre l'attaque de Rivière et al [91], ou dans le pipeline pour protéger contre Yuce et al. [106].

La section 6.1 précise les modèles de faute considérés dans ce chapitre. La section 6.2 présente les modèles de protections proposés contre les modèles de fautes considérés. La section 6.3 présente la mise en œuvre des modèles de protections dans le compilateur. La section 6.4 présente les résultats expérimentaux en termes de sécurité et de performance.

6.1 Modèles de fautes

Dans ce chapitre, l'objectif est de généraliser le modèle saut d'instructions (`skip{1, 1}`) utilisé dans le chapitre précédent. En utilisant les modélisations introduites dans la section 4.2, on considère les modèles suivants :

— `skip{N, M}` : `M` sauts de `N` instructions consécutives.

— `skip_block{size, M}` : saut de `M` blocs logiques de taille `size`.

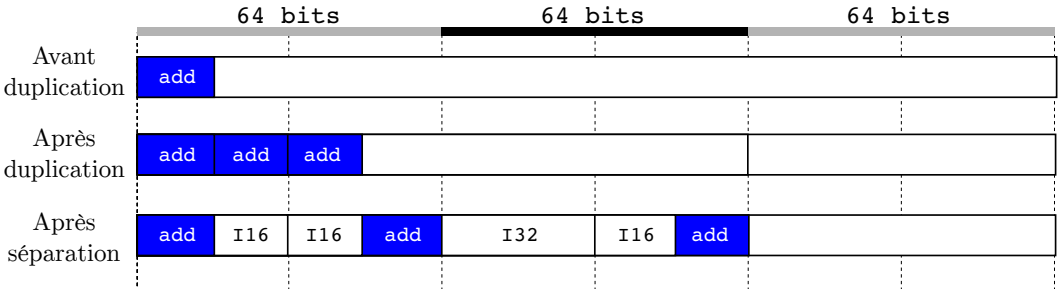


FIGURE 6.1 – Illustration du modèle `protect_skip{2, 2}`

6.2 Description du schéma de protections

Cette section présente le schéma de protection contre les modèles de fautes présentées dans la section précédente. Nous proposons un modèle de protection contre le modèle de fautes `skip{N, M}` (que l'on notera `protect_skip{A, B}`) et un modèle de protection contre le modèle de fautes `skip_block{size, M}` (que l'on notera `protect_skip_block{size, M}`), ces modèles sont présentés ci-dessous.

Notations :

- I_{org} et I_{dup} : une instruction originale et son duplicata (sa copie).
- $addr(I_{org})$ et $addr(I_{dup})$: adresses mémoire de I_{org} et I_{dup}.
- $nbNop_{(I_{org},I_{dup})}$: Nombre d'instructions `nop` à insérer entre I_{org} de I_{dup}.

6.2.1 Modèle de protection `protect_skip{A, B}`

modèle de fautes `skip{N, 1}` : Pour protéger contre ce modèle, une première solution consiste à dupliquer `N` fois chaque instruction du programme. Cela permet d'avoir une instruction originale et `N` duplicatas par instruction ($1 \times I_{org} +$ `N`$\times I_{dup}$), ce qui correspond à `N+1` instructions. Ainsi, un saut de `N` instructions consécutives épargnera toujours au moins un exemplaire des instructions sautées.

La seconde solution, qui est celle que nous avons mise en œuvre, consiste à dupliquer une seule fois chaque instruction, puis de déplacer chaque I_{dup} à `N-1` instructions plus loin de son I_{org}. Autrement dit, faire en sorte que le nombre d'instructions entre I_{org} et I_{dup} ($nbInstr_{(I_{org},I_{dup})}$) soit égal à `N-1`. Par la suite, on dira que le duplicata I_{dup} est à une distance de `N-1` instructions par rapport à son original I_{org}.

modèle de fautes `skip{1, M}` : Pour protéger contre ce modèle, il suffit de dupliquer `M` fois chaque instruction.

Nous proposons comme modèle de protection `protect_skip{A, B}` contre le modèle de fautes `skip{N, M}`, qui consiste à dupliquer `B` fois chaque instruction, avec `B`

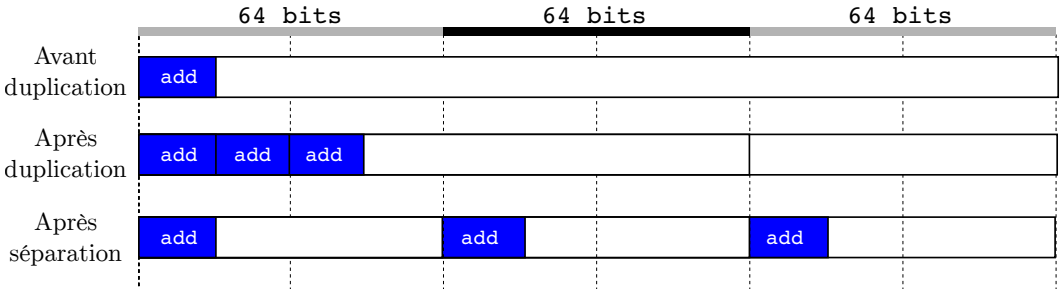


FIGURE 6.2 – Illustration du modèle `protect_skip_block{64 bits, 2}`

`= M`, puis séparer les duplicatas les uns des autres d'une distance de `A` instructions, avec `A = N - 1`.

Dupliquer `B` fois une instruction revient à faire `B` copies de cette instruction. Pour laisser une distance de `A` instructions entre I_{org} et I_{dup} nous insérerons des instructions `nop` entre elles. Cela aurait pu être toute instruction qui n'affecte pas la fonctionnalité du programme comme celles présentées dans la section 2.2.3. Le nombre d'instructions $nbNop_{(I_{org},I_{dup})}$ à insérer entre I_{org} et I_{dup} dépend du nombre d'instructions $nbInstr_{(I_{org},I_{dup})}$ qui se trouvent déjà entre elles, soit :

$$nbNop_{(I_{org},I_{dup})} = \texttt{A} - nbInstr_{(I_{org},I_{dup})} \tag{6.1}$$

La figure 6.1 illustre ce modèle de protection pour `A = 2` et `B = 2`.

6.2.2 Modèle de protection `protect_skip_block{size, M}`

Le modèle de fautes `skip_block{size, M}` (présenté dans la section 2.2.3.3) considère que les blocs sautés sont alignés sur la taille d'un mot mémoire (32 bits). Pour protéger contre ce modèle, il faut réaliser `M` duplications, puis faire en sorte que deux duplicatas ne se trouvent pas dans un même bloc logique, en les séparant.

Soit n le nombre de duplicatas. Pour vérifier que les n duplicatas sont dans des blocs différents, on vérifie successivement que I_{org} est dans un bloc différent de $I_{dup\{1\}}$, puis que $I_{dup\{i\}}$ est dans un bloc différent de $I_{dup\{i-1\}}$ pour $i \in [2, n]$. C'est une condition suffisante parce qu'on sait que, avant l'application de la séparation, $I_{dup\{i\}}$ est à une adresse strictement supérieure à $I_{dup\{i-1\}}$ quel que soit i. Cette hypothèse nous permet de faire n vérifications de distance (plutôt que $\frac{n*(n-1)}{2}$ dans le cas général).

Par la suite, on dira que I_{org} et I_{dup} sont à une distance $D_{(I_{org},I_{dup})} = 0$ lorsqu'elles sont dans le même bloc logique et à une distance $D_{(I_{org},I_{dup})} \neq 0$ lorsqu'elles sont dans des blocs logiques différents. On utilisera le mot *bloc* pour désigner un bloc logique. Soit `ET`, `OU` et `NON` respectivement le ET, le OU et le NON booléen.

Calcul de $D_{(I_{org},I_{dup})}$:

Si n est le nombre de bits nécessaires pour encoder `size` valeurs ($n = \log_2(\texttt{size})$), où `size` est une puissance de 2, alors toutes les adresses contenues dans un bloc de

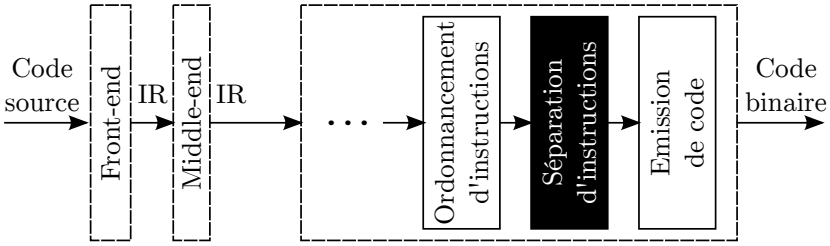


FIGURE 6.3 – Structure simplifiée du compilateur, montrant la passe *séparation d'instructions* et faisant abstractions des passes déjà présentées dans le chapitre 5

taille `size` ne diffèrent que de leurs n derniers bits (bits de poids faibles).

Par conséquent, une manière de calculer $D_{(I_{org},I_{dup})}$ est de déterminer si $addr(I_{org})$ et $addr(I_{dup})$ ne diffèrent que de leurs n derniers bits. Une façon de faire est de fixer les n derniers bits de chaque adresse à 0, puis de vérifier si les deux adresses sont identiques.

$$D_{(I_{org},I_{dup})} = (addr(I_{org}) \texttt{ ET } m) \oplus (addr(I_{dup}) \texttt{ ET } m) \quad (6.2)$$
$$\text{où } m = \texttt{NON(size} - 1)$$

Calcul de $nbNop_{(I_{org},I_{dup})}$:

Calculer $nbNop_{(I_{org},I_{dup})}$ consiste à déterminer le plus petit décalage que $addr(I_{dup})$ doit subir pour ne plus être dans la même bloc que $addr(I_{org})$. Cette opération revient à calculer la première prochaine adresse ($newAddr(I_{dup})$) qui ne satisfait pas l'équation 6.2. Une manière de calculer cette adresse est formulée par l'équation 6.3.

$$newAddr(I_{dup}) = (addr(I_{dup}) \texttt{ OU } m) + 1 \quad (6.3)$$

Ainsi l'équation 6.4 permet de calculer le nombre d'instructions `nop` de taille $sizeof(I_{dup})$ à insérer entre I_{org} et I_{dup}.

$$nbNop_{(I_{org},I_{dup})} = \frac{newAddr(I_{dup}) - (addr(I_{dup})}{sizeof(\texttt{nop})} \quad (6.4)$$

6.3 Mise en œuvre

La description présentée par la section précédente est mise en œuvre par la passe *Séparation d'instructions* représentée sur la figure 6.3.

Pour réduire le nombre d'instructions `nop` à insérer (et ainsi minimiser le coût du schéma), cette passe a été placée après la phase d'ordonnancement des instructions. En effet, comme introduit dans le chapitre précédent (section 5.2.6), l'ordonnanceur d'instructions permet de réduire le temps d'exécution du programme en réorganisant l'ordre d'exécution des instructions. Par effet de bord, cela peut créer une distance entre les I_{org} et leurs I_{dup} respectives. L'objectif de cette passe est d'ajuster cette distance conformément au schéma de protection.

TABLE 6.1 – Résultats de simulation du modèle de fautes `SKIP{N, M}` sur l'AES 32 bits[53] protégé avec le modèle de protection `protect_skip{A, B}`

	`SKIP{1, 1}`	`SKIP{2, 1}`	`SKIP{1, 2}`	`SKIP{2, 2}`
`protect_skip{0, 1}`	✓	✗	✗	✗
`protect_skip{1, 1}`	✓	✓	✗	✗
`protect_skip{0, 2}`	✓	✓	✓	✗
`protect_skip{1, 2}`	✓	✓	✓	✓

Le fonctionnement de cette passe est décrit par l'algorithme 5, qui s'applique sur les instructions machine dupliquées. Pour chaque instruction machine MI, cette passe identifie tous les couples $\{(MI_1, MI_2), (MI_2, MI_3), ..., (MI_{(n-1)}, MI_n)\}$ où n est le nombre de duplicatas. Lors du parcours des instructions, il est possible de différencier une instruction originale de ses dupliquées grâce aux modifications que nous avons introduites dans la classe `llvm::MachineInstr`. Nous y avons ajouté des attributs permettant d'identifier une instruction originale et d'obtenir la liste de ses duplicatas. Un extrait de cette classe montrant cette modification est présentée ci-dessous.

Listing 6.1 – Extrait de la classe `llvm::MachineInstr`

```
namespace llvm{
class MachineInstr : public ilist_node<MachineInstr> {
    private:
        unsigned int duplicationRank = 0;
        MachineInstr* nextDuplicate = nullptr;
...
    public:
        bool isOriginal() const {
            return this->duplicationRank == 0;
        }
...
```

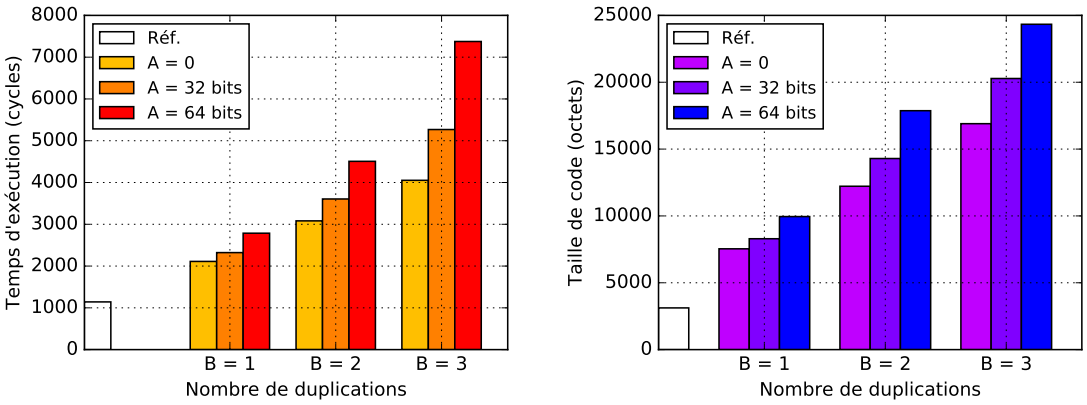
Puis des instructions `nop` sont éventuellement insérées entre chaque couple pour satisfaire le modèle de protection considéré.

6.4 Expérimentations

Nous avons réalisé une évaluation de sécurité, avec le simulateur de faute présenté dans la section 4, pour valider les modèles de protection vis-à-vis des modèles de fautes considérés et une évaluation de performances pour déterminer le coût en temps d'exécution et en taille de code engendré par l'application des modèles de protection sur un programme. Les évaluations sont réalisées sur l'implémentation AES 32 bits de Mibench [53] utilisée dans le chapitre précédent.

Algorithm 5 Séparation d'instructions

```
1: function instSeparation(listMachineInstructions, protectionModel)
2:     for all MI in listMachineInstructions do
3:         if isOriginal(MI) then
4:             MI_i ← MI
5:             MI_{i+1} ← NULL
6:             nbNop ← 0
7:             while hasNextDuplicate(MI_i) do
8:                 MI_{i+1} ← getNextDuplicate(MI_i)
9:                 if protectionModel = protect_skip then
10:                    nbInstr ← getNbInstrBetween(MI_i, MI_{i+1})
11:                    nbNop ← getNbNopToInsert(nbInstr)            ▷ eq. 6.1
12:                else                                  ▷ protectionModel = skip_block
13:                    if notInSameBlock(MI_i, MI_{i+1}) then       ▷ eq. 6.2
14:                        newAddr ← getNextValidAddr(MI_{i+1})     ▷ eq. 6.3
15:                        nbNop ← getNbNopToInsert(newAddr, addr(MI_{i+1}))
16:                                                                 ▷ eq. 6.4
17:                    end if
18:                end if
19:                insertNopBetween(nbNop, MI_i, MI_{i+1})
20:                MI_i ← MI_{i+1}
21:            end while
22:        end if
23:    end for
24: end function
```



(a) Temps d'exécution (b) Taille du code

FIGURE 6.4 – Impact sur le temps d'exécution et la taille du code de l'AES protégé avec le modèle `protect_skip{A, B}`

6.4.1 Évaluation de sécurité

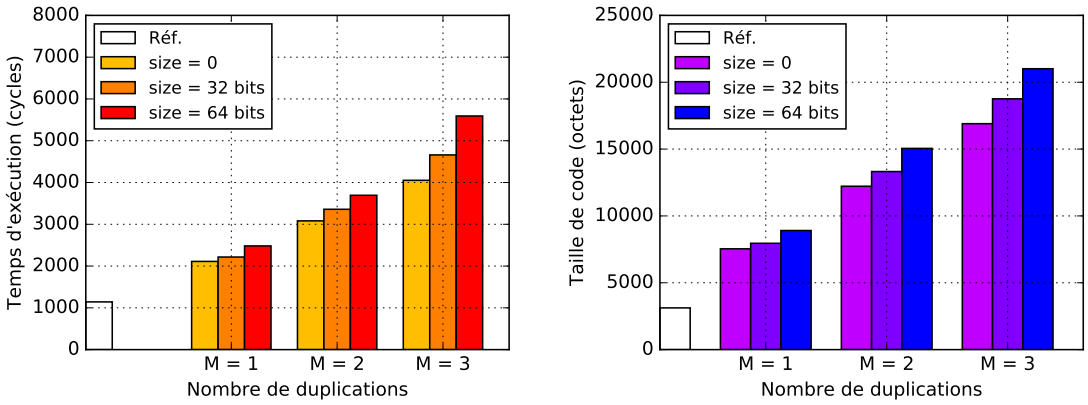
Le tableau 6.1 présente les résultats de simulation du modèle `SKIP{N, M}` sur le programme AES 32 bits [53] protégé avec le modèle de protection `protect_skip{N-1, M}`. Pour rappel, la notation `SKIP{N, M}` (décrite dans la section 4.2) désigne l'évaluation exhaustive du modèle de fautes `skip{N, M}`.

Lorsqu'une configuration du modèle de protection protège contre un modèle de fautes, la case correspondante dans le tableau est marquée par le symbole ✓. Le cas échéant, elle est marquée par le symbole ✗. Ces résultats montrent que le modèle de protection `protect_skip{N-1, M}` protège bien contre le modèle de fautes `skip{N, M}`.

6.4.2 Évaluation de performances

Nous avons réalisé l'évaluation de performances sur le microcontrôleur STM32 [69]. Les figures 6.4 et 6.5 montrent, pour chaque modèle de protection, l'augmentation du temps d'exécution et la taille du code en fonction du nombre de duplications et de la distance insérée entre les instructions.

Il apparait donc avantageux d'insérer des distances entre les instructions plutôt que d'augmenter le nombre de duplicatas. On voit sur ces figures que l'augmentation de la distance entre les instructions fait croître le temps d'exécution et la taille du code moins vite que l'augmentation du nombre de duplicatas. Cela s'explique par le fait que, d'une part, les instructions `nop` insérées s'exécutent sur 1 cycle, d'autre part, la duplication a été effectuée avant la phase d'ordonnancement qui insère déjà une certaine distance entre les instructions dans un but d'optimisation.



(a) Temps d'exécution

(b) Taille du code

FIGURE 6.5 – Impact sur le temps d'exécution et la taille du code de l'AES protégé avec le modèle `protect_skip_block{size, M}`

6.5 Conclusion

Dans ce chapitre, nous avons présenté une généralisation du schéma de tolérance au saut d'instructions présenté dans le chapitre précédente. Nous avons proposé et mis en œuvre deux modèles de protections : le modèle `protect_skip{A, B}` pour protéger contre le modèle de fautes `skip{N, M}` et le modèle `protect_skip_block{size, M}` contre le modèle de fautes `skip_block{size, M}`. Nous avons validé expérimentalement ces modèles de protection en utilisant le simulateur de fautes présenté dans le chapitre 4.

Pour protéger contre le saut de `N` instructions consécutives, nous avons utilisé une technique consiste à dupliquer une seule fois chaque instruction, au lieu de `N` fois, et insérer une distance de `N-1` instructions entre les duplicatas. L'évaluation de performances montre que cette technique permet de réduire le coût de la protection sans affecter le niveau de sécurité attendu. Cette optimisation est rendue possible grâce à une mise en œuvre qui exploite les avantages de l'approche compilation.

Chapitre 7

Combinaison du schéma de tolérance avec de l'intégrité de flot de contrôle

Sommaire

7.1 Modèles de fautes 86
7.2 Notations 86
7.3 Description du schéma CFI 87
7.3.1 Principe général 88
7.3.2 Cas de plusieurs prédécesseurs 88
7.3.3 Cas de plusieurs successeurs 89
7.3.4 Combinaison avec le schéma de tolérance 91
7.4 Mise en œuvre 91
7.4.1 Identification des sections à sécuriser 92
7.4.2 Initialisation des compteurs 92
7.4.3 Incrémentation des compteurs 92
7.4.4 Vérification des compteurs 94
7.4.5 Réaction 95
7.5 Illustrations 95
7.6 Expérimentations 100
7.6.1 Évaluation de performances 100
7.6.2 Évaluation de sécurité 100
7.7 Conclusion 101

Dans ce chapitre, nous nous intéressons aux fautes pouvant être utilisées par un attaquant pour faire dévier le programme de son chemin d'exécution initial.

Nous proposons une mise en œuvre d'un schéma d'intégrité de flot contrôle (CFI) inspiré du schéma proposé par Lalande et al. [58].

Ensuite, nous proposons de combiner ce schéma avec le schéma de tolérance au saut d'instructions présenté dans les chapitres précédents. Nous montrons dans les expérimentations que cette combinaison augmente le niveau de sécurité du schéma CFI, sans pour autant impacter significativement le temps d'exécution et la taille du code.

7.1 Modèles de fautes

Le principe du CFI est de détecter les altérations du flot de contrôle d'un programme. Nous avons vu dans l'état de l'art qu'il est possible d'altérer une instruction de sorte que l'état du processeur après la faute soit équivalent à l'exécution d'une instruction de branchement. Pour représenter ce type de fautes, on utilise le modèle `repl_br{addr}` décrit dans la section 4.2, qui désigne le remplacement d'une instruction du programme par une instruction de branchement (`br`) à une adresse `addr`. En appliquant ce modèle à l'échelle de tout le programme (`REPL_BR`), on vérifie qu'aucune des instructions ne pourra être remplacée avec succès par une instruction de branchement.

Une attaque possible sur l'intégrité du flot de contrôle procède en deux temps : (1) altérer le flot de contrôle, (2) puis attaquer le code de vérification d'intégrité du flot de contrôle de telle sorte que l'altération ne soit pas détectée. On utilise donc le modèle `skip{N, M}` déjà utilisé dans le chapitre précédent, en combinaison avec le modèle `repl_br`, pour vérifier qu'il n'est pas possible de contourner le mécanisme de vérification d'intégrité à l'aide de `M` fautes de `N` instructions injectées après.

