Sécurisation systématique d'applications embarquées contre les attaques physiques

Julien Proy

▶ **To cite this version:**

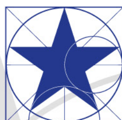
Julien Proy. Sécurisation systématique d'applications embarquées contre les attaques physiques. Cryptographie et sécurité [cs.CR]. Université Paris sciences et lettres, 2019. Français. NNT : 2019PSLEE048 . tel-03015143

HAL Id: tel-03015143
https://theses.hal.science/tel-03015143v1

Submitted on 19 Nov 2020

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.



PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure de Paris

Sécurisation systématique d'applications embarquées contre les attaques physiques

Soutenue par
Julien PROY
Le 17 Juin 2019

École doctorale nº386
Sciences Mathématiques

Spécialité
Informatique

Composition du jury :

Jean-Max DUTERTRE Professeur, EMSE	*Président*
Louis GOUBIN Professeur, UVSQ	*Rapporteur*
Erven ROHOU Directeur de recherche, INRIA	*Rapporteur*
Damien COUROUSSÉ Ingénieur de recherche, CEA-LIST	*Examinateur*
Alexandre BERZATI Ingénieur de recherche, INVIA	*Encadrant*
Karine HEYDEMANN Maître de conférences, Sorbonne Université	*Co-directrice*
Albert COHEN Directeur de recherche, Google et ENS	*Directeur de thèse*

ENS | PSL

Remerciements

Commencer une thèse pousse à en lire (beaucoup) d'autres. Plutôt que de lire un document anonyme, le chapitre des remerciements est souvent l'occasion de mieux connaître l'auteur et l'état d'esprit dans lequel le travail ainsi résumé s'est déroulé. Pour ma part, j'estime avoir été particulièrement chanceux de l'entourage que j'ai pu avoir pendant toute la thèse. Je tenais donc à remercier tous ceux qui, de près ou de loin, ont contribué à faire que cette thèse s'est très bien passée et s'achève tout aussi bien. Je souhaite également faire apparaitre ici tous ceux qui m'ont poussé à prendre des décisions qui m'ont amenées à m'engager dans une thèse, et dans celle-là en particulier. Remontons donc la ligne du temps.

Je souhaite tout d'abord remercier Messieurs Erven Rohou et Louis Goubin d'avoir accepté d'être rapporteurs de cette thèse et d'avoir eu le courage de lire avec attention ces 144 pages. Je tiens à remercier Damien Couroussé pour les conseils et discussions que l'on a pu avoir lors de nos différentes rencontres à Cadarache et Gardanne, et d'avoir accepté de faire partie de mon jury de thèse. Je tiens également à remercier Jean-Max Dutertre pour m'avoir permis de soutenir cette thèse à Gardanne en plus d'avoir accepté de faire aussi partie de mon jury de thèse.

Je souhaite remercier Albert Cohen pour avoir accepté de diriger cette thèse et d'avoir eu l'excellente idée de proposer Karine Heydemann comme co-directrice. Merci pour cette direction sous le signe de bons moments : lors de nos réunions très souvent devant un tableau à partir dans des discussions enflammées, qui se sont toujours soldées par des idées nouvelles et des conseils précieux, aussi lors des troisièmes mi-temps, d'usage et devenues un rituel. Les repas et discussions autour de verres bien choisis ont au moins autant contribué à la richesse de nos échanges. Merci également pour ton exigence sur la langue de Shakespeare, ta volonté et le temps dévoué à réécrire certaines parties (pour ne pas dire toutes) de nos articles, leur donnant d'un coup une forme bien plus claire voire étincelante.

Je souhaite remercier Karine Heydemann pour avoir aussi accepté de diriger cette thèse, et dont le tempérament et la complicité ont fait qu'on est devenus amis. Merci pour cette extraordinaire implication dans ces travaux, tes innombrables conseils donnés et ta patience remarquable lors des périodes de rédaction qui sont, il faut l'admettre, loin d'être mon point

fort. Merci pour tous les bons moments passés à Paris comme à Aix-en-Provence pendant nos séances de travail d'une intensité peu commune méritant de bonnes continuations autour de verres de vin délicatement choisis.

Je souhaite remercier Alexandre Berzati qui, après mon stage à INVIA, a fait en sorte que cette thèse ait lieu. Merci pour tout le temps consacré que ce soit pour la thèse comme pour le travail pour INVIA, d'avoir su me laisser le temps de chercher par moi-même tout en donnant le coup de pouce quand nécessaire. Merci pour tes conseils et avis toujours éclairés, surtout éclairants. Ils m'ont permis de progresser autant sur le plan académique que sur le plan industriel et m'ont appris à prendre du recul.

Merci à Robert Leydier et Alain Pomet pour m'avoir fait confiance et accueilli au sein de leur société INVIA. Grâce à votre enthousiasme, il règne une ambiance de travail à la fois efficace, enrichissante et souvent festive. Sans avoir connu INVIA, cette thèse n'aurait jamais eu lieu, je tenais donc à remercier également Sylvain Duquesne pour la confiance qu'il m'a accordé et la mise en relation avec INVIA pendant mon Master de Cryptographie.

Je voulais remercier aussi les anciens et actuels thésards avec qui j'ai pu échanger, qui ont pu me donner des conseils ou m'inspirer. Je pense en particulier à Thierno Barry, dont le sujet de thèse était proche du mien, qui m'a donné beaucoup de conseils et à Fabien Majeric qui m'a fait découvrir le banc d'attaques électromagnétiques et a participé au deuxième article publié. Merci également à Nicolas Moro, Louis Dureuil, Benjamin Lac, Jean-Baptiste Bréjon et Son Tuan Vu.

Un grand merci à tous mes collègues d'INVIA qui m'ont fait découvrir l'univers de l'embarqué, du design, du soft, du hard ou de la compilation, qui vont devoir me supporter encore pour une durée indéterminée et avec qui je continue à passer du bon temps au boulot comme à l'extérieur. Une pensée toute particulière à Bettina R., Yannick C., Vincent M., Thomas D., Emmanuel P., Sophie B., Sylvère T., Jorge P., Nico V., Julien G., Thierry F., Loic B., Alaa D.N. et XiangJun F.

Cette thèse est clairement l'aboutissement d'une période de 5 ans de ma vie qui a commencé par ma reprise d'études avec le Master de Cryptographie. Je ne me serais jamais lancé dans cette aventure sans Martin. Je ne pourrais jamais assez te remercier de m'avoir motivé. Allez Hum ! Depuis nos premières discussions interminables à base de fonction ζ, d'arctangentes ou encore de connexité par arcs, c'est grâce à toi que ma flamme des Maths est restée intacte et qu'elle m'a poussé à en arriver là. Je souhaite donc remercier celle qui a allumé cette flamme des Maths en moi en premier lieu. Merci Marie-Laure Ravard d'avoir été une super prof de Maths et de m'avoir fait découvrir cette véritable passion.

Merci à Martin, Pierre et Yoann d'être depuis de nombreuses années de super potes, toujours là dans les bons et mauvais moments même si on n'est pas forcément des spécialistes de

la communication téléphonique, ce sont des personnes sans qui la vie serait quand même vachement moins fun.

Merci à tous les autres, les aixois Eddy et Stéphanie, les bretons Flo, Nolwenn, Lucie, Laëtitia, José, Héloïse, Cédric, JC, Benoit, les ex-colocs Sylvie, Yang, Marine, Philippe, Natacha, Valérian, Koubi, Samy, et bien sûr ma petite Manon.

Il reste la famille. Sans elle, il est évident que je n'en serais pas là aujourd'hui. Je voudrais d'abord remercier Henri qui a toujours été soucieux de ma réussite, a pris soin de moi depuis mon plus jeune âge et qui m'a appris à dire merci quand j'étais petit. Un grand merci à Marie, ma deuxième maman, qui a toujours été présente et attentive, au point de stresser plus que moi dans les moments délicats. Un grand merci aussi à Maman, ma...première maman, qui a toujours tout fait pour que je sois heureux et a toujours cru en moi et m'a laissé suivre ma voie même lorsque cette dernière fut sinueuse. Enfin, un énorme merci à Jean-Marie, JM, le meilleur frère que l'on puisse avoir, qui a toujours été mon idole et ma référence.

Finalement, un infini merci à 王景春 (*Wang Jing Chun*), Spring, qui a partagé avec moi ces trois années et quelques mois particulièrement intenses et parfois compliqués que ce soit à Aix-en-Provence ou depuis la Chine. Merci en particulier d'avoir compris dans les moindres détails qui je suis, ainsi que pour la patience que tu as eue. Merci pour la sagesse et l'amour que tu m'as apportés sans lesquels j'aurais eu beaucoup de mal à venir à bout de ces trois années.

Résumé

La sécurité des systèmes embarqués contenant des données sensibles est un enjeu crucial. La disponibilité de ces objets en fait une cible privilégiée pour les attaques physiques, nécessitant l'ajout de protections matérielles et logicielles. La recherche de réduction des coûts de développement pousse les industriels à opter pour du déploiement automatique de protections. L'objet de la thèse consiste à étudier l'intégration de contre-mesures logicielles contre les attaques par faute dans les outils de développement, en particulier dans le compilateur, afin d'automatiser l'application de contre-mesures variées. Pour cela, nous proposons deux schémas de protection génériques et automatiquement déployables contre ces attaques : un dédié à la sécurisation des boucles et le deuxième à la sécurisation du graphe d'appel. Ces schémas spécifiques, intégrés dans un même compilateur (LLVM) permettent la sécurisation de parties sensibles et choisies du code limitant ainsi leur surcoût en performances. Les fautes exploitables variant d'un composant à l'autre, nous proposons également une caractérisation des effets des fautes au niveau du jeu d'instructions sur une plateforme intégrant un processeur superscalaire typique des téléphones mobiles. Ces travaux montrent la nécessité d'étudier les injections de faute sur des plateformes complexes, de concevoir de nouveaux schémas de protection adaptés, et de continuer à intégrer dans un même compilateur plus de schémas de sécurisation.

Abstract

The security of embedded systems containing sensitive data has become a main concern. These widely deployed devices are subject to physcial attacks, requiring protections both in hardware and software. The race for higher productivity and shorter time to market in the deployment of secure systems pushes for automatic solutions. This thesis studies the integration of software countermeasures against fault attacks in development tools, with a special focus on the compiler. The goal is to enable the automatic application, at compilation time, of a wide range of countermeasures. We propose two protection schemes against these attacks which can be automatically deployed : one scheme dedicated to loop control flow and the second dedicated to the protection of the call graph. These schemes, integrated in the LLVM compiler framework, allow to focus security application on sensitive areas of the targeted code, thus limitating the overhead. Faults that can be exploited are different from a device to another, we thus also provide an ISA-level characterization of fault effects on a superscalar processor representative of mobile phones. This work highlights the need of studying fault effects on more complex platforms, leading to the design of new protection schemes and automating their compilation-time application.

Table des matières

Remerciements **i**

Résumé **v**

Abstract **vii**

Acronymes **xv**

1 Introduction **1**
1.1 Problématique 2
1.2 Contributions 3
1.3 Plan du manuscrit 4

2 Attaques par injection de fautes et contre-mesures **7**
2.1 Introduction 7
2.2 Attaques par injection de faute 9
2.2.1 Vue d'ensemble 9
2.2.2 Moyens d'injection 10
2.2.2.1 Perturbation de la tension d'alimentation 11
2.2.2.2 Perturbation du signal d'horloge 11
2.2.2.3 Température 11
2.2.2.4 Injection électromagnétique 12
2.2.2.5 Rayonnement lumineux 12
2.2.3 Modèles de fautes 13
2.2.3.1 Persistance 13
2.2.3.2 Niveau logique 14
2.2.3.3 Niveau ISA 14
2.2.3.4 Niveau code source 15
2.2.3.5 Vers des architectures plus complexes 15
2.2.4 Exploitation des exécutions corrompues 16
2.3 Protections contre les fautes 17
2.3.1 Principes des contre-mesures 17
2.3.2 Contre-mesures matérielles 18

2.3.3 Contre-mesures logicielles . . . 19
2.3.4 Mise en œuvre des protections logicielles . . . 20
2.4 Sécurisation logicielle automatisée . . . 20
2.4.1 Automatisation au niveau du code source . . . 21
2.4.2 Automatisation au niveau du code binaire . . . 22
2.4.3 Compromis entre les deux approches . . . 23
2.5 Compilation et insertion de protection . . . 23
2.5.1 Compilation . . . 23
2.5.1.1 Architecture d'un compilateur . . . 23
2.5.1.2 Architecture de LLVM . . . 26
2.5.2 Automatisation à la compilation . . . 28
2.5.2.1 Approche *back-end* . . . 29
2.5.2.2 Approche *middle-end* . . . 30
2.6 Conclusion . . . 31

3 Sécurisation automatique des boucles **33**
3.1 Introduction . . . 34
3.2 Motivations . . . 35
3.2.1 Modèle de faute . . . 35
3.2.2 Attaques sur boucles . . . 36
3.2.3 Schémas de protection existants . . . 36
3.2.4 Protection manuelle des boucles et contraintes . . . 37
3.2.5 Vers un algorithme dédié aux boucles . . . 38
3.3 Algorithme de sécurisation des boucles : PLAF . . . 39
3.3.1 Prérequis . . . 39
3.3.2 Vue d'ensemble . . . 40
3.3.3 Détails des algorithmes . . . 42
3.3.3.1 Analyse arrière de dépendance . . . 42
3.3.3.2 Duplication des instructions . . . 45
3.3.3.3 Gestion des opérandes invariants . . . 45
3.3.3.4 Mise à jour du CFG intra-boucle . . . 47
3.3.3.5 Sécurisation des branchements internes . . . 47
3.3.3.6 Mise à jour finale du CFG . . . 47
3.3.3.7 Ordre de traitement des boucles . . . 48
3.3.3.8 Limitations du schéma de duplication . . . 49
3.4 Analyse sécuritaire . . . 49
3.4.1 Saut d'instruction . . . 51
3.4.2 Corruption de registre . . . 51
3.5 Interactions avec le flot de compilation . . . 52
3.5.1 Passes en amont . . . 53
3.5.1.1 Passes nécessaires . . . 53
3.5.1.2 Passes interférantes . . . 53

3.5.1.3 Passes bénéfiques 54
3.5.2 Placement dans le flot de compilation 54
3.5.3 Passes en aval et ajustements nécessaires 55
3.5.3.1 Placement de code 55
3.5.3.2 Allocation de registres 56
3.5.3.3 Factorisation des constantes 57
3.5.3.4 Élimination d'expression commune 57
3.5.3.5 Regroupement d'instructions 57
3.5.4 Conclusion des modifications du back-end 58
3.6 Expérimentations et résultats 58
3.6.1 Implémentation dans LLVM 58
3.6.1.1 Sélection des boucles à sécuriser 58
3.6.1.2 Limitations et avertissements à l'utilisateur 59
3.6.1.3 Visualisation du code généré sur exemples simples 59
3.6.2 Environnement expérimental 60
3.6.2.1 Environnement logiciel 62
3.6.2.2 Environnement matériel 62
3.6.3 Couverture de sécurisation 63
3.6.4 Évaluation fonctionnelle 64
3.6.5 Coût de la contre-mesure 65
3.6.5.1 Impact en performance 65
3.6.5.2 Impact sur la taille de code 66
3.6.5.3 Impact sur le temps de compilation 66
3.6.6 Évaluation sécuritaire de l'algorithme PLAF 67
3.7 Discussion sur une généralisation aux branchements 70
3.8 Conclusion 71

4 Sécurisation partielle du graphe d'appel 73
4.1 Introduction 73
4.2 Motivations 74
4.2.1 Attaques sur le graphe d'appel 75
4.2.2 Schémas de protection existants 76
4.2.3 Modèle d'attaquant 77
4.3 Schéma de suivi de l'arbre d'appel : *CalleeSimo* 78
4.3.1 Principe du schéma 78
4.3.1.1 Modes de sécurisation proposés 78
4.3.1.2 Concept de la sécurisation 79
4.3.1.3 Prérequis 80
4.3.2 Algorithmes principaux 80
4.3.2.1 Analyses et transformations des fonctions appelées 81
4.3.2.2 Analyses et transformations des fonctions appelantes 82

4.3.3 Suivi des fonctions . . . 83
4.3.3.1 Modes de suivi des fonctions . . . 83
4.3.3.2 Mise à jour des variables de suivi . . . 84
4.3.3.3 Cas statique . . . 85
4.3.4 Protection des arguments . . . 86
4.3.4.1 Protection des arguments depuis leur définition . . . 87
4.3.4.2 Protection des arguments jusqu'à leurs utilisations . . . 91
4.3.4.3 Mise à jour des variables de protection des arguments . . . 94
4.3.5 Intégration dans le flot de compilation . . . 94
4.3.6 Limites du schéma de protection . . . 95
4.4 Analyse sécuritaire pour architecture ARM . . . 96
4.4.1 Protection de l'arbre d'appel . . . 96
4.4.2 Protection des arguments . . . 98
4.5 Implémentation et résultats . . . 98
4.5.1 Implémentation dans LLVM . . . 99
4.5.1.1 Sélection des arguments et fonctions sensibles . . . 99
4.5.1.2 Gestion des attributs de variables . . . 99
4.5.2 Environnement expérimental . . . 100
4.5.3 Évaluation fonctionnelle . . . 100
4.5.4 Coût de la contre-mesure . . . 101
4.5.4.1 Impact sur les performances . . . 101
4.5.4.2 Impact sur la taille du code . . . 102
4.6 Discussion sur la complémentarité avec la sécurisation des boucles . . . 103
4.7 Conclusion . . . 104

5 Caractérisation des fautes au niveau ISA **107**
5.1 Introduction . . . 107
5.2 Environnement d'attaque . . . 109
5.2.1 Composant cible des attaques . . . 109
5.2.2 Banc d'attaque et paramétrage . . . 109
5.2.3 Processus expérimental . . . 112
5.2.3.1 Détermination de l'intervalle pour injection . . . 112
5.2.3.2 Instrumentation du code visé . . . 113
5.2.3.3 Processus d'analyse . . . 114
5.3 Première analyse de sensibilité aux fautes . . . 114
5.3.1 Codes cibles . . . 115
5.3.1.1 Boucle de type *memcpy* . . . 115
5.3.1.2 Boucle de type *memcmp* . . . 116
5.3.2 Campagne d'attaque sur les boucles standards . . . 117
5.3.3 Campagne d'attaque sur les boucles sécurisées . . . 118
5.4 Caractérisation de fautes . . . 121
5.4.1 Méthodologie . . . 121

5.4.2 Code initial : séquence de nops . . . 122
5.4.3 Séries d'additions . . . 122
5.4.3.1 Addition mono-registre . . . 123
5.4.3.2 Additions multi-registre indépendantes . . . 125
5.4.3.3 Isolation des effets . . . 128
5.4.4 Résumé des effets et classification . . . 129
5.5 Retour sur les boucles . . . 130
5.5.1 Analyse des fautes sur boucles . . . 130
5.5.2 Répétabilité des fautes . . . 132
5.5.3 Classification des fautes sur les boucles . . . 133
5.6 Conclusion . . . 136
5.7 Annexe des attaques . . . 138

6 Conclusion générale **141**
6.1 Bilan du travail réalisé . . . 141
6.2 Perspectives . . . 143

Bibliographie **145**

Table des figures **155**

Liste des tableaux **157**

Acronymes

AES
: *Advanced Encryption Standard*. 1, 11, 17, 19, 36, 65, 74, 75, 85

ANSSI
: *Agence nationale de la sécurité des systèmes d'information*. 100

AST
: *Abstract Syntax Tree*. 24

BTB
: *Branch Target Buffer*. 109, 131, 132

CFG
: *Control-Flow Graph*. 22, 26, 39, 42, 44, 47, 48, 55, 71, 76, 77, 89, 91, 98, 108, 131, 137, 142, 155

CFI
: *Control Flow Integrity*. 76

CRC
: *Cyclic Redundancy Check*. 18

CRT
: *Chinese Remainder Theorem*. 8

CSE
: *Common Subexpression Elimination*. 26, 53, 95

CVE
: *Common Vulnerabilities and Exposures*. 7

DCE
: *Dead Code Elimination*. 27, 38

DFA
: *Differential Fault Analysis*. 16

ECC

Elliptic Curve Cryptography. 8, 19, 75

FDTC

Fault Diagnosis and Tolerance in Cryptography. 8

GCC

GNU Compiler Collection. 25, 54, 143

GPIO

General Purpose Input-Output. 109, 110, 112, 113, 156

GVN

Global Value Numbering. 26, 37, 38, 53

IACR

International Association for Cryptologic Research. 9

IoT

Internet of Things. 1, 16, 108, 109, 144

IR

Représentation intermédiaire. 24

ISA

Instruction Set Architecture. 14–16, 20, 31, 97, 108, 114, 121, 122, 126, 129, 131, 137, 141, 142

JOP

Jump-Oriented Programming. 76

LICM

Loop-Invariant Code Motion. 27, 54

LLVM

Low-Level Virtual Machine. 3, 25, 27, 30, 53–58, 63, 72, 98, 113, 118

LSR

Loop Strength Reduction. 54, 55, 95

PC

Compteur de programme. 15, 131

PKI

Public Key Infrastructure. 1

PLAF

Protecting Loops Against Faults. 3, 5, 34, 39, 58, 71, 94, 96, 99–101, 103–105, 118, 120, 133, 141, 142

RNG

Random Number Generators ou *générateurs de nombres aléatoires*. 12

ROP

Return-Oriented Programming. 76

RSA

Rivest, Shamir, Adleman. 1, 8, 11, 12, 19

SCA

Side-channel Attacks. 8

SIMD

Single Instruction Multiple Data. 19, 30

SoC

System-on-Chip. 4, 5, 32, 109–111, 121, 156

SSA

Static Single Assignment. 25, 26, 39, 42, 43, 54, 55, 88

SSH

Secure SHell. 1

SWIFT

SoftWare Implemented Fault Tolerance. 120, 132–134

1 Introduction

Depuis de nombreuses années, nous nous sommes habitués à utiliser au quotidien des produits à base de cartes à puces, que ce soit avec nos cartes bancaires et autres cartes d'abonnements ou encore dans nos téléphones. Ces dernières années, une multitude d'objets électroniques miniatures est venu compléter ces habitudes, communément regroupés sous le nom d'Internet des Objets (*Internet of Things* (IoT)) : téléphones mobiles, passeports électroniques, étiquettes RFID ou encore bracelets connectés ont pour objectif d'améliorer notre confort. Pour mener à bien leur tâche, ces objets sont susceptibles de collecter ou d'échanger des données personnelles, convoitées pour leur haute valeur marchande, bien souvent au détriment de l'utilisateur via l'usage de services gratuits. La sécurité de ces systèmes embarqués est donc devenue un enjeu majeur à la fois pour l'industrie et pour les entités étatiques. Ces systèmes sont cibles d'attaques variées visant leur confidentialité, leur intégrité ou encore leur disponibilité.

Pour garantir la confidentialité de ces données sensibles, une solution consiste à faire appel à la cryptographie. Les standards tels que l'*Advanced Encryption Standard* (AES) et le RSA (pour *Rivest, Shamir, Adleman* (RSA)) peuvent être respectivement utilisés pour assurer la sécurité d'un canal de communication ou établir des schémas de confiance partagée entre les objets (e.g. respectivement canal *Secure SHell* (SSH) ou schéma *Public Key Infrastructure* (PKI)). Toutefois, les preuves de sécurité algébriques de ces algorithmes ne sont plus suffisantes pour garantir la sécurité des systèmes embarqués. En effet, ces systèmes peuvent être soumis à des attaques physiques : des attaques par canaux auxiliaires dont le but est de retirer des informations en exploitant des fuites physiques du système [Kocher et al., 1999] et des attaques par injection de faute provoquant une perturbation dans le composant ciblé pour obtenir, par exemple, des droits privilégiés [Boneh et al., 1997]. Les attaques par injection de faute sont puissantes et ne ciblent pas seulement les algorithmes cryptographiques : des codes système comme le démarrage de la plateforme ou la mise à jour du micrologiciel sont autant de points sensibles visés par ces attaques [Vasselle et al., 2017]. Cela fait de la sécurisation des systèmes embarqués pour se prémunir de ces attaques un enjeu important pour les industriels. De plus, la complexité de ces attaques en fait un vaste domaine d'étude. Nous considérons donc dans cette thèse exclusivement les attaques en faute.

L'efficacité de ces méthodes d'attaque combinée à la diversité des dispositifs électroniques rend très complexe la sécurisation de ces systèmes embarqués. De plus, l'implémentation de schémas de protection peut sévèrement impacter les performances du système (i.e. consom-

mation électrique, autonomie ou encore latence). L'objectif majeur de la sécurisation est de maximiser la sécurité sans pénaliser la performance de façon trop importante. La sécurisation peut se faire de façon matérielle ou logicielle. Si la performance est la principale contrainte du système, la première stratégie est la plus séduisante. En effet, les contre-mesures peuvent être appliquées au niveau matériel sans pénaliser la performance sous réserve d'accepter une augmentation substantielle de la taille du circuit et donc, de son coût. Le principal inconvénient de cette stratégie est qu'elle ne supporte pas de mise à jour suivant les évolutions de l'état de l'art. Cela peut engendrer d'importantes pertes financières ou un déficit d'image sur les produits concernés lorsqu'une faille de sécurité est exploitée. Les protections matérielles ne peuvent ainsi être utilisées seules et nécessitent l'ajout de protections logicielles.

Les contre-mesures logicielles offrent une possibilité de mise à jour pour suivre l'évolution des attaques physiques. Cependant, ajouter des contre-mesures au niveau logiciel n'apporte pas nécessairement toutes les garanties. En effet, sécuriser une implémentation logicielle nécessite une connaissance pointue de la plateforme cible ainsi que du code à sécuriser. Il faut déployer des contre-mesures adéquates en modélisant fidèlement les effets des attaques potentielles sur la cible.

1.1 Problématique

Le déploiement de contre-mesures logicielles dans l'industrie dépend de plusieurs contraintes, posant différents problèmes auxquels cette thèse s'attache à donner des réponses. L'objectif premier d'une contre-mesure est de protéger le code contre les attaques considérées. Les protections, issues d'expertises, visent à protéger un système contre un niveau d'attaquant défini vis à vis du niveau de certification ciblé pour le produit concerné. En effet, le processus de certification nécessite une analyse sécuritaire contre les attaques physiques à partir du niveau EAL5 [Criteria, 2017], niveau minimum requis pour une carte bancaire. Pour cela, les effets possibles des fautes sont modélisés, et les protections apportées sont dépendantes de la précision de cette modélisation. Les effets observables variant largement d'une architecture à l'autre, une caractérisation de ces effets est donc utile pour mieux comprendre comment intégrer des protections efficaces.

Une fois les effets possibles des fautes identifiés, il est important de déterminer quelles parties du code sont à sécuriser, comme les boucles et appels de fonction qui sont des exemples de portions sensibles de code, afin de définir des schémas de protection adaptés. Dans l'industrie, le déploiement de ces schémas est encore largement réalisé de façon manuelle, induisant des coûts et temps de développement importants, tout en étant source d'erreurs. Ce savoir-faire est même un argument de vente important des fabricants de produits sécurisés. Cela affecte directement le temps de mise sur le marché des produits et leur rentabilité. Les industriels sont donc en recherche d'approches automatisées de renforcement de code pour réduire

ces coûts. Néanmoins, l'ajout de contre-mesures de façon automatique possède ses propres contraintes, expliquant leur relativement faible utilisation de nos jours. Pour mettre en œuvre automatiquement la protection, il est nécessaire d'avoir des schémas de protection génériques afin de les appliquer sur des codes avec des formes variées, ainsi que des outils permettant leur implémentation automatisée. De plus, le code contenant les protections est susceptible de subir des transformations, notamment lors de l'étape de compilation qui traduit le langage de programmation de haut niveau – comme du C – en langage machine. En effet, le compilateur peut compromettre les protections ajoutées en amont [Balakrishnan et al., 2008], il est donc nécessaire de s'assurer que les protections insérées sont préservées dans le code binaire exécuté.

Les temps et coût de développement d'un produit sont des éléments essentiels à son succès. La performance du produit peut l'être tout autant. L'association de la performance et de la sécurité est ce qui garantit au produit une durée de vie conséquente : un passeport peut par exemple se retrouver sur le terrain pour 10 ans. Selon le marché ciblé, des contraintes de performances doivent être respectées (cartes bancaires par exemple). L'ajout de protections logicielles impliquant un surcoût que ce soit en termes de performances ou en taille de code, il est important de limiter ce surcoût afin de respecter les performances attendues. Une solution est de limiter au maximum l'application de la sécurisation aux seules parties sensibles du code considérées. Si en pratique, sécuriser manuellement des portions sensibles de code est faisable, son automatisation requiert des analyses qui peuvent se révéler être complexes afin de déterminer les éléments du code à sécuriser tout en limitant l'impact de la sécurisation sur les performances.

1.2 Contributions

Dans cette thèse, nous proposons deux schémas de protection contre les attaques en faute visant des parties sensibles de code et pouvant être déployés de manière automatisée à la compilation. En effet, le compilateur pouvant interférer avec l'ajout de protections, notre approche est de l'utiliser pour y intégrer des schémas de protection et ainsi limiter ses interactions plutôt que de les subir. Ces deux algorithmes ont été implémentés dans l'infrastructure de compilation *Low-Level Virtual Machine* (LLVM) [Lattner and Adve, 2004] :

- L'algorithme *Protecting Loops Against Faults* (PLAF) apporte une contre-mesure dédiée à la sécurisation des boucles. Les boucles, et notamment leur nombre d'itérations, sont souvent des points sensibles dans le code, que ce soit système [Nashimoto et al., 2016] ou cryptographique [Dehbaoui et al., 2013], qui, de plus, ne sont pas simples à sécuriser en matériel. Le schéma proposé est basé sur de la redondance assurant que lors de l'exécution, la boucle effectue le bon nombre d'itérations et sort par la bonne sortie de boucle. Il utilise une analyse des instructions utiles à dupliquer afin de garantir la propriété de sécurité souhaitée tout en limitant l'impact en performances et en taille de

code. Nous proposons également une étude du flot de compilation et des interactions entre les optimisations du compilateur et la sécurisation afin d'intégrer au mieux cet algorithme et détecter les dégradations.

- L'algorithme *CalleeSimo* apporte une contre-mesure dédiée à la sécurisation du graphe d'appel, en ajoutant un suivi des fonctions appelées (*callees*) et une protection de la valeur de leurs arguments. Ce schéma est proposé pour répondre à une demande de l'industrie de pouvoir automatiser des pratiques largement répandues de sécurisation manuelle des appels de fonctions et de leurs arguments, mais pas nécessairement disponibles publiquement dans l'état de l'art. L'implémentation dans LLVM proposée est configurable avec plusieurs options représentant des cas d'utilisations pratiques de ce type de protection. Il se base sur des équivalences de signatures des appels exécutés de part et d'autre des appels à sécuriser.

Ces schémas permettent une sécurisation automatique de points sensibles et fournissent des protections à faible coût facilement configurables afin de pouvoir être déployées simplement sur des codes existants en un temps limité et avec un impact limité sur le code protégé.

Comme toute protection, ces schémas sont conçus vis-à-vis de modèles d'attaquant. Les systèmes embarqués comme les cartes à puce représentent le contexte d'application typique de ces protections. Néanmoins, il est important de se poser la question de la validité de ces modèles pour d'autres architectures dans un futur proche. Les attaques récentes [Vasselle et al., 2017, Timmers and Spruyt, 2016] sur des composants complexes poussent à considérer des architectures plus complexes, comme les *System-on-Chip* (SoC) et les processeurs superscalaires et multi-cœurs. L'architecture de ce type de composant a peu été étudiée dans le contexte des attaques par fautes. L'état de l'art manque notamment de caractérisation des effets possibles de ces attaques sur ces plateformes permettant de définir des modèles de fautes réalistes. Nous présentons également une étude des effets possibles d'injections de faute par impulsions électromagnétiques sur processeur complexe. De cette étude, nous tirons de nouveaux modèles de fautes qu'il est intéressant de considérer pour de futurs schémas de protection ou l'adaptation de schémas existants.

1.3 Plan du manuscrit

La suite de cette thèse est décomposée en 5 chapitres. Le chapitre 2 présente l'état de l'art des attaques physiques en se concentrant sur les attaques par fautes. Il présente également les principes des contre-mesures existantes, qu'elles soient matérielles ou logicielles. L'implémentation de contre-mesures logicielles nécessitant de choisir sur quelle représentation du code agir, ce chapitre présente également les différentes approches de sécurisation logicielle automatisées au niveau du code source, du code binaire et à la compilation. Pour comprendre les enjeux de la sécurisation à la compilation, l'architecture d'un compilateur et les interactions liées à son but d'optimisation de code sont détaillées.

Le chapitre 3 présente l'algorithme PLAF dédié à la sécurisation des boucles contre des attaques par fautes pouvant être modélisées par un saut d'instruction ou une corruption de registre. Une analyse de la sécurité apportée par ce schéma est présentée, complétée de résultats de simulations d'injections de fautes, effectuées sur notre implémentation dans LLVM, pour valider la contre-mesure dans un environnement représentatif de systèmes embarqués. Le chapitre présente également l'analyse du placement dans le flot de compilation et des interactions avec les différentes optimisations.

Le chapitre 4 présente la seconde contre-mesure, *Calleesimo*, dédiée au suivi du graphe d'appel. Le schéma de protection étant composé de plusieurs variantes à coût et surface de protection variables, les différentes variantes ainsi que les contraintes associées pour leur application sont détaillées. Les conventions d'appels variant d'une architecture à l'autre, une analyse sécuritaire spécifique pour architecture ARM est proposée. Le chapitre précise les coûts en termes de performance et taille de code des variantes proposées.

Le chapitre 5 utilise un banc d'injection d'impulsions électromagnétique pour réaliser des campagnes d'injection de fautes sur un composant de type SoC basé sur un processeur ARM Cortex-A9. Nous présentons dans ce chapitre la caractérisation des effets résultant de ces campagnes d'injection afin de pouvoir proposer des modèles de fautes pour cette architecture. Nous utilisons également cet environnement pour étudier la résistance de la contre-mesure du chapitre 3 face à des attaques physiques réelles sur des architectures plus complexes que celles initialement envisagées, dont les effets sont plus complexes et plus variés que ceux décrits par le modèle initial dédié aux systèmes embarqués aux architectures plus simples.

Enfin, le chapitre 6 fait le bilan de ce manuscrit et présente les perspectives ouvertes par ces travaux.

Attaques par injection de fautes et contre-mesures

2

Sommaire

2.1 Introduction **7**
2.2 Attaques par injection de faute **9**
2.2.1 Vue d'ensemble 9
2.2.2 Moyens d'injection 10
2.2.3 Modèles de fautes 13
2.2.4 Exploitation des exécutions corrompues 16
2.3 Protections contre les fautes **17**
2.3.1 Principes des contre-mesures 17
2.3.2 Contre-mesures matérielles 18
2.3.3 Contre-mesures logicielles 19
2.3.4 Mise en œuvre des protections logicielles 20
2.4 Sécurisation logicielle automatisée **20**
2.4.1 Automatisation au niveau du code source 21
2.4.2 Automatisation au niveau du code binaire 22
2.4.3 Compromis entre les deux approches 23
2.5 Compilation et insertion de protection **23**
2.5.1 Compilation 23
2.5.2 Automatisation à la compilation 28
2.6 Conclusion **31**

2.1 Introduction

Les attaques auxquelles les systèmes embarqués sont sujets sont divisées en deux catégories. D'une part les attaques logicielles exploitent des vulnérabilités dans le code dont un grand nombre sont regroupées dans le dictionnaire *Common Vulnerabilities and Exposures* (CVE) [1]. D'autre part, des attaques nécessitant l'accès au composant, appelées attaques physiques, utilisent cet accès direct afin d'exploiter l'observation de propriétés physiques ou la déviation du comportement normal. Dans cette thèse, nous ne cherchons pas spécialement à empêcher les attaques logicielles. Nous nous concentrons sur les attaques physiques, qui sont particulièrement dangereuses dans le contexte des systèmes embarqués. Ces attaques physiques sont

1. https://cve.mitre.org/

classées en deux catégories selon que l'on cherche à observer le comportement du circuit (attaques par canaux auxiliaires), ou à le perturber (attaques par injection de faute) pour en retirer un bénéfice, comme une clé secrète ou un accès sans autorisation.

La première catégorie d'attaques consiste à observer des grandeurs physiques mesurables du composant. Appelées attaques par canaux auxiliaires (*Side-channel Attacks* (SCA)), elles sont dites passives ou non-invasives car elles ne modifient pas le comportement du circuit visé. Le but de ces attaques est principalement d'extraire des informations sensibles comme des clés de chiffrement et visent donc couramment les algorithmes cryptographiques. Parmi les grandeurs physiques utilisées, on peut notamment citer la consommation électrique [Kocher et al., 1999], le temps d'exécution [Kocher, 1996], le rayonnement électromagnétique [Dehbaoui et al., 2010] ou encore le signal acoustique [Genkin et al., 2017]. Les mesures de ces grandeurs peuvent être corrélées, entre autres, aux données utilisées, ce qui peut faire fuir de l'information sensible. Par exemple, dans le cas de la multiplication modulaire de Montgomery qui est l'une des opérations de base des algorithmes cryptographiques RSA et *Elliptic Curve Cryptography* (ECC), il est ainsi possible de retrouver la clé de chiffrement [Dugardin et al., 2016]. Ces attaques ne sont pas limitées à la communauté des experts en sécurité embarquée. Ils ont des retentissements jusque dans des journaux d'information généralistes, comme l'article [Aciiçmez et al., 2007], reporté dans le quotidien Le Monde[Morin, 2006], dans lequel les auteurs analysent les écarts de temps apportés par un prédicteur de branchements pour retrouver une clé de chiffrement RSA, notamment dans la librarire OpenSSL [2]. La thèse s'intéressant aux attaques par injection de faute, cette première catégorie d'attaques et les protections associées ne sont pas plus détaillées.

La deuxième catégorie d'attaques consiste à perturber le fonctionnement normal d'un circuit. Initialement, des perturbations de composants électroniques ont été observées dans les années 70 dans l'aérospatiale où des rayonnements cosmiques étaient la cause d'erreurs [C. May and H. Woods, 1978]. Ces sources d'erreurs sont devenues une source d'attaque potentielle des circuits quand, en 1997, Boneh et al. [Boneh et al., 1997] ont proposé d'exploiter une injection volontaire de faute pour attaquer l'algorithme RSA lorsque le théorème des restes Chinois (*Chinese Remainder Theorem* (CRT)) est utilisé pour accélérer les calculs. Depuis, les injections de fautes ont été la source de nombreuses attaques qui seront détaillées dans les prochaines sections. L'objectif de ces attaques est plus varié que les attaques par observations car un attaquant peut les utiliser pour :

- extraire des informations sensibles [Boneh et al., 2001, Dehbaoui et al., 2013] ;
- contourner des procédures d'authentification [Dureuil et al., 2016a] ;
- obtenir davantage de droits [Timmers and Spruyt, 2016].

Ces attaques sont devenues un champ de recherche très actif et sont même devenues le thème principal de conférences, comme le workshop *Fault Diagnosis and Tolerance in Cryptography*

2. https://www.openssl.org/

(FDTC) [3] organisé en coopération avec l'*International Association for Cryptologic Research* (IACR). De plus, elles ne sont pas restreintes à la communauté cryptographique. Elles touchent des codes plus généraux, en particulier plusieurs travaux montrent leur capacité à fauter des codes système (*secure boot*, *buffer overflow*) [Timmers and Spruyt, 2016, Vasselle et al., 2017, Nashimoto et al., 2016].

Dans ce chapitre, nous présentons l'état de l'art des attaques par fautes, ainsi que des schémas de protection associés. La section 2.2 présente les attaques physiques en détaillant les moyens d'injection de fautes et la modélisation de leurs effets. La section 2.3 présente les principes généraux des schémas de protection existants contre ces attaques. La section 2.4 détaille les moyens possibles d'automatiser l'application de ces protections, ainsi que les avantages et inconvénients des différentes approches. La section 2.5 présente dans un premier temps le fonctionnement d'un compilateur dans le but de comprendre les difficultés liées à l'automatisation de l'insertion de protections à la compilation. Puis, cette section présente les travaux existants d'application de contre-mesures à la compilation. Enfin, la section 2.6 résume ce chapitre et conclut en positionnant les objectifs de cette thèse.

2.2 Attaques par injection de faute

Cette section propose de détailler les capacités d'un attaquant à injecter des fautes à travers la présentation de travaux existants.

2.2.1 Vue d'ensemble

Les attaques par injection de fautes consistent à modifier le comportement normal du composant ciblé en perturbant une grandeur physique liée à son environnement. Si l'effet de l'injection est en premier lieu un effet physique, l'attaquant cherche généralement à l'exploiter au niveau logiciel. La figure 2.1 montre la vue d'ensemble des différents niveaux, matériels et logiciels, auxquels les effets d'une faute sont observables. L'attaquant observe les effets au niveau du code exécuté (en haut de la figure) alors que l'injection de faute induit d'abord des effets physiques (en bas de la figure).

La grandeur physique perturbée implique des altérations générant des courants électriques temporaires ou modifiant les caractéristiques des courants existants au niveau des composants du circuit comme les portes logiques ou cellules mémoires. Ces altérations peuvent être visibles au niveau micro-architectural lorsqu'elles sont capturées par une cellule mémoire ou un registre. Les instructions exécutées ou les données manipulées peuvent alors être corrompues. Au niveau logiciel, ces effets se traduisent par une corruption du flot de données ou du flot de contrôle du programme exécuté [Yuce et al., 2018]. Les effets observables

3. http://conferenze.dei.polimi.it/FDTC18/

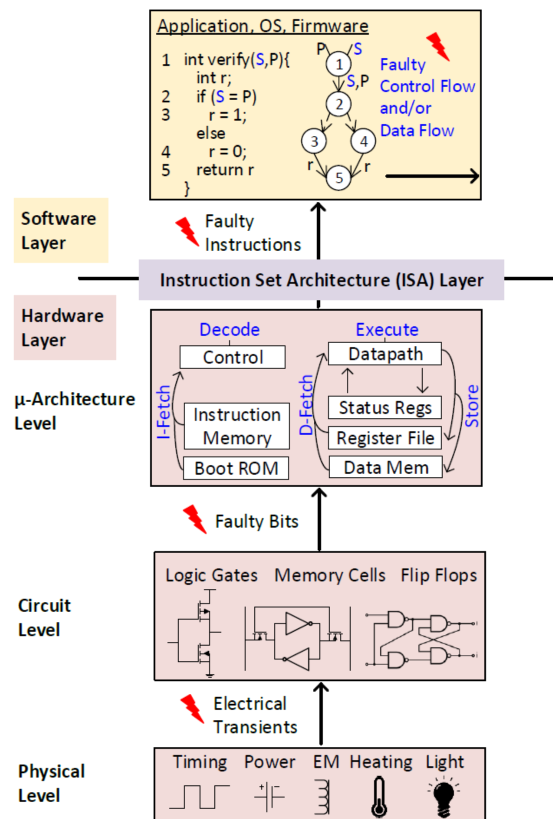


Figure 2.1: Vue d'ensemble des effets d'une faute aux différents niveaux ([Yuce et al., 2018])

dépendent d'un grand nombre de paramètres du moyen d'injection, de la plateforme ciblée et également du code s'exécutant [Dureuil et al., 2016b].

2.2.2 Moyens d'injection

Parmi les possibilités décrites dans la littérature pour injecter des fautes [Bar-El et al., 2006], on recense des perturbations du signal d'horloge ou de la tension d'alimentation, des altérations de la température ainsi que des injections par impulsions électromagnétiques ou par rayonnement lumineux (laser). Ces techniques nécessitent un investissement matériel dont le coût varie selon la technique utilisée et peut aller du simple au centuple [Guillen et al., 2017]. Certains proposent des plateformes à moindre coût [Guillen et al., 2017], et de plus en plus de moyens sont à disposition des attaquants, faisant des attaques par faute une menace à prendre sérieusement en compte. La précision temporelle et spatiale varie aussi selon le moyen utilisé. Chacun de ces moyens d'injection nécessite une étape de configuration, incluant l'instant et la durée de l'injection ainsi que son intensité, qui influe sur l'effet au niveau du circuit. Cette étape est cruciale pour l'attaquant afin de tenter d'obtenir une configuration qui lui permet de générer des effets exploitables. De plus, il est parfois nécessaire de préparer le composant ciblé, ce qui accentue le temps et le budget nécessaire à l'attaque [Dutertre et al., 2011]. Le procédé peut aussi s'avérer destructif pour le composant, compliquant d'autant

la tâche. Dans la suite de cette section, nous détaillons les différents moyens d'injection de faute.

2.2.2.1 Perturbation de la tension d'alimentation

Les attaques par perturbation de la tension d'alimentation, ou *power glitch*, consistent à faire varier cette tension pendant un très court instant. Ce type de perturbation est capable d'avoir des effets sur la vitesse de cheminement des données et induire des modifications de certaines portes logiques, donc de fauter certains calculs comme des valeurs intermédiaires d'un chiffrement AES [Zussa et al., 2012]. Par le caractère diffusant de ce type d'attaque, il est difficile de viser la corruption d'une donnée particulière. Dans leur article [Aumüller et al., 2003], Aumüller et al. montrent que les variables intervenant dans le calcul d'un chiffrement RSA peuvent être corrompues, et montrent qu'il est ainsi possible de mettre en péril la sécurité de l'algorithme implémenté sans contre-mesure. Le calcul lié à un chiffrement RSA étant relativement long, il est facile de localiser dans le temps l'exécution de l'algorithme pour le corrompre. Plus récemment, dans [Timmers and Mune, 2017] les auteurs s'attaquent au noyau Linux sur une plateforme de type smart-phone à base de processeur ARM Cortex-A9 afin d'obtenir des droits super-utilisateurs.

Cette méthode d'attaque est intéressante sur les dispositifs où l'on peut accéder à l'alimentation du composant pour réaliser une attaque à faible coût. Cet accès n'est pas toujours possible et la précision spatiale de l'attaque est faible : l'ensemble du composant ciblé est perturbé, ou au moins le domaine du composant qui partage la tension modifiée.

2.2.2.2 Perturbation du signal d'horloge

De manière similaire, il est possible de perturber le signal d'horloge du composant (*clock glitch*). Comme les circuits électroniques sont conçus pour fonctionner à une fréquence d'horloge contenue dans une plage spécifique, forcer le circuit à fonctionner à une fréquence hors de cette plage (*downclocking* ou *overclocking*) peut entrainer une violation des contraintes temporelles du circuit. À plus haut niveau, cela peut se matérialiser par des erreurs dans les calculs ou les transferts de mémoire. Il est ainsi possible de corrompre des instructions [Balasch et al., 2011], et à plus haut niveau, de retrouver en partie la clé d'un chiffrement AES [Agoyan et al., 2010].

Comme précédemment, cette méthode nécessite de pouvoir altérer de l'extérieur le ou les signaux d'horloge interne utilisés dans le composant.

2.2.2.3 Température

Toujours dans la même idée de faire fonctionner le circuit en dehors des conditions prévues, il est également possible d'agir sur la température de fonctionnement. Dans [Soucarros et al.,

2011], les auteurs perturbent un *Random Number Generators* ou *générateurs de nombres aléatoires* (RNG) en le faisant fonctionner en dehors des températures spécifiées. En 2009, dans [Skorobogatov, 2009], les auteurs ont réussi à modifier des données stockées en mémoire flash en chauffant localement une cellule mémoire. En chauffant le circuit dans son ensemble, les auteurs de [Hutter and Schmidt, 2014] ont pu attaquer une implémentation de l'algorithme RSA.

Cette méthode est limitée en précision à la fois temporelle et spatiale. Il est en effet difficile de chauffer ou refroidir une partie ciblée d'un composant ou provoquer des variations rapides de la température.