7.2 Notations

Cette section définit des notations qui seront utilisées tout au long de ce chapitre ainsi que dans le chapitre suivant.

— `BB#<i>` : désigne le bloc de base dont le numéro d'ordre est `i`.

— `entry` : le nom du premier bloc de base d'une fonction.

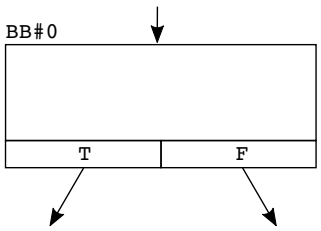


FIGURE 7.1 – Représentation d'un bloc de base

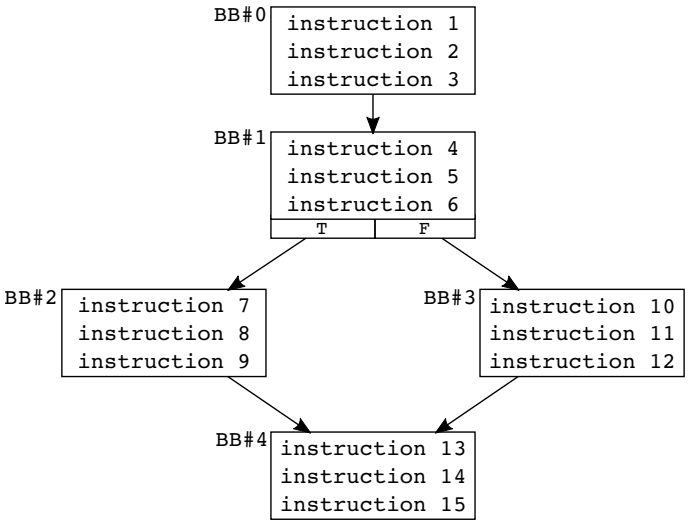


FIGURE 7.2 – CFG initial

— `#<VAR>` : une constante initialisé au moment de la compilation.

— `cnt<i>` : le compteur du bloc de base `BB#<i>`

— `cnt<i>++` : incrémentation du compteur `cnt<i>` de 1 (`cnt<i>` ← `cnt<i> + 1`)

La figure 7.1 représente un bloc de base possédant deux successeurs (arcs sortants) et un prédécesseur (arc entrant). La transition vers le premier ou le deuxième successeur est déterminée par une condition évaluée à la fin du bloc. Les labels `T` pour *True* et `F` pour *False* symbolisent la transition à réaliser selon que le résultat de la condition est respectivement vrai ou faux.

7.3 Description du schéma CFI

L'objectif du schéma CFI est de garantir que l'exécution d'un programme ne suive que des chemins prévus dans son graphe de flot de contrôle. Un *chemin* est une liste des points de passage empruntés lors de l'exécution d'un programme. La figure 7.2 illustre le CFG d'un programme. Les chemins d'exécutions prévus pour ce programme sont :

— `BB#0` → `BB#1` → `BB#2` → `BB#4`

— `BB#0` → `BB#1` → `BB#3` → `BB#4`

Le principe du schéma CFI est basé sur le calcul de signatures. Pour chaque bloc de base : (1) une signature lui est assignée, (2) cette signature est initialisée à la fin du bloc prédécesseur, (3) elle est mise à jour à l'intérieur du bloc à chaque point de passage et (4) sa validité est vérifiée à la fin du bloc de base, en la comparant à une valeur théorique pré-calculée (valeur attendue).

Un *sous-chemin* est associé à chacun des blocs de base. Chaque sous-chemin commence à l'initialisation d'une signature et se termine à la vérification de cette signa-

ture. L'initialisation d'une signature avant la vérification de la signature précédente permet le recouvrement des sous-chemins. Nous faisons les hypothèses suivantes :

— un sous-chemin est intègre si la valeur de sa signature lors de la vérification est conforme à la valeur attendue.

— un chemin est intègre si tous ses sous-chemins sont intègres.

Dans ce chapitre, nous utilisons un compteur comme signature. La mise à jour de la signature se fait par incrémentation du compteur.

Dans un CFG, un bloc de base peut avoir de zéro à plusieurs prédécesseurs/successeurs. On parle de configuration d'un bloc de base. Selon la configuration d'un bloc de base, le processus d'initialisation et de vérification de compteur peut varier par rapport au principe général présenté ci-après.

7.3.1 Principe général

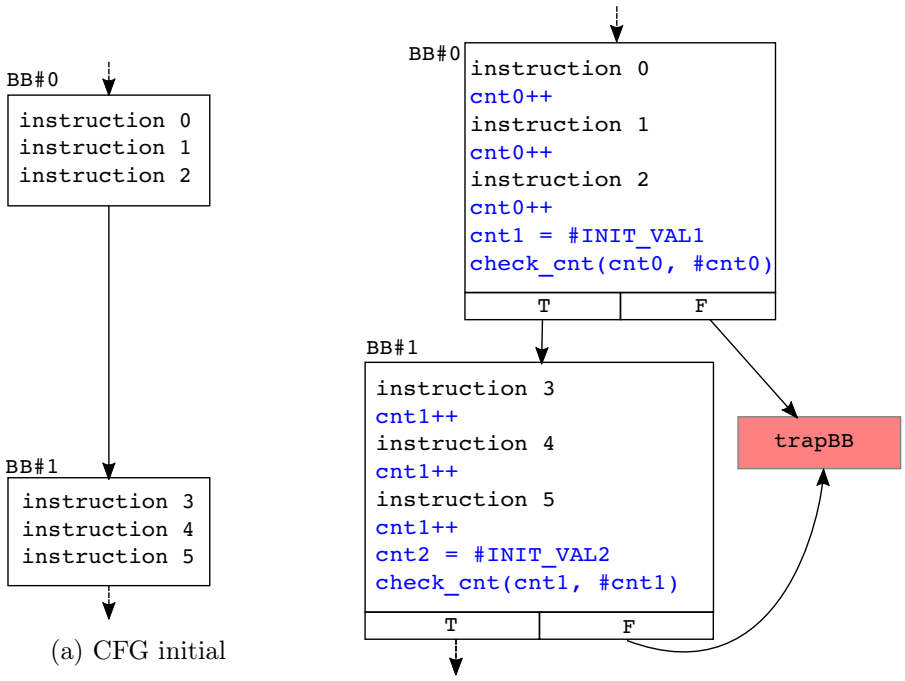
L'application du schéma CFI se déroule en trois étapes.

1. **Initialisation de compteurs :** un compteur unique est assigné à chaque bloc de base. Ce compteur est initialisé à la fin du bloc de base prédécesseur, à l'exception du bloc `entry`, pour lequel le compteur est initialisé en début du bloc.
2. **Incrémentation de compteurs :** le compteur de chaque bloc de base est incrémenté après chaque instruction de ce bloc.
3. **Vérification de compteurs :** La routine `check_cnt`, exécutée à la fin de chaque bloc de base vérifie la cohérence du compteur par rapport à sa valeur attendue au moment de la vérification. La valeur attendue d'un compteur en un point se calcule en additionnant sa valeur d'initialisation au nombre d'incrémentations effectuées jusqu'à ce point. La vérification est effectuée avant l'initialisation du compteur du bloc suivant. L'exécution du programme se poursuit normalement lorsque le compteur est vérifié avec succès ; dans le cas contraire, le flot d'exécution est redirigé vers un bloc de base spécifique (`trapBB`), contenant les instructions d'une routine de *réaction* à la faute. Le contenu de ce bloc peut être personnalisé par l'utilisateur.

La figure 7.3 illustre l'application du schéma sur bloc de base possédant un seul prédécesseur.

7.3.2 Cas de plusieurs prédécesseurs

Lorsqu'un bloc de base possède plusieurs prédécesseurs, son compteur est initialisé à la même valeur, à la fin de chacun de ses prédécesseurs. Sur la figure 7.4 qui illustre ce cas, le compteur `cnt2` du bloc de base `BB#2` est initialisé à la fois dans `BB#0` et `BB#1` à la valeur `#INIT_VAL2`.



(a) CFG initial

(b) CFG sécurisé

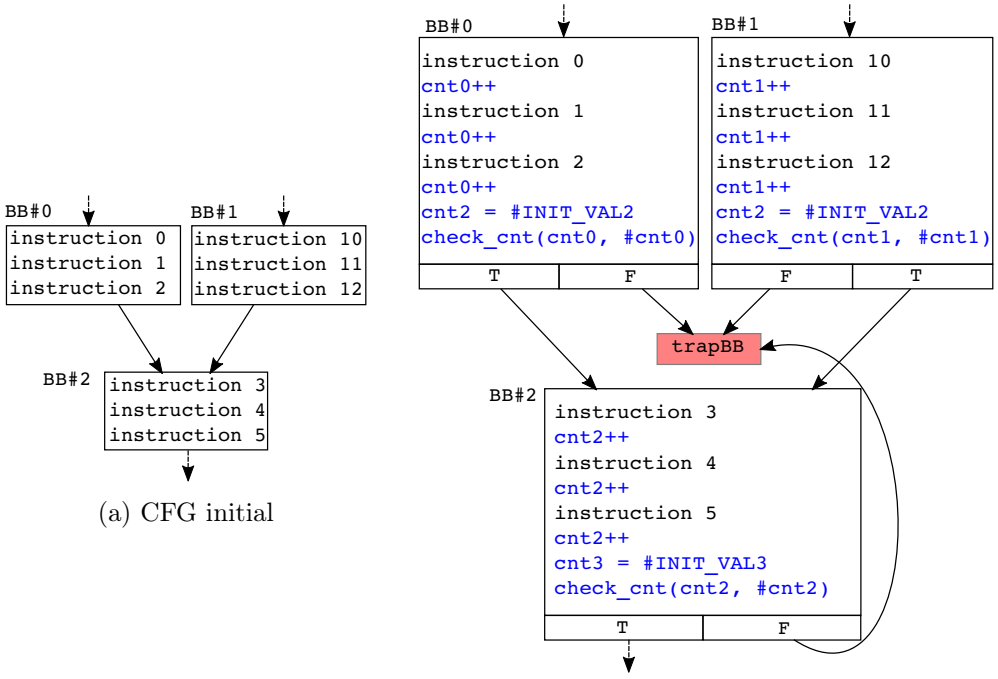
FIGURE 7.3 – Application du schéma CFI sur un bloc de base `BB#1` ne possédant qu'un seul prédécesseur

7.3.3 Cas de plusieurs successeurs

Un bloc de base qui possède plusieurs successeurs sous-entend que la transition vers un successeur plutôt qu'un autre est déterminée par une condition de saut évaluée dans ce bloc. Ce cas est illustré par la figure 7.5a. La question qui se pose est : faut-il évaluer le compteur avant ou après la condition de saut ? Par définition, un bloc de base ne peut avoir qu'un seul point de sortie. Effectuer la vérification du compteur dans le bloc de base, avant ou après le saut conditionnel, rajouterait une éventuelle sortie vers `trapBB` en cas d'échec, ce qui contredit la définition.

Une solution possible est d'insérer sur chaque arc sortant du bloc `BB#0`, un bloc de base intermédiaire dans lequel la validité du compteur `cnt0` sera vérifiée. Ce qui garantit la vérification du compteur indépendamment du résultat de la condition de branchement. Cependant, cette solution augmente le coût du schéma en taille de code, car on a deux routines de vérification pour le même compteur.

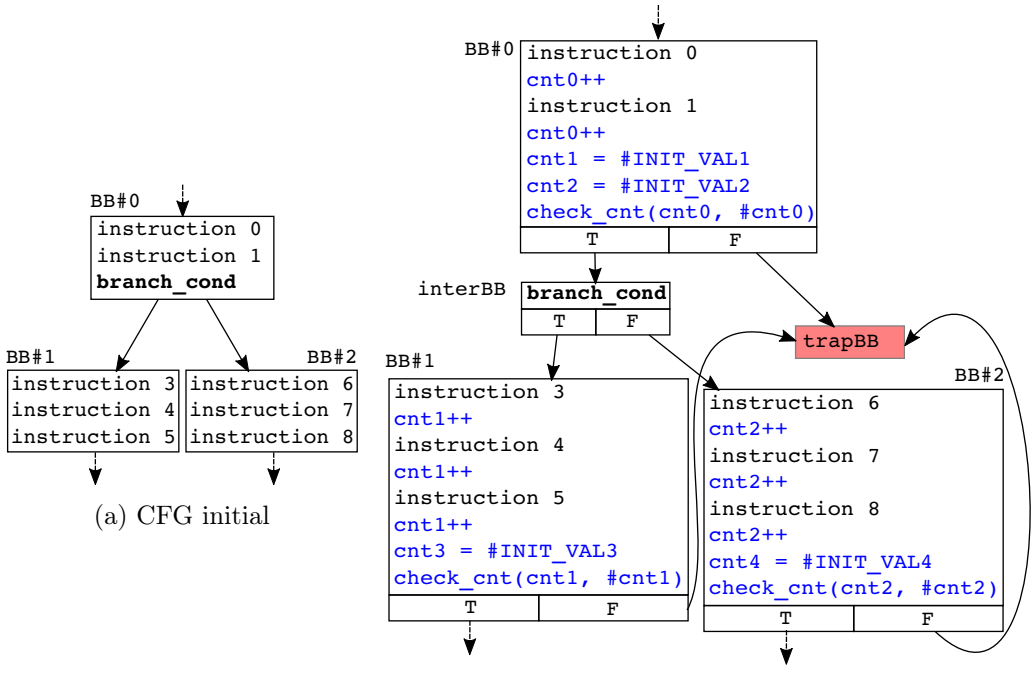
La solution proposée, illustrée par la figure 7.5b, consiste à effectuer la vérification du compteur dans le bloc de base `BB#0` avant l'instruction de branchement (`branch_cond` sur la figure). Mais, pour être en conformité avec la définition d'un bloc de base, `branch_cond` est déplacée dans un nouveau bloc de base `interBB`. Les compteurs des blocs de base `BB#1` et `BB#2` sont tous les deux initialisés dans `BB#0`.



(a) CFG initial

(b) CFG sécurisé

FIGURE 7.4 – Application du schéma CFI sur un bloc de base `BB#2` possédant plusieurs prédécesseurs



(a) CFG initial

(b) CFG sécurisé

FIGURE 7.5 – Application du schéma CFI pour le cas d'un bloc de base `BB#0` possédant plusieurs successeurs

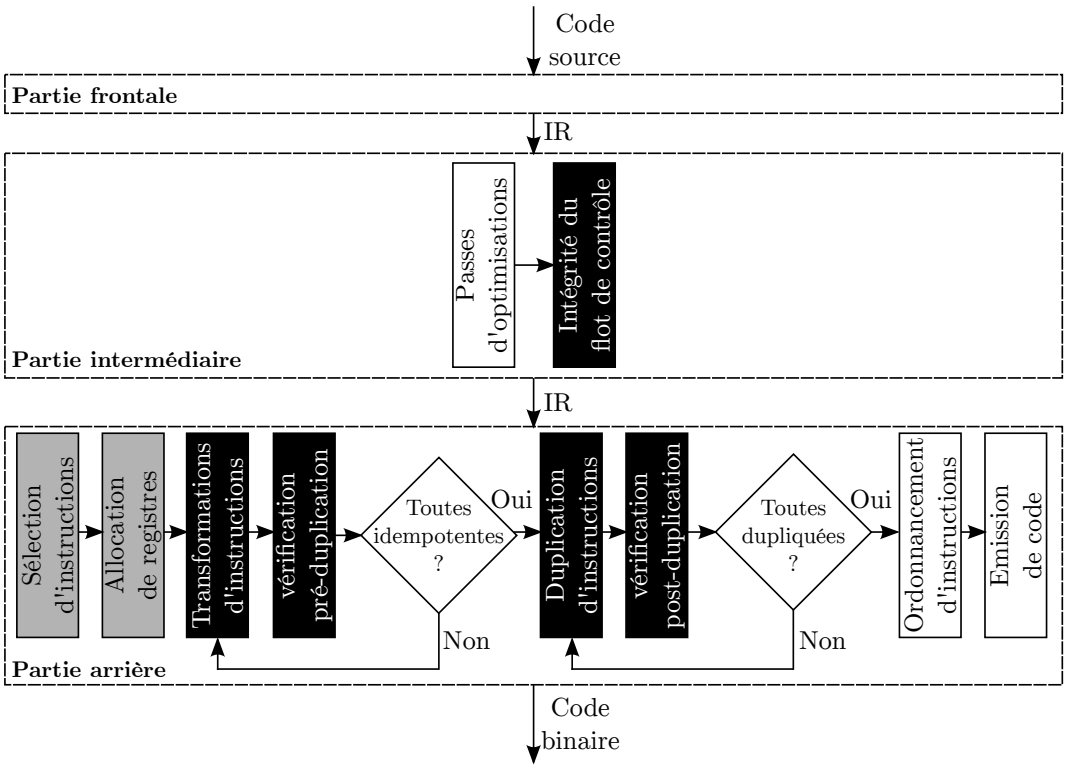


FIGURE 7.6 – Structure simplifiée du compilateur modifié. Les boites grises et les boites noires représentent respectivement les passes modifiées et celles ajoutées.

7.3.4 Combinaison avec le schéma de tolérance

La garantie apportée par un schéma CFI, de façon générale, repose sur le mécanisme de vérification de la signature. Dans le cas de notre schéma, cette tâche est réalisée par la routine `check_cnt`. Cependant, fauter cette partie du code, par exemple en sautant l'une des instructions de la routine `check_cnt`, compromet le mécanisme de vérification. Cette vulnérabilité sera illustrée dans la section 7.5.

Pour renforcer le mécanisme de vérification contre le saut d'instructions, nous proposons de combiner ce schéma CFI avec le modèle de protection `protect_skip{A, B}` (présenté dans le chapitre 6). Pour réduire le coût de la protection, nous appliquons le modèle `protect_skip{A, B}` uniquement sur code de la routine `check_cnt`.

7.4 Mise en œuvre

Ce schéma est mis en œuvre dans la partie intermédiaire du compilateur, dans une passe dédiée appelée *Intégrité du flot de contrôle*. Une illustration de la structure interne du compilateur faisant apparaître cette passe est présentée par la figure 7.6. Les passes du schéma de tolérance (dans la partie arrière du compilateur) ont déjà été présentées dans le chapitre 5. Les raisons du choix de la partie intermédiaire du compilateur pour implémenter ce schéma sont les suivantes :

— pour pouvoir appliquer le schéma CFI avec le schéma de tolérance aux sauts d'instructions (qui commence au moment de la sélection d'instructions), il est nécessaire que le schéma CFI soit appliqué avant, c'est-à-dire dans la partie intermédiaire.

— Mettre en œuvre un schéma dans la partie intermédiaire du compilateur le rend valide pour tous les langages et toutes les architectures supportées par le compilateur.

La passe *Intégrité du flot de contrôle* est une passe de type `LLVM function pass`, car elle traite le programme fonction par fonction. Les sections suivantes présentent les tâches effectuées par cette passe.

7.4.1 Identification des sections à sécuriser

Cette première étape consiste à répertorier toutes les sections délimitées par l'utilisateur. Pour qu'une section soit prise en compte, il faut que les annotations respectent le format suivant :

```
1 @begin_secure("CFI")
2 // code to secure
3 @end_secure("CFI")
```

Chaque annotation `@begin_secure("CFI")` doit être suivie par `@end_secure("CFI")` (et non l'inverse). Plusieurs sections de la même fonction peuvent être annotées séparément.

7.4.2 Initialisation des compteurs

Le processus d'initialisation des compteurs est décrit par l'algorithme 6. L'objectif est d'insérer une instruction d'initialisation du compteur de chaque bloc de base. Pour le bloc `entry`, l'instruction est insérée en début de bloc. Pour tous les autres blocs, elle est insérée à la fin des blocs prédécesseurs et avant l'instruction de branchement (dite instruction de *terminaison*). Car comme son nom l'indique, une instruction de terminaison doit toujours être la dernière instruction d'un bloc base, et le compilateur lève une exception dès que cette règle est dérogée par une passe.

— getRandomInteger( ) : retourne un entier 32 bits tiré aléatoirement.

— createMov32Instruction(`CNT_REG`, *cnt*) : crée une instruction qui déplace le compteur *cnt* dans le registre `CNT_REG`.

— terminator($predBB$) : retourne l'instruction de terminaison (dernière instruction) du bloc $predBB$.

7.4.3 Incrémentation des compteurs

Le processus d'incrémentation des compteurs est décrit par l'algorithme 7. L'objectif est d'insérer une instruction qui incrémente un compteur de 1 après chaque instruction d'un bloc de base, sauf après l'instruction de terminaison.

Algorithm 6 Initialisation des compteurs

```
1: function initCounter(listBasicBlocks)
2:     for all BB in listBasicBlocks do
3:         cnt ← getRandomInteger( )
4:         movInstr ← createMov32Instruction(CNT_REG, cnt)
5:         if isEntryBB(BB) then
6:             insertInstr(BB, movInstr)
7:         else
8:             for all predBB in predecessors(BB) do
9:                 termInstr ← terminator(predBB)
10:                insertInstructionBefore(movInstr, termInstr)
11:            end for
12:        end if
13:    end for
14: end function
```

Algorithm 7 Incrémentation des compteurs

```
1: function incCounters(listBasicBlocks)
2:     for all BB in listBasicBlocks do
3:         for all instr in BB do
4:             if instr ≠ terminator(BB) then
5:                 cntppInstr ← createAddInstr(CNT_REG, CNT_REG, 0x01)
6:                 insertInstrAfter(cntppInstr, instr)
7:             end if
8:         end for
9:     end for
10: end function
```

La fonction createAddInstr(`CNT_REG`, `CNT_REG`, `0x01`) crée une instruction qui réalise l'opération `CNT_REG` ← `CNT_REG` +1.

7.4.4 Vérification des compteurs

Le mécanisme de vérification est décrit par l'algorithme 8. L'objectif est d'insérer des instructions permettant de vérifier que la valeur du compteur à la fin d'un bloc de base correspond à sa valeur attendue lors d'une exécution normale. Le traitement

Algorithm 8 Vérification des compteurs

```
1:  function checkCounters(listBasicBlocks)
2:      trapBB ← createBasicBlock()
3:      for all BB in listBasicBlocks do
4:          if nbSuccessors(BB) = 1 then
5:              succBB ← Successor(BB)
6:              insertCheckRoutine(BB, succBB, trapBB)
7:          else                                  ▷ 0 ou plusieurs successeurs
8:              interBB ← createBasicBlock()
9:              termInstr ← terminator(BB)
10:             insertInstr(termInstr, interBB)
11:             deleteInstr(termInstr, BB)
12:             transferSuccessors(BB, interBB)
13:             addPredecessor(interBB, BB)
14:             insertCheckRoutine(BB, interBB, trapBB)
15:         end if
16:     end for
17: end function
```

réalisé pour chaque bloc de base dépend de sa configuration, selon les cas présentés dans la section 7.3. La fonction insertCheckRoutine( ), appelée à la ligne 14 de l'algorithme 8, est présentée en détail par l'algorithme 9.

Cette fonction est chargée d'insérer la routine de vérification d'un compteur. Elle prend les trois paramètres suivants :

— $fromBB$: le bloc de base à la fin duquel la routine de vérification sera insérée.

— $ifBB$: le bloc de base vers lequel le flot d'exécution sera redirigé si la vérification du compteur réussit.

— $elseBB$: le bloc de base vers lequel le flot d'exécution sera redirigé si la vérification du compteur échoue.

La ligne 3 récupère la valeur d'initialisation du compteur assigné au bloc de base $fromBB$. La ligne 4 détermine le nombre d'instructions d'incrémentation de compteur insérées entre l'instruction qui initialise le compteur et l'instruction de terminaison de $fromBB$. Ce qui permet de calculer, à la ligne 5, la valeur attendue du compteur. À la ligne 6, nous créons une instruction `cmp CNT_REG,` $expectedCntVal$,

Algorithm 9 Insertion de la routine `check_cnt`

```
1:  function insertCheckRoutine(fromBB, ifBB, elseBB)
2:      termInstr ← terminator(formBB)
3:      initCnt ← getInitCounter(fromBB)
4:      nbInc ← getInbIncrementation(fromBB)
5:      expectedCntVal ← initCnt + nbInc
6:      cmpInstr ← createCmpInstr(CNT_REG, expectedCntVal)
7:      condBranchInstr ← createCondBranchInstr(EQ, ifBB, elseBB)
8:      pushBackAnnotation(@begin_secure("skip"))
9:      pushBackInstr(cmpInstr, fromBB)
10:     pushBackInstr(condBranchInstr, fromBB)
11:     pushBackAnnotation(@end_secure("skip"))
12: end function
```

qui compare la valeur attendue du compteur à sa valeur effective, c'est-à-dire contenue dans le registre `CNT_REG`. Nous créons ensuite, à la ligne 7, une instruction de branchement conditionnel, avec le prédicat `EQ` (*Equal*), qui réalise un saut vers le bloc de base $ifBB$ si les deux valeurs comparées sont égales et vers $elseBB$ dans le cas contraire. Les lignes 9 et 10 insèrent les instructions précédemment crées à la fin du bloc $fromBB$. Les lignes 8 et 11 annotent les instructions qui viennent d'être insérées pour que celles-ci soient prises en compte lors de l'application du schéma de tolérance au saut d'instructions.

7.4.5 Réaction

La réaction correspond à l'exécution des instructions contenues dans le bloc de base `trapBB`. Le contenu de ce bloc de base peut être fourni par l'utilisateur à l'intérieur du code source, en utilisant l'annotation `@CFI_handler`. Par exemple :

```
1 @CFI_handler
2 void reaction(){
3     // corps de la fonction
4 }
```

Si aucune fonction n'est fournie par l'utilisateur, une fonction `default_reaction()` sera automatiquement proposée par le compilateur :

```
1 void default_reaction(){
2     raise_interrupt();
3 }
```

`default_reaction()` déclenche une interruption matérielle.

7.5 Illustrations

Dans cette section, nous illustrons, sur un extrait du programme verifyPIN [94] présenté par le listing 7.1, l'application du schéma CFI combiné au schéma de to-

Listing 7.1 – Extrait du programme VerifyPIN [94] écrit en langage C

```
1  BOOL byteArrayCompare(unsigned char * a1, unsigned char* a2)
2  void verifyPIN(){
3      g_authenticated = BOOL_FALSE;
4      if(g_ptc > 0) {
5          if(byteArrayCompare(g_userPin, g_cardPin) == BOOL_TRUE) {
6              g_ptc = 3;
7              g_authenticated = BOOL_TRUE; // Authentication();
8          }
9          else {
10             g_ptc--;
11         }
12     }
13 }
```

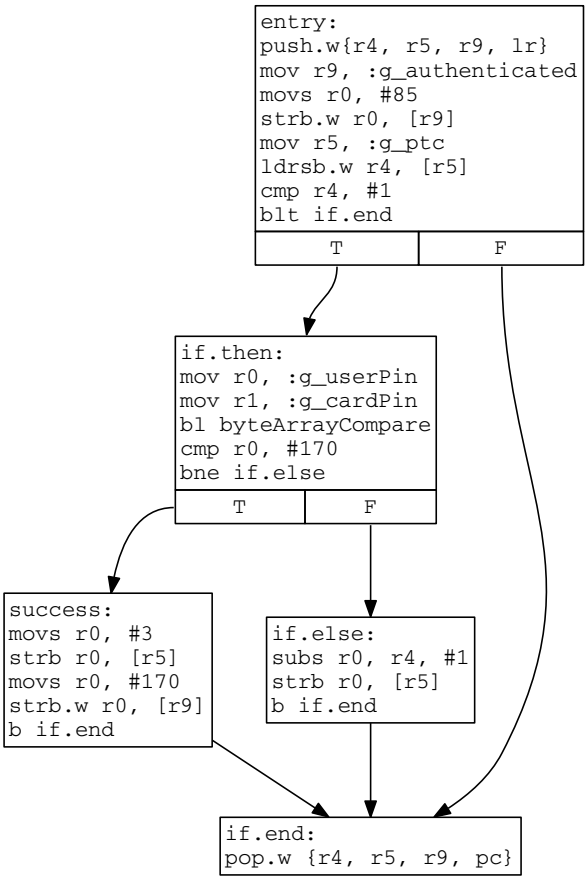


FIGURE 7.7 – CFG non sécurisé de la fonction `VerifyPIN` (listing 7.1)

lérance au saut d'instructions. Le programme est écrit en langage C. La variable `g_authenticated` est initialisée à `BOOL_FALSE`, et mise à `BOOL_TRUE` si l'authentification est réussie. La variable `g_ptc` contient le nombre de tentatives restantes avant le blocage de la carte. La fonction `byteArrayCompare`, dont l'implémentation n'est pas présentée sur cet exemple, renvoie `BOOL_TRUE` si le PIN fourni par l'utilisateur (`g_userPin`) est identique au PIN de la carte (`g_cardPin`). La figure 7.7 présente le CFG non sécurisé de la fonction `verifyPIN`. Le CFG a été extrait du code assembleur ARM généré par notre compilateur.

La figure 7.8 illustre l'application du schéma CFI sans combinaison avec le schéma de tolérance au saut d'instructions. On peut remarquer les nouveaux blocs de base intermédiaires (`interBB0` et `interBB1`) et le bloc `trapBB` ajoutés conformément au schéma. Les nouvelles instructions insérées sont colorées en bleu. Le registre `r6` contient le compteur de chaque bloc de base. Il est initialisé avec l'instruction `mov r6, #val` et incrémenté avec `add r6, #1`. La validité du compteur est vérifiée avec l'instruction `cmp r6, #cnt`, qui compare le registre `r6` contenant la valeur calculée du compteur et la constante `#cnt` qui est la valeur attendue du compteur. Cette instruction met à jour les bits de statut (*flag*) du registre APSR, ce qui permet à l'instruction *Branch if Not Equal* (`bne trapBB`) de réaliser un branchement dans le bloc `trapBB` si les deux valeurs comparées précédemment ne sont pas égales.

À partir de l'illustration de la figure 7.8, on peut réaliser que sauter l'instruction de comparaison ou l'instruction de branchement conditionnel pourrait empêcher de brancher dans le bloc `trapBB` même si les deux valeurs comparées ne sont pas égales. Autrement dit, sauter une instruction dans le code de la routine `check_cnt` peut empêcher de garantir la validité du chemin d'exécution.

Pour se prémunir de cette attaque, nous avons appliqué le schéma de tolérance mais, pour des raisons de coûts, seulement aux instructions de la routine `check_cnt`. La figure 7.9 illustre la combinaison du schéma CFI au schéma de tolérance au saut d'instructions. Pour simplifier l'illustration, le schéma de tolérance est configuré pour protéger contre le modèle de fautes *saut d'une seule instruction machine* (`protect_skip{1, 1}`). Cependant, il est possible d'appliquer le modèle `protect_skip{A, B}` quelle que soient les valeurs de `A` et de `B`. On peut remarquer sur la figure 7.9 la duplication des instructions `cmp` et `bne`.

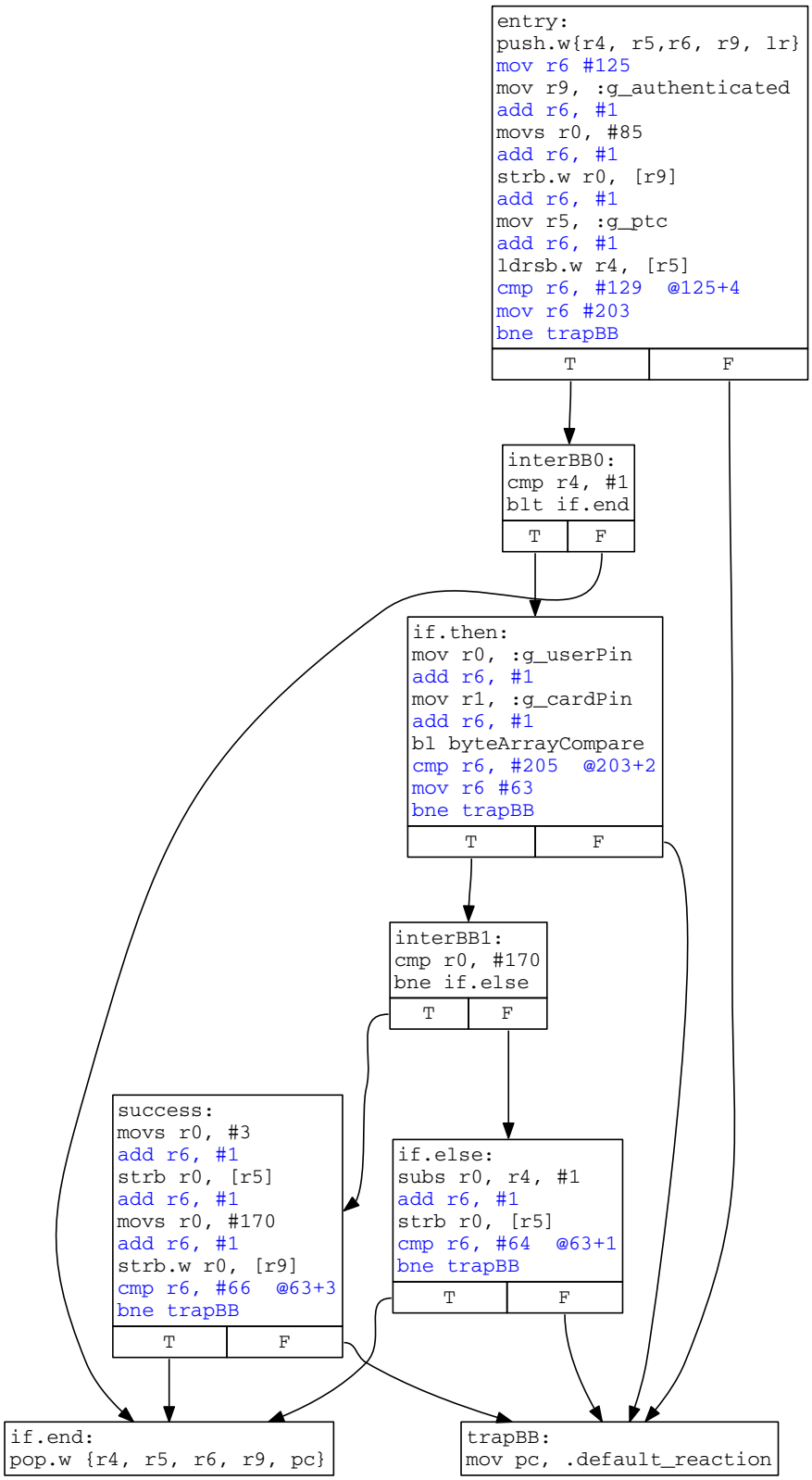


FIGURE 7.8 – CFG de la fonction `VerifyPIN`, sécurisé avec le schéma CFI sans combinaison avec le schéma de tolérance au saut d'instructions

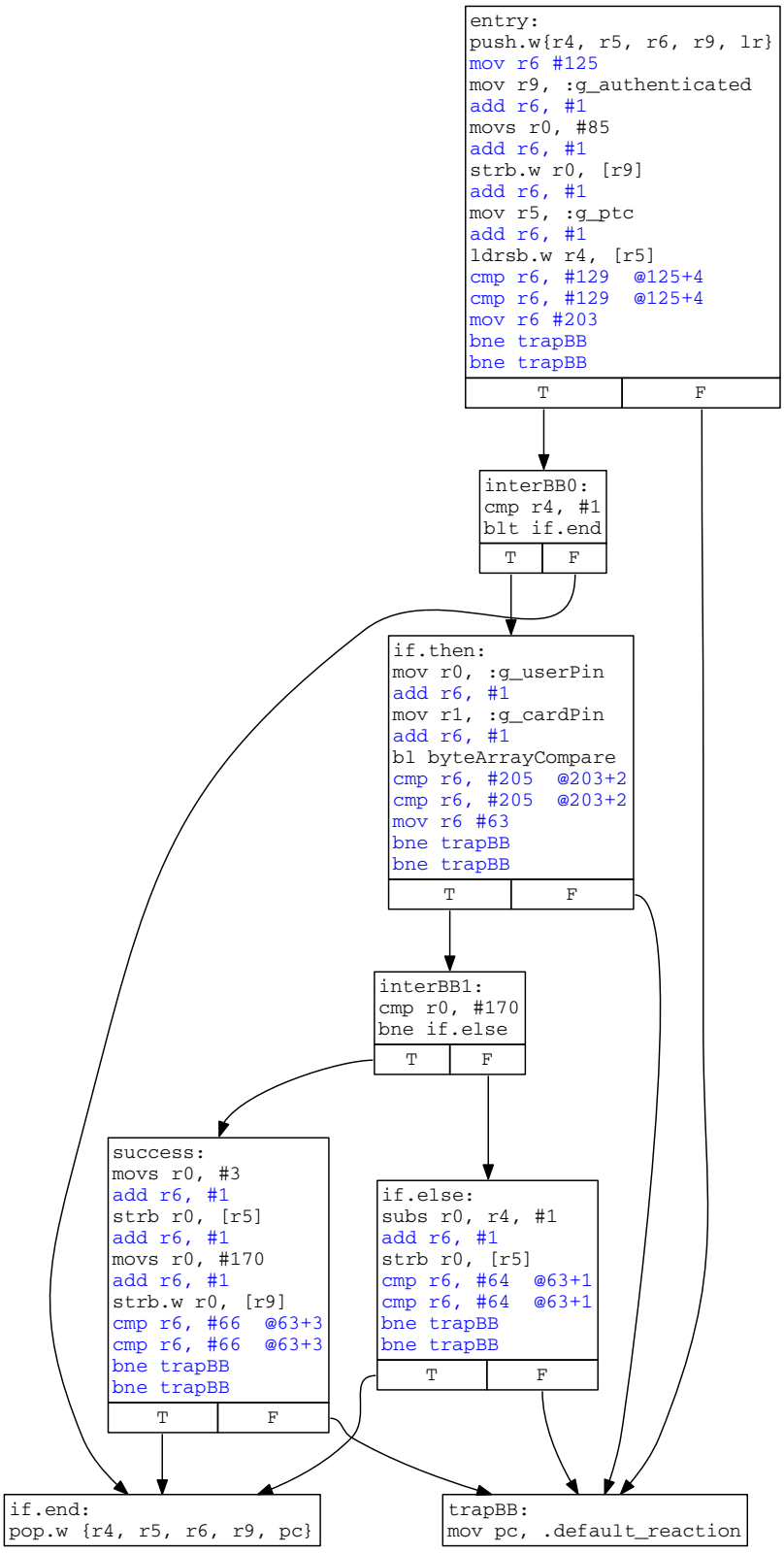


FIGURE 7.9 – CFG de la fonction `VerifyPIN`, sécurisé avec le schéma CFI combiné avec le modèle de protection `protect_skip{1, 1}`.

TABLE 7.1 – Résultat de l'évaluation de performances du schéma CFI combiné au schéma de tolérance au saut d'instructions (CFI + `protect_skip{A, B}`)

	Modèles de protections	**Surcoûts**	
		Temps d'exéc.	**Taille de code**
AES 8 bits	CFI	$\times 2,31$	$\times 1,81$
	CFI + `protect_skip{1, 1}`	$\times 2,70$	$\times 1,99$
	CFI + `protect_skip{2, 1}`	$\times 3,11$	$\times 2,17$
AES 32 bits	CFI	$\times 2,17$	$\times 1,65$
	CFI + `protect_skip{1, 1}`	$\times 2,54$	$\times 1,82$
	CFI + `protect_skip{2, 1}`	$\times 2,92$	$\times 1,98$
Verify-PIN	CFI	$\times 1,23$	$\times 1,54$
	CFI + `protect_skip{1, 1}`	$\times 1,44$	$\times 1,69$
	CFI + `protect_skip{2, 1}`	$\times 1,65$	$\times 1,85$

7.6 Expérimentations

Pour valider ce schéma, nous avons réalisé des expérimentations sur les programmes utilisés dans le chapitre 5. Nous avons effectué une évaluation de sécurité avec le simulateur, présenté dans le chapitre 4, et une évaluation de performances sur le microcontrôleur STM32 [69].

7.6.1 Évaluation de performances

Le tableau 7.1 présente les surcoûts en taille de code et en temps d'exécution de l'application du schéma CFI combiné au schéma de tolérance au saut d'instructions (CFI + `protect_skip{A, B}`). Les deux dernières colonnes reportent pour chaque modèle de protection les surcoûts par rapport au code de référence (non protégé).

Nous pouvons constater que la combinaison avec le schéma de tolérance augmente très peu le surcoût par rapport au schéma CFI seul. Cela s'explique par le fait que le schéma de tolérance n'est appliqué que sur une petite portion du schéma CFI (sur les routines de vérification de compteur). Mais aussi par le fait que la duplication des instructions de branchement, présentes dans les routines de vérification, n'augmente pas le temps d'exécution global, car seulement l'instruction originale (la première rencontrée) sera exécutée.

7.6.2 Évaluation de sécurité

Le tableau 7.2 présente les résultats de la simulation des modèles de fautes `REPL_BR` et `REPL_BR_SKIP{N, M}` sur le programme verify-PIN protégé par les modèles de protections CFI et CFI + `protect_skip{A, B}`. Les modèles de fautes simulés sont détaillés dans la section 4.2.

Ces résultats montrent, d'une part, que le schéma CFI seul protège bien contre le modèle de fautes `REPL_BR`, mais pas contre le modèle `REPL_BR_SKIP{N, M}`. D'autre

TABLE 7.2 – Résultat de la simulation des modèles de fautes `REPL_BR` et `REPL_BR_SKIP{N, M}` sur le programme verify-PIN protégé par les modèles de protections CFI et CFI + `protect_skip{A, B}`

Modèles de protections	**Modèles de faute**		
	`REPL_BR`	`REPL_BR_SKIP{1, 1}`	`REPL_BR_SKIP{2, 1}`
CFI	✓	✗	✗
CFI + `protect_skip{1, 1}`	✓	✓	✗
CFI + `protect_skip{2, 1}`	✓	✓	✓

part, le modèle de protection CFI + `protect_skip{A, B}` protège bien contre les modèles de fautes `REPL_BR_SKIP{N, M}`.

7.7 Conclusion

Dans ce chapitre, nous avons décrit et mis en œuvre un schéma CFI permettant de vérifier la validité du chemin d'exécution d'un programme. Nous avons proposé une combinaison du schéma CFI avec le schéma de tolérance au saut d'instructions présenté dans le chapitre précédent pour augmenter le niveau de sécurité apporté.

Les résultats de la simulation ont montré que le schéma combiné protège bien contre les modèles de fautes considérés. L'évaluation de performances a montré que le coût introduit par la combinaison est relativement faible par rapport au coût du CFI seul.

Dans le chapitre suivant, nous proposons un nouveau schéma basé sur le principe du CFI et qui garantit, entre autres, non seulement que le chemin d'exécution suivi est prévu dans le CFG du programme, mais aussi que ce chemin est valide d'un point de vue fonctionnel.

Chapitre 8

Conception d'un schéma d'intégrité de code et de flot de contrôle (CCFI)

Sommaire

8.1 Modèles de fautes 104
8.2 Principe de fonctionnement du schéma CCFI 105
8.3 Mise en œuvre 105
8.3.1 Un seul prédécesseur et un seul successeur 107
8.3.2 Plusieurs prédécesseurs 107
8.3.3 Prédécesseur commun 108
8.3.4 Branchements conditionnels 111
8.3.5 Alignement des instructions 113
8.3.6 Réservation de registres 114
8.4 Expérimentations 115
8.4.1 Évaluation de performances 115
8.4.2 Évaluation sécuritaire 117
8.5 Conclusion 117

Dans ce chapitre, nous présentons la description et la mise en œuvre d'un schéma d'intégrité de code et de flot de contrôle appelé CCFI pour *Code and Control-Flow Integrity*. Le schéma reprend les propriétés de sécurité apportées par le schéma du chapitre précédent et les étend, en garantissant l'intégrité contre le saut de n'importe quel bloc de code de 32 bits ou plus, et en assurant l'intégrité des conditions de branchement. Cette dernière propriété est importante. En effet, le schéma de protection du chapitre précédent permet de vérifier que les transitions prises par le programme sont conformes au graphe de flot de contrôle théorique mais il est cependant possible de réaliser une faute non détectée, par exemple en altérant un branchement conditionnel de façon à modifier le nombre d'itérations d'une boucle. Une telle faute permet de corrompre l'exécution du programme tout en respectant son CFG original. La faute ne serait donc pas détectée par le schéma du chapitre précédent ; elle sera par contre détectée par le schéma de protection présenté dans ce chapitre.