2.2.2.4 Injection électromagnétique

De façon plus localisée, l'injection d'impulsions électromagnétiques (EM) consiste à créer un champ électromagnétique à la surface du circuit ciblé pour induire des courants sur certains fils électriques et perturber le fonctionnement du circuit. Cette technique a été introduite dans [Quisquater and Samyde, 2002] puis appliquée dans [Schmidt and Hutter, 2007] pour attaquer l'algorithme RSA. Depuis, de nombreuses attaques à base d'injection EM ont été présentées dans la littérature. Des algorithmes cryptographiques ont été, à de multiple reprises, la cible d'attaques à base d'injection d'impulsions EM sur des plateformes diverses [Ordas et al., 2015, Dehbaoui et al., 2012a, Dehbaoui et al., 2012b, Dureuil et al., 2016b, Moro et al., 2013, Majéric et al., 2016].

Par rapport aux trois moyens d'injection précédents, les injections d'impulsions EM sont plus précises dans le temps et dans l'espace. Elles ne nécessitent pas forcément de préparer le composant ciblé, selon que l'attaque se fasse face avant ou face arrière du composant.

2.2.2.5 Rayonnement lumineux

Le dernier moyen d'injection couramment utilisé dans la littérature est l'injection de lumière focalisée, généralement émise par un laser. L'énergie apportée par le rayonnement peut rendre les transistors ciblés conducteurs et ainsi perturber le fonctionnement du circuit. Cette technique a été d'abord utilisée dans [Skorobogatov and Anderson, 2003]. Une explication du phénomène photoélectrique responsable de la perturbation peut être trouvée dans [Dutertre et al., 2014]. Le diamètre du faisceau lumineux en fait le moyen d'injection le plus précis, permettant selon la cible choisie de perturber un groupe de transistors voire un unique transistor [Viera et al., 2018]. Le diamètre est à adapter à la technologie comme le montre la figure 2.2 [4].

4. image tirée de la présentation de Jean-Max Dutertre au Pré-GDR Sécurité Informatique, Jussieu, 2018

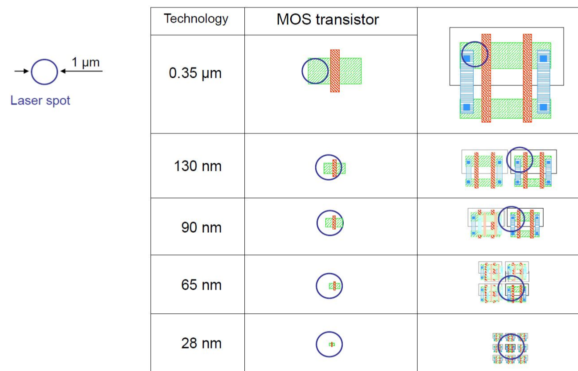


Figure 2.2: Taille comparative du faisceau laser par rapport à la technologie ciblée

En plus de l'équipement nécessaire, il est souvent indispensable d'effectuer une étape de préparation du circuit visé, consistant à décapsuler le boitier afin d'avoir accès au silicium [Dutertre et al., 2011].

L'injection laser est le moyen le plus précis pour injecter une faute mais aussi le plus onéreux. Dans [Breier and Jap, 2015], les auteurs décrivent un équipement d'injection dont le prix est de l'ordre de grandeur de 150 000 €. Néanmoins, la certification des composants (niveau EAL4+) requiert la réalisation d'attaques physiques notamment par injection laser ou EM. Il est donc essentiel pour les industriels proposant des systèmes certifiés de considérer ce type d'attaques.

2.2.3 Modèles de fautes

La section précédente a présenté différents moyens pour injecter une faute. L'attaquant a besoin d'une compréhension suffisante des effets possibles des fautes pour arriver à obtenir ce qu'il souhaite, le plus précis étant le mieux. Le concepteur de contre-mesures a besoin de comprendre précisément ces effets, et ce, à différents niveaux pour pouvoir s'en protéger (cf. figure 2.1). Pour cela, il est commun de chercher à modéliser les effets d'une faute à un niveau donné (physique, assembleur, code source ou algorithme). Ce modèle dépend à la fois du moyen d'injection et de l'architecture visée. Dans [Verbauwhede et al., 2011], les auteurs classifient des modèles de fautes communs de la littérature selon la partie du programme visé, le niveau d'abstraction et le nombre de bits impactés. Avant de préciser les niveaux auxquels décrire ces modèles de fautes, la section suivante décrit un élément important d'un effet qui est sa persistance dans le temps.

2.2.3.1 Persistance

Une caractéristique importante d'une faute est la durée de son effet. Deux catégories de fautes sont couramment observées :

- Permanente : même après une réinitialisation, l'effet d'une faute permanente est toujours présent. Cela se produit par exemple, lorsqu'un fusible ou une cellule mémoire est altéré voire détruit. Ce type d'effet n'est pas recherché par les attaquants car il n'est pas possible de revenir à l'état initial, ce qui limite les attaques différentielles par exemple [Biham and Shamir, 1999].
- Transitoire : l'effet de la faute est temporaire et le circuit revient dans son état initial ensuite. Selon le moyen d'injection, la précision dans le temps peut être plus ou moins importante. Par exemple, un seul cycle peut être affecté par la faute alors que dans certains cas, l'effet dure jusqu'à la réinitialisation du composant.

La plupart des attaques de la littérature considèrent des fautes transitoires. Dans ces deux cas, les effets peuvent être décrits à plusieurs niveaux (cf. figure 2.1), détaillés dans les sections suivantes.

2.2.3.2 Niveau logique

Les effets physiques d'une injection peuvent se traduire au niveau du circuit par des corruptions de bits. Un bit peut être soumis à plusieurs effets :

- sa valeur est inversée (*bit flip*)
- sa valeur est forcée à 0 (*bit reset*)
- sa valeur est forcée à 1 (*bit set*)

À part dans certains cas d'injections laser suffisamment précises pour ne cibler qu'un seul bit, il est rare que l'effet de la faute soit limité à la corruption d'un seul bit. Généralement, l'effet d'une faute est une combinaison de ces effets sur plusieurs bits d'un même mot (registre ou mémoire), voire de plusieurs mots simultanément. Dans [Moro et al., 2013], les auteurs corrompent par le biais d'une impulsion EM la valeur d'une donnée chargée depuis la mémoire flash, dont le nombre de bits affectés dépend de la tension de l'injection utilisée, jusqu'à forcer les 32 bits de la valeur chargée à 1. Dans ce cas, la valeur avec laquelle le bus de transfert de données a été préchargé semble avoir un impact sur la valeur corrompue. Toutefois, dans [Colombier et al., 2018], les auteurs utilisent une injection laser pour parvenir à forcer à 1 un unique bit d'un mot chargé depuis la mémoire flash.

2.2.3.3 Niveau ISA

En augmentant le niveau d'abstraction, des corruptions de bits observées au niveau logique corrompent un mot binaire, qui peut correspondre à des valeurs manipulées par le programme ou à des valeurs résidant dans des éléments mémorisant qui peuvent être des données ou du code. Les effets de ces corruptions modélisables au niveau *Instruction Set Architecture* (ISA) peuvent être des corruptions de registres ou des remplacements d'instructions. Une majorité d'articles considèrent des modèles de fautes à ce niveau [Trichina and Korkikyan, 2010, Balasch et al., 2011, Berthomé et al., 2012, Barenghi et al., 2012]. Selon les bits

corrompus dans le mot correspondant à une instruction, celle-ci peut être remplacée par un `nop` ou une autre instruction qui n'a pas d'effet sur le code exécuté, ce qui revient à ne pas exécuter l'instruction. C'est pourquoi, un effet souvent observé dans la littérature est le saut d'instruction [Moro, 2014, Barry et al., 2016].

Lorsque plusieurs mots sont affectés, ces effets peuvent alors se combiner. De plus, certains éléments micro-architecturaux peuvent être pris en considération. Dans [Rivière et al., 2015], les auteurs corrompent 4 instructions consécutives (128 bits) en ciblant le cache d'instructions ou le *prefetch buffer* d'une plateforme à base de processeur ARM Cortex-M4. Dans [Yuce et al., 2016], les auteurs montrent que toutes les instructions du pipeline peuvent être affectées simultanément par une unique perturbation du signal d'horloge. Cela se traduit par une combinaison de remplacements d'instructions et de corruptions de données.

2.2.3.4 Niveau code source

Au niveau du code source, ces registres et instructions corrompues peuvent impacter le flot de données et le flot de contrôle du programme en cours d'exécution. Ces deux flots sont interdépendants. En effet, une variable corrompue peut modifier les futurs chemins exécutés et inversement, exécuter une chemin non prévu peut altérer la valeur de certaines variables.

Plus précisément, au niveau source, certaines variables peuvent être corrompues, des tests peuvent être inversés faisant prendre au programme un mauvais chemin. Des sauts imprévus peuvent même être générés, notamment lorsqu'une instruction de branchement est corrompue ou lorsque l'instruction est remplacée par un branchement. Des corruptions du flot de contrôle ont également être réalisées en altérant la valeur du registre *Compteur de programme* (PC) [Timmers et al., 2016].

2.2.3.5 Vers des architectures plus complexes

Les modèles de fautes décrits dans cette section ont, pour la plupart, été réalisés sur deux types d'architectures : des micro-contrôleurs 8-bits AVR [Balasch et al., 2011] ou Intel 8051 [Berthomé et al., 2012], ou des cartes à puces 32-bits à base de processeur ARM Cortex-M3 ou équivalent [Moro et al., 2013, Colombier et al., 2018]. Ces architectures sont très répandues dans le domaine de la carte à puce pour la Pay-TV, les cartes SIM ou cartes bancaires, qui sont historiquement les cibles des attaques physiques. De plus, dans ces domaines, les exigences de certification expliquent les importants efforts pour caractériser les modèles de faute possibles sur ces architectures. Au niveau ISA, les modèles de faute le plus souvent cités sont le saut d'instruction et la corruption de registre. Comme détaillé dans les chapitres 3 et 4, ces modèles seront ceux considérés pour la conception des contre-mesures présentées dans cette thèse.

Avec l'avènement de l'IoT, de plus en plus d'architectures complexes (superscalaire, exécution dans le désordre, pipeline profond...) sont utilisées comme cibles d'attaques physiques [Vasselle et al., 2017, Timmers and Spruyt, 2016]. Au vu de la complexité de ces architectures, peu d'études détaillent les modèles de faute possibles aux niveaux inférieurs, or ceux-ci sont essentiels à comprendre lorsque l'on cherche à sécuriser ces plateformes. La caractérisation des effets observables au niveau ISA est une donnée manquante, aujourd'hui, pour ce type de plateformes.

2.2.4 Exploitation des exécutions corrompues

Le but d'un attaquant est d'exploiter une exécution corrompue pour retrouver une donnée secrète, accéder à des droits supplémentaires ou contourner des mécanismes d'authentification. Pour cela, une fois la ou les fautes injectées, il est nécessaire d'exploiter les données corrompues.

Pour accéder à des droits privilégiés ou contourner des mécanismes d'authentification, plusieurs scénarios d'attaque se basent sur la corruption du flot de contrôle du programme [Timmers et al., 2016, van Woudenberg et al., 2011]. Si l'attaquant est capable de générer des sauts d'instruction, il peut par exemple sauter le branchement associé à la condition, vérifiant les droits et ainsi obtenir l'effet souhaité, comme par exemple dans le cas d'une vérification de PIN [Dureuil et al., 2016a].

Afin de retrouver une donnée secrète, une technique appelée *safe error* consiste à exploiter le fait qu'une faute n'a pas d'impact sur le résultat de l'algorithme [Yen and Joye, 2000]. Fauter un calcul qui n'a pas d'effet sur le résultat final peut donner des indications sur le flot de contrôle exécuté comme dans le cas d'un algorithme *square-and-multiply-always* [5] par exemple.

Une autre technique d'attaque régulièrement employée contre les algorithmes cryptographiques est l'analyse différentielle (*Differential Fault Analysis* (DFA)) [Boneh et al., 1997]. Elle est basée sur la comparaison entre le chiffré fauté et le chiffré original utilisant le même texte clair et la même clé. L'analyse de plusieurs de ces couples permet de retrouver tout ou partie de la clé secrète par des propriétés mathématiques [Blömer and Seifert, 2003]. Un seul chiffré fauté peut même suffire dans certains cas [Tunstall et al., 2011]. Toutefois, cette méthode ne s'applique que quand l'attaquant peut maîtriser les entrées fournies à l'algorithme. Ce type d'attaque a été en particulier utilisé récemment pour plusieurs algorithmes dédiés à la cryptographie légère comme les algorithmes PRIDE [Lac et al., 2016], Trivium [Mohamed et al., 2011], SCREAM [Lac et al., 2017], SIMON et SPECK [Tupsamudre et al., 2014]. Pour

5. À une itération de la boucle principale de l'algorithme, fauter la multiplication qui n'est pas accumulée dans le résultat final n'a pas d'impact et la connaissance de cette information fournit à l'attaquant la valeur du bit de l'exposant associé

éviter l'exploitation de ces données, il est donc nécessaire d'ajouter des protections dédiées à ces attaques physiques.

2.3 Protections contre les fautes

Pour limiter l'impact des attaques par injection de faute, des protections peuvent être ajoutées de deux manières : soit au niveau matériel du composant ciblé, soit au niveau logiciel. Dans cette section, nous présentons les approches couramment utilisées afin d'améliorer la résistance aux attaques par fautes.

2.3.1 Principes des contre-mesures

Pour se prémunir des effets d'attaques par fautes, il existe 3 types de contre-mesures :

- La détection de fautes consiste à détecter un comportement anormal et de réagir en conséquence, en réinitialisant le système par exemple ;
- La tolérance aux fautes consiste à s'assurer que même en présence de faute, le résultat retourné reste inchangé ;
- L'infection de faute consiste à infecter le résultat de façon à ce que ce dernier ne soit pas exploitable par l'attaquant.

Les techniques de protection les plus répandues se basent sur de la redondance spatiale ou temporelle. En effet, effectuer deux fois le calcul et comparer les résultats permet de détecter si l'un des calculs a été fauté [Bar-El et al., 2006]. De plus, si au moins trois calculs sont réalisés, il est possible d'être tolérant à une faute en utilisant un système à base de vote majoritaire sur les résultats obtenus. La redondance temporelle a un impact sur les performances alors que la redondance spatiale a un impact sur la taille du composant ou de la mémoire utilisée.

L'infection de faute est spécifique aux opérations cryptographiques pour lesquelles on cherche à limiter ou empêcher l'attaquant de réaliser une cryptanalyse à partir des résultats fautés, comme dans le cas du chiffrement AES présenté dans [Patranabis et al., 2017]. Il est également possible de ne pas renvoyer les données en cas de faute détectée. En revanche, cette méthode ne peut pas s'appliquer à tous les codes. De plus, il est toujours possible d'analyser le comportement du circuit par des méthodes d'attaques combinées (injection de faute et observations) [Roche et al., 2011].

2.3.2 Contre-mesures matérielles

Les contre-mesures au niveau matériel sont majoritairement utilisées pour détecter des fautes. Elles sont basées sur l'ajout de composants (détecteurs, blocs) qui ont un impact sur la surface du composant à protéger.

Il est possible d'intégrer au circuit des détecteurs physiques. Il en existe capables de déceler des violations de contraintes temporelles, issues par exemple de perturbations de la tension d'alimentation ou du signal d'horloge [Zussa et al., 2014]. Dans [El-Baze et al., 2016], les auteurs proposent un détecteur d'injection d'impulsions électromagnétiques. De manière similaire, dans [He et al., 2016], les auteurs ont conçu un détecteur d'attaques par injection laser basé sur un oscillateur en anneau haute-fréquence. Il en existe également d'autres types, comme des détecteurs photosensibles ou encore des détecteurs d'intrusion [Bar-El et al., 2006]. Généralement, ils lèvent une alarme lorsque le composant fonctionne en dehors de sa spécification, qui peut être utilisée pour traiter la faute, comme réinitialiser la plateforme par exemple. L'avantage de ces détecteurs est de pouvoir détecter au plus vite un comportement anormal, et d'arrêter l'exécution dès la détection. En revanche, ils ne détectent pas forcément toutes les perturbations, et inversement, peuvent aussi générer des fausses alarmes.

Une autre technique pour détecter des fautes est l'utilisation de code détecteurs d'erreurs comme le *Cyclic Redundancy Check* (CRC). Cette technique couramment utilisée pour détecter des erreurs dans les communications peut en effet s'appliquer à la détection de fautes intentionnelles. Par exemple, dans [Barenghi et al., 2010], les auteurs utilisent un bit de parité comme code de détection d'erreur.

Lorsqu'un calcul sensible est effectué par un bloc matériel, il est possible d'utiliser un schéma basé sur de la redondance. Cette redondance peut être temporelle avec l'insertion d'un délai en entrée du bloc de calcul, ou spatiale en dupliquant ce bloc. Cette technique peut coûter cher en surface du circuit selon la taille du bloc à dupliquer, mais fournit une bonne protection contre les attaques laser. En effet, il est difficile de cibler deux blocs distincts simultanément. En revanche, cette technique montre ses limites face à des attaques plus globales sur le circuit comme la perturbation du signal d'horloge ou de la tension d'alimentation.

Ajouter des protections au niveau matériel a un avantage certain : le surcoût en termes de performances du circuit est faible voire nul, sous réserve de pouvoir supporter la surface supplémentaire pour implémenter la contre-mesure sur le composant. En revanche, l'ajout de contre-mesures impacte le coût de la conception du matériel et de sa validation. De plus, leur principal défaut est leur absence de flexibilité. Il est illusoire de penser qu'une protection sera efficace sur de longues périodes de temps tant l'état de l'art des attaques évolue à la fois dans les techniques utilisées et dans leur efficacité. Une protection ajoutée au niveau matériel peut très bien, une fois le composant mis sur le marché se retrouver face à une nouvelle

attaque contre laquelle il est difficile voire impossible d'apporter des mises à jour. En plus de compromettre les composants concernés, de telles attaques peuvent même nuire à l'image de la marque. C'est pourquoi, même en présence de protection matérielles, il est indispensable de considérer également l'ajout de contre-mesures logicielles.

2.3.3 Contre-mesures logicielles

Au niveau logiciel, il est possible d'ajouter de la redondance au niveau d'une instruction, d'une fonction ou de l'algorithme. Dans sa thèse [Moro, 2014], Nicolas Moro propose un schéma de tolérance à un saut d'instruction basé sur de la redondance temporelle des instructions. Il utilise la propriété d'idempotence [6] de certaines instructions pour les dupliquer directement et transforme les instructions qui n'ont pas cette propriété en séquence d'instructions qui le sont toutes afin de pouvoir tout dupliquer. De manière similaire, dans [Barenghi et al., 2010], les auteurs ajoutent de la détection de faute à une implémentation de l'algorithme AES en dupliquant ou tripliquant certaines instructions considérées comme sensibles. Au niveau logiciel, utiliser de la redondance ne se limite pas à dupliquer les calculs avec les mêmes valeurs. Il est ainsi possible d'utiliser certaines propriétés arithmétiques pour que le calcul dupliqué se fasse sur des données inversées bit à bit [Hillebold, 2014]. Un schéma basé sur de la redondance spatiale est proposé dans [Chen et al., 2017] où les auteurs utilisent les extensions *Single Instruction Multiple Data* (SIMD) des processeurs modernes pour effecteur les calculs sur plusieurs données en parallèle. Dans [Patrick et al., 2017], les auteurs appliquent un mélange entre redondance spatiale et temporelle intra-instruction, où les données dupliquées sont entrelacées avec les données originales.

Plutôt que de dupliquer le code, il est parfois possible d'ajouter des vérifications au niveau algorithmique. Pour des algorithmes cryptographiques, il est par exemple possible de déchiffrer le résultat pour vérifier le bon chiffrement d'un texte et s'assurer de retrouver le texte initial. Selon l'algorithme à protéger, certaines propriétés mathématiques peuvent également être utilisées comme des égalités modulaires pour le RSA [Vigilant, 2008] ou l'appartenance à la bonne courbe dans le contexte de l'ECC [Ciet and Joye, 2005].

Dans [Lalande et al., 2014], les auteurs utilisent des compteurs d'instructions pour protéger du code écrit en langage C contre des attaques provoquant des sauts correspondant à au moins 2 lignes de code source. Les compteurs sont incrémentés après chaque instruction C et sont potentiellement vérifiées à ce moment ou à la sortie d'un bloc, en comparant leur valeur à celle précalculée attendue.

Pour se protéger des fautes affectant le flot de contrôle du programme, d'autres schémas de protection ont été proposés à base de signatures de parties de code. Dans [Oh et al., 2002], le

6. Une instruction idempotente est une instruction qui produit le même effet qu'elle soit exécutée une ou plusieurs fois (ex : `add r0, r1, r2`)

schéma proposé permet de détecter au moment de l'exécution que le chemin suivi est bien un chemin statiquement autorisé.

Les protections ajoutées au niveau logiciel offrent une flexibilité plus importante. La prise en compte de nouvelles attaques dans la sécurisation d'applications peut se faire de manière rapide après l'apparition d'une nouvelle attaque. En revanche, selon le schéma utilisé, le coût en performances ou en taille de code peut être élevé, en particulier si le schéma est appliqué sur l'ensemble du code [Moro, 2014]. Le coût en temps de développement peut aussi être important si le schéma est appliqué manuellement.

2.3.4 Mise en œuvre des protections logicielles

Tous ces schémas de protection au niveau logiciel nécessitent l'ajout de code. Pour les systèmes embarqués, les ressources étant limitées, il est essentiel que l'ajout de protection se fasse avec un coût en performances et taille de code limité. On peut toutefois profiter de la flexibilité de la sécurisation au niveau logiciel pour se limiter aux parties critiques du code pour la sécurité, plutôt que d'appliquer des schémas lourds et génériques.

En pratique, que ce soit au niveau du code source ou au niveau ISA, la sécurisation de code déployée actuellement dans l'industrie utilise des techniques ad-hoc de protections logicielles pour se protéger contre des attaques considérées comme réalistes pour le produit ciblé aux niveaux de certification visés. Ce procédé, majoritairement manuel, est souvent réalisé en deux temps. Des experts ajoutent des protections au code puis des expérimentations ou des simulations sont effectuées pour tester l'efficacité du code sécurisé. Ces étapes manuelles font de la sécurisation logicielle un processus long, coûteux, et source d'erreurs. De plus, il est souvent nécessaire de les réitérer en cas de multiples plateformes cibles. C'est pourquoi les industriels sont en demande d'approches automatisées pour réduire ces coûts.

2.4 Sécurisation logicielle automatisée

La recherche constante de réduction des coûts et donc du temps de conception dans l'industrie font que l'application automatique de sécurité est un domaine qui se développe depuis plusieurs années. Pour autant, pour déployer des protections de manière automatisée, il est nécessaire d'avoir à disposition des schémas de protection génériques et des outils permettant leur application automatique. De plus, l'automatisation de l'insertion de protections pose la question de la représentation du code sur laquelle effectuer la sécurisation. Deux approches sont naturellement utilisées selon où elles agissent : au niveau du code source ou au niveau du code binaire. Le reste de cette section présente ces deux approches.

2.4.1 Automatisation au niveau du code source

Dans [Akkar et al., 2003], les auteurs proposent un schéma de protection du flot de contrôle contre les attaques en faute. Ce schéma est basé sur le contrôle de différents points de passage sensibles dans le code définis par le développeur. Des annotations sous forme de *pragma* sont ajoutées au début d'une zone à sécuriser, à chaque point de passage et au niveau où le contrôle doit être effectué. Afin de transformer le code et d'y ajouter les contre-mesures, les auteurs utilisent un outil de réécriture de code source basé sur le préprocesseur. Cet outil transforme les annotations sous forme de *pragma* pour générer un code source contenant la protection, qui peut ensuite être compilé.