L'originalité de ce schéma de protection réside dans l'exploitation d'un mécanisme de calcul de signature typique du CFI, que nous utilisons à la fois pour vérifier l'intégrité du flot de contrôle du programme et pour vérifier l'intégrité d'exécution des instructions d'un bloc de base.

Dans la section 8.1, nous précisons les modèles de fautes considérés. Dans la section 8.2, nous présentons le principe de fonctionnement du schéma CCFI. Dans la section 8.3, nous présentons la mise en œuvre de ce schéma dans le compilateur. Nous présentons les résultats des expérimentations dans la section 8.4. Une présentation plus détaillée de la mise en œuvre du schéma, avec des illustrations sur un programme exemple, est présentée dans l'annexe A.

8.1 Modèles de fautes

Ce schéma de protection vise à apporter indépendamment (1) de l'intégrité du flot de contrôle en tenant compte des conditions de branchement et (2) de l'intégrité d'exécution d'une séquence d'instructions sans branchement.

1. Intégrité du flot de contrôle : les altérations du flot de contrôle sont modélisées par le modèle `repl_br`, de la même manière que dans le chapitre précédent. En complément, on veut se prémunir de tous les sauts d'instructions de branchement conditionnel ; ce comportement est modélisé par le modèle `skip_br`.

2. Intégrité d'une séquence d'exécution d'instructions : Nous nous appuyons sur l'observation de Moro et al. [75] qui montrent qu'il est possible de sauter une séquence de code de 32 bits en attaquant le chargement des instructions du processeur. Nous généralisons ce modèle en considérant qu'il est possible de remplacer un bloc aligné de la mémoire programme de 32 bits ou plus. Ce comportement est modélisé par le modèle `skip_block{size, 1}`.

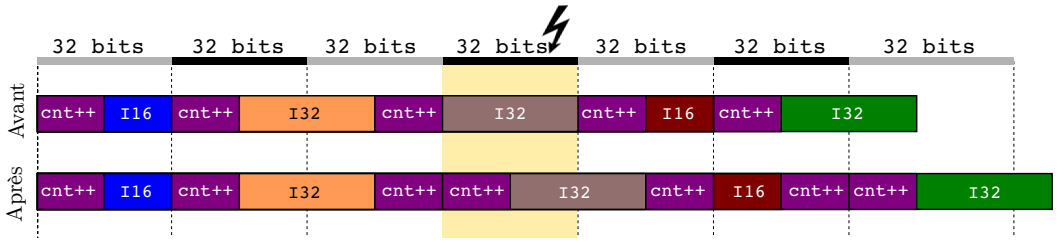


FIGURE 8.1 – Alignement des instructions dans des blocs de taille 32 bits. `I16` et `I32` représentent respectivement des instructions de taille 16 bits et 32 bits

8.2 Principe de fonctionnement du schéma CCFI

Dans ce chapitre, nous réutilisons les notations définies dans la section 7.2.

Le schéma CCFI s'inscrit dans la continuité du schéma CFI présenté dans le chapitre 7. Il repose sur un calcul de signature. La signature est portée par un compteur `cnt`, et la mise à jour de la signature se fait par une incrémentation du compteur (`cnt++`). Les instructions `cnt++` sont insérées entre chaque instruction. En outre, afin d'obtenir l'intégrité d'exécution d'une séquence de code par rapport à l'injection d'une faute de type `skip_block{size, 1}`, nous faisons en sorte que chaque instruction utile du programme soit associée à une incrémentation de compteur `cnt++` dans chaque bloc de taille `size`. Ainsi, le saut du bloc ne peut se faire sans sauter en même temps une instruction `cnt++`, de sorte que cette faute sera détectée au moment de la vérification de la signature. Cette association entre les instructions utiles du programme et les instructions `cnt++` est calculée à l'aide d'une analyse de l'alignement des instructions. La figure 8.1 illustre le cas où on ajuste l'alignement des instructions pour qu'une instruction `cnt++` soit présente dans chaque bloc de taille 32 bits.

Le schéma CCFI permet de vérifier que le chemin d'exécution suivi est non seulement prévu mais qu'il est aussi valide d'un point de vue fonctionnel, en prenant en compte dynamiquement la validité des conditions de branchements. Cela permet de détecter, par exemple, l'exécution du bloc *else* à la place du bloc *if* ou la modification du nombre d'itérations d'une boucle. Dans ce dernier cas, cependant, CCFI permet de détecter toutes les altérations sur la séquence de code exécutée, mais non les fautes qui toucheraient au chemin de données du processeur : par exemple, l'altération de la valeur d'un indice de boucle ne sera pas détectée.

8.3 Mise en œuvre

Nous avons mis en œuvre le schéma CCFI dans la partie arrière du compilateur. Les critères qui ont conduit à ce choix sont :

— le besoin de modifier l'alignement mémoire des instructions machine. Pour que cette modification soit garantie dans le code binaire, il faut qu'elle soit

effectuée le plus tard possible du processus de compilation (juste avant la phase d'émission de code).

— le besoin de manipuler les conditions des branchements et les *flags* du processeur, nécessite de cibler une architecture bien définie.

La mise en œuvre a été longue et complexe. Elle a nécessité environ 2000 lignes de code reparties entre plusieurs passes de compilation. Certains choix d'implémentation qui tiennent compte de l'écosystème LLVM seront difficiles à décrire sans rentrer en détails dans la structure interne de LLVM, ce qui pourrait s'avérer difficile à suivre pour le lecteur.

Dans ce chapitre, nous nous limitons aux seuls détails permettant de comprendre la transcription du schéma théorique décrit précédemment vers une mise en œuvre dans LLVM. Dans l'annexe A, nous présentons plus en détail les passes de compilation qui entrent en jeu dans la mise en œuvre de ce schéma. Nous discutons des choix d'ordonnancement pour l'exécution de ces passes, et nous illustrons sur un exemple le rôle de chaque passe dans l'application du schéma.

Les points suivants résument les principales étapes de la mise en œuvre du schéma :

— un compteur unique est initialisé en début de chaque bloc de base (`cnt`)

— ce compteur est incrémenté avant chaque instruction (`cnt++`)

— la validité d'un compteur est vérifiée en début du bloc de base suivant, avec la routine `check_cnt`

— le compteur d'un bloc de base est initialisé avant la vérification de la validité du compteur du bloc précédent

— le processus de vérification inclut une vérification de la validité de la condition qui a permis la transition qui précède.

— l'alignement des instructions est effectué de sorte que chaque instruction soit immédiatement suivie d'une instruction d'incrémentation de compteur.

Nous réservons un seul registre physique pour stocker le compteur `cnt` dans chaque bloc de base, bien que dans les illustrations suivantes on utilisera la notation `cnt`i pour désigner le compteur du bloc `BB#`i, où i est le numéro d'ordre du bloc. Le processus de réservation de registres sera présenté dans la section 8.3.6

Puisque l'objectif est d'initialiser le compteur du bloc courant avant la vérification du compteur du bloc précédent, il est nécessaire de réserver un deuxième registre dans lequel le compteur du bloc précédent sera copié avant l'initialisation du compteur du bloc courant. Dans la suite, on appellera *compteur précédent* d'un bloc le compteur de son bloc prédécesseur.

Notations

`cnt_prec` : le compteur précédent d'un bloc

`#cnt_prec` : la valeur attendue du compteur précédent

$\oplus$: l'opérateur ou exclusif

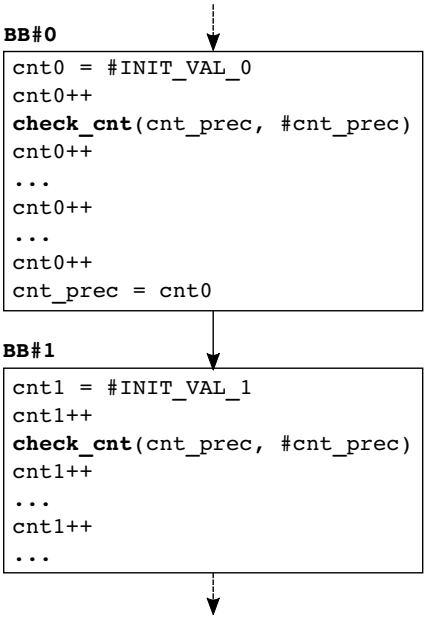


FIGURE 8.2 – Cas d'un seul prédécesseur et un seul successeur

Comme dans le chapitre 7, le schéma s'applique sur CFG du programme, ce qui nécessite de prendre en compte les différentes configurations des blocs de base : un bloc de base peut avoir zéro à plusieurs prédécesseur/successeurs. Pour simplifier la description, nous allons décrire séparément l'application du schéma pour chacune de ces configurations.

8.3.1 Un seul prédécesseur et un seul successeur

Cette configuration concerne le cas d'un bloc de base ne possédant qu'un seul prédécesseur et un seul successeur. La figure 8.2 montre l'initialisation d'un compteur `cnt0` par une valeur aléatoire `#INIT_VAL_0`, tirée statiquement lors de la compilation.

À la fin de `BB#0`, `cnt0` est copié dans `cnt_prec` pour devenir le compteur précédent de `BB#1`. La validité de `cnt_prec` est vérifiée dans `BB#1` avec la routine `check_cnt`, et cela, après que le compteur du bloc `BB#1` ait été initialisé.

8.3.2 Plusieurs prédécesseurs

Dans le cas d'un bloc de base `BB` possédant au moins deux prédécesseurs, un problème se pose lors de la vérification de la validité du compteur précédent. D'une part, vu que chacun des prédécesseurs de `BB` possède son propre compteur initialisé à une valeur unique, il y aurait autant de `cnt_prec` valides que de prédécesseurs. D'autre part, il est impossible de connaître, au moment de la compilation, par lequel des prédécesseurs `BB` sera atteint lors de l'exécution. Par conséquent, il n'est pas possible de connaître statiquement la valeur de `#cnt_prec` nécessaire à la vérification.

La solution proposée pour résoudre ce problème est illustrée par la figure 8.3.

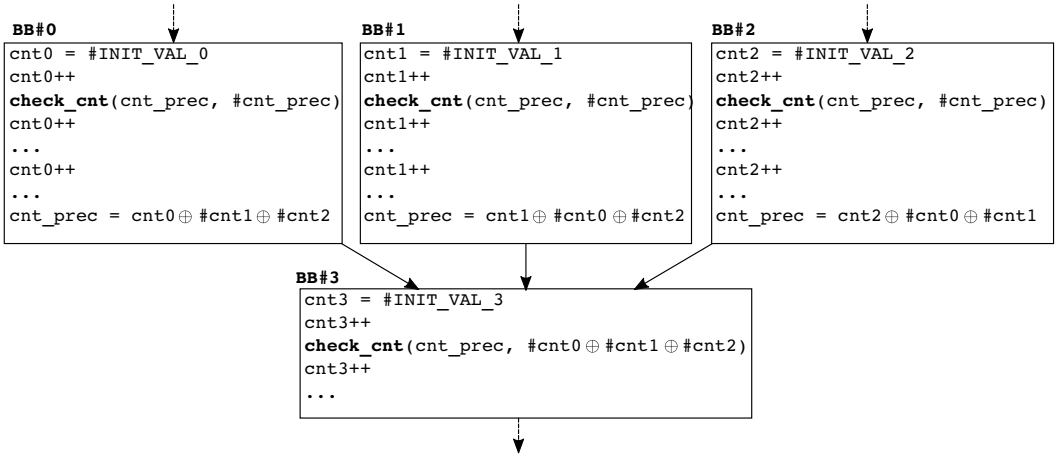


FIGURE 8.3 – Cas de plusieurs prédécesseurs

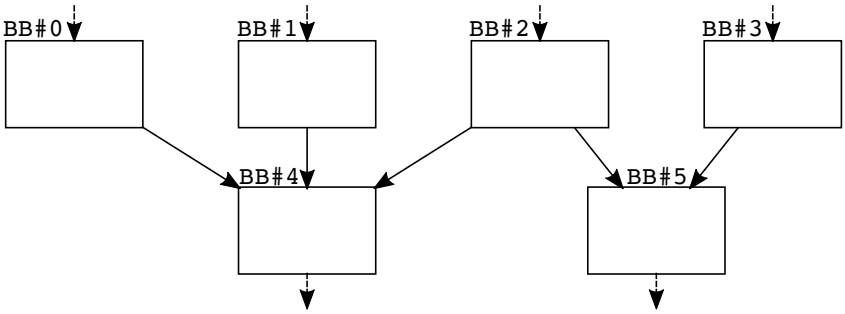


FIGURE 8.4 – Cas d'un prédécesseur commun

Elle consiste à appliquer l'opérateur $\oplus$ entre les compteurs en sorties de tous les prédécesseurs. Cette opération permet au bloc de base `BB#3` d'avoir une seule valeur attendue du compteur précédent `#cnt_prec`.

8.3.3 Prédécesseur commun

Le cas d'un bloc de base possédant plusieurs prédécesseurs est traité dans la section précédente, et celui d'un bloc de base possédant plusieurs successeurs est identique au cas d'*un seul prédécesseur et un seul successeur* (traité dans la section 8.3.1). Dans cette nouvelle configuration, on s'intéresse au cas d'un bloc de base possédant plusieurs prédécesseurs, sachant qu'au moins un de ces prédécesseurs possède, à son tour, plusieurs successeurs. Sur la figure 8.4, qui illustre cette configuration, les blocs de base `BB#0`, `BB#1` et `BB#2` sont les prédécesseurs de `BB#4`. Parmi lesquels, le bloc de base `BB#2` possède `BB#4` et `BB#5` comme successeurs.

Le problème que pose cette configuration se situe au niveau du bloc de base `BB#2`, qui est à la fois prédécesseur de `BB#4` et de `BB#5`. D'après la solution proposée pour la configuration précédente (*plusieurs prédécesseurs*), `BB#4` attend un compteur en sortie de `BB#2` égal à `#cnt0` $\oplus$ `#cnt1` $\oplus$ `#cnt2`, alors que `BB#5` attend `#cnt2` $\oplus$ `#cnt3`.

La solution consiste à faire en sorte que le même compteur en sortie de `BB#2` vaut `#cnt0` $\oplus$ `#cnt1` $\oplus$ `#cnt2` pour la transition `BB#2` $\rightarrow$ `BB#4`, et `#cnt2` $\oplus$ `#cnt3` pour `BB#2` $\rightarrow$ `BB#5`. Pour cela, nous insérons un bloc de base intermédiaire sur chacune des transitions partant de `BB#2`. Le rôle de ces blocs de base intermédiaires est d'affecter la bonne valeur au compteur en fonction de la destination de la transition. Cette solution est illustrée par la figure 8.5.

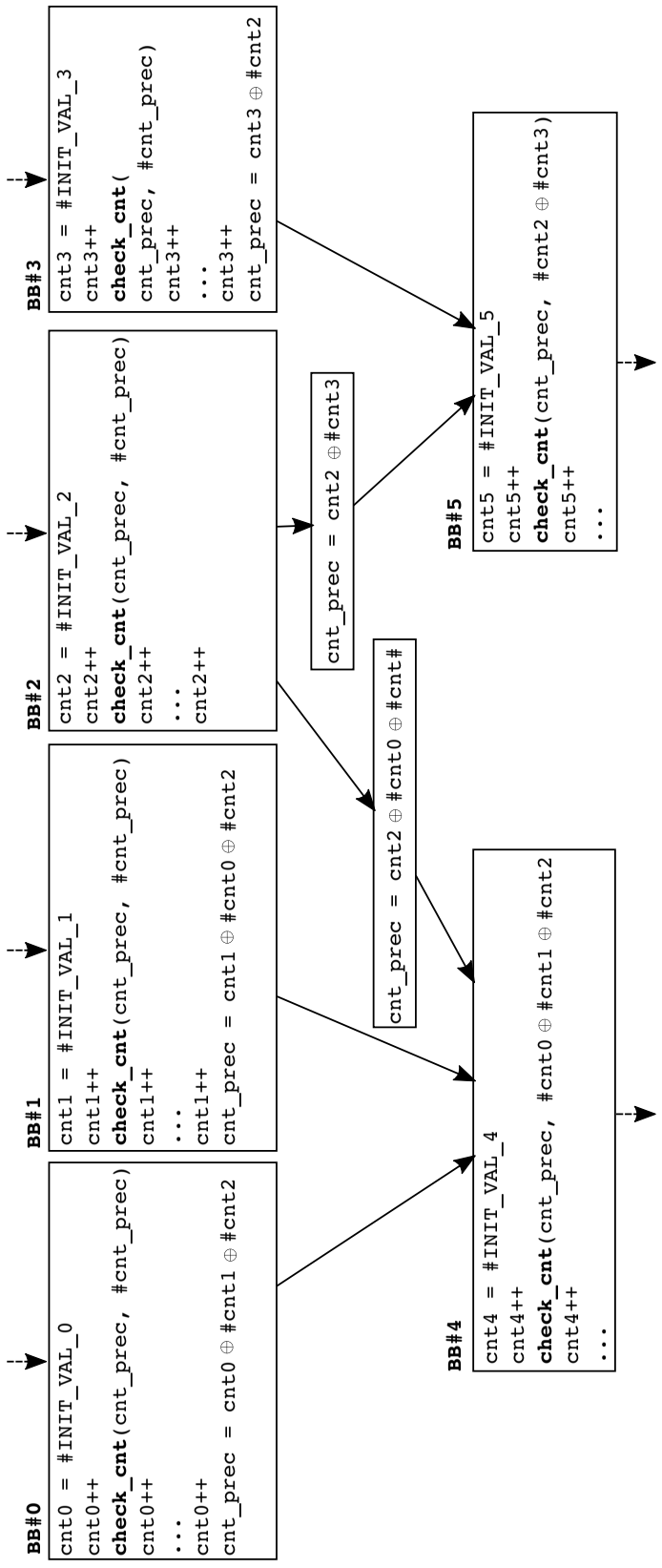


FIGURE 8.5 – Solution pour le cas d'un prédécesseur commun

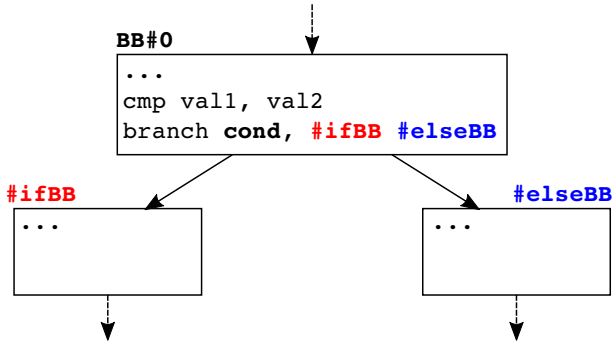


FIGURE 8.6 – Cas d'un branchement conditionnel



FIGURE 8.7 – Découpage du compteur 32 bits

8.3.4 Branchements conditionnels

L'objectif est de prendre en compte, lors de la vérification, la validité du chemin emprunté pour chaque contexte d'exécution. Pour cela, l'idée est d'inspecter le contenu du registre de statut pour connaître le résultat de la dernière condition évaluée. Sur l'architecture ARM, ce registre est le APSR (*Application Program Status Register*), qui est un registre spécial, découpé en plusieurs champs de bits. Le tableau 8.1 présente la fonction de chacun de ces bits. Les 4 bits qui nous intéressent dans ce cas-ci sont les bits NZCV, qui encodent l'état de la dernière opération arithmétique.

Cette configuration est illustrée par la figure 8.6. L'instruction `cmp` du bloc `BB#0` effectue une comparaison de `val1` et `val2`, cette opération met à jour les bits NZCV en fonction du résultat de la comparaison. L'instruction `branch` réalise un saut vers le bloc de base `#ifBB` ou `#elseBB` en fonction de la condition `cond`, qui peut être par exemple :

— EQ (*Equal*) : vraie si le bit Z est égal à 1, qui signifie que `val1` est égal à `val2`.

— NE (*Not equal*) : vrai si le bit Z est égal à 0, qui signifie `val1` différent de `val2`

La liste complète des conditions est disponible dans le manuel de référence ARM pour le microcontrôleur Cortex-M3 [105].

La connaissance de la nature de la condition évaluée dans `BB#0` permet de déduire la configuration des bits NZCV attendue pour chacune des transitions `BB#0` → `#ifBB` et `BB#0` → `#elseBB`. De ce fait, pour vérifier la validité d'une transition, il suffit dans un premier temps, de sauvegarder l'état des bits NZCV juste après l'instruction qui les modifie (l'instruction `cmp` sur la figure 8.6). Dans un deuxième temps, avant de rentrer dans chacun des blocs de bases destination `#ifBB` et `#elseBB`, de vérifier que la configuration des bits NZCV sauvegardée correspond à celle attendue pour la transition correspondante.

TABLE 8.1 – Découpage du registre APSR

Bits	**Nom du flag**	**Fonction du flag**
[31]	N	Négatif
[30]	Z	Zéro
[29]	C	Retenue (*Carry*)
[28]	V	Débordement (*Overflow*)
[27]	Q	Saturation
[26 :0]	-	Réservé

TABLE 8.2 – Résultats des routines `get_flags` et `get_mask` en fonction de `cond` et du bloc de base destination

	`#ifBB`		`#elseBB`	
`cond` (NZCV)	`get_flags`	`get_mask`	`get_flags`	`get_mask`
EQ (Z=1)	0100	0100	0000	0100
NE(Z=0)	0000	0100	0100	0100

Pour éviter l'utilisation d'un registre supplémentaire ou de la mémoire pour sauvegarder la configuration attendue des bits NZCV pour chaque bloc de base destination, nous proposons d'encoder la configuration attendue dans le compteur. Nous découpons le compteur en trois champs de bits. Ce découpage, illustré par la figure 8.7, est constitué des champs suivants :

1. `flags` [31:28] (4 bits) : la configuration attendue des bits NZCV.
2. `mask` [27:24] (4 bits) : un masque pour indiquer lesquels des bits NZCV sont concernés par la condition.
3. `counter` [23:0] (24 bit) : le compteur proprement dit. Pour éviter un débordement dû aux incrémentations des compteurs les bits [23:22] sont mis à 0 lors de l'initialisation du compteur.

Dans la suite de cette description, nous utiliserons les notations suivant : `cnt[flags]`, `cnt[mask]` et `cnt[counter]` pour désigner respectivement à la partie `flags`, `mask` et `counter` du compteur `cnt`.

La figure 8.8 montre l'application de cette solution sur la figure 8.6. La routine `get_flags` renseigne la configuration des bits NZCV attendue dans `cnt[flags]`. Cette configuration dépend de la nature de la condition évaluée dans `BB#0` et du bloc de base destination. La routine `get_mask` renseigne la position des bits concernés. À titre d'exemple, le tableau 8.2 liste pour les conditions EQ et NE les résultats produits par les routines `get_flags` et `get_mask` selon le bloc de base destination. L'instruction `mrs` copie dans la variable `nzcv` le contenu du registre APSR. La routine `check_cond`, décrite par l'algorithme 10, vérifie si `flags` est égal à `nzcv` sur les bits pointés par `mask`.

L'objectif de l'algorithme 10 est de vérifier que le résultat de la condition attendue (*flags*) correspond au résultat constaté de la condition évaluée (*nzcv*). La

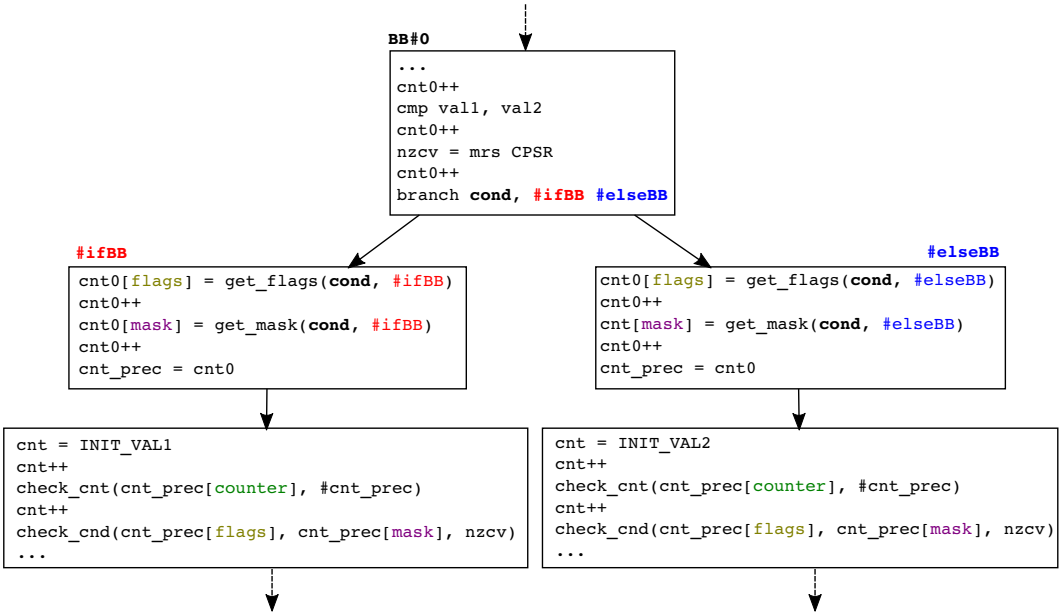


FIGURE 8.8 – Solution pour le cas d'un branchement conditionnel (version protégée de la figure 8.6)

première tâche consiste à identifier, dans *nzcv*, les bits concernés par la condition évaluée. Pour cela, on réalise un ET booléen entre *mask* et *nzcv*. Puis on vérifie que le résultat obtenu correspond à *flag*.

Algorithm 10 `check_cond`

```
1: function CHECK_COND(cnt, nzcv)
2:     mask ← cnt[mask]
3:     flags ← cnt[flags]
4:     if flags == (mask & nzcv) then
5:         return True
6:     else
7:         return False
8:     end if
9: end function
```

8.3.5 Alignement des instructions

8.3.5.1 Principe de l'alignement

Le principe d'alignement en mémoire consiste à placer les données et les instructions sur des adresses mémoires multiples d'un mot mémoire pour optimiser leur temps de chargement par le processeur. En effet, les bus qui relient le processeur à la mémoire chargent les données/instructions à partir d'adresses multiples d'un mot mémoire (4 octets pour les architectures 32 bits). Par exemple, un bus dont la

capacité de chargement est de 4 octets doit effectuer deux chargements pour charger une donnée de 4 octets lorsque l'adresse de celle-ci n'est pas multiple de 4. Les compilateurs essayent donc d'aligner le mieux possible les instructions et les données sur des adresses multiples de 4, en tenant compte de certains paramètres tels que l'architecture du processeur, de la mémoire cache, de la pagination de la mémoire virtuelle. Pour aligner les instructions, les compilateurs insèrent des instructions `nop` (opération de *padding*). Cela est particulièrement fréquent pour les processeurs supportant un jeu d'instruction de taille variable.

8.3.5.2 Alignement des instructions `cnt++`

L'objectif de cette modification est d'aligner les instructions de telle sorte qu'une instruction `cnt++` se retrouve dans chaque bloc de `size` bits de la mémoire programme. Cette opération est réalisée juste avant la phase émission de code afin de garantir les propriétés d'alignement souhaitées dans le code final.

La première tâche consiste à aligner tous les blocs de base du programme sur des adresses multiples d'un mot mémoire, en utilisant la fonction LLVM :

```
1 void llvm::MachineBasicBloc::setAlignment(unsigned align)
```

La seconde tâche consiste à parcourir toutes les instructions du programme, bloc de base par bloc de base, pour s'assurer que chaque bloc de `size` bits d'instructions (à partir du début du bloc de base) contient au moins une instruction qui incrémente le compteur (`cnt++`). Lorsqu'un bloc ne contenant pas un `cnt++` est identifié, une instruction `cnt++` est insérée avant ce bloc pour le décaler.

Cependant, pour que cela fonctionne, il faudrait que la taille de l'instruction `cnt++` soit inférieure à `size`. Sur un processeur supportant des instructions de tailles 16 bits et 32 bits, qui est le cas de la cible de cette implémentation, la taille de `cnt++` doit donc être de 16 bit.

8.3.6 Réservation de registres

Dans LLVM, le mécanisme d'allocation de registres est le même pour tous les backends. Cependant, chaque cible doit *spécialiser* l'implémentation d'un certain nombre d'interfaces exportées par le module *LLVM Target-independent code generator*. Parmi ces interfaces on a :
`llvm/Target/TargetRegisterInfo.h`. Cette interface permet à l'allocateur de registres d'obtenir des informations sur les registres physiques du processeur cible, notamment le nombre et la liste de registres disponibles. Dans le back-end ARM cette interface est implémentée par la classe :
`llvm/lib/Target/ARM/ARMBaseRegisterInfo.cpp`. C'est cette classe que nous avons modifiée pour réserver trois registres physiques. Un registre réservé est un registre qui ne sera pas visible par l'allocateur de registre et donc pas utilisé par le programme. Un extrait de cette modification est présenté ci-dessous :

```
1  #include "ARMBaseRegisterInfo.h"
2  #include "CCFI/CCFIContext.h"
3  ...
4  using namespace llvm;
5
6  BitVector ARMBaseRegisterInfo::
7  getReservedRegs(const MachineFunction &MF) const {
8  ...
9      if (isCCFIEnabled()){
10         Reserved.set(CCFIContext::CNT_REG); // compteur
11         Reserved.set(CCFIContext::PREV_CNT_REG); // compteur précédent
12         Reserved.set(CCFIContext::TMP_REG); // compteur temporaire
13 ...
14 }
```

La condition testée à la ligne 9 permet de réserver ces registres seulement lorsque l'application de ce schéma est activée dans le compilateur. Les trois registres réservés (lignes 10-12) sont définis dans l'interface `llvm/lib/Target/ARM/CCFI/CCFIContext.h`, que nous avons définie pour fournir aux autres passes de ce schéma des informations sur le contexte général du schéma. Un extrait de cette interface, faisant apparaître la définition des registres réservés, est présenté ci-dessous. Les trois registres physiques réservés sont donc `R6, R7` et `R8`, respectivement pour le compteur précédent, le compteur et un registre temporaire (dont l'utilité sera montrée dans l'annexe A).

```
1  #include "ARM.h"
2  ...
3  namespace CCFIContext{
4
5  const unsigned int PREV_CNT_REG   = ARM::R6;
6  const unsigned int CNT_REG        = ARM::R7;
7  const unsigned int TMP_REG        = ARM::R8;
8  ...
9  }
```

8.4 Expérimentations

Pour expérimenter ce schéma, nous avons réalisé une évaluation de performances sur le microcontrôleur STM32 [69], et une évaluation de sécurité en utilisant le simulateur de fautes présenté dans le chapitre 4.

8.4.1 Évaluation de performances

L'évaluation de performances a été effectuée avec les codes sources de référence déjà utilisés dans les chapitres précédents. Le tableau 8.3 présente le résultat des mesures de temps d'exécution et de taille de code des programmes testés. Chaque programme est compilé avec différentes options d'optimisation. Le temps d'exécution

TABLE 8.3 – Résultat de l'évaluation de performances du schéma CCFI. Le temps d'exécution est mesuré en nombre de cycle d'horloge et la taille du code en octets.

	Opt.	Référence		Protégé		Surcoût	
		Temps	Taille	Temps	Taille	Temps	Taille
AES 8-bit	-O0	17940	1736	56152	6093	×3.13	×3.51
	-O1	9814	1296	25614	4160	×2.61	×3.21
	-O2	5256	1936	11037	5440	×2.10	×2.81
	-O3	5256	1936	11037	5440	×2.10	×2.81
	-Os	7969	1388	18009	3678	×2.26	×2.65
AES 32 bits	-O0	1890	6140	5802	20016	×3.07	×2.87
	-O1	1226	3120	3653	8954	×2.98	×2.26
	-O2	1142	3120	2295	6708	×2.01	×2.15
	-O3	1142	3120	2295	6708	×2.01	×2.15
	-Os	1144	3116	3088	8195	×2.70	×2.63
Verify-PIN	-O0	212	248	682	756	×3.22	×3.05
	-O1	101	144	253	407	×2.51	×2.83
	-O2	42	200	90	498	×2.15	×2.49
	-O3	42	200	89	498	×2.12	×2.49
	-Os	81	180	135	471	×2.39	×2.62

et la taille du code sont mesurés avec la méthode utilisée dans le chapitre 5 (section 5.3).

On peut constater que le surcoût engendré par le schéma de protection est de ×2.5 en moyenne pour chacun des programmes évalué. Il est possible d'expliquer ces coûts observés en s'intéressant aux modifications introduites par ce schéma à l'échelle d'un bloc de base. Rappelons que notre architecture d'expérimentation (ARMv7m) supporte un jeu d'instruction de taille variable : 16 bits (Thumb1) et 32 bits (Thumb2). Un programme compilé pour cette architecture contient en moyenne autant d'instructions de taille 16 bits que 32 bits dans un bloc de base. Ainsi, insérer une instruction `cnt++` de taille 16 bits avant chaque instruction du bloc de base ne double pas la taille du bloc (en moyenne). En plus, le temps d'exécution d'une instruction `cnt++` est de 1 cycle, ce qui ne double pas non plus le temps d'exécution du bloc de base. Ensuite, deux instructions sont ajoutées en début du bloc pour initialiser son compteur et une ou deux instructions pour préparer le compteur suivant, ce qui représente un coût fixe pour chaque bloc de base. Il découle de cette analyse que plus un bloc de base contient d'instructions, plus le coût du schéma est amorti à l'échelle de ce bloc.

On peut noter aussi la corrélation entre le niveau d'optimisation et le coût du schéma protection en termes de temps d'exécution ; plus le code du programme évalué est optimisé moins le surcoût engendré est élevé. Cela s'explique par le fait que les optimisations de code tendent à réduire le nombre de branchements et donc le nombre de blocs de base présents dans un programme. Dans le paragraphe précé-

TABLE 8.4 – Résultat de la simulation des modèles de fautes sur le programme Verify-PIN protégé par le schéma CCFI

Modèles de fautes	fautes détectées
`REPL_BR`	✓
`SKIP_BLOCK{16 BITS, 1}`	✗
`SKIP_BLOCK{32 BITS, 1}`	✓
`SKIP_BR`	✓
`REPL_BR`, `SKIP_BLOCK{16 BITS, 1}`	✗
`REPL_BR`, `SKIP_BLOCK{32 BITS, 1}`	✓
`SKIP_BR`, `SKIP_BLOCK{16 BITS, 1}`	✗
`SKIP_BR`, `SKIP_BLOCK{32 BITS, 1}`	✓

dent, nous avons conclu que le schéma engendre, pour chaque bloc de base, un coût variable qui dépend de la taille du bloc et un coût fixe. Par conséquent, moins un programme contient de blocs de base plus le coût fixe est réduit. Et de manière indirecte, pour un programme donné, moins il contient de blocs de base plus la taille des blocs est important et donc, plus le coût variable est réduit.

8.4.2 Évaluation sécuritaire

Le tableau 8.4 présente les résultats de la simulation des modèles de fautes sur le programme Verify-PIN protégé par le schéma CCFI. Ces résultats montrent que le schéma CCFI protège bien conte les modèles de fautes considérés.

8.5 Conclusion

Dans ce chapitre, nous avons présenté la conception et la mise en œuvre d'un schéma de protection CCFI combinant l'intégrité du flot de contrôle (CFI) et l'intégrité des instructions exécutées. Le schéma CCFI garantit non seulement que le chemin d'exécution suivi est prévu, mais aussi qu'il est valide d'un point de vue fonctionnel (en tenant compte des différentes conditions de sauts).

Les évaluations expérimentales ont montré que le schéma CCFI protège bien contre les modèles de fautes considéré (dans la section 8.1).

Chapitre 9

Conclusion et perspectives

9.1 Conclusion

Il est établi qu'une injection de faute dans un système embarqué peut porter atteinte à la confidentialité, à l'intégrité ou à l'authenticité des données qu'il contient. Cette thèse a été développée autour de l'application automatisée de contre-mesures logicielles contre les attaques par injection de fautes sur les systèmes embarqués.

L'approche *code source* qui consiste à insérer les contre-mesures dans le code source de l'application à sécuriser ne permet pas de garantir la conservation des propriétés de sécurité de la contre-mesure dans le code binaire de l'application, en raison des optimisations réalisées par le compilateur. L'approche *code binaire* qui consiste à insérer les contre-mesures dans le code binaire (ou assembleur) de l'application impacte considérablement ses performances à cause des différentes transformations nécessaires. Cette thèse a exploré l'approche *compilation* qui consiste à intégrer les contre-mesures dans le compilateur.

Nous avons proposé un compilateur LLVM permettant l'application automatisée de plusieurs schémas de contre-mesures lors de la compilation : l'utilisateur fournit un code source avec des annotations décrivant les paramètres de la contre-mesure à appliquer puis le compilateur produit automatiquement un code binaire fonctionnel et sécurisé.

Schéma de tolérance au saut d'instructions : Nous avons commencé par intégrer le schéma de tolérance au saut d'instructions initialement proposé et formellement vérifié par Moro et al. [76]. Le schéma consiste à dupliquer toutes les instructions après les avoir transformées dans des formes idempotentes. Cette première mise en œuvre nous a permis de valider expérimentalement l'avantage de l'approche compilation par rapport à l'approche binaire sur les critères de temps d'exécution et d'empreinte mémoire, en comparant nos résultats avec ceux de Moro et al. sur les mêmes codes de références et le même processeur cible.

Généralisation du schéma de tolérance : Nous avons ensuite généralisé le schéma de tolérance pour protéger contre le saut de N instructions, qui peuvent être consécutives ou non. Cette généralisation renforce le niveau de sécurité apporté par le schéma et permet notamment de résister aux modèles d'attaques utilisés par Rivière et al. [91] et Yuce et al. [106]. Plutôt que de dupliquer N fois chaque instruction pour résister à un saut de N instructions consécutives, nous avons proposé de dupliquer une seule fois puis de séparer l'instruction originale de son duplicata d'une distance

D proportionnelle à N. Nous avons montré expérimentalement que cette solution permet de réduire le coût du schéma tout en maintenant le même niveau de sécurité.

Intégrité de flot de contrôle (CFI) combiné à la tolérance aux sauts d'instructions : Nous avons conçu et intégré un schéma CFI permettant de protéger contre les scénarios d'attaques consistant à faire dévier un programme de son chemin d'exécution. Nous avons découvert lors des expérimentations la vulnérabilité de ce schéma aux sauts d'instructions. Nous avons donc combiné le schéma CFI à celui de tolérance déjà présent dans notre compilateur. Nous avons illustré expérimentalement que le schéma combiné permet de résister à une attaquant combinant détournement de flot de contrôle et saut d'instructions.

Schéma intégrité de code et de flot de contrôle (CCFI) : Nous avons conçu et mis en œuvre un schéma combinant intégrité de flot de contrôle et intégrité des instructions exécutées. Le schéma garantit la validité du chemin d'exécution suivi ainsi que l'exécution de toutes les instructions le long de ce chemin. Il prend en compte les conditions de sauts pour vérifier dynamiquement la validité d'un chemin pour un contexte d'exécution donné.

Ce schéma combine les avantages des schémas précédents tout en coûtant moins cher (en temps d'exécution et en empreinte mémoire) et en apportant plus de garanties que ces schémas appliqués de manière superposée.

Simulateur de fautes : Nous avons développé un simulateur de fautes pour évaluer nos différents schémas. Le simulateur, basé sur QEMU, prend en entrée un fichier binaire, le modèle de fautes à évaluer et un distingueur pour déterminer si une faute a eu un effet sur une instance d'exécution du binaire.

9.1.1 Bilan

Cette thèse a pu montrer que l'approche compilation est un bon compromis entre l'approche source qui ne garantit pas la conservation des propriétés de sécurité dans le code binaire et l'approche binaire qui impacte considérablement les performances de l'application finale.

La sécurité a toujours un coût et une application sécurisée coutera toujours plus cher en temps d'exécution et en empreinte mémoire qu'une application non sécurisée. Mais ce coût peut être réduit lorsque la sécurité est prise en compte tôt dans le processus de génération de code.

Le compilateur est une brique importante dans l'ingénierie logicielle : il est le pont reliant l'univers du logiciel à l'univers du matériel. Nous sommes convaincus que le compilateur est l'endroit idéal pour intégrer la sécurité dans les applications logicielles. Cependant, les utilisateurs de compilateurs propriétaires ou encore ceux qui n'ont accès qu'au code binaire d'une application, n'ont d'autres choix que de se tourner vers les autres approches de sécurisation.

9.2 Perspectives

L'utilisation d'un compilateur pour appliquer automatiquement des contre-mesures contre les fautes ouvre plusieurs pistes de recherches et d'améliorations.

- **Vérification formelle** Une perspective d'amélioration des travaux de cette thèse pourrait être de vérifier formellement l'équivalence fonctionnelle entre une version sécurisée par notre compilateur et une version non sécurisée d'un même programme.
- **Étude de l'impact sur les attaques par observation** À cause de la duplication d'instructions et la manipulation répétée de certaines données, il serait intéressant d'étudier l'impact de ces schémas de contre-mesure sur les attaques par observation.
- **Combinaison avec des contre-mesures aux attaques par observation** S'il s'avère que ces schémas augmentent la fuite des données, une piste d'amélioration pourrait être d'étudier la possibilité de les combiner avec des schémas de contre-mesures aux attaques par observation, par exemple le masquage ou le polymorphisme de code.
- **Implémentation hybride logicielle/matérielle du schéma CCFI** Les contre-mesures implémentées en matériel ayant un net avantage en temps d'exécution comparées à celles implémentées en logiciel, il pourrait être envisageable d'ajouter un support matériel permettant d'accélérer la réalisation de certaines tâches du schéma, notamment la vérification de la validité des compteurs.
- **Étude de l'impact du langage source** Il serait intéressant de voir si le langage de programmation dans lequel le programme à sécuriser est écrit, a un impact sur les coûts des schémas de contre-mesures appliqués au moment de la compilation.

La sécurité des systèmes embarqués sera d'avantage un enjeu majeur dans les années à venir. Si certains systèmes dont la fonction principale est dédiée à la sécurité, tels que les cartes à puces ou les *secure elements*, embarquent déjà un certain nombre de mécanismes de protection robustes face aux attaques physiques ; les systèmes connectés avec une composante physiques telle que les systèmes cyber-physiques et plus particulièrement l'internet des objets (IOT) doivent encore être renforcés par une couche de sécurité. En effet, ces systèmes utilisent souvent des piles logicielles et matérielles *sur l'étagère* (COTS) dont les vulnérabilités font déjà beaucoup parler et ils sont la cible de nombreuses attaques logicielles. Il est très probable que dans les années à venir, suite au déploiement massif des IOT dans notre quotidien, ces systèmes deviennent une nouvelle cible des attaques physiques. L'automatisation de l'application de protections logicielles contre les attaques apparaît comme une piste de recherche prometteuse.

Annexe A

Description des passes de compilation du schéma CCFI

A.1 Détails d'implémentation

L'objectif dans cette section est de présenter plus en détails les passes de compilation impliquées dans la mise en œuvre de ce schéma. La figure A.1 présente une vue sur ces passes, faisant apparaître leur ordre d'exécution. Pour détailler le fonctionnement de ces passes, nous allons partir d'un exemple initial sur lequel le processus d'application du schéma CCFI sera déroulé étape par étape pour expliciter le rôle de chaque passe dans ce processus. L'exemple ci-dessous, écrit en C, a été volontairement simplifié et choisi pour faire apparaître plusieurs configurations des blocs de base, tout en restant court.

```
1 volatile int x;
2 int test(int *a, int *b, int *res){
3     if(x == 5)
4         *res = *a + *b;
5     *res = *a + *b;
6     return 0;
7 }
```

Le mot clé `volatile` est utilisé pour empêcher l'optimisation de la variable `x` par le compilateur, et ainsi conserver la structure de contrôle `if`.