Dans [Lalande et al., 2014], un schéma générique de protection de code C est présenté. Il se base sur l'ajout de compteurs dans le code source. Le schéma de protection à l'aide de compteurs étant générique, cette contre-mesure peut être appliquée de manière automatique à l'aide d'un script en python de réécriture du code source. Le code sécurisé est ensuite compilé afin de générer le binaire à exécuter. En revanche, pour que ce binaire garde les protections insérées, il est nécessaire de désactiver les optimisations.

À ce niveau, les protections souhaitées sont plus simples à exprimer, que ce soit pour définir les variables sensibles ou pour identifier les parties de code à protéger. En revanche, les modèles d'attaquant à ce niveau peuvent être éloignés de la réalité des attaques qui visent le code binaire exécuté. Il n'y a pas de correspondance directe entre les fautes à bas niveau et celles modélisables au niveau du code source, ce qui rend l'exercice délicat.

Le principal défaut de l'application de contre-mesures au niveau du code source est l'ensemble des transformations dues aux optimisations que le code protégé subit lors de la compilation. Dans le but de produire du code optimisé, le compilateur élimine les redondances ou toute partie de code qu'il détermine comme inutile au fonctionnement du programme. Il peut donc supprimer les duplications ou compteurs ajoutés pour la sécurité. Pour éviter les interactions avec le compilateur, il est nécessaire d'utiliser des astuces pour le contourner. Une technique possible pour des schémas de duplication est de déclarer `volatile` certaines variables afin d'empêcher le compilateur de les optimiser [Eide and Regehr, 2008]. Il est aussi possible d'insérer du code assembleur que le compilateur ne peut analyser afin de l'empêcher de supprimer certaines parties de code. Plus radicalement, il est parfois nécessaire de compiler le code sans optimisation (niveau `-O0`). L'impact en performances et taille de code peut alors être rédhibitoire, surtout dans le contexte des systèmes embarqués où le prix de la mémoire est une part importante du prix du produit final, et ou des exigences de performances peuvent être requises comme dans le standard EMV[7]. La réduction de taille de code et l'amélioration des performances apportées par les optimisations pour les systèmes embarqués sont des atouts difficiles à ignorer dans l'industrie.

7. https://www.emvco.com/emv-technologies/contact/

Cette approche ayant comme principal problème d'avoir le compilateur en aval des contre-mesures, la deuxième approche propose d'appliquer les contre-mesures après les transformations liées au compilateur.

2.4.2 Automatisation au niveau du code binaire

Une approche opposée à la précédente consiste à appliquer des contre-mesures au niveau du code binaire. Le schéma proposé par Nicolas Moro dans sa thèse [Moro, 2014], présenté dans la section 2.3.3, propose de dupliquer des instructions du jeu d'instruction ARMv7. Il propose un algorithme afin d'appliquer cette contre-mesure automatiquement sur du code binaire ARMv7 désassemblé. Le manque d'analyse sémantique du code à sécuriser implique une duplication localisée de chaque instruction pas nécessairement optimisée. De plus, l'ajout de cette contre-mesure peut nécessiter l'utilisation de registres additionnels.

Dans [De Keulenaer et al., 2015], les auteurs utilisent une infrastructure logicielle de réécriture de code binaire au niveau de l'édition de liens, *Diablo* [Put et al., 2005], pour insérer plusieurs contre-mesures contre les attaques en faute. Ce puissant outil possède des capacités d'analyse importantes et recréé le *Control-Flow Graph* (CFG) des fonctions en utilisant une représentation intermédiaire propre de bas niveau. Grâce à cet outil, les auteurs ont pu automatiser l'implémentation de schémas de contre-mesures génériques, comme la duplication de branchements conditionnels, la vérification d'écritures mémoire, ou encore la duplication de compteurs de boucle. Dans ce cas également, le manque d'information sémantique ne permet pas de localiser précisément les parties sensibles à sécuriser. De plus, ces schémas sont limités à des cas simples dans lesquels les analyses de l'outil sont capables de retrouver les éléments nécessaires à la sécurisation. Par exemple, la duplication du compteur de boucle n'est possible que si la boucle possède un compteur qui n'est incrémenté qu'une seule fois, comparé à une constante de la boucle et qu'aucun appel à une fonction est contenu dans la boucle.

Agir à ce niveau possède un avantage indéniable dans le cas où le code source de l'application à protéger n'est pas disponible. Dans ce cas où seul le binaire est disponible, il n'y a alors pas d'autre solution pour intégrer des contre-mesures. De plus, le code binaire étant plus proche de l'effet physique des fautes, les modifications apportées à ce niveau permettent de considérer des modèles de fautes plus près de la réalité.

En revanche, pour pouvoir introduire du code à ce niveau, ces approches nécessitent de disposer de registres disponibles. Il est donc nécessaire d'avoir recours à de complexes analyses afin de pouvoir identifier les intervalles de vie des registres disponibles pour les ré-allouer, voire devoir les stocker sur la pile engendrant des surcoûts. De plus, le manque d'information sémantique à ce niveau ne permet pas forcément de pouvoir retrouver finement

les informations sensibles du code à sécuriser et limite donc le type de schémas de protection déployables à ce niveau.

2.4.3 Compromis entre les deux approches

L'automatisation est attractive car c'est un moyen de capitaliser les différentes protections existantes et de pouvoir les déployer à la demande. Néanmoins, avec ces deux approches, on constate que d'un côté, l'utilisation du compilateur en aval peut compromettre la sécurité apportée et que de l'autre côté, on aurait besoin des informations disponibles à la compilation que le code binaire n'a plus. Une troisième approche qui est d'utiliser le compilateur, outil central disposant de toutes les informations et responsable de toutes les transformations de code, pour mettre en œuvre les contre-mesures semble réunir les avantages des deux approches précédentes. L'utilisation du compilateur pour la sécurité, en particulier de sa représentation intermédiaire, est le choix fait pour cette thèse.

2.5 Compilation et insertion de protection

Afin de mieux comprendre les enjeux et les difficultés liées à cette approche d'insertion de protection à la compilation, cette section décrit plus en détail le fonctionnement d'un compilateur, ainsi que des travaux existants de sécurisation à la compilation.

2.5.1 Compilation

De nos jours, la plupart des codes d'application sont écrits dans un langage de haut-niveau comme le C, même si certaines portions de programmes sont toujours écrites en assembleur, notamment pour des raisons de performance. Le compilateur est donc un outil indispensable de traduction entre ce langage source et le langage machine ciblé. Pour réaliser cette transformation de code, un programme passe par au moins 3 représentations : du code source au code assembleur en passant par la ou les représentations intermédiaires utilisées lors de la compilation. Les paragraphes suivants décrivent en détail cette étape de compilation et les modifications effectuées qui peuvent impacter la sécurité qui serait ajoutée en amont.

2.5.1.1 Architecture d'un compilateur

Il existe de nombreux compilateurs correspondant aux nombreux langages source et architectures matérielles existants. Un compilateur est généralement découpé en trois étapes principales : la partie frontale, la partie intermédiaire et la partie arrière. La figure 2.3 représente les étapes principales du flot de compilation détaillées dans cette section.

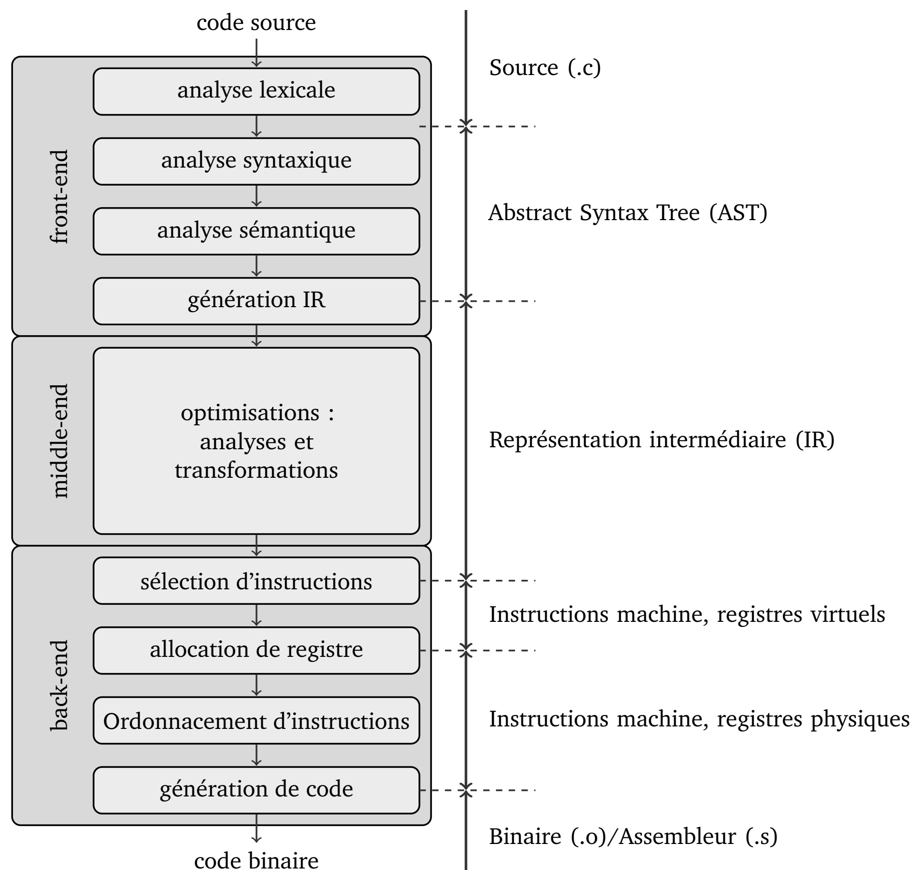


Figure 2.3: Flot de compilation et représentations du code associées

Partie frontale La partie frontale, ou *front-end*, a pour but d'analyser le code source pour le transformer dans la représentation interne du compilateur (*Représentation intermédiaire* (IR)). Cette étape est dépendante du langage source utilisé et a pour but de supprimer cette dépendance grâce à une IR agnostique du langage source. L'analyse se décompose de la façon suivante :

- L'analyse lexicale parcours le code source afin d'en repérer les mots-clés, variables et éléments de structure appelés jetons ;
- L'analyse syntaxique structure cette liste de jetons sous forme d'arbre (*Abstract Syntax Tree* (AST)) et vérifie la validité du code par rapport à la grammaire du langage.
- L'analyse sémantique vérifie sur l'AST que les propriétés du langage source sont respectées et, au besoin, retourne des messages d'erreur.
- Enfin, l'IR correspondante à l'AST du code est générée.

Partie intermédiaire La partie intermédiaire, ou *middle-end*, a pour rôle principal d'optimiser le code afin de réduire sa taille, son empreinte mémoire et/ou son temps d'exécution. Cette étape est une succession de passes d'analyses et d'optimisations dont l'organisation dépend du compilateur considéré. Elle agit sur l'IR du code qui est non seulement indépendante du langage source mais également de l'architecture cible. Cette étape est critique pour générer

du code performant. Les optimisations apportées au niveau IR sont génériques, applicables pour n'importe quels langages source et architectures cibles.

L'IR est souvent représentée sous forme *Static Single Assignment* (SSA) [Rastello and Bouchez-Tichadou, 2019]. Dans cette forme, chaque variable n'est définie qu'une seule fois. Le flot de données est capturé syntaxiquement, facilitant les analyses et optimisations de code. Cette forme est utilisée dans les compilateurs de production de nos jours, notamment dans les compilateurs libres *GNU Compiler Collection* (GCC) [8] et LLVM [Lattner and Adve, 2004].

Partie arrière La partie arrière, ou *back-end*, s'occupe de générer le code final pour l'architecture ciblée au travers des étapes suivantes :

- Une première étape de transformation, appelée sélection d'instructions, consiste à remplacer les instructions au niveau IR par des instructions machine spécifiques à l'architecture cible. Il n'y a pas nécessairement équivalence entre ces instructions donc plusieurs instructions IR peuvent être représentées par une unique instruction machine et réciproquement. Après cette étape, le code utilise des instructions spécifiques à l'architecture mais utilise une infinité de registres virtuels correspondant à une forme SSA.
- Ensuite, une autre phase de transformation, appelée allocation de registre, consiste à affecter les registres physiques en nombre restreint aux registres virtuels utilisés jusque là, éliminant au passage la forme SSA. En cas d'impossibilité, il est nécessaire de stocker temporairement la valeur en mémoire afin de la réutiliser, opération appelée *spill de registre*.
- Afin de profiter des spécificités de l'architecture cible, une passe d'ordonnancement des instructions est effectuée. Elle permet d'améliorer le temps d'exécution du programme en utilisant, par exemple, au mieux le pipeline du processeur, en fonction des dépendances entre les données, les instructions et du niveau de parallélisme.
- Enfin, le code final est émis, soit sous forme de code objet, de binaire exécutable ou d'assembleur.

Dans la thèse nous avons choisi d'utiliser l'infrastructure de compilation LLVM. LLVM est constitué d'une collection de modules de compilation, développé depuis 2004. Depuis, sa flexibilité et sa simplicité d'utilisation pour développeurs (comparé à *gcc*) en a fait le compilateur privilégié d'importants industriels tels que Apple, Samsung, Nvidia ou Google qui soutiennent le projet. Cet engouement et la relative facilité de le modifier, notamment pour y ajouter de la sécurisation de code, sont les principales raisons d'un tel choix. Le paragraphe suivant détaille les spécificités du compilateur LLVM et en particulier les interactions potentielles avec la sécurisation de code.

8. https://gcc.gnu.org/

2.5.1.2 Architecture de LLVM

LLVM est une infrastructure de compilation modulaire regroupant de multiples outils autour de la compilation. Les 3 parties *front-end*, *middle-end* et *back-end* sont explicitement séparées ce qui permet de ne développer que la partie frontale pour rajouter un nouveau langage supporté, et qu'une partie arrière pour supporter une nouvelle architecture. La figure 2.4 représente le schéma général de la structure du compilateur.

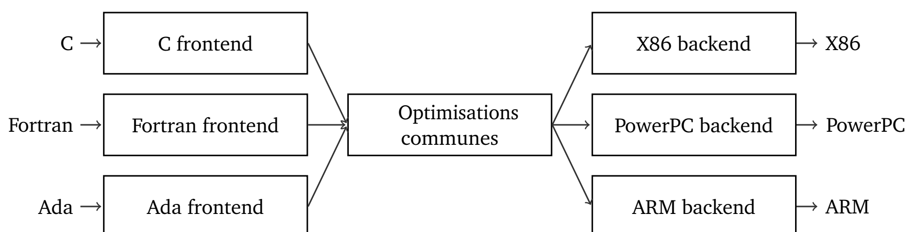


Figure 2.4: Structure modulaire du compilateur LLVM

LLVM utilise une représentation intermédiaire sous la forme SSA fournissant des opérations bas-niveau tout en ayant la capacité à représenter les structures des langages haut-niveau, commes les *switchs*. Dans cette IR, le corps d'une fonction est représenté sous la forme de son graphe de flot de contrôle (CFG) décomposé en blocs de base [9]. La majorité des optimisations sont réalisées sur cette forme par le *middle-end*. Les optimisations sont composées de passes d'analyses et de passes de transformations utilisant les données de ces analyses. Elles ont pour but d'éliminer au maximum les calculs inutiles, les accès mémoires généralement coûteux, et de structurer le code pour faciliter les optimisations les plus importantes, utilisées dans les niveaux d'optimisations standards (`-O1`, `-Os` par exemple). Les optimisations majeures utilisées dans LLVM sont les suivantes :

- La passe *Register Promotion* (*Mem2Reg*) a pour but de supprimer les accès mémoire en promouvant les variables dans des registres, simplifiant les structures de haut-niveau le cas échéant (structures C, tableaux...). Cette passe permet notamment de simplifier les analyses de dépendances de données qui peuvent être difficiles voire impossibles en cas d'accès mémoire omniprésents.
- Les passes *Common Subexpression Elimination* (CSE) et *Global Value Numbering* (GVN) ont pour but de simplifier les opérations arithmétiques redondantes utilisées dans le calcul de plusieurs variables, et de simplifier les variables ayant les mêmes valeurs afin d'éliminer des calculs communs en dépendant. Elles sont complémentaires et leur différence réside dans la façon de définir la redondance : GVN détecte des motifs redondants par analyse sémantique tandis que CSE utilise une analyse syntaxique. La combinaison de ces deux passes peut éliminer de la sécurisation basée sur de la duplication d'instructions mais est également capable d'éliminer des chaines entières de dépendances *def-use* dupliquées.

9. suite linéaire d'instructions se terminant par un saut

- De multiples variantes de *Dead Code Elimination* (DCE) ont pour but d'éliminer toute partie de code inutile, comme des branches du flot de contrôle jamais atteintes ou des variables inutilisées. Les contre-mesures ajoutent souvent des blocs de gestion d'erreurs, pour traiter les cas où une faute est détectée. En l'absence de faute, ces blocs ne doivent jamais être exécutés et peuvent ainsi être supprimés par le compilateur étant donné qu'il n'a pas conscience d'une éventuelle attaque par faute. Un exemple classique de double vérification est présenté dans le listing 2.1 qui est simplifié après élimination de code mort pour le code du listing 2.2 (les codes exemples fournis sont écrits en C pour une meilleure lisibilité mais ces passes agissent sur l'IR).

```
if (isAuthenticated == true) {
    if (isAuthenticated == false) {
        CounterMeasure();
    } else {
        RunSecureProgram();
    }
} else {
    Mute();
}
```

Listing 2.1: Code sécurisé avant optimisation

```
if (isAuthenticated == true) {



        RunSecureProgram();

} else {
    Mute();
}
```

Listing 2.2: Code après optimisation

- Les boucles étant souvent des points critiques pour la performance de code, plusieurs passes sont dédiées aux boucles. Elles transforment leur flot de contrôle soit pour faciliter les analyses et transformations futures (*loop canonicalization*), soit pour améliorer leurs performances (*loop rotation*, *loop unrolling*, *loop unswitching*, *Loop-Invariant Code Motion* (LICM))
- Plusieurs passes d'optimisation inter-procédurales d'*inlining* de fonction permettent de fusionner tout ou partie [10] du corps de certaines fonctions dans d'autres.

L'ensemble des passes d'analyses et de transformations de la version courante de LLVM peuvent être trouvées sur le site du compilateur [11].

À titre d'exemple, la figure 2.5 montre les principales étapes de la transformation du code d'une fonction de calcul d'un *pgcd* de deux nombres entiers écrite en C (figure 2.5 (1)) à travers le flot de compilation. On peut voir l'IR générée par le front-end de LLVM à partir du code source (figure 2.5 (2)). Avant optimisation, on peut noter que chaque variable est allouée sur la pile (bloc *.entry*). On peut aussi remarquer que le flot de contrôle de la fonction correspond à celui du code source (le bloc *while.cond* correspond à la condition `while(r)`). Au contraire, après que le *middle-end* ait transformé le code avec le niveau d'optimisation `-Os` (figure 2.5 (3)), les variables sont directement utilisées dans des registres libérant ainsi de la mémoire et évitant de stocker des variables inutilement. De plus, on peut voir que le flot de contrôle a été modifié : la boucle a été retournée (optimisation *loop rotation*) pour placer la condition de sortie à la fin de la boucle et ne contenir qu'un seul bloc. Enfin, le

10. LLVM possède une passe d'inlining partiel de fonction : http://llvm.org/doxygen/PartialInlining_8cpp_source.html
11. https://llvm.org/docs/Passes.html

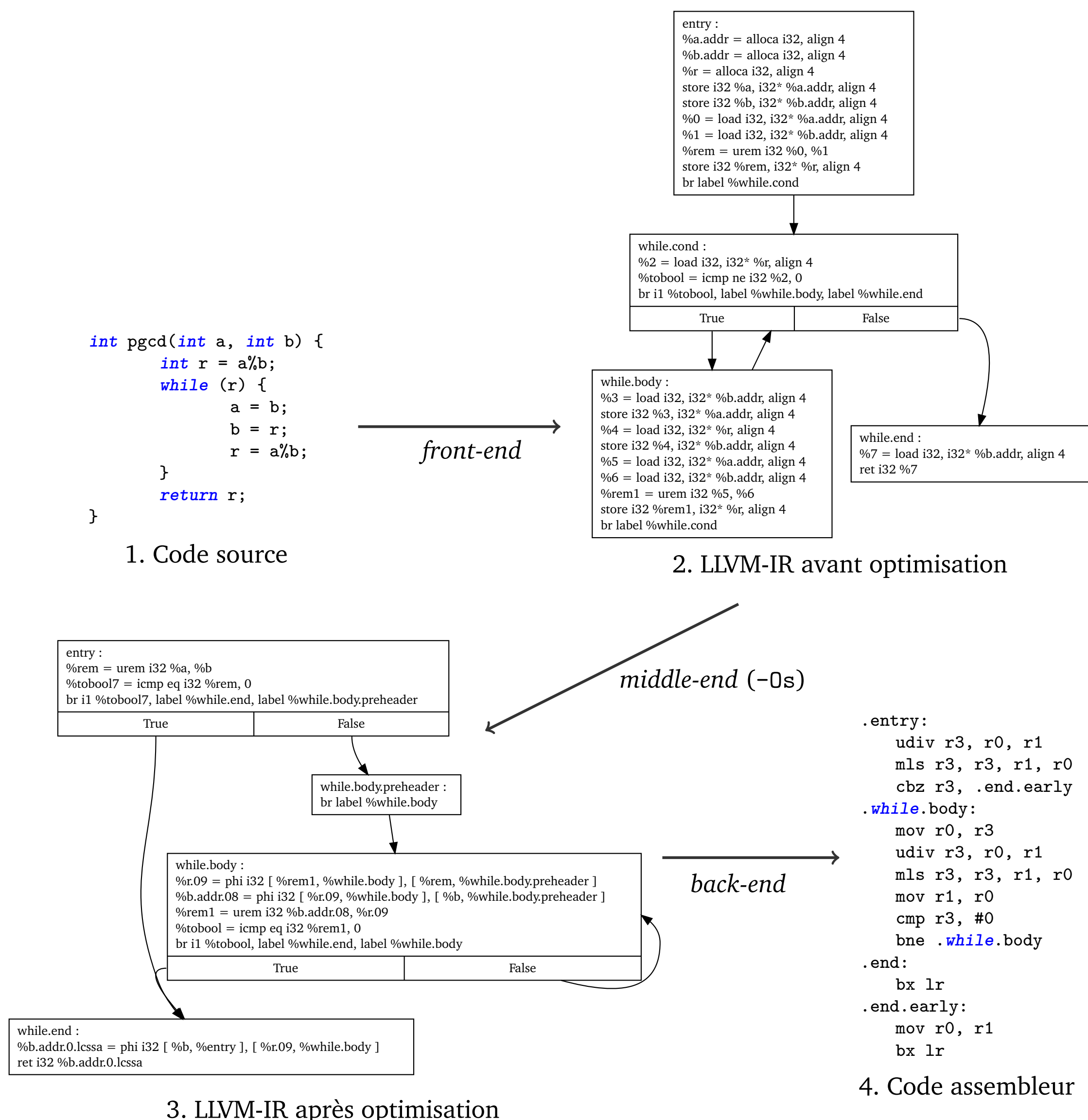


Figure 2.5: Transformations d'un code source (1) par le *front-end* en IR (2), optimisé par le *middle-end* au niveau d'optimisation -Os (3), transformé par le *back-end* pour architectures implémentant l'ISA ARMv7 en code assembleur (4)

back-end génère le code assembleur qu'on peut noter être proche de l'IR après optimisation (figure 2.5 (4)). Malgré tout, le back-end transforme encore le code de façon à utiliser au mieux les propriétés de l'architecture cible. On peut ainsi remarquer que la fonction possède maintenant deux sorties (instructions `bx lr`) car il serait plus coûteux de les réunir en une seule.