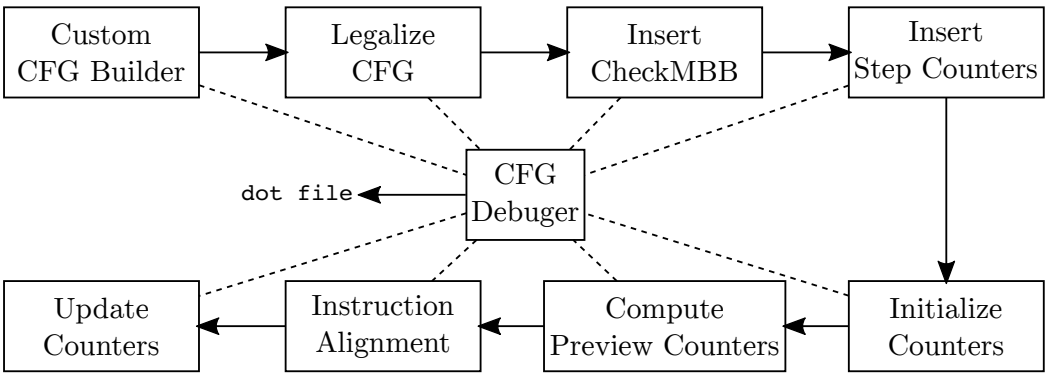


Figure A.1 – Ordre d'exécution des passes du schéma CCFI

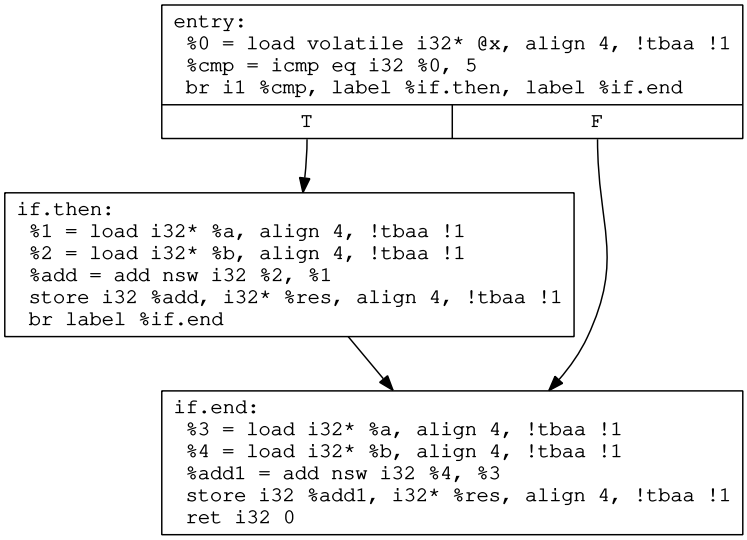


FIGURE A.2 – CFG de la fonction `test`

Les passes de ce schéma sont insérées dans le back-end du compilateur, juste après la phase d'allocation de registres. À ce stade de la compilation, le programme est représenté par un CFG, et les instructions à l'intérieur du CFG sont dans la forme `llvm::MachineInstr` (instructions machine). La figure A.2 représente le CFG de la fonction `test` de cet exemple.

A.1.1 Passe `Custom CFG Builder`

Cette passe étend le CFG initial de LLVM pour construire un nouveau CFG qui prend en compte les besoins du schéma CCFI. En effet, ce schéma introduit de nouveaux attributs pour les blocs de base. Chaque bloc de base a désormais :

— `inputCounter` : un compteur propre initialisé par une valeur unique à l'entrée du bloc de base.

— `nbIncrement` : le nombre de fois que le compteur a été incrémenté dans le bloc de base.

— `outputCounter` : la valeur attendue du compteur à la sortie du bloc de base.

— `prevCounter` : la valeur attendue du compteur précédent, qui d'après le schéma peut être différente de chacune des `outputCounter` des prédécesseurs.

— `nxtPrevCounter` : la valeur de `prevCounter` du successeur du bloc.

— `nbPred` et `nbSucc` : le nombre de prédécesseurs et successeurs du bloc de base.

— `type` : le type du bloc de base, qui peut être :

`INTER` pour les blocs intermédiaires ajoutés, par la passe `Legalize CFG` présentée plus loin, pour adapter la structure du CFG aux différentes configurations prises en compte par le schéma.

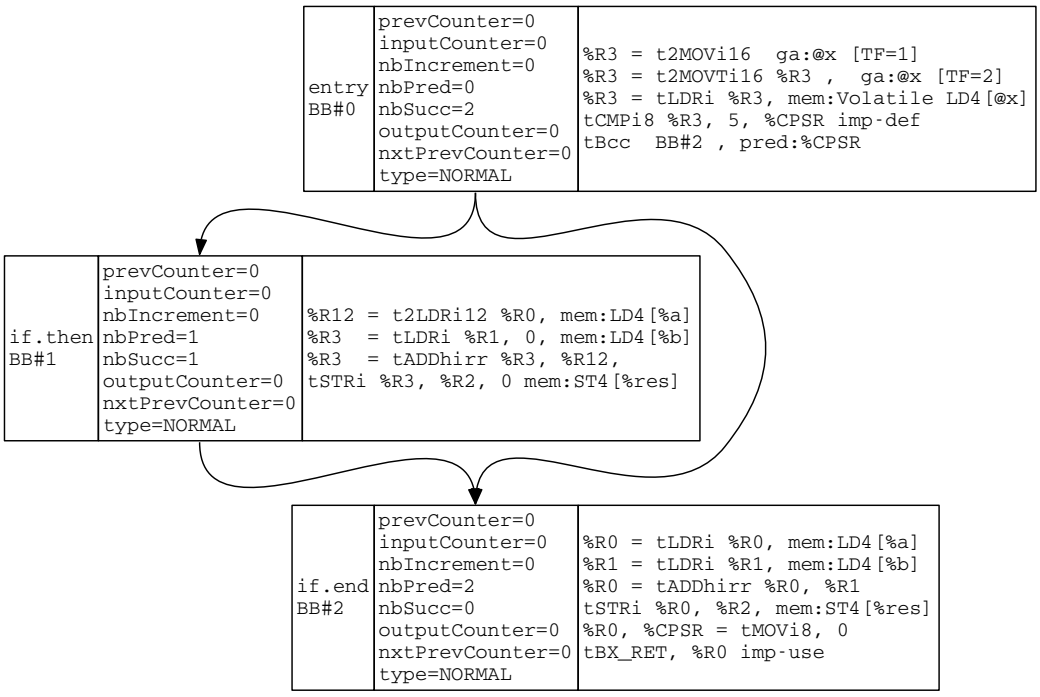


FIGURE A.3 – CFG étendu de la fonction `test`

`CHECK` pour les blocs insérés, par la passe `Insert CheckMBB` présentée plus loin, pour vérifier la validité d'un compteur à l'entrée d'un bloc de base.

`TRAP` pour le bloc *trapBB* qui est le bloc de destination en cas d'échec de vérification d'un compteur. Ce bloc est inséré par la passe `Insert CheckMBB`, et il est successeur de tous les blocs de type `CHECK`

`NORMAL` pour tous les autres blocs de base.

Certains de ces attributs pourraient être calculés à tout moment, c'est le cas par exemple de `outputCounter` qui est égale à `inputCounter + nbIncrement`. Mais nous avons fait le choix de les garder en tant qu'attributs pour des raisons de maintenabilité, de clarté du code et surtout pour simplifier l'implémentation des autres passes qui utilisent ces informations.

A.1.2 Passe `CFG Debuger`

Le rôle de cette passe est de faciliter le débogage des autres passes du schéma. Elle collecte et affiche (en temps réel) dans la console les étapes intermédiaires de la passe en cours d'exécution. Elle permet aussi d'exporter le CFG, à tout moment, dans un format `dot` que nous utilisons pour afficher graphiquement le CFG. Cette passe peut être considérée comme une passe *utilitaire* d'après la classification des passes faite par LLVM. Elle est connectée à toutes les autres passes du schéma opérant sur le CFG du programme.

La figure A.3 présente le CFG étendu de la fonction `test`. Cette figure est

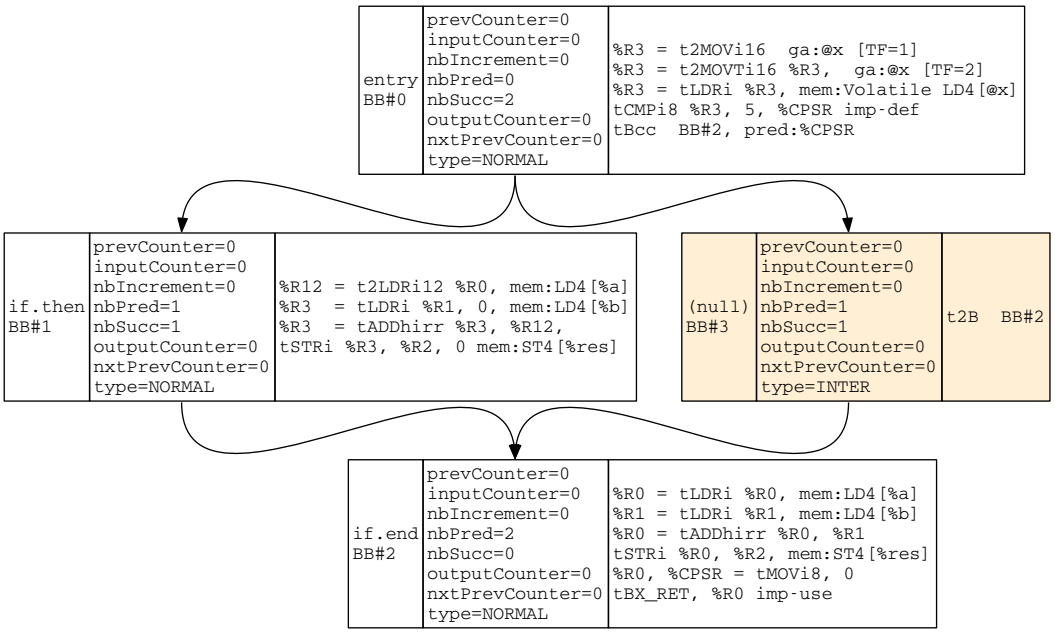


FIGURE A.4 – CFG de la fonction `test` après la passe `Legalize CFG`

construite à partir du fichier `.dot` exporté par cette passe juste après la passe `Custom CFG Builder`. Chaque bloc de base est représenté par un rectangle composé de trois parties : la première contient le nom et le numéro du bloc. Le numéro d'un bloc est l'ordre de ce bloc dans la structure de donnée utilisée par LLVM pour implémenter le CFG, et n'a rien à voir avec l'ordre d'exécution des blocs. Les noms sont assignés par LLVM dès le début de partie intermédiaire. Le nom de chaque bloc est en rapport avec sa position dans les structures de contrôle du code source (if.then, else, etc.). Les nouveaux blocs créés pendant la compilation, n'ayant aucun lien direct avec la structure du code source, sont nommés `(null)`. La deuxième partie contient les attributs du bloc, présentés dans la section précédente. La dernière partie contient les instructions du bloc de base.

A.1.3 Passe `Legalize CFG`

Le rôle de cette passe est de préparer le CFG et de garantir sa validité pour les autres passes. Cette passe analyse le CFG en effectuant un parcours en profondeur (DFS) à partir du bloc `entry`, et réalise les transformations décrites par les différentes configurations. À chaque parcours, cette passe retourne vrai ou faux selon qu'une transformation du CFG ait été réalisée ou non et elle est ré-exécutée tant qu'elle retourne vrai.

```
1 while (legalizeCFG());
```

Sur notre exemple, le CFG de la fonction `test` pose problème. Pour comprendre ce problème, rappelons qu'à la fin de chaque bloc de base est calculé le compteur

précédent des blocs successeurs. D'une part, d'après la configuration *plusieurs prédécesseurs* (section 8.3.2), le compteur précédent du bloc `BB#2` est le ou-exclusif des compteurs des blocs `BB#1` et `BB#0`, et doit donc être calculé à la fin de ces blocs. D'autre part, d'après la configuration *un seul prédécesseur* (section 8.3.1), le compteur précédent du bloc `BB#1` doit aussi être calculé à la fin du bloc `BB#0`. Par conséquent, `BB#0` devrait calculer deux compteurs précédents, ce qui n'est pas possible d'après le schéma qui ne prévoit qu'une seule variable (un seul registre) pour stocker le compteur précédent.

Pour trouver ce type de configuration dans le CFG, cette passe recherche le pattern suivant : un bloc de base ayant plusieurs prédécesseurs et dont un de ses prédécesseurs possède au moins deux successeurs, c'est-à-dire un autre successeur que lui. Ce pattern correspond à celui de la configuration *plusieurs prédécesseurs et plusieurs successeurs* (section 8.3.3). Cette passe applique donc la transformation préconisée pour cette configuration afin de résoudre le problème, à savoir l'insertion d'un bloc de base intermédiaire entre `BB#0` et `BB#2`. La figure A.4 montre le nouveau CFG de la fonction `test` où ce problème est résolu.

Un autre problème de ce CFG est que le compteur du dernier bloc de base ne pourra pas être évalué d'après le schéma. Ce qui pose un problème de sécurité car toutes les instructions de ce bloc pourraient être sautées sans que cela ne puisse être détecté. Cette passe résout ce problème en insérant un nouveau bloc après le dernier (figure A.4). Pour que ce nouveau bloc puisse être atteint lors de l'exécution : (1) la dernière instruction (l'instruction de retour de fonction) est supprimée du dernier bloc et insérée dans le nouveau, (2) le nouveau bloc est déclaré comme étant successeur du dernier.

A.1.4 Passe `Insert CheckMBB`

Le rôle de cette passe est d'insérer les instructions et les blocs de base nécessaires à la vérification de la validité de chaque compteur. Appliquer le schéma tel quel ajouterait un deuxième point de sortie pour chaque bloc de base, ce qui contredit sa définition (section 3.2.3.3). Cette passe résout cette limitation en insérant de nouveaux blocs de base. Les transformations réalisées par cette passe sont illustrées par le CFG de la figure A.5. Cette passe commence par insérer le bloc `trapBB` puis procède à l'insertion des blocs de type `CHECK` comme détaillé ci-après.

A.1.4.1 Insertion du bloc de base `trapBB`

La première tâche de cette passe est la création et l'insertion d'un bloc de base *puits*, c'est-à-dire de type `TRAP` (le bloc gris sur le CFG de la figure A.5). Ce bloc a la particularité de ne pas avoir de successeurs, et tous ses prédécesseurs sont des blocs de base de type `CHECK` (présentés dans la section suivante). Si l'utilisateur avait fourni une fonction annotée avec `@CFI_handler`, une instruction réalisant l'appel à cette fonction aurait été insérée dans ce bloc de base. Par défaut, cette passe insère une instruction de branchement vers le même bloc (une boucle infinie).

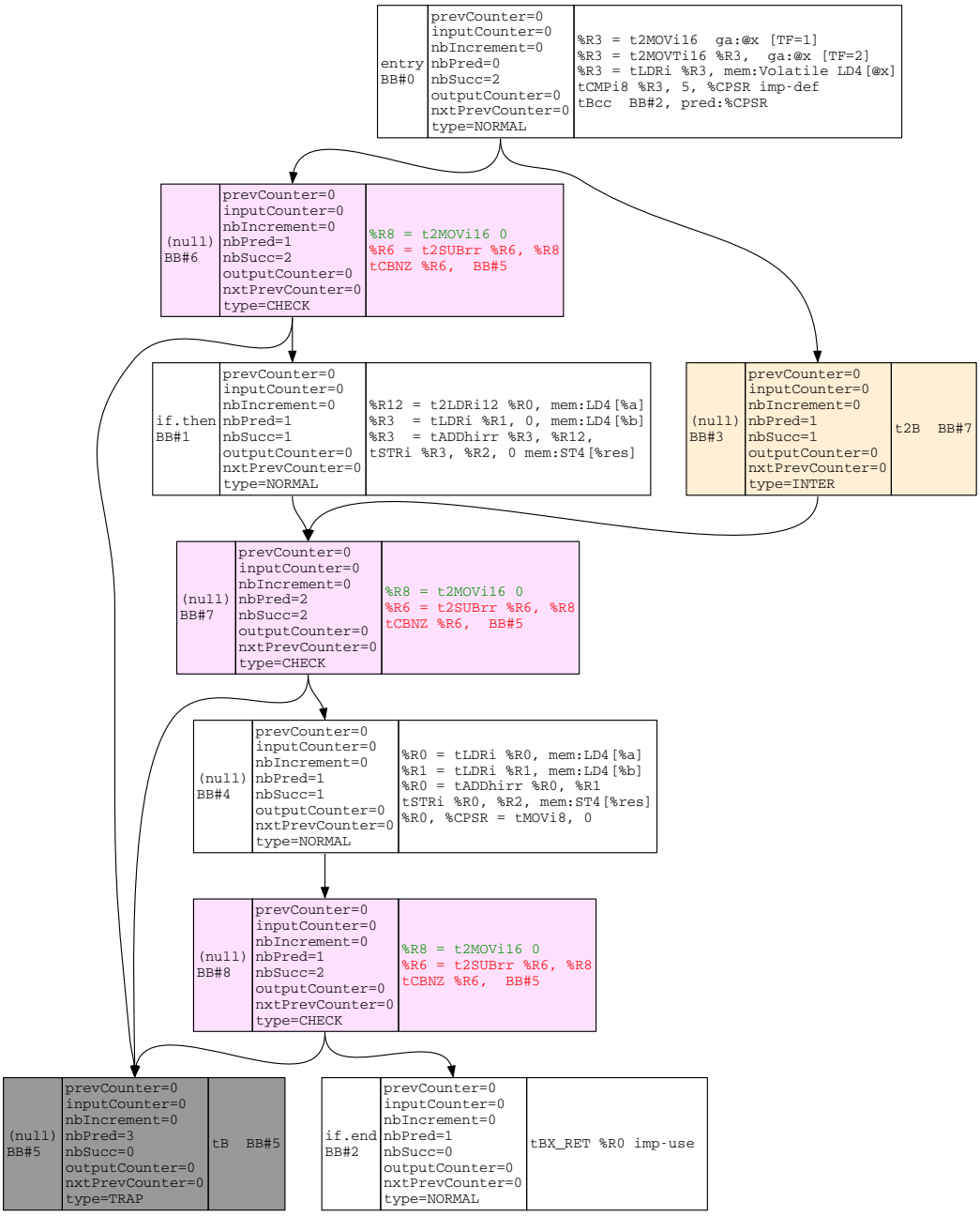


FIGURE A.5 – CFG de la fonction `test` après la passe `Insert CheckMBB`

A.1.4.2 Insertion des blocs de base `CHECK`

Les blocs de type `CHECK`, représentés en rose sur la figure A.5, sont les blocs dans lesquels les compteurs sont évalués. Ils sont insérés avant tous les blocs de base de type `NORMAL`, sauf avant le bloc `entry`. Chaque bloc `CHECK` :

— est un successeur de tous les prédécesseurs du bloc `NORMAL` avant lequel il est inséré.

— a deux successeurs : le bloc `NORMAL` qu'il précède et le bloc `TRAP`.

Dans chaque bloc `CHECK`, cette passe insère les instructions qui réalisent la vérification du compteur, qui sont :

— `mov TMP_REG, #cnt_prec` : mettre dans le registre temporaire (`R8`) la valeur attendue du compteur précédent. `#cnt_prec` est remplacé par `0` puisqu'au moment où cette passe s'exécute les compteurs n'ont pas encore été initialisés et la valeur attendue du compteur précédent ne peut donc pas être calculée.

— `sub PREV_CNT_REG, TMP_REG` : cette instruction réalise une soustraction entre la valeur attendue du compteur précédent et sa valeur effective contenue dans le registre `PREV_CNT_REG` (`R6`), et stocke le résultat dans `R6`.

— `cbnz R6, TRAP_BB` : cette instruction réalise un saut dans le bloc `TRAP` seulement si le contenu du registre `R6` est différent de `0`. En effet, si le `R6` est différent de `0` cela signifie que la valeur attendue du compteur précédent et sa valeur effective sont différentes, et qu'une faute a donc eu lieu.

A.1.5 Passe `Insert Step Counters`

Cette passe parcourt le CFG pour insérer des instructions qui incrémentent les compteurs entre les instructions. L'instruction insérée est `add CNT_REG, #1`, elle est de taille 16 bits et incrémente le compteur `CNT_REG` (`R7`) de 1. Comme illustré par la figure A.6, ces instructions ne sont insérées que dans les blocs de base de type `NORMAL` et `CHECK`, car c'est dans ces blocs que se trouvent les instructions à protéger. Pour chaque bloc de base, cette passe met à jour de l'attribut `nbIncrement`.

A.1.6 Passe `Initialize Counters`

Le rôle de cette passe est d'initialiser les compteurs de chaque bloc de base. Chaque compteur est initialisé à une valeur aléatoire unique tirée au moment de la compilation. Le compteur de chaque bloc de base (de type `NORMAL`) est initialisé dans le bloc `CHECK` qui le précède et avant les instructions qui vérifient la validité du compteur précédent. Le bloc `entry` n'étant précédé par aucun bloc `CHECK`, son compteur est initialisé avant sa première instruction. Le dernier bloc `CHECK` étant le dernier à vérifier la validité d'un compteur n'initialise aucun nouveau compteur.