2.5.2 Automatisation à la compilation

Les deux principaux objectifs d'un compilateur sont de produire un code binaire sémantiquement équivalent au code initial, et de produire un code optimisé. Il est toutefois possible de

s'en servir pour ajouter de la sécurisation de code à différentes étapes du flot de compilation. Au niveau du *front-end*, l'approche est similaire à une approche côté code source et présente les mêmes défauts : les optimisations en aval peuvent altérer les protections ajoutées et le modèle d'attaquant à ce niveau peut être éloigné de la réalité. Au niveau du *back-end*, l'approche ressemble à une approche sur le code binaire, dépendante de l'architecture cible, mais avec plus d'information disponible selon le placement dans le *back-end*. Enfin, il est possible d'agir sur l'IR et ainsi être plus portable. Ces approches sont évidemment limitées aux cas où le compilateur peut être modifié pour lui ajouter les contre-mesures proposées et ne peut donc pas s'appliquer aux contextes où un compilateur propriétaire est utilisé en boite noire. Dans la littérature, les protections ajoutées à la compilation sont souvent contre les attaques par observations, les schémas présentés dans la suite ne sont donc pas limités aux attaques par faute.

2.5.2.1 Approche *back-end*

Dans [Bayrak et al., 2015], les auteurs automatisent des contre-mesures connues contre les attaques par observations (masquage booléen, préchargement aléatoire des bus) pour l'architecture AVR, appliquées à la fin du *back-end* sur le code assembleur. Avec cette automatisation, ils ajoutent une analyse de fuite d'information afin de sélectionner les instructions sensibles à protéger, et ainsi limiter le coût en performance et en taille de code. Il est important d'agir à ce niveau pour ce type de protections qui nécessitent d'agir sur chaque accès mémoire.

Le schéma de duplication des instructions de la thèse de Nicolas Moro présenté précédemment a été intégré au *back-end* du compilateur LLVM pour l'architecture ARMv7-Thumb2 [Barry et al., 2016]. Cette automatisation bénéficie directement d'être au niveau du compilateur pour empêcher au maximum la génération d'instructions non idempotentes et ainsi limiter le coût de la sécurisation. De plus, l'ordonnancement des instructions a été modifié afin d'ordonnancer les instructions dupliquées et de réduire l'impact sur les performances. Cela permet également d'étendre la protection à un saut de plus d'une instruction : en cas de faute sur le cache comme présenté dans [Rivière et al., 2015] qui permet de sauter jusqu'à 4 instructions consécutives, il est possible d'écarter suffisamment les instructions dupliquées des originales pour ne jamais pouvoir fauter les deux en une seule faute. De manière similaire, une instrumentation open-source [12] du compilateur LLVM intégrant une contre-mesure de duplication d'instructions est proposée dans [Cojocar et al., 2018] pour l'architecture ARMv7m/Thumb2 contre le modèle de saut d'instruction.

Appliquer des contre-mesures au niveau du back-end permet de profiter d'analyses et d'informations plus précises que sur le code binaire directement. De plus, en modifiant l'allocation de registres et la sélection d'instructions, il est possible de mieux interagir avec les schémas de protection à ajouter et ainsi, potentiellement limiter leur coût. Toutefois, ces implémentations

12. Sources disponible à l'adresse `https://github.com/cojocar/llvm-iskip`

sont spécifiques à une architecture cible unique et nécessitent d'être transposées pour les autres architectures avec un coût de développement important.

2.5.2.2 Approche *middle-end*

Dans [Agosta et al., 2013], les auteurs fournissent un outil de protection contre les attaques par observation appliquant du masquage de variables sensibles après une analyse de sensibilité des instructions via l'analyse du flot de données du programme. Ces analyses et transformations du code sont intégrées au compilateur LLVM. Elles fournissent une contre-mesure à faible coût en localisant précisément l'application du masquage aux seules variables déterminées sensibles, que seule une protection à la compilation peut automatiser.

Dans [Chen et al., 2017], les auteurs utilisent les extensions SIMD pour dupliquer des calculs et limiter l'impact des attaques par faute. Leur schéma de protection vise à utiliser de la redondance spatiale pour proposer un impact en performance faible voire nul au prix d'un surcoût en mémoire. Ce schéma est également intégré au niveau IR du compilateur LLVM. Comme précisé précédemment, le code binaire ne contient pas les informations suffisantes pour réaliser des analyses complexes. Il est difficile de mettre en œuvre la conversion d'instructions classiques en instructions SIMD d'un tel schéma de manière automatique sur du code binaire [Clark et al., 2007]. La modification du code source pour l'utilisation de ces extensions SIMD a déjà été utilisé manuellement pour des raisons de performances dans la librairie OpenSSL, il n'est donc pas nécessairement correct de supposer que les instructions et registres SIMD ne sont pas utilisés par le code avant sécurisation.

L'avantage d'agir au niveau IR est l'indépendance vis à vis de l'architecture cible, fournissant ainsi une portabilité aux contre-mesures développées. En revanche, les optimisations au niveau IR peuvent compromettre l'efficacité de la contre-mesure ajoutée comme dans le cas d'une protection au niveau du code source. L'implémentation de [Agosta et al., 2013] est placée après toutes les optimisations au niveau IR, avant le back-end. Dans [Chen et al., 2017], ce n'est pas explicitement décrit mais le placement semble être le même. Cette approche semble réunir les avantages des autres méthodes :

- De puissantes analyses permettent d'utiliser des schémas complexes pour, entre autres, localiser au mieux l'application de la contre-mesure et limiter son coût.
- Les transformations sont à la fois indépendantes du langage source et de l'architecture cible.
- Toutes les informations du code source sont disponibles, permettant ainsi d'exprimer simplement les protections souhaitées.
- En particulier dans LLVM, l'IR est bas-niveau, proche de l'assembleur, permettant ainsi de définir un modèle d'attaquant assez proche de la réalité.

En revanche, aucun des articles cités ne décrit les éventuels problèmes liées aux transformations du code dans le back-end. En effet, l'IR n'est pas la représentation finale du code exécuté

et le back-end ne se contente pas de transformer l'IR en code binaire. Des optimisations plus localisées sont aussi présentes dans le back-end et peuvent également compromettre la sécurité ajoutée au niveau IR. L'insertion d'intrinsèques au niveau IR permet de limiter les problèmes liées aux optimisations du *back-end*. La question de la robustesse du code obtenu est donc un point essentiel d'une telle approche, notamment dans le cas d'architectures complexes dans lesquelles les transformations du *back-end* peuvent être encore plus importantes. Cette approche a donc besoin d'analyses des interactions possibles avec le *back-end* et, en cas de besoin, d'avoir à disposition des moyens de contournements.

2.6 Conclusion

Dans ce chapitre, nous avons présenté les risques auxquels sont exposés les systèmes embarqués en mettant l'accent sur les attaques par injection de faute.

Ces attaques, consistant à perturber le comportement du circuit visé, peuvent être réalisées par différents moyens qui ont été décrits, comme le laser ou les impulsions électromagnétiques. Les effets de ces attaques au niveau physique ont, par transitivité, des effets aux niveaux circuit, micro-architectural, ISA et logiciel qui peuvent être modélisés et qui dépendent de l'architecture ciblée, des paramètres d'injection et du code exécuté. Pour limiter l'impact de ces attaques, des schémas de protections matérielles et logicielles ont été conçus et sont déployés en pratique.

Les contre-mesures matérielles manquent de souplesse et les contre-mesures logicielles sont encore souvent insérées de façon manuelle et coûteuse dans l'industrie (de la carte à puce notamment). Au niveau du code source, elles nécessitent d'utiliser des astuces pour contourner les optimisations du compilateur, voire de désactiver complètement les optimisations, tout en n'affranchissant pas d'une analyse manuelle du code généré pour s'assurer que la sécurisation est toujours présente. Au niveau assembleur, elles nécessitent une analyse manuelle longue et fastidieuse par des experts en sécurité afin de modifier le code pour y apporter la sécurité voulue. Cette démarche est coûteuse en temps de développement et peut nécessiter des étapes de débuggage particulièrement complexes.

En pratique, une combinaison de protections matérielles et logicielles sont souvent ajoutées, notamment pour des raisons de certification des produits. De plus, certains composants n'embarquent pas de protections matérielles et nécessitent alors des contre-mesures logicielles. Dans ce chapitre, nous avons indiqué que des méthodes d'automatisation de l'insertion de contre-mesures existent. Ces méthodes, lorsqu'elles sont appliquées sur le code source ou le code binaire, ont des limitations impliquant des vérifications manuelles longues et coûteuses ou sont trop ciblées à une architecture particulière et nécessitent alors des coûts de développement importants pour les porter sur d'autres cibles. Les coûts introduits sont

généralement trop élevés pour être adoptés par l'industrie. Certains schémas de protection contre les attaques par observation ont été mis en œuvre à la compilation, mais très peu de protections contre les attaques par injection de faute sont insérées par le compilateur. Pour permettre aux industriels de disposer de davantage de possibilités d'insertion automatique de protections, il est nécessaire de développer d'autres contre-mesures applicables à la compilation à d'autres granularités ou contre d'autres modèles de fautes par exemple. Ainsi, cette thèse vise à :

- Proposer des schémas de protections appliqués à la compilation. Ces schémas proposent des protections localisées à des parties sensibles de code (boucles dans le chapitre 3 et appels de fonctions dans le chapitre 4) afin d'être compatibles avec des contraintes fortes en taille de code et en performances pour répondre à un besoin industriel. Ils sont conçus en considérant un modèle de faute classique, hérité des nombreuses études caractérisant les modèles de faute largement observés sur des architectures typiques de l'industrie de la carte à puce.
- Étudier le placement dans le flot de compilation pour l'insertion de ces contre-mesures et étudier les effets du compilateur afin de mettre en évidence les interactions de ce dernier avec la sécurisation de code.
- Projeter les protections conçues dans les systèmes embarqués futurs, tels les SoC et micro-architectures complexes de plus en plus cibles d'attaques [Vasselle et al., 2017, Timmers and Mune, 2017], en étudiant les effets des fautes sur une architecture à processeur complexe et en soumettant les codes protégés à des attaques physiques sur ces architectures.

3 Sécurisation automatique des boucles

Sommaire

3.1 Introduction **34**
3.2 Motivations **35**
3.2.1 Modèle de faute 35
3.2.2 Attaques sur boucles 36
3.2.3 Schémas de protection existants 36
3.2.4 Protection manuelle des boucles et contraintes 37
3.2.5 Vers un algorithme dédié aux boucles 38
3.3 Algorithme de sécurisation des boucles : PLAF **39**
3.3.1 Prérequis 39
3.3.2 Vue d'ensemble 40
3.3.3 Détails des algorithmes 42
3.4 Analyse sécuritaire **49**
3.4.1 Saut d'instruction 51
3.4.2 Corruption de registre 51
3.5 Interactions avec le flot de compilation **52**
3.5.1 Passes en amont 53
3.5.2 Placement dans le flot de compilation 54
3.5.3 Passes en aval et ajustements nécessaires 55
3.5.4 Conclusion des modifications du back-end 58
3.6 Expérimentations et résultats **58**
3.6.1 Implémentation dans LLVM 58
3.6.2 Environnement expérimental 60
3.6.3 Couverture de sécurisation 63
3.6.4 Évaluation fonctionnelle 64
3.6.5 Coût de la contre-mesure 65
3.6.6 Évaluation sécuritaire de l'algorithme PLAF 67
3.7 Discussion sur une généralisation aux branchements **70**
3.8 Conclusion **71**

3.1 Introduction

Dans le chapitre précédent, nous avons montré les différents choix possibles pour l'ajout de contre-mesures logicielles. Les avantages des approches au niveau code source et au niveau code binaire sont mutualisés en appliquant si possible la contre-mesure à la compilation.

Ce chapitre présente la conception et l'implémentation d'une première contre-mesure à appliquer à la compilation, les difficultés rencontrées ainsi que les solutions apportées. Le schéma proposé, PLAF, permet de sécuriser automatiquement les boucles sensibles d'une application où la propriété de sécurité recherchée est la bonne exécution des boucles, autant dans le nombre d'itérations effectuées que dans la sortie de boucle empruntée. De plus, le schéma est capable de prendre en compte des boucles avec des flots de contrôle et de données complexes.

Des résultats expérimentaux sur cibles à architecture ARM montrent que le schéma PLAF permet de sécuriser de façon automatique les boucles sensibles de code cryptographiques connus pour être vulnérables. En moyenne 95% des boucles peuvent être sécurisées sur ces codes, et 97% des fautes correspondantes au modèle considéré peuvent être éliminées. De plus, localiser précisément la contre-mesure en l'appliquant qu'aux parties sensibles du code limite le surcoût en taille introduit à 14% en moyenne comparé à des schémas de protection plus génériques qui doublent la taille [Moro, 2014].

Après avoir précisé nos différentes choix et les raisons les motivant dans la section 3.2, l'étude comprend la conception de l'algorithme de sécurisation dans la section 3.3 et une analyse sécuritaire de l'algorithme proposé dans la section 3.4. La section 3.5 décrit les interactions entre la sécurisation des boucles et les optimisations du compilateur dans le but de déterminer un placement adéquat dans le flot de compilation. Les résultats expérimentaux sont présentés dans la section 3.6. La section 3.7 propose une discussion sur la généralisation de cet algorithme à la sécurisation d'un branchement quelconque.

Les travaux de ce chapitre ont été publiés dans la revue ACM *Transactions on Architecture and Code optimization* (TACO) en 2017 [Proy et al., 2017]. Ils ont été présentés à la conférence *HiPEAC18* [1] en Janvier 2018 à Manchester (UK), ainsi que dans le workshop *PHISIC2018* [2] en Mai 2018 à Gardanne (France).

1. https://www.hipeac.net/2018/manchester/
2. http://phisic2018.emse.fr/

3.2 Motivations

Avant de présenter en détails notre schéma de sécurisation des boucles, cette section présente les différents paramètres à prendre en compte à la conception d'un schéma de protection comme le modèle d'attaquant, le code ciblé à sécuriser ainsi que la représentation du code à laquelle appliquer la contre-mesure. Nos choix d'attaquant et de code cible ainsi que les limitations associées sont également discutés.

3.2.1 Modèle de faute

Pour définir une contre-mesure au niveau logiciel, il est essentiel de préciser les capacités de l'attaquant considéré en définissant un modèle de faute. Dans ce chapitre, nous considérons deux effets possibles d'injection de fautes transitoires dans le corps de la boucle : le cas d'un unique saut d'instruction d'une part, et le cas d'une unique corruption de registre d'autre part. On peut noter qu'en cas d'accès mémoire corrompu, l'effet de la faute peut persister. Les transferts mémoire sont indirectement couverts par ce modèle de faute car corrompre une lecture en mémoire revient à corrompre le registre contenant le résultat de cette lecture. De même, les écritures en mémoire sont prises en compte via la corruption du registre contenant la donnée à écrire.

Le saut d'instruction peut être vu comme un cas particulier de remplacement d'instruction dans lequel l'instruction fautée n'a aucun effet visible et est donc équivalente à un `nop`. Cet effet est fréquent parmi les effets observés sur différentes architectures et avec différents moyens d'injection [Karaklajic et al., 2013, Moro et al., 2014]. Les attaques en faute résultantes d'injection laser provoquent très souvent des corruptions de registre. De plus, les évaluations sécuritaires lors d'un processus de certification considèrent ces deux types de fautes [Criteria, 2017], expliquant pourquoi les industriels cherchent à s'en prémunir en premier lieu.

Plus récemment, l'injection de fautes multiples (plusieurs injections séparées dans le temps) a également été prise en compte dans la littérature [Dottax et al., 2009, Moro, 2014, Yuce et al., 2016] ainsi que pour la certification de produits sécurisés [Criteria, 2017]. Les attaques à base de fautes multiples ont en général pour but de viser deux instants bien différents de l'exécution du code. Il est, par exemple, possible d'induire une première faute sur une boucle puis une deuxième faute plus tard lors de la vérification du calcul par une contre-mesure afin de la contourner. Les mécanismes sécuritaires de détection de fautes des algorithmes sont souvent basés sur le calcul inverse sur les données de sortie afin de comparer avec les données initiales (déchiffrer le résultat pour vérifier qu'on n'obtient bien la donnée originale pour le cas d'un algorithme de chiffrement). Ces opérations supplémentaires comportent souvent des boucles, ce qui appuie notre démarche de sécurité se concentrant sur des boucles. Pour la

conception de l'algorithme de sécurisation, nous avons considéré une unique faute par boucle, ce qui correspond donc au scenario le plus probable même en cas de fautes multiples.

Si se prémunir contre plusieurs fautes localisées dans une même boucle devient un besoin, le schéma de duplication proposé peut être étendu pour répliquer plusieurs fois les instructions au prix d'une sécurisation plus coûteuse dont le niveau de protection apporté reste à étudier plus finement. Toutefois, cette contre-mesure dédiée aux boucles est proposée comme une première brique d'un compilateur orienté sécurité dans lequel d'autres techniques de sécurisation seraient potentiellement plus adaptées contre les fautes multiples. De plus, le schéma proposé ne protège pas contre des fautes multiples en général, mais est capable d'en détecter (cf. section 5.5).

3.2.2 Attaques sur boucles

Depuis de nombreuses années, des attaques de la littérature exploitent des fautes ayant comme cible particulière une boucle, que ce soit lié au domaine cryptographique ou venant d'applications système. Altérer le nombre d'itérations d'un chiffrement AES peut mener à une cryptanalyse permettant de retrouver la clé secrète [Dehbaoui et al., 2013, Demirci and Selçuk, 2008]. D'autres algorithmes cryptographiques sont sensibles à des attaques similaires sur les boucles, comme par exemple le couplage sur courbes elliptiques [El Mrabet, 2009, Mrabet, 2013, Page and Vercauteren, 2006] ou l'algorithme Khazad [Muller, 2003]. Côté système, on peut notamment citer des attaques de type *buffer overflow* [Nashimoto et al., 2016] ou encore le contournement de la vérification d'un PIN [Dureuil et al., 2016a].

Ces attaques visent les boucles, et plus particulièrement l'altération du nombre d'itérations de la boucle, que ce soit par défaut [Demirci and Selçuk, 2008] ou par excès [Nashimoto et al., 2016]. En cas de boucles avec un flot de contrôle complexe, cela peut vouloir signifier sortir de la boucle prématurément par une mauvaise sortie. Cela permet notamment de retrouver la clé d'un schéma de signature à base de réseaux via une telle sortie prématurée de boucle [Espitau et al., 2018].

La propriété de sécurité essentielle à préserver pour se prémunir contre ces attaques est que le bon nombre d'itérations soit effectué et, également, que la sortie de boucle prise soit la bonne.

3.2.3 Schémas de protection existants

Parmi les différentes contre-mesures présentées dans [De Keulenaer et al., 2015], un schéma de réécriture automatique de code binaire permettant une sécurisation des compteurs de boucle est proposé. Cette approche est malheureusement limitée à des boucles simples dans

lesquelles la variable d'induction doit être incrémentée ou décrémentée d'un pas constant à chaque itération avec une condition de sortie unique.

Certains schémas de protection génériques basés sur de la duplication permettent de protéger les boucles en tant que cas particulier, comme le schéma de [Moro, 2014] qui protège contre des attaques par saut d'instruction. Ces schémas génériques impliquent une sécurisation plus étendue engendrant un surcoût élevé étant donné qu'ils ne cherchent pas à protéger une région du code spécifique mais plutôt le code dans son ensemble [Reis et al., 2005].

Les schémas de protection dédiés à la sécurisation du flot de contrôle peuvent être utilisés pour détecter la bonne sortie de la boucle mais ne permettent en général pas de garantir le bon nombre d'itérations. Ils permettent, par exemple, d'éviter un déroutement du code résultant de la corruption d'un branchement [Oh et al., 2002]. De plus, ils ne tracent pas le flot de données qui pourrait aboutir à des sorties de boucle prématurées [Lalande et al., 2014].

À notre connaissance, il n'existe donc pas de contre-mesure dédiée aux boucles capables de prendre en compte la diversité des flots de contrôle possibles pour le corps d'une boucle.

3.2.4 Protection manuelle des boucles et contraintes

En pratique, une sécurisation classique du nombre d'itérations d'une boucle consiste à dupliquer la variable d'induction au niveau du code source [Dureuil et al., 2016a]. Dans cette section, différents code sont présentés, dans lesquels le code écrit en rouge correspond à l'ajout de code de sécurisation.

Afin de mettre en avant les problèmes liés à la sécurisation d'une boucle au niveau du code source, un premier exemple de boucle très simple est présenté dans le listing 3.1. Dans cet exemple, la variable d'induction est dupliquée et la condition de sortie de la boucle est vérifiée avec cette variable à la sortie de la boucle. Cette forme de duplication temporelle avec détection est une contre-mesure largement utilisée, avec ici, une granularité plus fine qu'une duplication de fonction entière. C'est un exemple typique de déploiement de protection au niveau du code source.

Si ce code est compilé avec un compilateur optimisant, les optimisations décrites dans la section 2.5 vont modifier ce code. L'optimisation GVN va détecter que la variable `j` n'est qu'une copie de la variable `i` et va donc remplacer toutes les occurrences de `j` par `i`. Ainsi, le code résultant est équivalent à dupliquer le calcul de la condition de sortie et sa vérification en utilisant seulement la variable `i`. Ainsi, une faute capable de corrompre la valeur de la variable `i` pendant la boucle ne peut être détecté car les deux vérifications seraient alors faussées. De plus, la vérification finale devenant la même instruction que la condition de

sortie, elle est toujours vraie en ce point du code et va, elle aussi, être supprimée par l'une des optimisations de type DCE.

```
int i, j;
for (i = 0, j = 0;; i < n; i++) {
    foo(i);
    j++;
}
if (j < n) error();
```

Listing 3.1: Simple boucle

```
int i = n, j = n;
while (i != 0) {
    foo(i);
    if (Array[i] == chkVal) break;
    i--; j--;
}
if (j != 0 && Array[j] != chkVal) error();
```

Listing 3.2: Boucle avec sortie multiple

Cet exemple de code simple entre facilement dans des cas où les optimisations du compilateur suppriment sans difficulté la contre-mesure ajoutée, et ce cas est loin d'être isolé. Un exemple de boucle avec deux sorties est présenté dans le listing 3.2. Afin de s'assurer du bon nombre d'itérations, la vérification finale inclut donc un contrôle basé sur les deux sorties de la boucle. Même dans ce cas, la combinaison des optimisations GVN et DCE est toujours capable de détecter cette duplication et de la supprimer.

Par ailleurs, il n'est pas toujours possible de déterminer les variables d'induction : l'exemple du listing 3.3 montre un cas plus complexe, tiré d'un exemple de calcul de *pgcd*. Dans cet exemple, on pourrait supposer que la variable d'induction est la variable `r` et que la dupliquer suffit à protéger la condition de sortie de la boucle. Or, une faute provoquant le saut de l'instruction `b=r;` affecte en même temps la variable originale `r` et la variable dupliquée `r'`, et ne sera donc pas détectée. La duplication de la variable utilisée directement dans la condition de sortie n'est alors pas suffisante.

D'autres problèmes peuvent également survenir lorsque le flot de contrôle de la boucle influence aussi le nombre d'itérations. L'exemple fourni dans le listing 3.4 montre une condition de sortie de boucle (`i>0`) dont la validité dépend des chemins exécutés lors des précédentes itérations (`i--` ou `i-=2`). En conséquence, le nombre d'itérations dépend non seulement de la condition de sortie, mais également des conditions internes (`i==5`) à chaque itération. Cet exemple met en évidence la nécessité de sécuriser une partie du flot de contrôle dans le corps de la boucle : celui qui influence une des conditions de sortie. Ce dernier exemple ne contient pas de protection additionnelle car, au vu des dernières remarques, il n'est pas trivial de savoir comment sécuriser manuellement une telle boucle. Tous les exemples présentés ont montré leur inefficacité potentielle après compilation.