Cette passe utilise deux instructions pour initialiser un compteur, qui sont :

— `eor CNT_REG, CNT_REG` qui permet d'initialiser le registre `CNT_REG` à 0, en réalisant un ou-exclusif du registre avec lui-même. Car, vu que le registre est

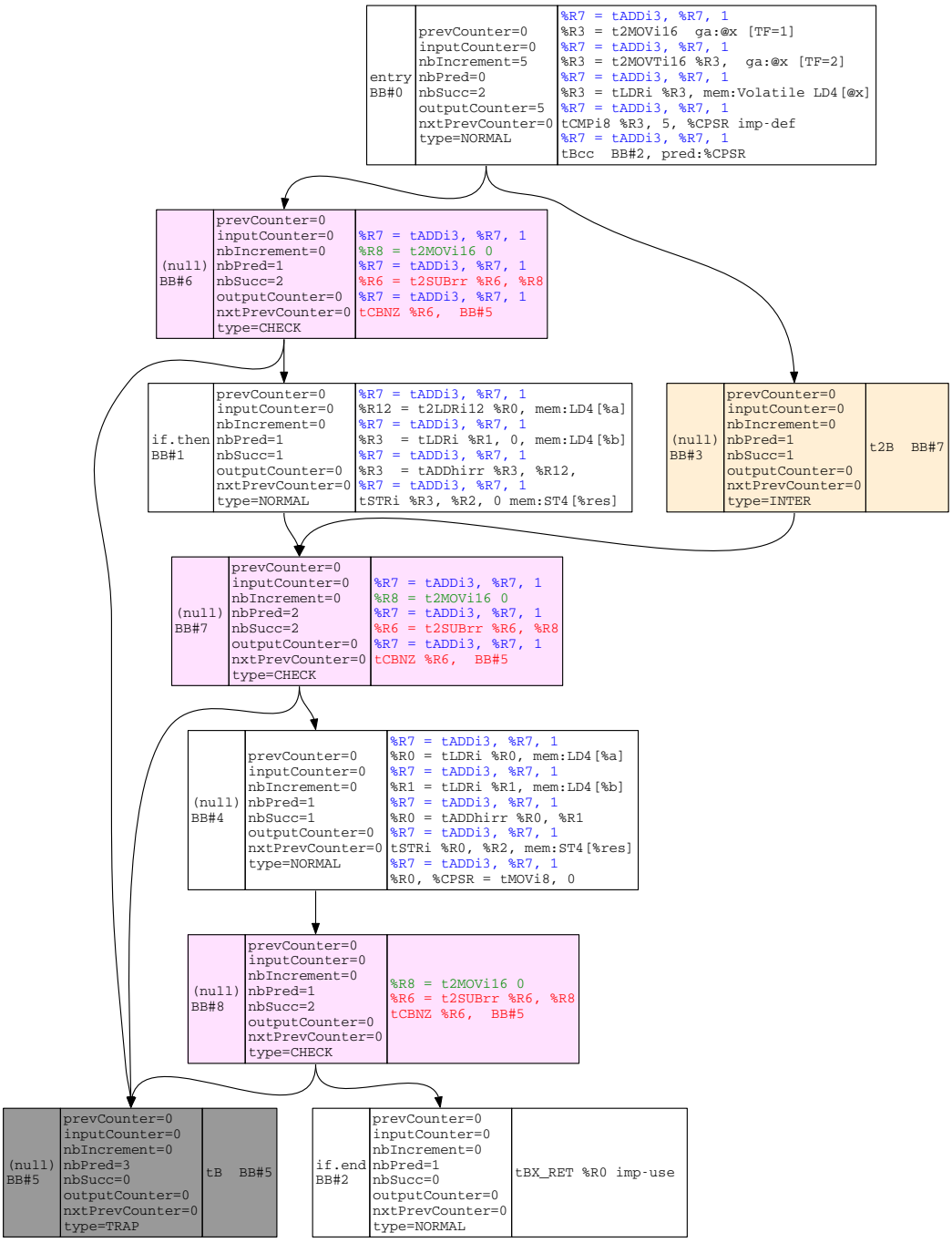


FIGURE A.6 – CFG de la fonction `test` après la passe `Insert Step Counters`

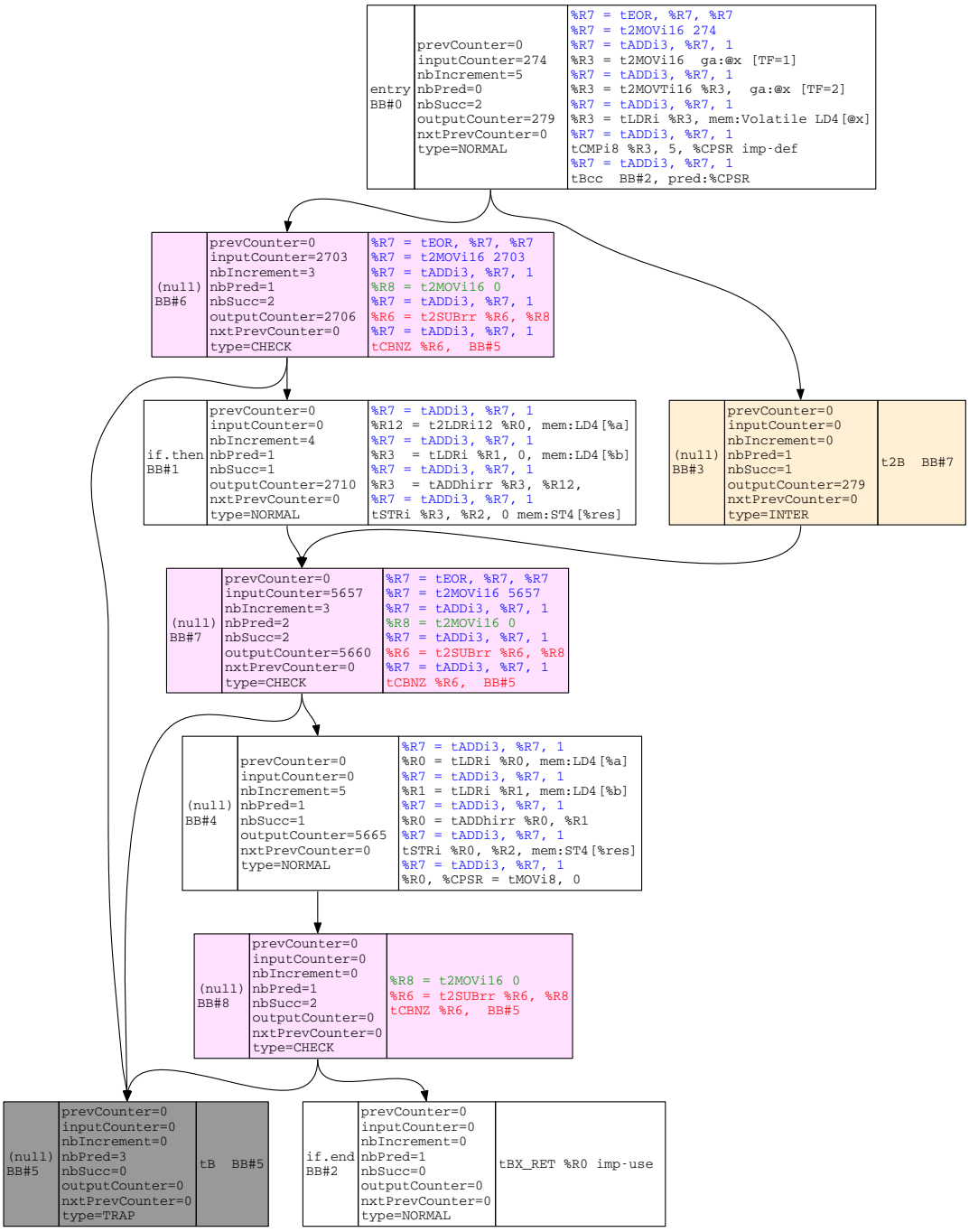


FIGURE A.7 – CFG de la fonction `test` après la passe `Initialize Counters`

divisé en trois zones de bits (`cnt`, `flags` et `mask`) et est le même pour tous blocs de base, il est nécessaire de commencer par le réinitialiser à 0 avant d'y charger la valeur d'un nouveau compteur `cnt`.

— `mov CNT_REG, #RANDOM_VALUE` qui charge dans `CNT_REG` la valeur du compteur.

Les modifications introduites par cette passe dans le CFG sont illustrées par la figure A.7. Elle termine par mettre à jour l'attribut `inputCounter` de chaque bloc avec `#RANDOM_VALUE`.

A.1.7 Passe `Compute Preview Counters`

Maintenant que les compteurs ont été initialisés et incrémenté plusieurs fois dans chaque bloc de base, cette passe calcule les valeurs attendues des compteurs précédents. Les compteurs précédents étant évaluées dans les blocs `CHECK`, leurs valeurs sont calculées dans les prédécesseurs de ces blocs : les blocs `NORMAL` et `INTER`. La figure A.8 montre les nouvelles instructions insérées pour calculer les compteurs précédents.

Cette passe itère les blocs `CHECK`s. Si un bloc `CHECK` a un seul prédécesseur, alors son compteur précédent (`cnt_prec`) est égal au compteur courant (`cnt`) de son prédécesseur. C'est le cas du bloc `BB#6` qui a pour prédécesseur le bloc `BB#0`. Une instruction est donc insérée à la fin du bloc prédécesseur pour copier `CNT_REG` dans `PREV_CNT_REG` (`mov R6, R7`).

Si un bloc `CHECK` possède plusieurs prédécesseurs, alors son compteur précédent est le ou-exclusif des compteurs courants de ses prédécesseurs. C'est le cas du bloc `BB#7` qui a pour prédécesseurs les blocs `BB#1` et `BB#3`. Les instructions suivantes sont insérées dans les blocs prédécesseurs pour réaliser ce calcul et mettre à jour le registre `PREV_CNT_REG`.

— `mov PREV_CNT_REG, #neighborCNT` qui récupère la valeur attendue du compteur courant du bloc de base voisin (`#neighborCNT`), c'est-à-dire de l'autre bloc qui est aussi prédécesseur de `BB#7`.

— `eor PREV_CNT_REG, CNT_REG` qui réalise le ou-exclusif des deux compteurs et stock le résultat dans `PREV_CNT_REG`.

A.1.8 Passe `Instruction Alignement`

Cette passe met en œuvre l'algorithme d'alignement d'instruction décrit dans la section 8.3.5. Elle insère, quand cela est nécessaire, des instructions `cnt++` pour ajuster l'alignement des instructions d'un bloc de base. Cette passe, en modifiant le nombre de `cnt++` insérés dans un bloc de base, invalide la valeur attendue de certains compteurs du CFG. Une mise à jour de ces valeurs est donc nécessaire après cette passe.

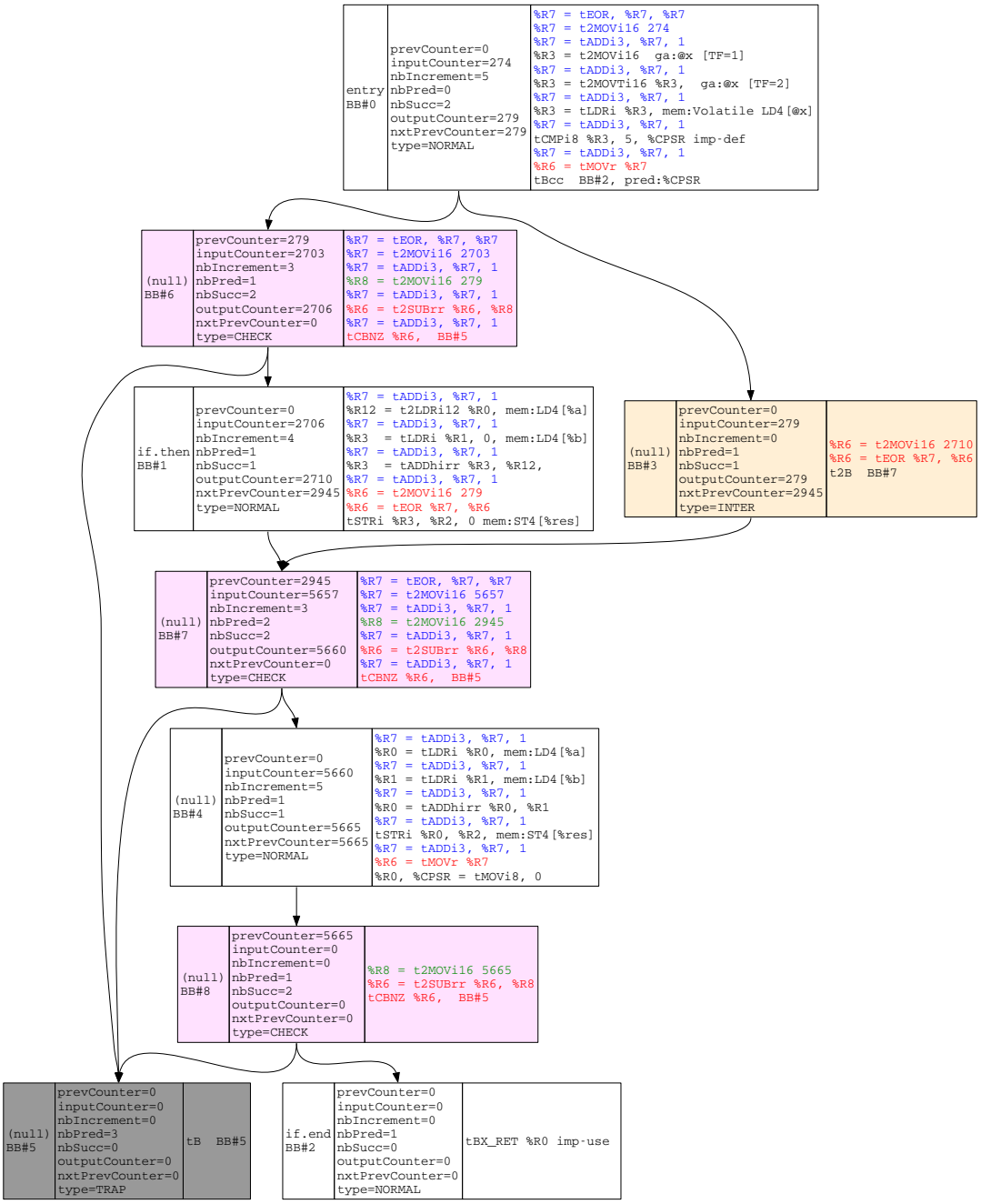


FIGURE A.8 – CFG de la fonction `test` après la passe `Compute Preview Counters`

A.1.9 Passe `Update Counters`

Le rôle de cette passe est de mettre à jour tous les compteurs du CFG suite aux éventuelles modifications intervenues lors de l'alignement des instructions. Elle est exécutée après toutes les passes du schéma pour garantir la validité du schéma dans le code binaire. Lorsqu'une instruction qui incrémente un compteur a été ajoutée dans un bloc de base pour ajuster l'alignement, cette passe recalcule les attributs `outputCounter` et `nxtPreviewCounter` de ce bloc, puis propage les nouvelles valeurs dans les blocs de base concernés.

Bibliographie personnelle

Brevet

- Procédé d'exécution d'un code machine d'une fonction sécurisée
  T. Barry, D. Couroussé et K. Heydemann
  Institut National de la Propriété Industrielle, 11/04/2017
  Numéro de demande : FR 1753175

Conférence avec acte

- Automated Combination of Tolerance and Control Flow Integrity Countermeasures against Multiple Fault
  T. Barry, D. Couroussé, B. Robisson and K. Heydemann
  2017 European LLVM Developers Meeting, Saarbrücken, Germany
- Compilation of a Countermeasure Against Instruction-Skip Fault Attacks
  T. Barry, D. Couroussé and B. Robisson
  3th Workshop on Cryptography and Security in Computing Systems, Prague, Czech, 2016
- Runtime Code Polymorphism as a Protection against Side Channel Attacks
  D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin and J-L. Lanet
  10th International Conference on Information Security Theory and Practice, Crete, Greece, 2016
- A Template Attack Against VERIFY PIN Algorithms
  H. Le Bouder, T. Barry, D. Couroussé, J-L. Lanet and R. Lashermes
  13th International Conference on Security and Cryptography, Lisbon, Portugal, 2016
- COGITO : Code Polymorphism to Secure Devices
  D. Couroussé, B. Robisson, J-L. Lanet, T. Barry, H. Noura, P. Jaillon, and P. Lalevée
  11th International Conference on Security and Cryptography, Vienna, Austria, 2014

Publications en cours

- The Multiple Ways to Automate the Application of Software Countermeasures against Physical Attacks : Pitfalls and Guidelines
  N. Belleville, T. Barry, D. Couroussé, K. Heydemann, H-P. Charles and B. Robisson
  Cyber-Physical Security Education, Paris, France, 2017

Conférence sans acte

- Compilation of Countermeasures Against Fault Injection Attacks
  T. Barry, D. Couroussé and B. Robisson
  6ème édition de la rencontre Crypto'Puces, Île de Porquerolles, France, 2017
- Compilation for the Composition of Software Protections for Embedded Systems
  T. Barry, D. Couroussé and B. Robisson
  5ème édition de la rencontre Crypto'Puces, Île de Porquerolles, France, 2015

Posters

- Compiler-based Countermeasure Against Fault Attacks
  T. Barry, D. Couroussé and B. Robisson
  17th Workshop on Cryptographic Hardware and Embedded Systems, Saint-Malo, France, 2015
- Runtime Code Polymorphism as a Protection Against Physical Attacks
  D. Couroussé, T. Barry and B. Robisson
  17th Workshop on Cryptographic Hardware and Embedded Systems, Saint-Malo, France, 2015

Bibliographie

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM CCS'05*, New York, NY, USA, 2005. ACM. (Cité en page 24.)

[2] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the 49th Annual Design Automation Conference*, pages 77–82. ACM, 2012. (Cité en pages 13 et 42.)

[3] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail : On critical paths and clock faults. In *CARDIS*, volume 10, pages 182–193. Springer, 2010. (Cité en page 14.)

[4] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley Boston, 1986. (Cité en page 28.)

[5] Dorothy M Andrews. Using executable assertions for testing and fault tolerance. In *9th Fault-Tolerance Computing Symposium, Madison, Wisconsin, USA*, 1979. (Cité en page 24.)

[6] Jean Arlat, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Jean-Claude Laprie, and David Powell. Tolérance aux fautes. (Cité en page 21.)

[7] ARM Info Center. Application Binary Interface for the ARM Architecture v2.10. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0036b/index.html`, 2017. (Cité en page 39.)

[8] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and J-P Seifert. Fault attacks on RSA with CRT : Concrete results and practical countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 260–275. Springer, 2002. (Cité en page 14.)

[9] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pages 105–114. IEEE, 2011. (Cité en page 16.)

[10] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2) :370–382, 2006. (Cité en pages 14 et 20.)

[11] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java card operand stack : fault attacks, combined attacks and countermeasures. In *International Conference on Smart Card Research and Advanced Applications*, pages 297–313. Springer, 2011. (Cité en page 17.)

[12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault Injection Attacks on Cryptographic Devices : Theory, Practice, and Countermeasures. *Proceedings of the IEEE*, 100(11) :3056–3076, 2012. (Cité en page 16.)

[13] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented aes : effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, page 7. ACM, 2010. (Cité en pages 21, 22, 24, 25 et 147.)

[14] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A First Step Towards Automatic Application of Power Analysis Countermeasures. In *Proceedings of the 48th Design Automation Conference*, pages 230–235. ACM, 2011. (Cité en page 42.)

[15] Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. Automatic application of power analysis countermeasures. *IEEE Transactions on Computers*, 64(2) :329–341, 2015. (Cité en page 42.)

[16] Ali Galip Bayrak, Nikola Velickovic, Paolo Ienne, and Wayne Burleson. An architecture-independent instruction shuffler to protect against side-channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4) :20, 2012. (Cité en page 12.)

[17] Nick Benton. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25. ACM, 2004. (Cité en page 40.)

[18] Pascal Berthomé, Karine Heydemann, Xavier Kauffmann-Tourkestansky, and Jean-Francois Lalande. High level model of control flow attacks for smart card functional security. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 224–229. IEEE, 2012. (Cité en pages 16 et 17.)

[19] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. *Advances in Cryptology—CRYPTO'97*, pages 513–525, 1997. (Cité en page 19.)

[20] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of aes. In *International Workshop on Selected Areas in Cryptography*, pages 69–83. Springer, 2004. (Cité en page 13.)

[21] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 37–51. Springer, 1997. (Cité en pages 14 et 19.)

[22] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In *International Conference on Smart Card Research and Advanced Applications*, pages 283–296. Springer, 2011. (Cité en pages 16 et 17.)

[23] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. Laser profiling for the back-side fault attacks : with a practical laser skip instruction attack on aes.

In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, pages 99–103. ACM, 2015. (Cité en page 19.)

[24] Drew Campbell, Jason Harper, Vinodhkumar Natham, Funian Xiao, and Raji Sundararajan. A compact high voltage nanosecond pulse generator. *Proc. ESA AME*, pages 1–12, 2008. (Cité en page 15.)

[25] Rafael Boix Carpi, Stjepan Picek, Lejla Batina, Federico Menarini, Domagoj Jakobovic, and Marin Golub. Glitch it if you can : parameter search strategies for successful fault injection. In *International Conference on Smart Card Research and Advanced Applications*, pages 236–252. Springer, 2013. (Cité en page 14.)

[26] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 181–194. Springer, 2007. (Cité en page 19.)

[27] Corbet, Jonathan. Fun with NULL pointers, part 1. `https://lwn.net/Articles/342330/`, 2009. (Cité en page 39.)

[28] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009. (Cité en page 37.)

[29] Jean-Sébastien Coron and Louis Goubin. On boolean and arithmetic masking against differential power analysis. In *Cryptographic Hardware and Embedded Systems—CHES 2000*, pages 1–14. Springer, 2000. (Cité en page 13.)

[30] Thomas Coudray, Arnaud Fontaine, and Pierre Chifflier. Picon : Control flow integrity on llvm ir. 2015. (Cité en page 43.)

[31] Damien Couroussé, Thierno Barry, Bruno Robisson, Philippe Jaillon, Olivier Potin, and Jean-Louis Lanet. Runtime code polymorphism as a protection against side channel attacks. In *IFIP International Conference on Information Security Theory and Practice*, pages 136–152. Springer, 2016. (Cité en pages 13 et 42.)

[32] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard : automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998. (Cité en page 19.)

[33] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *NDSS*, pages 8–11, 2015. (Cité en page 43.)

[34] CWE. CWE-14 : Compiler Removal of Code to Clear Buffers. `https://cwe.mitre.org/data/definitions/14.html`, 2017. (Cité en page 40.)

[35] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4) :451–490, 1991. (Cité en page 35.)

[36] Jean-Luc Danger, Sylvain Guilley, Shivam Bhasin, and Maxime Nassar. Overview of dual rail with precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors. In *Signals, Circuits and Systems (SCS), 2009 3rd International Conference on*, pages 1–8. IEEE, 2009. (Cité en page 13.)

[37] Jean-Luc Danger, Sylvain Guilley, Thibault Porteboeuf, Florian Praden, and Michaël Timbert. Hcode : Hardware-enhanced real-time cfi. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, PPREW-4, pages 6 :1–6 :11, New York, NY, USA, 2014. ACM. (Cité en page 24.)

[38] Ronald De Keulenaer, Jonas Maebe, Koen De Bosschere, and Bjorn De Sutter. Link-time smart card code hardening. *International Journal of Information Security*, 15(2) :111–130, 2016. (Cité en pages 22 et 25.)

[39] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, Philippe Orsatelli, Philippe Maurine, and Assia Tria. Injection of transient faults using electromagnetic pulses-practical results on a cryptographic system. *IACR Cryptology EPrint Archive*, 2012 :123, 2012. (Cité en page 15.)

[40] Dicofr.com. Dictionnaire de l'informatique et d'internet. `http://www.dicofr.com/cgi-bin/n.pl/dicofr/definition/20010101002601`, 2014. (Cité en page 59.)

[41] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. From code review to fault injection attacks : Filling the gap using fault model inference. In *International Conference on Smart Card Research and Advanced Applications*, pages 107–124. Springer, 2015. (Cité en page 71.)