3.2.5 Vers un algorithme dédié aux boucles

Les exemples précédents montrent qu'un schéma de protection des boucles nécessite une analyse du flot de données et du flot de contrôle qui influencent une condition de sortie d'une boucle et que l'application manuelle de protection sur des boucles complexes s'avère délicate, voire non réaliste.

```c
int r = a%b, r' = a%b;
while (r != 0) {
    a = b;
    b = r;
    r = a%b, r' = a%b;
}
if (r' != 0) error();
```

Listing 3.3: Induction complexe

```c
int i = n;
while (i > 0) {
    if (i == 5) {
        i -= 2; /* ... */
    } else {
        i--; /* ... */
    }
}
```

Listing 3.4: Induction conditionelle

Nous avons cherché à répondre au besoin de protection des boucles en concevant un schéma générique de sécurisation et des algorithmes permettant de le déployer à la compilation. Le principe de sécurisation repose sur la duplication de toutes les instructions du corps de boucle qui influencent directement ou indirectement une condition de sortie. Des vérifications des conditions de sortie, recalculées à partir de condition dupliquées, sont effectuées à chaque rebouclage ou lors de sortie de boucle afin de détecter toute attaque en faute pendant l'exécution de la boucle qui pourrait changer le nombre d'itérations ou forcer une mauvaise sortie.

3.3 Algorithme de sécurisation des boucles : PLAF

Dans cette section, nous allons présenter l'algorithme de sécurisation des boucles PLAF que nous avons conçu. L'algorithme décrit est dédié à une représentation intermédiaire (IR) bas niveau de compilateur. Afin de simplifier les différents schémas, on suppose de plus que cette IR est indépendante de la cible et est en forme SSA.

3.3.1 Prérequis

Le graphe de flot de contrôle (CFG) d'une fonction est découpée en *blocs de base* qui sont une suite linéaire d'instructions qui se termine par un saut. Au niveau IR, ce saut peut prendre la forme d'un retour de fonction, d'un branchement conditionnel, inconditionnel ou d'un *switch*.

Le CFG peut prendre des formes très diverses en fonction de l'application, du langage de programmation utilisé ou encore du flot de compilation. Comme la plupart des optimisations de code visant des boucles, notre algorithme de sécurisation proposé opère sur une forme normalisée des boucles d'une fonction construite par analyse du flot de contrôle, comme une forêt de boucles imbriquées [Ramalingam, 1999].

Cette forme normalisée de boucle dite *boucle naturelle* est une composante fortement connexe d'un graphe qui ne possède qu'un bloc d'entrée, appelé l'*entête*, et un unique arc arrière qui retourne à cet entête. Le bloc source de cet arc arrière est nommé *latch*. Le corps de la boucle

est constitué des blocs desquels il existe un chemin menant à l'entête en passant par l'arc arrière. Le(s) bloc(s) de la boucle possédant un successeur hors de la boucle sont appelés *blocs sortants*, et un tel successeur est appelé *bloc de sortie*. Un bloc sortant étant dans la boucle, il possède également par définition un successeur dans la boucle et donc au moins deux successeurs au total. Sa dernière instruction est donc soit un branchement conditionnel, soit un *switch*. En cas de *switch*, un bloc sortant peut avoir comme successeurs plusieurs blocs de sortie. Inversement, un bloc de sortie peut avoir plusieurs prédécesseurs dans la boucle, et donc être le successeur de plusieurs blocs sortants.

Ces différentes notions de bloc de boucle sont résumées dans la figure 3.1 qui présente un exemple (à gauche) de boucle possédant deux blocs sortants e_A^{in} et e_B^{in}. Le premier se termine par un branchement conditionnel alors que le deuxième se termine par un *switch* : il possède quatre successeurs dont deux blocs de sortie e_1^{out} et e_2^{out}. Afin de représenter les différents cas sur un seul exemple, le bloc e_2^{out} est un bloc de sortie commun aux deux blocs sortants.

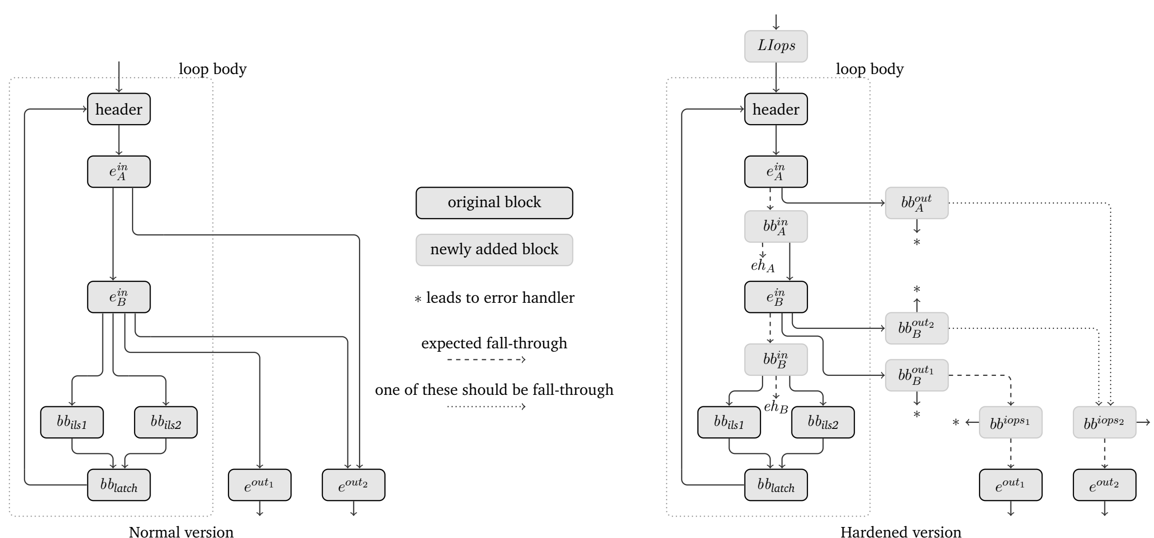


Figure 3.1: Exemple de boucle avec plusieurs blocs sortants et plusieurs blocs de sortie (à gauche), et sa version sécurisée (à droite)

3.3.2 Vue d'ensemble

L'algorithme générique proposé a pour but de renforcer la sécurité d'une boucle sensible contre toute altération du nombre d'itérations et/ou de son chemin de sortie. Le schéma est basé sur la duplication des instructions impliquées dans le calcul des différentes conditions de sorties de la boucle afin de vérifier la valeur de ces conditions à chaque itération et en sortie de boucle. En cas de différence, une fonction personnalisable de gestion d'erreur est appelée.

Les étapes d'analyses et de transformations nécessaires au renforcement d'une boucle sont décrites dans l'algorithme 1. Pour illustrer le résultat des modifications du flot de contrôle,

on considère l'exemple donné en figure 3.1 (à gauche). Le résultat de la sécurisation est représenté aussi (à droite).

Algorithm 1: Sécurisation des sorties d'une boucle

```
L ← processLoop(L)
    Input: Loop L
    Output: Secured loop L
1   LIops = []
2   foreach exiting block e^in of L do
3       ei = getJump(e^in)
4       ci = getCondInst(e^in)
5       [slice, LBr, LIops] = getBackSlice(L, ci, LIops)
6       dupInsts(L, instToDup)
7       bb_in = newBlock([ci, ei])
8       bb_eh = newBlock([call errorHandler])
9       updateCFGin(L, e^in, bb_in, eh)
10      secureBranches(L, LBr)
11      foreach exit block e^out successor of e^in do
12          bb_out = newBlock(ei)
13          updateExitCondCFGout(L, e^out, eh, bb_out)
14  bb_iops = dupLoopInvOps(L, LIops)
15  foreach exit block e^out do
16      updateIOpsCFGout(L, e^out, eh, dupBB(bb_iops))
```

L'algorithme de traitement d'une boucle commence par analyser chaque bloc sortant e^{in} et procède aux modifications associées. Dans un second temps, des modifications sont apportées aux blocs de sortie e^{out} en fonction de l'analyse de l'ensemble des blocs sortants.

Pour chaque bloc sortant e^{in}, l'algorithme identifie d'abord ei son instruction de saut ainsi que ci l'instruction qui définit la condition de ce saut. Partant de cette instruction ci, la seconde étape est de déterminer l'ensemble *slice* des instructions impliquées dans le calcul de cette condition par une analyse arrière de dépendance, ainsi que l'ensemble *LBr* des branchements définissant, dans la boucle, des chemins d'exécution influençant également le calcul de cette condition ci. Ces branchements sont sensibles car ils peuvent induire un chemin faussant la valeur de la condition (cf. exemple au niveau source 3.4). Par conséquent, ils sont aussi à sécuriser et une analyse arrière de dépendance partant de chacun d'entre eux inclut dans l'ensemble *slice* les instructions dont ils dépendent. Lors de ces analyses de dépendances, certains opérandes des instructions rencontrées sont indépendants de la boucle. Ceux-ci sont stockés dans l'ensemble *LIops* dédié.

Ensuite, a lieu la phase de transformation pour le bloc e^{in} : toutes les instructions de l'ensemble *slice* sont dupliquées. Un bloc de base bb_{in} contenant l'instruction ci dupliquée est inséré dans la boucle derrière le bloc e^{in}. Il devient l'un des successeurs de e^{in} et permet de vérifier que la direction prise en sortie de e^{in} est la bonne (à savoir, rester dans la boucle). Un autre bloc bb_{eh} pour la gestion d'erreurs lui est également ajouté comme successeur. La sécurisation des branchements sensibles, contenus dans *LBr* et associés à e^{in}, est assurée par la fonction *secureBranches*. De plus, pour chaque bloc de sortie e^{out} associé à e^{in}, un bloc

supplémentaire bb_{out} vérifie grâce à la condition dupliquée que la sortie de la boucle devait bien avoir lieu. Il est inséré dans le CFG de la fonction comme prédécesseur de e^{out} par la fonction *updateExitCondCFGout*.

Enfin, tous les opérandes invariants de boucle stockés dans LIops sont gérés par la fonction *duplicateLoopInvOps* dont le fonctionnement est le suivant : ces opérandes sont sauvegardés sur la pile avant la première itération de la boucle, dans un bloc dominant l'entête (le bloc $LIops$ dans la figure 3.1). Pour chaque bloc de sortie e^{out}, un nouveau bloc bb_{iops} est créé et placé comme prédécesseur du bloc e^{out}. Dans ces blocs, les valeurs de ces invariants de boucle sont comparées aux valeurs sauvegardées en mémoire et une unique vérification globale finale s'assure qu'aucune valeur n'a été corrompue pendant l'exécution de la boucle.

3.3.3 Détails des algorithmes

Cette partie est consacrée à la présentation détaillée des différentes étapes de l'algorithme principal.

3.3.3.1 Analyse arrière de dépendance

Le rôle de l'analyse arrière de dépendances réalisée par la fonction *getBackwardSlice* est triple. Pour un bloc sortant e^{in}, elle détermine l'ensemble des instructions, l'ensemble des branchements et l'ensemble des variables indépendantes de la boucle influençant la condition de sortie du bloc e^{in}. À partir d'une condition de sortie, l'algorithme 2 décrit comment déterminer ces différents ensembles en un seul parcours.

L'analyse part de l'instruction passée en paramètre en l'incluant dans une liste de travail (*ligne 2*). Cette liste de travail est dédiée à ne recevoir que des instructions de la boucle définissant un opérande influençant la condition de sortie. À chaque étape, une instruction est retirée de la liste de travail et est ajoutée à la liste *slice* si elle n'y est pas déjà présente (*ligne 6*). L'analyse remonte les chaînes de dépendances *use-def* en parcourant les différents opérandes de l'instruction en cours d'analyse (*lignes 17 à 22*). Si l'opérande est indépendant de la boucle, il est ajouté à l'ensemble *LIops* (*lignes 18 à 20*). Sinon, les instructions définissant ces opérandes sont à leur tour ajoutées à la liste de travail. L'analyse ne nécessite qu'un seul parcours des instructions, ainsi, lorsqu'une instruction retirée de la liste est déjà présente dans l'ensemble *slice*, l'analyse arrière ne l'analyse pas afin de ne pas boucler à l'infini (*ligne 5*).

Un cas particulier à prendre en compte est le cas des nœuds ϕ. En forme SSA, chaque variable est définie une seule fois. Les nœuds ϕ sont utilisés dans les blocs jonctions de plusieurs chemins lorsqu'une variable vivante jusqu'à un tel bloc a été définie sur au moins l'un des chemins. Ils permettent de définir quelle valeur utiliser en fonction du chemin suivi et d'assigner le résultat à une nouvelle variable. C'est grâce à cette propriété que la forme SSA

Algorithm 2: Analyse arrière de dépendance et invariants de boucle

```
[slice, LBr, LIops] ← getBackSlice(L, start, LIops)
    Input: Loop L ; Instruction start ; Loop-invariant operand list LIops
    Output: List of instructions slice ; List of branches to secure LBr ; List of loop-invariant
            operands LIops
1   worklist = [start] ; slice = [] ; LBr = []
2   while worklist is not empty do
3       cur = dequeue(worklist)
4       if cur ∈ slice then continue
5       append(slice, cur)
6       if isPhiNode(cur) then
7           phiBB = getBB(cur)
8           foreach pair (Pred_i, Pred_j) ∈ inLoopPredsOf(phiBB) do
9               if valFrom(cur, Pred_i) = valFrom(cur, Pred_j) then
10                  continue
11              ancs = relevantCommonAncestors(Pred_i, Pred_j)
12              foreach a ∈ ancs do
13                  ei = getJump(a)
14                  appendIfNotListed(LBr, ei)
15                  appendIfNotListed(worklist, getCondInst(ei))
16      foreach operand next ∈ Operands(cur) do
17          if next is loop-independent then
18              appendIfNotListed(LIops, next)
19              continue
20          inst = getDefinitionInstOf(next)
21          if inst ∈ L then append(worklist, inst)
22  return [slice, LBr, LIops]
```

peut exister malgré des flots de contrôle non linéaires. De plus, la définition d'une variable domine ainsi nécessairement toutes ses utilisations. De cette façon et contrairement aux autres instructions, les nœuds ϕ possèdent, en plus de leurs opérandes, une information supplémentaire sur les prédécesseurs ayant produit les opérandes, donc une information implicite sur le flot de contrôle entre la définition de l'opérande et le nœud ϕ. Cette information supplémentaire doit être prise en compte afin de sécuriser les instructions dont la condition dépend.

Une instruction de la liste de travail peut être un nœud ϕ qui définit une variable opérande d'une instruction impliquée dans le calcul de la condition d'un branchement ou d'une sortie. Pour mieux visualiser ce cas, considérons l'exemple de la figure 3.2. Dans cet exemple, on suppose que seuls les blocs nommés *header*, *if3*, *then3* et e^{in} contiennent des instructions impliquées dans le calcul de la condition de sortie de e^{in}. Plusieurs chemins permettent d'arriver au bloc e^{in}. En particulier, le bloc *then3* contient la définition d'un opérande influençant la condition. Ainsi, à cause de cette représentation en forme SSA, un nœud ϕ est nécessairement présent dans le bloc e^{in} et contenu dans la slice [3].

3. Sinon, la définition du bloc *then3* ne dominerait pas son utilisation dans le bloc e^{in}

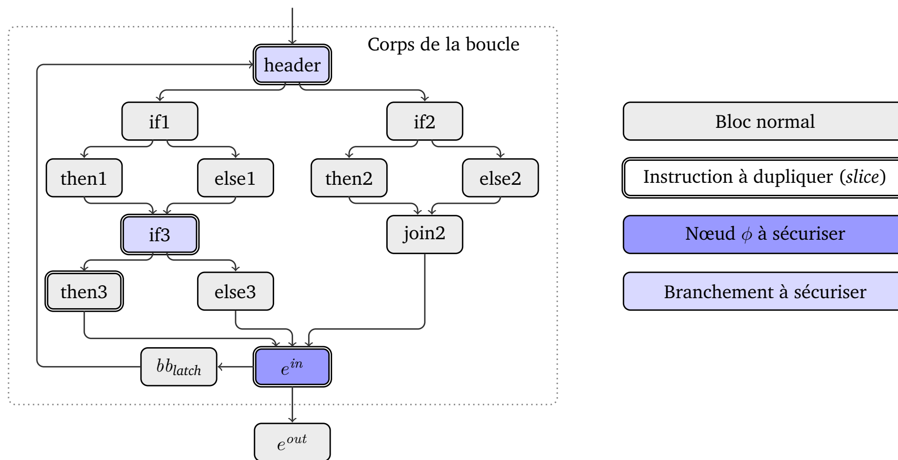


Figure 3.2: Schéma présentant les blocs du CFG contenant des branchements à sécuriser en fonction d'un nœud ϕ

Pour garantir que la bonne valeur est choisie par un tel nœud ϕ, il est nécessaire de s'assurer que le chemin exécuté pour arriver au bloc e^{in} est celui sélectionnant cette bonne valeur. Afin d'assurer que le programme vient d'un prédécesseur correct, il est possible de sécuriser tous les branchements qui permettent d'atteindre le bloc e^{in} depuis son dominateur immédiat (le bloc *header* dans cet exemple). Bien que cela soit suffisant de sécuriser tous ces branchements, cela n'est pas nécessaire et potentiellement très coûteux. En fait, deux cas de branchements n'ont pas besoin d'être sécurisés. Le premier cas concerne les branchements des blocs dont tous les chemins issus passent par le même prédécesseur du nœud ϕ. Dans l'exemple, c'est le cas du bloc *if2* dont tous les successeurs mènent au bloc *join2*. Le second cas concerne les blocs dont les successeurs mènent à des chemins qui assignent la même valeur à la variable définie par le nœud ϕ. La sortie de tels blocs n'a alors pas d'influence sur la valeur sélectionnée au nœud ϕ. Ainsi, seuls les branchements des ancêtres communs des paires de prédécesseurs d'un nœud ϕ qui le nourrissent avec des valeurs différentes doivent être considérés.

De plus, parmi ces branchements, ceux des blocs post-dominés par un autre ancêtre commun n'ont pas besoin d'être sécurisés non plus : c'est la sortie de cet ancêtre commun qui influencera le chemin suivi jusqu'au nœud ϕ. Dans l'exemple présenté, le bloc *if1* est post-dominé par le bloc *if3*. Son branchement n'a donc pas besoin d'être sécurisé lors du traitement du nœud ϕ du bloc e^{in}. En effet, supposons qu'une faute affectant le branchement du bloc if1 impacte la valeur prise par le nœud ϕ de e^{in}. Comme tous les chemins sortants du bloc *if1* passent par le bloc *if3*, la condition de sortie du bloc *if3* dépend nécessairement du chemin pris en sortie de *if1*, la faute n'aurait sinon pas d'impact. Dans un tel cas, la condition du branchement du bloc *if3* dépend d'un nœud ϕ de ce dernier dont la valeur est sélectionnée en fonction du chemin suivi en sortie du bloc *if1*. Ainsi, sécuriser la sortie du bloc *if3* est suffisant lors du traitement du nœud ϕ de e^{in}, car itérativement, le nœud ϕ dans if3 sera sécurisé. De façon plus générale, les branchements à sécuriser sont déterminés récursivement en cas de nœud ϕ rencontré par les analyses arrières partant des branchements déjà analysés. La sortie du bloc *if1* sera, le cas échéant, sécurisée lors du traitement du nœud ϕ de *if3*.

Finalement, afin de déterminer l'ensemble des branchements impactant la valeur d'un nœud ϕ, il faut considérer chaque paire de ses blocs prédécesseurs définissant des valeurs différentes (*ligne 8*) puis, ajouter à la liste *LBr* des branchements à sécuriser (*ligne 14*) ceux des blocs qui

1. appartiennent aux chemins partant du bloc ancêtre commun le plus proche des deux prédécesseurs arrivant jusqu'au nœud ϕ ;
2. ne sont pas post-dominés par un autre ancêtre commun de ces deux prédécesseurs.

La fonction *relevantCommonAncestors* (*ligne 11*) est en charge de déterminer les blocs contenant ces branchements à sécuriser.

De la même façon que pour protéger une sortie de boucle, la protection d'un branchement implique de dupliquer toutes les instructions impliquées dans le calcul de sa condition et d'insérer des blocs de contrôle dans chaque direction possible pour vérifier qu'elle est la bonne. Ainsi, l'instruction définissant la condition de chaque branchement à sécuriser est ajoutée à la liste de travail (*ligne 15*) afin de continuer l'analyse arrière depuis cette instruction.

3.3.3.2 Duplication des instructions

En ce qui concerne la duplication des instructions en pratique, l'ensemble *slice* des instructions à dupliquer est trié dans un ordre topologique afin de préserver les dépendances de type *def-use* lors du traitement séquentiel des instructions. Chaque boucle est traitée séparément par un appel à l'algorithme 1, dans lequel chaque bloc sortant d'une boucle donnée est également traité séparément. En revanche, des instructions peuvent apparaitre dans les ensembles *slices* associés à plusieurs conditions de sortie ou lors de la sécurisation précédente d'une boucle interne ou externe selon l'ordre de traitement des boucles. Une attention particulière est portée lors de la duplication afin d'éviter de multiples duplications de la même instruction.

Les instructions dupliquées sont placées juste avant l'instruction originale correspondante dans le corps de la boucle. De plus, les opérandes des instructions dupliquées sont mis à jour avec les opérandes dupliqués correspondants qui sont conservés dans une liste de renommage, construisant ainsi un sous-graphe de flot de données entièrement indépendant de celui composé des données originales. Cette étape traite donc de la duplication de toutes les variables non invariantes pour la boucle nécessaire à la sécurisation des conditions de sortie.

3.3.3.3 Gestion des opérandes invariants

Lors des analyses arrière effectuées pour la sécurisation d'une boucle, les opérandes invariants pour cette boucle sont rangés dans la liste *LIops*. Les copies des instructions utilisant ces opérandes utilisent le même opérande que l'instruction originale. La sécurisation doit pourtant

garantir que si la corruption de l'un de ces opérandes mène à une sortie de boucle corrompue, elle doit être détectée.

Pour cela, l'algorithme utilise une autre méthode que la duplication de ces opérandes afin de limiter l'augmentation du nombre de variables vivantes et donc la pression sur les registres. De plus, les passes d'optimisation du *back-end* du compilateur sont enclines à simplifier ce type de duplication sans analyse globale du flot de données. La solution choisie consiste à stocker chacun de ces opérandes invariants, dans un emplacement dédié en mémoire, dans un bloc dominant la boucle et laisser les copies des instructions utiliser le même opérande invariant. La détection de corruption de l'un de ces opérandes est réalisée en sortie de boucle par l'ajout d'un nouveau bloc nommé bb_{iops}. Dans ce bloc, chaque valeur copiée en mémoire est relue et est comparée avec la valeur utilisée dans la boucle à l'aide d'une opération `xor`. L'ensemble des résultats de ces comparaisons est également combiné à l'aide de `xor` afin de limiter le contrôle de ces invariants à un unique branchement basé sur la comparaison du résultat final à zéro. Un tel bloc doit être exécuté le long de chaque chemin de sortie de boucle possible. De plus, il doit contenir la vérification des opérandes invariants de la boucle rencontrés lors des analyses arrières réalisées pour toutes les sorties de la boucle.

En effet, la corruption d'un opérande invariant peut permettre à l'exécution de quitter la boucle par une autre sortie que celle prévue, même si cette dernière ne dépend pas de la variable corrompue. Considérons l'exemple présenté dans le listing 3.5 où la boucle possède deux sorties. La première sortie est définie via l'expression `k<n` et dépend de `n` comme unique opérande invariant. La deuxième sortie est définie via l'expression `break` et dépend à la fois de `A` et de `B` comme opérandes invariants, mais pas de `n`. Si cette fonction est appelée pour vérifier que deux zones mémoire sont différentes, l'exécution doit sortir de la boucle par le `break`. Si, dans ce cas, une corruption du pointeur mémoire `A` (prenant la valeur de `B`) conduit à une comparaison de deux zones mémoire identiques, la sortie de la boucle se fera par la première sortie (`k<n`). Afin de détecter une telle corruption, il faut donc bien vérifier à la première sortie qu'en plus de `n`, les pointeurs `A` et `B` n'ont pas été corrompus.

```
int compare(int *A, int *B, int n) {
   int k, diff = -1;
   for (k=0; k<n; k++) {
      if (A[k] != B[k]) {
         diff = k;
         break;
      }
   }
   return diff;
}
```

Listing 3.5: Boucle avec deux sorties

De plus, lors de la sécurisation d'un nid de boucles, les opérandes invariants peuvent être communs à plusieurs boucles. Il n'est pas pour autant nécessaire de sauvegarder un tel opérande invariant plusieurs fois. Il suffit de réaliser la sauvegarde dans un bloc dominant

toutes les boucles qui l'utilisent, idéalement dans le bloc qui définit cet opérande, juste après sa définition. Dans le cas d'un opérande pas défini dans la fonction (paramètre d'une fonction par exemple), il peut être sauvegardé à l'entrée de la fonction. Cette solution, consistant à sauvegarder la valeur de l'opérande invariant le plus près possible de sa définition, conduit à réduire au maximum la surface d'attaque de cet opérande. Elle est donc également utilisée dans le cas d'une boucle simple.

3.3.3.4 Mise à jour du CFG intra-boucle

Pour chaque bloc sortant e^{in}, deux blocs additionnels sont créés. Un nouveau bloc bb_{in} contient l'instruction de saut *ei* dupliquée, qui est mise à jour pour que chacun de ses successeurs en dehors de la boucle soit remplacé par un unique nouveau bloc bb_{eh} de gestion des erreurs. Un tel bloc bb_{in} est inséré dans le CFG entre e^{in} et chacun de ses successeurs dans la boucle, potentiellement plusieurs en cas de *switch*. Ce bloc détectera si une faute a permis de rester dans la boucle.

L'ordre de placement de ces différents blocs doit être géré avec précaution. Lors d'un saut d'instruction visant un branchement, l'exécution continue dans le bloc positionné juste derrière séquentiellement (*fall-through block*). Pour assurer la détection d'une telle faute, le bloc bb_{eh} doit être placé juste derrière le bloc bb_{in} qui doit lui même être placé juste derrière le bloc e^{in}. En effet, rien n'assure que le bloc e^{in} est initialement suivi par un bloc de la boucle. Ainsi, sauter le branchement de e^{in} amène au calcul de la condition dupliquée de bb_{in}. De même, sauter le branchement de bb_{in} amène directement au gestionnaire d'erreur de bb_{eh}.

3.3.3.5 Sécurisation des branchements internes

Pour chaque successeur bb_s d'un branchement *br* de la liste LBr, une copie du branchement, utilisant la condition dupliquée, est insérée dans un nouveau bloc de vérification. Ce bloc est ensuite inséré dans le CFG comme successeur de l'arc sortant du branchement *br* vers ce successeur bb_s de telle sorte que bb_s soit placé séquentiellement après ce nouveau bloc. Le branchement original est donc modifié pour pointer vers ce nouveau bloc et tous les successeurs du branchement dupliqué excepté bb_s sont remplacés par le bloc de gestion d'erreur bb_{eh} précédemment créé. Si un saut d'instruction vise le branchement *br*, l'exécution continuera dans le bloc de vérification associé au successeur séquentiel du bloc contenant le branchement initial et une faute sera ainsi détectée si elle induisait un mauvais chemin pris.

3.3.3.6 Mise à jour finale du CFG

Lors de l'étape de sécurisation de chaque bloc de sortie e^{out} associé à un bloc sortant e^{in}, un nouveau bloc bb_{out} est créé. De la même façon que précédemment pour les blocs bb_{in} créés et insérés entre e^{in} et ses successeurs dans la boucle, le bloc bb_{out} contient l'instruction *ei*

dupliquée. A l'inverse, cette fois, tous les successeurs dans la boucle sont remplacés par le bloc de gestion des erreurs. La fonction *updateExitCondCFGout* ajoute au CFG le bloc bb_{out} comme cible du saut de fin du bloc e^{in} à la place de e^{out}.

De plus, le bloc bb_{iops} créé pour la vérification des opérandes invariants doit être exécuté avant d'arriver dans chaque bloc de sortie e^{out}. Pour chaque bloc de sortie e^{out}, la fonction *updateIOpsCFGout* insère une copie du bloc bb_{iops} dans le CFG en remplaçant les arcs sortant de tous les blocs bb_{out} menant à cette sortie e^{out} afin que bb_{iops} devienne le successeur de tous ces blocs bb_{out} et qu'il soit le prédécesseur de e^{out} et de bb_{eh} en cas de détection de corruption de l'un des opérandes invariant (Algorithm 1 *ligne 15*).

Comme lors de l'ajout des blocs dans la boucle, le placement des blocs en sortie de boucle est important. Afin d'éviter d'ajouter un nouveau bloc de gestion d'erreur à chaque sortie, tous les blocs bb_{out} (créés pour un ou plusieurs blocs e^{in}) doivent être placés séquentiellement avant le bloc bb_{iops}. De plus, ce placement permet également d'éviter d'avoir du code inconnu entre les blocs de vérification. Le bloc de gestion d'erreur créé pour la détection d'erreurs en bb_{in} peut être utilisé. Ce placement assure que si le branchement d'un bloc de vérification n'est pas exécuté suite à un saut d'instruction, l'exécution ne pourra pas continuer dans une partie de code imprévisible avec un effet inconnu sur la boucle. En cas de faute sur le branchement d'un bloc bb_{out}, l'exécution continue soit dans un autre bloc bb_{out} soit dans un bloc bb_{iops}. En cas de faute sur le branchement d'un bloc bb_{iops}, l'exécution continue dans le bloc e^{out}. Dans tous les cas, la faute n'aura pas d'impact sur le nombre d'itérations de la boucle.

L'ajout et le placement de ces différents blocs de contrôle sont présentés dans la figure 3.1 (à droite). Les lignes en pointillés indiquent les blocs placés séquentiellement.

3.3.3.7 Ordre de traitement des boucles

En présence de nid de boucles, l'ordre de traitement des boucles peut avoir un impact sur le surcoût total de la sécurisation. Lorsque deux boucles d'un même nid possèdent un bloc sortant commun (souvent le cas lors d'une instruction `return`), l'ensemble des instructions calculé par l'analyse arrière de la boucle interne est nécessairement contenu dans celui calculé pour la boucle externe. Aussi, des opérandes peuvent être invariants pour la boucle interne mais ne pas l'être pour la boucle externe. Par conséquent, sécuriser la sortie de la boucle externe est suffisant pour sécuriser en même temps cette sortie pour la boucle interne.

L'algorithme traite donc les boucles les plus externes en premier et marque au passage les blocs sortant déjà sécurisés afin de ne pas chercher à les sécuriser plusieurs fois. En revanche, si le bloc sortant se termine par une instruction `switch`, un traitement particulier est nécessaire afin de rajouter correctement tous les blocs de vérification. Le schéma de sécurisation doit en effet être adapté lorsqu'un successeur bb_s du bloc sortant est dans la boucle externe mais en dehors de la boucle interne. Pour rappel, en cas de `switch`, un unique

bloc de vérification bb_{in} est inséré pour l'ensemble des successeurs qui appartiennent à la boucle alors que chaque successeur extérieur à la boucle est protégé par l'ajout de blocs bb_{out} et bb_{iops} propres. C'est pourquoi dans le cas précédent, le successeur bb_s étant un bloc de sortie de la boucle interne, il doit avoir ses blocs de vérification dédiés. Lors du traitement de la boucle externe, le schéma de sécurisation traite donc séparément ces cas particuliers des autres successeurs qui sont à la fois contenus dans la boucle interne et externe.

3.3.3.8 Limitations du schéma de duplication

Le schéma de sécurisation est basé sur la duplication des instructions rencontrées lors des analyses arrière effectuées pour les différentes sorties de la boucle à sécuriser. Cependant, certaines instructions ne peuvent être dupliquées sans éviter de potentiels effets de bord. Par exemple, les accès en mémoire volatile ainsi que les appels de fonction sans information sur la fonction appelée ne peuvent être dupliqués. Par conséquent, lorsque l'analyse arrière rencontre une instruction non duplicable, le bloc sortant correspondant ne peut pas être protégé par l'algorithme.

Pour informer le développeur de l'impossibilité de sécuriser la boucle, l'algorithme émet des avertissements pour tous les blocs sortants ne pouvant être protégés. Le développeur peut ainsi investiguer si certains de ces blocs sortants ne sont pas sensibles. Certaines sorties prématurées de boucles sont en effet souvent utilisées à des fins de gestion d'erreur et ne sont pas nécessairement sensibles même si la boucle elle-même l'est. Le développeur peut aussi modifier son code pour que la protection puisse être appliquée, voire sécuriser la sortie correspondante manuellement en cas d'impossibilité de sécurisation automatique en ayant conscience des modifications liées au compilateur vues au début de ce chapitre.

Un algorithme alternatif peut également être utilisé pour sécuriser partiellement la sortie correspondante. L'analyse arrière s'arrête alors lorsqu'une instruction non duplicable est rencontrée. Dans ce cas, la duplication est limitée aux premières instructions rencontrées, permettant ainsi de detecter une corruption visant l'une de ces instructions mais restant limitée à cette sous-partie du flot de données. En effet, une faute qui modifie la valeur d'une instruction qui aurait été rencontrée ultérieurement par l'analyse arrière et qui aurait donc dû être dupliquée se répercutera sur les deux flot de données et ne sera donc pas detectée.

3.4 Analyse sécuritaire

Dans cette partie, on analyse la robustesse du schéma de protection proposé en présence de faute unique de type saut d'instruction ou corruption de registre général (hors registre `pc`) pendant l'exécution d'une boucle sensible. On raisonne au niveau IR en considérant des instructions IR et leurs opérandes comme registres à corrompre.

Pour rappel, le schéma de protection est basé sur la détermination des différentes instructions impliquées dans le calcul de la condition de sortie et la duplication de ces instructions afin de créer un graphe de flot de données indépendant du graphe original, menant à un double calcul de cette condition de sortie. Les opérandes invariants sont quant à eux sauvegardés en mémoire et sont communs aux deux graphes de flot de données. Les possibles fautes affectant la sortie d'une boucle sont détectées via l'ajout des différents blocs de vérification : entre le bloc sortant et son successeur séquentiel, ainsi qu'avant chaque successeur en dehors de la boucle afin de vérifier que la sortie devait bien être prise. De plus, avant chaque sortie, l'intégrité des opérandes invariants est également vérifiée. Les fautes affectant les branchements internes à la boucle sont également détectées par l'ajout de blocs de vérification placés sur les différents chemins issus de ces branchements.

L'analyse sécuritaire doit prendre en compte les deux modèles de fautes considérés à savoir le saut d'une instruction et la corruption de registre (autre que le registre `pc`). Dans le cas où une instruction possède un registre destination, sauter cette instruction revient à corrompre son registre de destination. Si l'instruction modifie également les drapeaux et que ceux-ci sont vivants après l'instruction, la corruption peut également affecter ces drapeaux. Cette corruption ne peut atteindre les deux graphes de flot de données car l'instruction n'appartient qu'à un seul à la fois. Pour cette raison, seuls les sauts d'instruction concernant des instructions sans registre de destination sont à étudier. Les seules instructions à considérer sont donc les instructions de branchement car les instructions impliquées dans les conditions calculées par l'analyse arrière ne peuvent comprendre d'écriture mémoire.

En effet, l'analyse arrière remonte les chaînes *def-use*, mais pas les dépendances de type *read-after-write*. Lorsqu'une condition de sortie dépend de données précédemment écrites en mémoire dans la boucle, il est possible qu'une des valeurs relues en mémoire affectant à la fois la condition originale ainsi que la condition dupliquée qui en dépend soit corrompue. Se protéger contre de telles fautes nécessiterait d'ajouter à l'analyse arrière les variables écrites en mémoire à des adresses potentiellement lues dans la boucle et qui impactent une condition de sortie. Une vérification de chacune de ces écritures mémoires devrait alors être ajoutée afin d'étendre l'algorithme de sécurisation. Cependant, cela peut devenir particulièrement coûteux et nous n'avons pas rencontré de telles dépendances lors de nos expérimentations. Ces cas particuliers se produisent notamment lorsque la compilation s'effectue sans optimisation (`-O0`), ce qui est rarement réalisé en pratique pour des raisons de performance et de taille de code. Ce point est même un avantage du schéma de protection à la compilation contrairement à la protection de code source qui nécessite souvent de ne pas optimiser le code lors de la compilation [Dureuil et al., 2016a].

3.4.1 Saut d'instruction

Comme indiqué précédemment, seules les instructions de branchement sont à analyser. Plusieurs cas peuvent se présenter mais tous ont comme point commun de faire continuer le flot d'exécution dans le bloc placé séquentiellement juste derrière le bloc du branchement fauté :

- Le saut d'une instruction de branchement interne dupliquée n'aura pas d'effet sur la propriété de sécurité car le flot d'exécution continuera dans le bloc initialement cible du saut original dont la direction n'était pas fautée.
- Le saut d'une instruction de branchement d'un bloc bb_{in} ajouté forcera le flot d'exécution à atterrir dans le bloc de gestion d'erreur.
- Sauter le branchement d'un bloc bb_{out} ou bb_{iops} amène à continuer le flot d'exécution en dehors de la boucle dans le bloc e^{out}, ce qui était déjà le cas et n'a donc pas d'impact sur le nombre d'itérations. Ces blocs n'étant pas dans la boucle, ils ne font pas à proprement parler des cas considérés par le modèle de faute, mais sont toutefois ajoutés par la contre-mesure et méritaient donc d'être précisés. De plus, en cas de nid de boucles, si bb_{out} appartient à une boucle externe, alors e^{out} aussi, la faute n'a donc pas d'impact sur la sécurité de la boucle externe.
- Le saut d'une instruction de branchement interne originale fera continuer l'exécution dans l'un des blocs bb_s de vérification ajoutés. S'il correspond au bon successeur, la faute n'aura pas d'impact. S'il correspond au mauvais successeur, le bloc de vérification constatera une différence et la faute sera ainsi détectée.
- De la même façon, sauter l'instruction de branchement originale du bloc sortant de la boucle aboutit à l'exécution d'un des blocs de vérification ajouté. Dans le cas où le chemin est le bon, la faute n'a pas d'impact, alors que dans l'autre cas, la faute est détectée par ce bloc.

Il se peut également que le corps de la boucle contienne des branchements inconditionnels. Si un tel branchement est sauté, l'exécution continue dans le bloc placé séquentiellement juste derrière, bloc qui peut ne pas faire partie du corps de la boucle, en particulier dans le cas d'un bloc *latch* qui n'est pas bloc de sortie de la boucle. Une telle sortie de boucle prématurée corrompt le nombre d'itérations de la boucle. Afin de sécuriser ces branchements inconditionnels, il suffit de les dupliquer. Les branchements inconditionnels dupliqués de cette façon sont considérés par le compilateur comme code mort. Pour garantir la sécurité avec le schéma proposé, il faut donc les dupliquer après toute passe du flot de compilation capable de supprimer du code mort.

3.4.2 Corruption de registre

Dans le cas où la faute se produit sur un registre contenant un opérande invariant, les deux conditions de sortie de boucle, originale et dupliquée, sont impactées. Toute sortie de la

boucle passe par un bloc bb_{iops} en charge de la validité de ces opérandes. La corruption sera donc détectée à ce moment, que l'opérande invariant intervienne dans le calcul direct de la condition de sortie ou dans le calcul d'un branchement interne à la boucle.

Sinon, la faute corrompt la valeur d'une variable utilisée dans l'un des graphes de flot de données du calcul d'une condition de sortie. En cas de corruption d'une variable du flot de données dupliqué, le flot de données initial n'est pas corrompu grâce à l'indépendance des deux flots. Dans ce cas, la boucle exécute le bon nombre d'itérations sauf si la faute est détectée par la condition dupliquée erronée.

Si la faute vise une variable impliquée dans le flot de données initial du calcul d'une condition de sortie, alors elle peut engendrer une itération supplémentaire qui sera détectée par le bloc de vérification bb_{in} à l'intérieur de la boucle portant sur la condition dupliquée non affectée. Si elle engendre une sortie prématurée, c'est la vérification par le bloc bb_{out} à l'extérieur de la boucle qui détectera la faute.

Lorsque la corruption vise un registre impliqué dans la condition d'un branchement interne de la boucle, alors, toujours par indépendance des graphes, la condition dupliquée n'est pas affectée. Si la direction prise n'est pas la bonne, cela sera détecté par le bloc de vérification ajouté sur le chemin pris. En revanche, si la condition dupliquée d'un tel branchement interne est corrompue, la faute peut être détectée ou pas mais elle n'aura de toute façon aucun impact sur le nombre d'itérations de la boucle : le bon chemin avait été pris.

Enfin, certaines fautes peuvent aboutir à une levée d'exception (division par zéro par exemple), ou rendre infinie l'exécution de la boucle (forcer à zéro l'incrément du compteur de boucle par exemple). Cela ne rentre pas dans un cas de violation de la propriété de sécurité étant donné qu'avec de telles conséquences, le système sera interrompu ou ne répondra plus. Ces problèmes ne sont donc pas directement gérés par la contre-mesure mais ils sont reportés à un niveau supérieur, par le traitement des exceptions ou grâce à une éventuelle utilisation de *timers* pour gérer des temps d'exécution anormalement longs.

3.5 Interactions avec le flot de compilation

Comme expliqué dans la section 2.5, l'ajout de contre-mesure au niveau du code source peut être mis en défaut par les optimisations du compilateur. Cette complication s'applique également à l'ajout de contre-mesure dans le compilateur selon le placement de la passe de sécurisation dans le flot de compilation, en fonction des optimisations ultérieures à la passe. En l'absence de support interne aux compilateurs permettant de raisonner sur la préservation systématique des propriétés de sécurité, nous avons opté pour une analyse pragmatique des passes de compilation et de leurs impacts potentiels sur la sécurisation des boucles. Une telle

approche a permis de déterminer une place précise dans le flot de compilation permettant à la fois de bénéficier des optimisations les plus agressives du compilateur existantes pour la taille de code ou la performance tout en générant du code contenant la sécurisation proposée.

Au niveau intermédiaire du *middle-end*, les informations portant sur le flot de contrôle ainsi que sur le flot de données sont disponibles. De plus, les modifications apportées à ce niveau sont à la fois indépendantes du langage source ainsi que de l'architecture cible visée. Nous avons donc choisi de privilégier une intégration du schéma de sécurité dans le compilateur à ce niveau intermédiaire, dans le but d'aboutir à plus long terme à la construction d'une chaîne de compilation optimisante orientée sécurité pour différentes cibles. Afin de déterminer une position idéale de la passe de sécurisation dans le flot de compilation à ce niveau intermédiaire, nous avons analysé les passes d'optimisation classiques d'un compilateur et avons fait le choix de LLVM comme exemple type.

3.5.1 Passes en amont

Certaines passes de compilation doivent être appliquées en amont de la passe de sécurisation comme les passes d'analyses et de transformations fournissant des informations et un formatage du code essentiels pour la sécurisation, comme la mise en forme des boucles par exemple. Il en est de même des passes qui interfèrent avec la sécurisation. D'autres passes, améliorant la qualité du code sécurisé résultant, ont également intérêt à être placées en amont. La suite présente les passes d'optimisations retenues selon leurs interactions avec la sécurisation. Enfin, le placement de la passe de sécurisation qui découle de cette analyse est détaillée dans le cas du compilateur LLVM.

3.5.1.1 Passes nécessaires

La passe nommée *register promotion* tente de remplacer les différents accès mémoire par des variables scalaires dans le but que ces dernières soient allouées dans des registres. Dans les boucles, l'importance d'éliminer de fausses dépendances mémoire est d'autant plus grande que cela permet également de convertir des dépendances inter-itération en dépendances explicites grâce à l'utilisation des nœuds ϕ. Cette transformation est très utile pour construire, par analyse arrière, un ensemble des instructions à dupliquer exempt d'accès mémoire ou avec un nombre limité. Cette passe doit donc être appliquée en amont de la passe de sécurisation.

3.5.1.2 Passes interférantes

Les passes d'élimination de redondance classiques comme l'élimination de sous-expression commune (CSE) ou la passe *Global Value Numbering* (GVN) ont pour objectif de simplifier voire d'éliminer les calculs et variables redondants. De telles passes ont donc une sérieuse capacité à altérer voire supprimer le code ajouté par les algorithmes de sécurisation. De

plus, en étant appliquées avant, ces passes peuvent être bénéfiques en réduisant le nombre d'instructions à dupliquer pour la sécurisation des boucles.

La passe de simplification des variables d'induction (*Induction Variable Substitution*) a pour but de déterminer un cycle d'induction canonique à partir duquel dériver les autres variables d'induction. Cette passe est capable de modifier les variables impliquées dans les conditions de sorties, mais également ces conditions elles-mêmes, et ainsi potentiellement simplifier les conditions de sorties dupliquées.

La réduction de force appliquée aux boucles (*Loop Strength Reduction* (LSR)) transforme les opérations coûteuses en opérations potentiellement moins coûteuses. Dans le cas des boucles, cela peut radicalement transformer les expressions intervenant dans le calcul des conditions de sortie en introduisant de nouvelles variables d'induction. Cette passe est ainsi capable de remplacer les variables d'inductions originales et dupliquées par une nouvelle qui ne sera donc pas sécurisée.

Toutes ces passes doivent être appliquées en amont des passes de sécurisation pour éviter de compromettre la robustesse du code final.

3.5.1.3 Passes bénéfiques

La passe de déplacement de code invariant dans les boucles (LICM) déplace dans des blocs dominants ou post-dominants d'une boucle les instructions invariantes de celle-ci. Le corps de la boucle est ainsi réduit et certaines de ces instructions auraient potentiellement pu faire partie des instructions à dupliquer. Cette passe a donc un effet positif sur le surcoût induit par la contre-mesure en termes de performance et de taille de code.

Dans certains compilateurs (dont GCC et LLVM), la forme SSA dans le cas des boucles est normalisée par une propriété connue sous le nom de *Loop-closed SSA*. Sous cette forme, des nœuds ϕ sont insérés à la sortie des boucles pour chaque variable définie dans la boucle et qui est utilisée en dehors de la boucle. Cette forme permet d'isoler du flot de contrôle externe les transformation de code dont la portée est limitée à une boucle. Comme l'algorithme de sécurisation des boucles modifie les sorties des boucles, cette passe a aussi un impact positif sur l'application de la contre-mesure en limitant l'impact en dehors de la boucle.

3.5.2 Placement dans le flot de compilation

Dans le compilateur LLVM considéré comme illustration de compilateur moderne, la passe de réduction de force (LSR) est celle appliquée en dernier dans le flot de compilation parmi toutes les passes citées dans la section précédente qui sont des transformations au niveau IR en forme SSA. Étant en partie dépendante de la cible, c'est une des rares passes au niveau IR

implémentée dans le *back-end*, juste avant le passage à une représentation du code plus proche des instructions machine. Nous avons donc fait le choix de placer notre passe de sécurisation des boucles au début du *back-end*, immédiatement après LSR et avant la génération de code (cf. figure 2.3).

De cette façon, la passe de sécurisation bénéficie des étapes de normalisation et simplification tout en évitant d'interférer avec les optimisations agressives du *middle-end*. En agissant au niveau IR, la passe est indépendante de la cible. Nous avons donc pu l'utiliser dans plusieurs *back-ends* (x86, ARM, propriétaire).

3.5.3 Passes en aval et ajustements nécessaires

Une fois le placement de la passe de sécurisation fixé dans le flot de compilation, il est nécessaire d'analyser les potentiels effets néfastes des passes d'optimisations appliquées en aval jusqu'à l'émission du code. En raison de la diversité des compilateurs et des *back-ends* dédiés aux différentes cibles, une telle analyse ne peut être réalisée de manière générique. En revanche, il est possible d'analyser les passes d'optimisation typiquement rencontrées après le remplacement de la forme SSA pour une représentation intermédiaire de plus bas niveau afin de pointer les transformations susceptibles d'interférer avec la sécurisation. Nous avons choisi le *back-end* ARM du compilateur LLVM afin de vérifier nos hypothèses sur un *back-end* concret. Nous avons ainsi pu constater que les transformations liées à l'élimination de redondance sur le code de bas niveau ou à la sélection d'instruction n'ont pas compromis la sécurisation.

Nous avons effectué une analyse plus poussée pour la sécurisation des boucles et il s'est avéré que certains ajustements sont toutefois nécessaires pour éviter toute interaction néfaste. Cette section présente les modifications nécessaires au niveau du *back-end* pour ne pas compromettre la sécurisation des boucles.