[42] Jean-Max Dutertre, Stephan De Castro, Alexandre Sarafianos, Noémie Boher, Bruno Rouzeyre, Mathieu Lisart, Joel Damiens, Philippe Candelier, Marie-Lise Flottes, and Giorgio Di Natale. Laser attacks on integrated circuits : from CMOS to FD-SOI. In *Design & Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On*, pages 1–6. IEEE, 2014. (Cité en page 15.)

[43] Jean-Max Dutertre, Jacques JA Fournier, Amir-Pasha Mirbaha, David Naccache, Jean-Baptiste Rigaud, Bruno Robisson, and Assia Tria. Review of fault injection mechanisms and consequences on countermeasures design. In *Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2011 6th International Conference on*, pages 1–6. IEEE, 2011. (Cité en page 20.)

[44] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 255–264. ACM, 2008. (Cité en page 41.)

[45] Hassan Eldib and Chao Wang. Synthesis of Masking Countermeasures Against Side Channel Attacks. In *International Conference on Computer Aided Verification*, pages 114–130. Springer, 2014. (Cité en page 42.)

[46] Aydin Ersoz, DM Andrews, and EJ McCluskey. The watchdog task : Concurrent error detection using assertions. 1985. (Cité en page 24.)

[47] EUROSMART. Secure element shipments from 2010 to 2016 and 2017 forecasts. http://www.eurosmart.com/, 2017. (Cité en page 4.)

[48] Pierre-Alain Fouque, Delphine Leresteux, and Frédéric Valette. Using faults for buffer overflow effects. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1638–1639. ACM, 2012. (Cité en pages 17 et 19.)

[49] Christophe Giraud. DFA on AES. In *International Conference on Advanced Encryption Standard*, pages 27–41. Springer, 2004. (Cité en page 19.)

[50] Google. The Go Projet - The Go Programming Language. https://golang.org/project/, 2017. (Cité en page 32.)

[51] Sylvain Guilley, Laurent Sauvage, Jean-Luc Danger, Tarik Graba, and Yves Mathieu. Evaluation of power-constant dual-rail logic as a protection of cryptographic applications in fpgas. In *Secure System Integration and Reliability Improvement, 2008. SSIRI'08. Second International Conference on*, pages 16–23. IEEE, 2008. (Cité en page 13.)

[52] Sylvain Guilley, Laurent Sauvage, Florent Flament, Vinh-Nga Vong, Philippe Hoogvorst, and Renaud Pacalet. Evaluation of power constant dual-rail logics countermeasures against dpa with design time security metrics. *IEEE Transactions on Computers*, 59(9) :1250–1263, 2010. (Cité en page 13.)

[53] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench : A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001. (Cité en pages 69, 80, 82 et 151.)

[54] Philippe Hoogvorst, Guillaume Duc, and Jean-Luc Danger. Software implementation of dual-rail representation. *COSADE, Darmstadt, Germany*, pages 24–25, 2011. (Cité en page 13.)

[55] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013. (Cité en page 15.)

[56] Paul Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996. (Cité en page 12.)

[57] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in cryptology—CRYPTO'99*, pages 789–789. Springer, 1999. (Cité en page 12.)

[58] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card C codes. In *ESORICS'14*, pages 200–218. Springer, 2014. (Cité en pages 8, 22, 24, 25 et 86.)

[59] Ronan Lashermes, Guillaume Reymond, Jean-Max Dutertre, Jacques Fournier, Bruno Robisson, and Assia Tria. A DFA on AES based on the Entropy of Error Distributions. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*, pages 34–43. IEEE, 2012. (Cité en page 19.)

[60] Chris Lattner and Vikram Adve. LLVM : A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization : feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004. (Cité en pages 32 et 35.)

[61] Lattner, Chris. What Every C Programmer Should Know About Undefined Behavior. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html, 2011. (Cité en page 39.)

[62] Xavier Leroy. Formal Certification of a Compiler Back-end or : Programming a Compiler With a Proof Assistant. In *ACM SIGPLAN Notices*, volume 41, pages 42–54. ACM, 2006. (Cité en page 40.)

[63] LLVM. LLVM Related Publications. http://llvm.org/pubs/, 2017. (Cité en page 32.)

[64] LLVM. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html, 2017. (Cité en page 33.)

[65] LLVM. The LLVM Target-Independent Code Generator. http://llvm.org/docs/CodeGenerator.html, 2017. (Cité en page 61.)

[66] David J. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, 31(7) :681–685, 1982. (Cité en page 24.)

[67] Aamer Mahmood and Edward J McCluskey. *Watchdog processors : error coverage and overhead*. Center for Reliable Computing, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, 1984. (Cité en page 24.)

[68] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks : Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008. (Cité en page 12.)

[69] Trevor Martin. *The Insider's Guide To The STM32 ARM Based Microcontroller*. Hitex (UK) Ltd., 2008. (Cité en pages 69, 82, 100 et 115.)

[70] David May, Henk L Muller, and Nigel P Smart. Non-deterministic processors. In *ACISP*, volume 1, pages 115–129. Springer, 2001. (Cité en page 12.)

[71] Phoronix Media. LLVM Clang 4.0 vs. GCC 7 January 2017 Compiler Benchmarks. http://openbenchmarking.org/result/1701270-PTS-CLANG4GC49, 2017. (Cité en page 33.)

[72] Mentor Graphics. Sourcery CodeBench. https://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview, 2017. (Cité en pages 48 et 69.)

[73] Mohindra, Dhruv. MSC06-J. Do not modify the underlying collection when an iteration is in progress. https://www.securecoding.cert.org/confluence/display/java/MSC06-J.+Do+not+modify+the+underlying+collection+when+an+iteration+is+in+progress, 2017. (Cité en page 67.)

[74] Amir Moradi, Mohammad T Manzuri Shalmani, and Mahmoud Salmasizadeh. A Generalized Method of Differential Fault Attack Against AES Cryptosystem. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 91–100. Springer, 2006. (Cité en page 19.)

[75] Nicolas Moro. *Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2014. (Cité en pages 66, 69, 72 et 104.)

[76] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3) :145–156, 2014. (Cité en pages 7, 21, 22, 23, 25, 58, 69, 70, 72, 119 et 151.)

[77] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler Assisted Masking. *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 58–75, 2012. (Cité en page 42.)

[78] Minh Huu Nguyen. *Sécurisation de processeurs vis-à-vis des attaques par faute et par analyse de la consommation*. PhD thesis, Paris 6, 2011. (Cité en page 21.)

[79] CORPORATE NIST. The digital signature standard. *Communications of the ACM*, 35(7) :36–40, 1992. (Cité en page 4.)

[80] Sébastien Ordas, Ludovic Guillaume-Sage, Karim Tobich, J-M Dutertre, and Philippe Maurine. Evidence of a larger EM-induced fault model. In *International Conference on Smart Card Research and Advanced Applications*, pages 245–259. Springer, 2014. (Cité en page 15.)

[81] Martin Otto. *Fault attacks and countermeasures*. PhD thesis, University of Paderborn, 2005. (Cité en page 18.)

[82] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the AES and khazad. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 77–88. Springer, 2003. (Cité en page 19.)

[83] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5) :895–913, 1999. (Cité en page 31.)

[84] F Poucheret, L Chusseau, B Robisson, and P Maurine. Local electromagnetic coupling with CMOS integrated circuits. In *Electromagnetic Compatibility of Integrated Circuits (EMC Compo), 2011 8th Workshop on*, pages 137–141. IEEE, 2011. (Cité en page 15.)

[85] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA) : Measures and counter-measures for smart cards. *Smart Card Programming and Security*, pages 200–210, 2001. (Cité en pages 12 et 15.)

[86] Francesco Regazzoni, Luca Breveglieri, Paolo Ienne, and Israel Koren. Interaction between fault attack countermeasures and the resistance against power analysis attacks. In *Fault Analysis in Cryptography*, pages 257–272. Springer, 2012. (Cité en page 6.)

[87] Francesco Regazzoni, Thomas Eisenbarth, Luca Breveglieri, Paolo Ienne, and Israel Koren. Can knowledge regarding the presence of countermeasures against fault attacks simplify power attacks on cryptographic devices ? In *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, pages 202–210. IEEE, 2008. (Cité en page 6.)

[88] Regehr, John. A Guide to Undefined Behavior in C and C++, Part 1. `https://blog.regehr.org/archives/213`, 2010. (Cité en page 39.)

[89] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. SWIFT : Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005. (Cité en page 22.)

[90] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 413–427, 2010. (Cité en page 13.)

[91] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of ARMv7-M architectures. In *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*, pages 62–67. IEEE, 2015. (Cité en pages 17, 19, 25, 76 et 119.)

[92] Bruno Robisson and Pascal Manet. Differential Behavioral Analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–426. Springer, 2007. (Cité en page 19.)

[93] Jörn-Marc Schmidt and Michael Hutter. *Optical and em fault-attacks on CRT-based RSA : Concrete results*. na, 2007. (Cité en pages 15 et 19.)

[94] SERTIF. Simulation for the Evaluation of Robustness of embedded Applications against Fault injection – ANR. `http://sertif-projet.forge.imag.fr`, 2016. (Cité en pages 69, 95 et 96.)

[95] Adi Shamir and Eran Tromer. Acoustic cryptanalysis. *presentation available from http ://www. wisdom. weizmann. ac. il/ tromer*, 2004. (Cité en page 12.)

[96] Sergei Skorobogatov. Local heating attacks on flash memory devices. In *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*, pages 1–6. IEEE, 2009. (Cité en page 14.)

[97] NIST-FIPS Standard. Announcing the advanced encryption standard (AES). *Federal Information Processing Standards Publication*, 197 :1–51, 2001. (Cité en page 4.)

[98] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8) :761–763, 1984. (Cité en page 39.)

[99] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security*, volume 26, pages 27–40, 2014. (Cité en page 43.)

[100] Elena Trichina and Roman Korkikyan. Multi Fault Laser Attacks on Protected CRT-RSA. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 75–86. IEEE, 2010. (Cité en pages 16, 17 et 19.)

[101] Ingrid Verbauwhede, Dusko Karaklajic, and Jorn-Marc Schmidt. The Fault Attack Jungle - A Classification Model to Guide You. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pages 3–8. IEEE, 2011. (Cité en page 16.)

[102] Wikipedia. George Hotz - Jailbreak iOS devices. `https://en.wikipedia.org/wiki/George_Hotz`, 2017. (Cité en page 5.)

[103] Wikipédia. Idempotence. `https://fr.wikipedia.org/wiki/Idempotence`, 2017. (Cité en page 23.)

[104] Sung-Ming Yen and Marc Joye. Checking Before Output May not be Enough Against Fault-based Cryptanalysis. *IEEE Transactions on computers*, 49(9) :967–970, 2000. (Cité en page 19.)

[105] Joseph Yiu. *The definitive guide to the ARM Cortex-M3*. Newnes, 2009. (Cité en pages 31, 69 et 111.)

[106] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software Fault Resistance is Futile : Effective Single-Glitch Attacks. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on*, pages 47–58. IEEE, 2016. (Cité en pages 25, 76 et 119.)

[107] Loic Zussa, Amine Dehbaoui, Karim Tobich, Jean-Max Dutertre, Philippe Maurine, Ludovic Guillaume-Sage, Jessy Clediere, and Assia Tria. Efficiency of a glitch detector against electromagnetic fault injection. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014. (Cité en page 20.)

[108] Loic Zussa, Jean-Max Dutertre, Jessy Clédiere, Bruno Robisson, Assia Tria, et al. Investigation of timing constraints violation as a fault injection means. In *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*, 2012. (Cité en page 14.)

Table des figures

2.1 Illustration de l'ajout du bruit dans un programme pour modifier les observables du circuit qui l'exécute . . . 13
2.2 Équipement d'injection laser du la plate-forme Micro Packs du centre microélectronique de Provence . . . 15
2.3 Illustration du modèle de fautes saut d'instructions `skip{1, 2}` . . . 16
2.4 Illustration du modèle de fautes `skip_block{64 bits, 1}` . . . 18
2.5 Redondance spatiale d'exécution . . . 20
2.6 Redondance temporelle d'exécution . . . 21
2.7 Procédé de duplication d'instructions selon Barenghi et al. [13] . . . 22
2.8 Schéma de tolérance . . . 22
2.9 Duplication d'une instruction idempotente . . . 23
2.10 Duplication d'une instruction non idempotente . . . 23

3.1 Structure interne d'un compilateur . . . 29
3.2 Transformation d'un programme en arbre syntaxique (AST) . . . 30
3.3 Aperçu de la structure du compilateur LLVM . . . 32
3.4 illustration de la passe d'optimisation `-loop-unswitch` . . . 34
3.5 illustration de la passe d'optimisation `-indvars` . . . 34
3.6 illustration de la passe d'optimisation `-instcombine` . . . 34
3.7 Illustration de la représentation intermédiaire LLVM-IR . . . 35
3.8 Représentation d'un programme en graphe de flot de contrôle . . . 36
3.9 Représentation d'un programme par un DAG . . . 37
3.10 Représentation de la fonction `foo` en `MachineInstr` juste après l'étape sélection d'instructions . . . 38
3.11 Représentation de la fonction `foo` en `assembleur` . . . 39

4.1 Aperçu du simulateur de faute . . . 48
4.2 Processus de simulation . . . 49
4.3 Architecture interne du simulateur . . . 52
4.4 Extrait du code assembleur de la fonction `AES_Encrypt`. Le code est sécurisé avec le schéma de tolérance au saut d'instructions . . . 53
4.5 Manuel d'utilisation du simulateur . . . 56

5.1 Structure simplifiée du compilateur modifié. Les boites grises et les boites noires représentent respectivement les passes modifiées et celles ajoutées . . . 60
5.2 Extrait de l'interface `class Instruction` proposée par LLVM pour définir une instruction machine . . . 61
5.3 Illustration du processus de sélection d'instructions . . . 63

5.4 Extrait de la passe `Thumb2SizeReduce` responsable de la réduction des opérandes d'une instruction . . . 63
5.5 Intervalles de vie des variables a, b et c à l'entrée de l'ALU, pour l'opération $a = b + c$. . . . 64
5.6 Ordonnancement des instructions dupliquées . . . 68
5.7 Impact du l'ordonnancement des instructions dupliquées sur le temps d'exécution . . . 68

6.1 Illustration du modèle `protect_skip{2, 2}` . . . 77
6.2 Illustration du modèle `protect_skip_block{64 bits, 2}` . . . 78
6.3 Structure simplifiée du compilateur, montrant la passe *séparation d'instructions* et faisant abstractions des passes déjà présentées dans le chapitre 5 . . . 79
6.4 Impact sur le temps d'exécution et la taille du code de l'AES protégé avec le modèle `protect_skip{A, B}` . . . 82
6.5 Impact sur le temps d'exécution et la taille du code de l'AES protégé avec le modèle `protect_skip_block{size, M}` . . . 83

7.1 Représentation d'un bloc de base . . . 86
7.2 CFG initial . . . 87
7.3 Application du schéma CFI sur un bloc de base `BB#1` ne possédant qu'un seul prédécesseur . . . 89
7.4 Application du schéma CFI sur un bloc de base `BB#2` possédant plusieurs prédécesseurs . . . 90
7.5 Application du schéma CFI pour le cas d'un bloc de base `BB#0` possédant plusieurs successeurs . . . 90
7.6 Structure simplifiée du compilateur modifié. Les boites grises et les boites noires représentent respectivement les passes modifiées et celles ajoutées. . . . 91
7.7 CFG non sécurisé de la fonction `VerifyPIN` (listing 7.1) . . . 96
7.8 CFG de la fonction `VerifyPIN`, sécurisé avec le schéma CFI sans combinaison avec le schéma de tolérance au saut d'instructions . . . 98
7.9 CFG de la fonction `VerifyPIN`, sécurisé avec le schéma CFI combiné avec le modèle de protection `protect_skip{1, 1}`. . . . 99

8.1 Alignement des instructions dans des blocs de taille 32 bits. `I16` et `I32` représentent respectivement des instructions de taille 16 bits et 32 bits . . . 105
8.2 Cas d'un seul prédécesseur et un seul successeur . . . 107
8.3 Cas de plusieurs prédécesseurs . . . 108
8.4 Cas d'un prédécesseur commun . . . 108
8.5 Solution pour le cas d'un prédécesseur commun . . . 110
8.6 Cas d'un branchement conditionnel . . . 111
8.7 Découpage du compteur 32 bits . . . 111

8.8 Solution pour le cas d'un branchement conditionnel (version protégée de la figure 8.6) 113

A.1 Ordre d'exécution des passes du schéma CCFI 123
A.2 CFG de la fonction `test` 124
A.3 CFG étendu de la fonction `test` 125
A.4 CFG de la fonction `test` après la passe `Legalize CFG` 126
A.5 CFG de la fonction `test` après la passe `Insert CheckMBB` 128
A.6 CFG de la fonction `test` après la passe `Insert Step Counters` . . . 130
A.7 CFG de la fonction `test` après la passe `Initialize Counters` 131
A.8 CFG de la fonction `test` après la passe `Compute Preview Counters` 133

Liste des tableaux

3.1 Schémas de contre-mesure et les passes d'optimisations susceptibles de les altérer 40

5.1 Surcoût en taille code (en octets) et en temps d'exécution (en cycles d'horloge) pour chaque implémentation. Les deux dernières colonnes reportent les résultats présentés par Moro et al. [76] 70
5.2 Comparaison (pour des niveaux de sécurité équivalents) des approches *exhaustive* et *rétroactive* sur les surcoûts (en taille/temps d'exécution) engendrés par la duplication d'instructions. 72

6.1 Résultats de simulation du modèle de fautes `SKIP{N, M}` sur l'AES 32 bits[53] protégé avec le modèle de protection `protect_skip{A, B}` 80

7.1 Résultat de l'évaluation de performances du schéma CFI combiné au schéma de tolérance au saut d'instructions (CFI + `protect_skip{A, B}`) 100
7.2 Résultat de la simulation des modèles de fautes `REPL_BR` et `REPL_BR_SKIP{N, M}` sur le programme verify-PIN protégé par les modèles de protections CFI et CFI + `protect_skip{A, B}` 101

8.1 Découpage du registre APSR 112
8.2 Résultats des routines `get_flags` et `get_mask` en fonction de `cond` et du bloc de base destination 112
8.3 Résultat de l'évaluation de performances du schéma CCFI. Le temps d'exécution est mesuré en nombre de cycle d'horloge et la taille du code en octets. 116
8.4 Résultat de la simulation des modèles de fautes sur le programme Verify-PIN protégé par le schéma CCFI 117

École National Supérieur des Mines
de Saint-Étienne

NNT : 2017LYSEM037

Thierno BARRY

SECURING SOFTWARE AGAINST FAULT ATTACKS AT COMPILE TIME

Specialty : Microelectronic

Keywords : fault attacks, countermeasures, fault tolerance, CFI, combined countermeasures, compilation, LLVM, ARM Cortex-M3

Abstract :

Embedded systems are increasingly present in our daily lives (e.g. credit cards, smartphones and biometric passports). Given the sensitivity of the data they handle, the safety of these systems has become a major concern for industry and state organizations. It is established that a fault injection in an embedded system can compromise the security of the data it contains, for example obtaining a secret key or bypassing an authentication mechanism.

The goal of this thesis is the automatic generation of software protections against fault injection attacks on embedded systems. The source and binary approaches consist in inserting the protections respectively in the source and binary code of the application ; this thesis explores the use of a compilation approach that consists in integrating the protections in the compiler.

We propose an LLVM-based compiler allowing the automated application of several protection schemes during compilation : (1) a tolerance scheme against instruction-skip, (2) a control flow integrity scheme (CFI) to ensure the validity of the followed execution path and (3) and a scheme that combines CFI and instruction integrity, guaranteeing both the validity of followed execution path but also that no instruction along this path has been skipped or altered. Our approach based on a modified compiler allows making code protection and code optimization coexist, thus enabling the generation of a secure and optimized binary code in terms of memory footprint and execution time. We developed a fault simulator to validate the robustness of our protection schemes with respect to the considered fault models.

This thesis shows that the compilation approach is a good compromise between the source approach that does not guarantee the integrity of security properties in the final binary code due to optimizations performed by the compiler, and the binary approach that considerably impacts the performance of the secure application due to the various necessary transformations.

École National Supérieur des Mines
de Saint-Étienne

NNT : 2017LYSEM037

Thierno BARRY

SÉCURISATION À LA COMPILATION DE LOGICIELS CONTRE LES ATTAQUES EN FAUTES

Spécialité : Microélectronique

Mots-clés : attaques en fautes, contre-mesures, tolérance aux fautes, CFI, combinaison de contre-mesures, compilation, LLVM, ARM Cortex-M3

Résumé :

Les systèmes embarqués sont de plus en plus présents dans notre quotidien (e.g. cartes de crédits, smartphones, passeports biométriques). Compte tenu de la sensibilité des données qu'ils manipulent, la sécurité de ces systèmes est aujourd'hui une préoccupation majeure pour les industrielles et les organismes étatiques. Il est établi qu'une injection de faute dans un système embarqué permet de compromettre la sécurité des données qu'il contient, par exemple obtenir une clé secrète ou outrepasser un mécanisme d'authentification.

L'objet de cette thèse est la génération automatisée de protections logicielles contre les attaques par injection de fautes sur les systèmes embarqués. Les approches *source* et *binaire* consistent à insérer les protections respectivement dans le code source et binaire de l'application. Cette thèse explore l'utilisation d'une approche compilation consistant à intégrer les protections dans le *compilateur*.

Nous proposons un compilateur basé sur LLVM permettant l'application de plusieurs schémas de protection : (1) un schéma de tolérance aux sauts d'instructions, (2) un schéma d'intégrité de flot de contrôle (CFI) permettant de garantir la validité du chemin d'exécution suivi et (3) un schéma combinant CFI et intégrité des instructions, garantissant à la fois la validité du chemin d'exécution suivi et aussi qu'aucune instruction le long de ce chemin n'a été sautée ou altérée. Notre approche, basée sur un compilateur modifié, permet de faire coexister *protection de code* et *optimisation de code*, permettant ainsi de générer un code binaire sécurisé et optimisé en termes d'empreinte mémoire et de temps d'exécution. Nous avons développé un simulateur de faute afin de valider la robustesse de nos implémentations vis-à-vis des modèles de fautes considérés.

Cette thèse montre que l'approche compilation est un bon compromis entre l'approche *source* qui ne garantit pas la conservation des propriétés de sécurité dans le code binaire dû aux optimisations réalisées par le compilateur et l'approche *binaire* qui impacte considérablement les performances de l'application sécurisée dû aux différentes transformations nécessaires.