3.5.3.1 Placement de code

L'ordre de placement séquentiel des blocs de vérification insérés par la contre-mesure est vital pour la sécurité du code final comme expliqué dans les sections 3.3 et 3.4. Forcer un placement spécifique des blocs est réalisable en ajoutant des contraintes sur le CFG [Holloway and Smith, 2000].

Dans le *back-end*, une fois l'IR convertie en représentation bas niveau, le CFG est linéarisé pour obtenir un code séquentiel. Cette linéarisation peut introduire de nouveaux branchements inconditionnels invisibles au niveau IR. Ces nouveaux branchements inconditionnels sont susceptibles de faire partie du corps de boucles sensibles. Comme expliqué dans la section 3.4, il est nécessaire de les dupliquer pour éviter en cas de saut de l'instruction de branchement de tomber dans un bloc qui serait en dehors de la boucle. Ainsi, seuls les branchements

inconditionnels d'une boucle sensible dont le successeur séquentiel est en dehors de la boucle ont besoin d'être dupliqués. En l'absence d'information sur le placement des blocs, doubler tous les branchements inconditionnels apparaissant dans une boucle sécurisée permet à fortiori de garantir la robustesse souhaitée.

Dans le *back-end* ARM de LLVM, deux transformations peuvent modifier le placement des blocs de base : l'optimisation du flot de contrôle (*Control-flow Optimizer*) et le *Branch Folding*. Nous avons modifié cette dernière pour ne pas toucher aux blocs de vérification et ainsi garantir le bon placement séquentiel des blocs. La passe de *Control-flow Optimizer* a dû être désactivée car elle interférait avec le positionnement des blocs nécessaires pour garantir l'efficacité de la protection de boucle. Le surcoût induit est pris en compte dans les résultats donnés en section 3.6.5.

Enfin, une passe dédiée à la duplication des sauts inconditionnels a été ajoutée tout à la fin du *back-end*, avant l'émission du code, pour sécuriser les sauts inconditionnels d'une boucle précédant un bloc hors de la boucle.

3.5.3.2 Allocation de registres

L'allocateur de registres détermine les variables à conserver sur la pile en cas de pression sur les registres. Ces variables sont dites *spillées*. Celles qui sont uniquement en lecture seule dans une boucle sont des cibles de choix car elles n'ont pas besoin d'être écrites en mémoire après une ou plusieurs utilisations. Ainsi, lorsque la pression sur les registres est trop forte, une telle variable est lue depuis la pile avant son utilisation, mais, le registre contenant la variable n'est pas sauvegardé de nouveau après son utilisation. La variable est relue depuis la pile pour la ou les utilisations futures.

Une corruption du registre contenant la valeur lue peut se répandre sur les calculs dont elle dépend et, dans le cas d'un invariant d'une boucle sécurisée, est capable d'impacter à la fois les calculs originaux et dupliqués d'une condition de sortie. Dans le bloc de vérification des opérandes invariants, à cause d'un éventuel spill, la valeur à vérifier risque d'être relue depuis la pile au lieu de comparer directement le registre corrompu. La valeur relue de la pile étant bonne, la corruption ne peut être détectée dans ce cas.

Afin d'assurer la robustesse du schéma de protection, il est donc nécessaire de forcer une écriture en mémoire des variables indépendantes de la boucle *spillées* après toute utilisation. Une corruption d'une variable *spillée* lue sera répercutée sur l'emplacement en pile correspondant et pourra ainsi être détectée lors de la vérification en sortie.

3.5.3.3 Factorisation des constantes

Certaines constantes au niveau IR peuvent être placées dans un registre par l'allocateur de registres. En effet, selon la taille des opérandes immédiats, il est préférable, voire nécessaire, de mettre une constante dans un registre pour des raisons de contraintes de mode d'adressage. Par exemple, si l'on considère le jeu d'instructions ARMv7, l'expression `1<<x` en langage C, traduite par l'instruction `sll(#1, %x)` en IR LLVM n'a pas d'équivalent direct. Il faut placer la constante `1` dans un registre `Rm` et utiliser l'instruction `lsl(Rd, Rm, Rx)` correspondante du jeu d'instructions[4]. Si une telle constante apparait dans une instruction à dupliquer, le même registre va être utilisé pour cette constante dans l'instruction originale et dans son duplicata, créant un point de vulnérabilité. La corruption de ce registre affectera le code initial et le code dupliqué sans détection de faute possible. La solution à ce problème consiste à désactiver la factorisation de constantes identiques dans les boucles sécurisées. Dans le compilateur LLVM, cela doit être mis en œuvre au moment de la sélection d'instructions qui réalise cette optimisation.

3.5.3.4 Élimination d'expression commune

De la même manière que cité en Section 3.5.1.2, le *back-end* comporte également une passe d'élimination des expressions communes (*Machine CSE*). Moins agressive qu'au niveau IR, cette passe peut tout de même supprimer une instruction dupliquée si les deux versions ont des opérandes identiques. Le cas se présente notamment pour les lectures en mémoire.

Afin de contourner le problème, cette passe du *back-end* peut être désactivée. Une solution moins brutale est de qualifier de `volatile` les accès mémoires à dupliquer. Ainsi la passe *Machine CSE* peut toujours optimiser le code sans toucher à ces accès mémoire. De plus, cette solution est moins coûteuse que d'ajouter un qualificatif `volatile` directement à la variable sur le code source. En effet, en ajoutant le qualificatif au moment de la sécurisation dans le flot de compilation, toutes les optimisations agressives en amont ont pu être appliquées normalement. Notre choix s'est donc porté sur cette deuxième option.

3.5.3.5 Regroupement d'instructions

Le *back-end* est capable de regrouper plusieurs instructions IR en une seule instruction machine. Par exemple, le jeu d'instructions ARMv7 fournit une instruction de lecture mémoire double (`ldrd`[5]) qui permet de regrouper deux lectures en une seule instruction. En conséquence, sauter une instruction qui en regroupe plusieurs se traduit par une faute multiple au niveau IR. Un unique saut d'instruction est autorisé par le modèle de faute considéré mais cela n'est pas couvert par le schéma de protection. De tels regroupements d'instructions doivent donc être désactivés dès lors qu'ils comportent des instructions originales et dupliquées. En

4. http://infocenter.arm.com/help/topic/com.arm.doc.dui0204j/Cjacbgca.html
5. http://infocenter.arm.com/help/topic/com.arm.doc.dui0489h/CIHGJHED.html

effet, regrouper plusieurs instructions uniquement contenues dans le flot de données d'origine ou celui dupliqué n'affecte pas l'autre flot de données et donc la robustesse du schéma de protection est préservée. En pratique, aucun regroupement d'instructions s'avérant dangereux n'a été rencontré dans nos expérimentations.

3.5.4 Conclusion des modifications du back-end

Cette section montre qu'il n'est pas trivial de sécuriser au niveau IR. Selon le *back-end* ciblé, des ajustements de certaines passes sont nécessaires afin de conserver la sécurité introduite au niveau IR. Cumuler une passe principale au niveau IR et des ajustements dans le *back-end* est donc le prix à payer pour obtenir une sécurité générique.

3.6 Expérimentations et résultats

Cette section présente la mise en oeuvre de ce schéma de protection dans l'infrastructure de compilation LLVM ainsi que les résultats des expérimentations associées.

3.6.1 Implémentation dans LLVM

L'algorithme PLAF a été initialement implémenté en tant que passe de compilation dans le compilateur LLVM version 4.0. Les résultats présentés dans cette section correspondent à cette implémentation. Par la suite, elle a été adaptée pour être compatible avec la version 6.0 de LLVM. La passe, désactivée par défaut, s'active via l'ajout d'une option `-loop-hardening=all` à l'appel du driver de compilation `clang`. La passe a pu être validée fonctionnellement grâce à l'utilisation de trois *back-ends* : ARMv7m/Thumb2, ARMv7a et x86_64, comme présenté dans cette section.

3.6.1.1 Sélection des boucles à sécuriser

Toutes les boucles d'une application ne sont pas sensibles. Il n'est donc pas nécessaire d'appliquer la sécurisation à toutes les boucles d'un code et ainsi éviter un surcoût inutile. L'activation de la passe s'accompagne, pour cette raison, de la possibilité de choisir les boucles à sécuriser. L'utilisation de l'attribut de fonction `__attribute__((annotate("SecLoops")))` limite l'application de la sécurisation uniquement aux boucles des fonctions ainsi annotées. Pour gagner encore en précision, il est possible de sécuriser une boucle en particulier en insérant un appel de fonction intrinsèque immédiatement avant la boucle à sécuriser (à la manière d'un `#pragma`). L'utilisation de l'option `-loop-hardening=tagged-only` à l'appel du driver de compilation permet d'activer la passe avec ces précisions.

3.6.1.2 Limitations et avertissements à l'utilisateur

Comme évoqué dans la section 3.3.3.8, la présence de certaines instructions dans l'ensemble déterminé par l'analyse arrière de dépendance empêche la sécurisation de la sortie correspondante. La version actuelle de l'algorithme fonctionne sur un principe de liste blanche indiquant les instructions duplicables et non de liste noire. Par conséquent, les instructions arithmétiques, les instructions de transtypage, les nœuds ϕ et les lectures mémoire non volatiles peuvent être dupliquées. À l'inverse, si une autre instruction apparait lors de l'analyse arrière et que la sortie de boucle correspondante ne peut être sécurisée, un avertissement est émis pour informer l'utilisateur que la protection de cette sortie n'a pas pu être déployée. Les avertissements de compilation sont traditionnellement émis par le *front-end* du compilateur et sont liés au code source. Avertir lors d'optimisations au niveau IR est nécessairement moins explicite pour l'utilisateur. En revanche, l'avertissement gagne en précision en présence d'information de *debug* où l'instruction non duplicable peut être associée à une déclaration au niveau source.

3.6.1.3 Visualisation du code généré sur exemples simples

Cette section donne des exemples de code assembleur ARMv7 de boucle avec et sans sécurisation. Ces codes ont été obtenus en activant le niveau d'optimisation `-Os` du compilateur, avec ou sans la passe de protection des boucles. Le premier exemple est une boucle qui compare deux tableaux en temps constant, donné dans le listing 3.6. Le code assembleur présenté dans le listing 3.7 est la version originale du code généré pour cette fonction.

```
int compare(int *A, int *B, int size) {
    int k, diff=0;
    for (k=0; k<size; k++)
        if (A[k] != B[k])
            diff++; /* Constant time - no early exit */
    return diff;
}
```

Listing 3.6: Comparaison de tableaux en temps constant

Le code présenté en bleu correspond à la traduction au niveau assembleur des instructions IR impliquées dans la condition de sortie de la boucle, et déterminées par l'algorithme. La version sécurisée de ce même code est présenté dans le listing 3.8, dans lequel le code rouge correspond aux instructions dupliquées du calcul de la condition de sortie ainsi que la vérification intra-boucle de cette condition. Le code orange correspond au bloc de vérification de la condition en dehors de la boucle, tandis que le code vert correspond au contrôle des opérandes invariants.

Les listings 3.9 et 3.10 montrent les versions originales et sécurisées du code présenté précédemment dans le listing 3.4. D'une part, les instructions en bleu correspondent à la sortie de la boucle ainsi qu'à la sécurisation associée. D'autre part, les instructions en rouge

```
.entry1:
    push {r4, r5, r7, lr}

    mov lr, r0
    movs r0, #0
    cmp r2, #1

    blt .end
    movs r3, #0

.for.body:
    ldr.w r4, [r1, r3, lsl #2]
    ldr.w r5, [lr, r3, lsl #2]
    adds r3, #1

    cmp r5, r4
    it ne
    addne r0, #1
    cmp r3, r2
    blt .while.body










.end:

    pop {r4, r5, r7, pc}
```

Listing 3.7: Code assembleur original de l'exemple du listing 3.6

```
.entry1:
    push {r4, r5, r6, lr}
    sub sp, #8
    mov lr, r0
    movs r0, #0
    cmp r2, #1
    str r2, [sp, #4]
    blt .end
    movs r4, #0
    movs r3, #0
.for.body:
    ldr.w r5, [r1, r4, lsl #2]
    ldr.w r6, [lr, r4, lsl #2]
    adds r4, #1
    adds r3, #1
    cmp r6, r5
    it ne
    addne r0, #1
    cmp r4, r2
    bge .exit
    cmp r3, r2                      /* bb_in */
    blt .while.body
.errorHandler:
    bl ErrorHandler
.exit:
    cmp r3, r2                      /* bb_out */
    blt .errorHandler
    ldr r6, [sp, #4]               /* bb_iops */
    teq.w r2, r6
    bne .errorHandler
.end:
    add sp, #8
    pop {r4, r5, r6, pc}
```

Listing 3.8: Version sécurisée correspondante de l'exemple du listing 3.6

correspondent au branchement interne de la boucle à sécuriser ainsi qu'à sa sécurisation associée également. L'alignement des deux codes permet une vision plus claire des instructions et de leur duplicatas.

3.6.2 Environnement expérimental

Les prochaines sections présentent les expérimentations ainsi que les résultats obtenus. Le but de ces expérimentations est d'évaluer l'automatisation de la sécurisation d'un bout à l'autre du flot de compilation. Pour cela, les objectifs sont multiples. Le premier objectif est de s'assurer de la non altération fonctionnelle du code généré. Les expérimentations servent ensuite à évaluer la sécurité du code protégé ainsi que l'estimation du surcoût induit en performance et en taille de code. Il est également possible d'évaluer la capacité à sécuriser automatiquement tout type de boucles.

Les expériences décrites dans les prochaines sections ont été réalisées en activant les passes de sécurisation. La sécurisation des boucles a été activée pour toutes les boucles rencontrées indépendamment de leur sensibilité. Cela permet de mettre en avant la généricité des schémas

```
.entry2:
    push {r4, lr}

    mov r4, r0
    cmp r4, #1
    it lt
    poplt {r4, pc}


.while.body:
    cmp r4, #5
    bne .if.else



.if.then:
    /* ...then... */
    movs r4, #3

    b .while.cond







.if.else:
    /* ...else... */
    subs r4, #1

.while.cond:
    cmp r4, #0
    bgt .while.body











.end

    pop {r4, pc}
```

Listing 3.9: Code assembleur de l'exemple du listing 3.4

```
.entry2:
    push {r4, r5, r6, lr}
    sub sp, #8
    mov r4, r0
    cmp r4, #1
    str r4, [sp, #4]
    blt .end
    mov r5, r4
    mov r6, r4
.while.body:
    cmp r5, #5
    bne .bb.else
.bb.then:
    cmp r6, #5
    bne .error
.if.then:
    /* ...then... */
    movs r5, #3
    movs r6, #3
    b .while.cond
    b .while.cond
.bb.else:
    cmp r6, #5
    bne .if.else
.error:
    bl errorHandler
.if.else:
    /* ...else... */
    subs r6, #1
    subs r5, #1
.while.cond:
    cmp r5, #1
    blt .bb$_{out}$
    cmp r6, #0                      /* bb_in */
    bgt .while.body
    b .error
    b .error
.bb_out:
    cmp r6, #0                      /* bb_out */
    bgt .error
    ldr r6, [sp, #4]                /* bb_iops */
    teq.w r4, r6
    bne .error
.end:
    add sp, #8
    pop {r4, r5, r6, pc}
```

Listing 3.10: Version sécurisée correspondante de l'exemple du listing 3.4

de protection, et d'établir une estimation de la borne supérieure du surcoût des contre-mesures. Dans la pratique, seules quelques boucles et arguments sensibles sont sécurisés, pouvant fortement diminuer les chiffres des surcoûts présentés.

3.6.2.1 Environnement logiciel

Pour l'évaluation de la sécurisation des boucles, nous avons choisi trois protocoles cryptographiques connus : de la cryptographie symétrique (`aes`), de la cryptographie à clé publique (`ecc`) et une fonction de hashage (`sha`). Ces codes de production contiennent des boucles sensibles qui peuvent faire fuir des informations secrètes si leur nombre d'itérations est altéré. Souvent embarqués dans des produits sensibles, ces codes illustrent des cas réels nécessitant d'être sécurisés contre les attaques physiques par les industriels.

En supplément, nous avons caractérisé la couverture de la contre-mesure sur une large variété de boucles provenant d'applications génériques et d'autres orientées sécurité. Nous avons donc sélectionné 15 codes spécifiques :

- `pgp` et `blowfish` provenant de la suite de tests MiBench [Guthaus et al., 2001] ;
- l'application de compression `gzip` [6] contenant de nombreuses boucles sur un code réduit ;
- trois applications contenant des codes cible typiques : `openssl`, `gmp` et `sqlite` [7] ;
- les tests de la suite SPEC CPU 2006 dédiés aux calculs sur nombres entiers qui sont écrits en langage C.

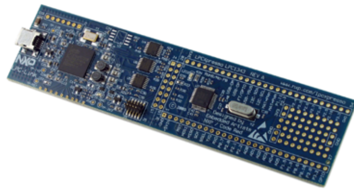
Les deux derniers cas ont été choisis pour stresser la passe de sécurisation dans les cas limites. En effet, ils contiennent de nombreuses boucles avec une large variété de flots de contrôle et de flots de données, en particulier dans les boucles.

3.6.2.2 Environnement matériel

La cible principale est un processeur 32-bit de la famille Cortex-M implémentant le jeu d'instructions ARMv7m/Thumb2, processeur typique de ceux largement déployés dans les systèmes embarqués. Les évaluations pour cette cible ont été réalisées sur une carte NXP (*LPCXpresso* 1343 rev. A) qui intègre un processeur ARM Cortex-M3. La seconde cible est un Raspberry Pi-3 embarquant un Cortex-A53 implémentant le jeu d'instructions ARMv7a sur lequel s'exécute le système d'exploitation Linux Raspbian [8]. Cette seconde cible a été choisie pour ses performances et mémoires bien supérieures au Cortex-M3 et ainsi être capable d'exécuter des codes plus complexes avec une vitesse de d'exécution raisonnable. Ces types de processeurs plus performants sont de plus en plus ciblés [Vasselle et al., 2017] comme nous le détaillons dans le chapitre 5. Sur la carte LPCXpresso, l'exécutable est chargé

6. `http://www.gzip.org`
7. `http://www.openssl.org`, `http://www.gmplib.org` et `http://www.sqlite.org`
8. `https://www.raspbian.org`

depuis un ordinateur via le port JTAG de la carte. Le contenu de la mémoire ainsi que les registres généraux peuvent aussi être récupérés à la fin d'une exécution. Les deux cibles sont représentées sur la Figure 3.3.



(a) Une carte NXP LPCXpresso 1343 implémentant un ARM Cortex-M3

(b) Un Raspberry Pi-3 implémentant un ARM Cortex-A53

Figure 3.3: Les deux plateformes embarquées cibles des évaluations des contre-mesures

Pour mettre en avant la portabilité, nous avons aussi rejoué les tests fonctionnels sur une plateforme x86_64 : un processeur Intel Core-i3 5010 sur lequel s'exécute le système d'exploitation Linux Debian 8. Enfin, nous avons compilé les applications avec tous les niveaux d'optimisation offerts par LLVM à savoir `-O1`, `-O2`, `-O3`, `-Os` et `-Oz`, ce dernier étant celui qui optimise de manière agressive la taille du code final.

3.6.3 Couverture de sécurisation

Afin de déterminer la capacité de l'algorithme à sécuriser tout type de boucles, nous avons analysé, pour chacune des applications précitées, le pourcentage de boucles automatiquement sécurisées. Les résultats, présentés en figure 3.4 sont répartis selon trois catégories : les boucles entièrement sécurisées dont toutes les sorties sont sécurisées, les boucles partiellement sécurisées dont au moins une des sorties a été sécurisée et enfin les boucles qui n'ont pas pu être sécurisées (aucune sortie).

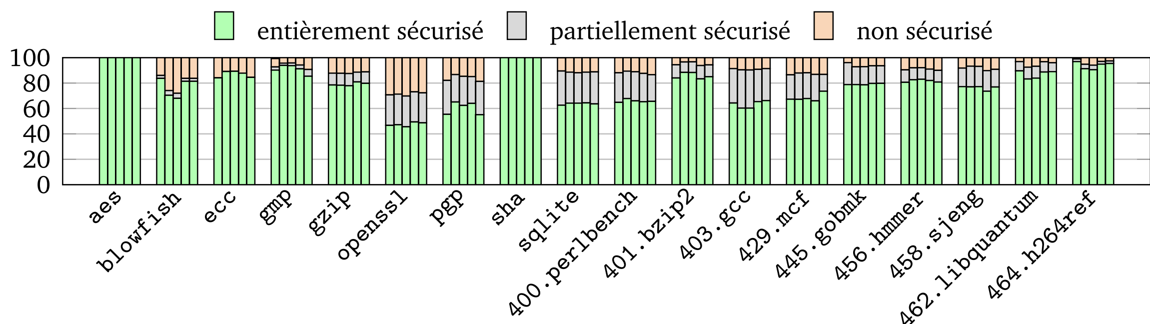


Figure 3.4: Répartition des boucles en fonction de la réussite de la sécurisation pour les options de compilation `-O1`, `-O2`, `-O3`, `-Os` et `-Oz`

En considérant l'ensemble des applications testées, le pourcentage de boucle sécurisées varie de 45% à 100% et ne descend jamais sous les 70% si l'on accepte une sécurisation partielle. Ces résultats récompensent notre effort à concevoir une passe de sécurisation générique capable de couvrir des corps de boucle avec des flots de contrôle complexes.

Nous nous sommes également intéressés aux causes de non sécurisation. En effet, lorsqu'une instruction qui doit être dupliquée ne peut pas l'être, la sortie de boucle correspondante ne peut alors être sécurisée. Cela se produit environ dans un cas sur deux pour `openssl`, `sqlite` et `403.gcc`. Approximativement 80% des instructions non duplicables sont des appels de fonction et les 20% restants sont des accès mémoire volatiles.

On peut noter qu'aucun schéma de protection logiciel basé sur de la duplication est capable de gérer ces cas d'accès mémoire non volatile [Barry et al., 2016, Moro et al., 2014, Reis et al., 2005]. En revanche, le nombre de cas dûs à des appels de fonction pourrait être réduit en utilisant des informations sémantiques sur les fonctions appelées. Lorsque celles-ci sont pures, l'appel pourrait par exemple être dupliqué si l'information est présente.

Une option est d'offrir à l'utilisateur la possibilité d'indiquer les fonctions ou appels de fonction qui sont duplicables. Actuellement, les avertissements émis par le compilateur permettent à l'utilisateur d'*inliner* les appels de fonction bloquant la sécurisation ou de changer le corps de ces boucles pour les rendre sécurisables.

3.6.4 Évaluation fonctionnelle

Avant d'évaluer la robustesse du schéma de sécurisation et son impact sur le code final, nous avons effectué une évaluation fonctionnelle pour s'assurer que le code produit était correct. La méthodologie, différente selon les applications, est décrite dans cette section.

Nous avons d'abord implémenté un gestionnaire d'erreur spécifique arrêtant l'exécution et indiquant par conséquent un nombre d'itérations erroné. Les trois applications cryptographiques (`aes`, `ecc` et `sha`) ont été validées grâce à une procédure de test dédiée couvrant les différentes options de ces librairies, où le résultat produit pour une entrée donnée est comparé à celui attendu, précalculé. Pour `gzip`, `pgp` et `blowfish`, nous avons compressé (resp. chiffré) puis décompressé (resp. déchiffré) un fichier de 420 MB. Le fichier final est alors comparé au fichier initial. Les librairies `openssl` et `gmp` possèdent des campagnes de validation auto-testantes que nous avons utilisées pour confirmer le bon fonctionnement de ces deux applications. De la même manière, toutes les applications de la suite SPEC CPU2006 ont pu profiter de l'infrastructure de validation offerte dans la suite. Enfin, pour `sqlite`, nous avons utilisé un script disponible en ligne sur le site de l'application [9].

Nous avons réalisé des tests fonctionnels pour les 5 principaux niveaux d'optimisation mentionnés précédemment. Aucune erreur n'a été reportée ni mise en évidence permettant de valider la bonne exécution des applications pour les scénarios de test considérés.

9. https://sqlite.org/speed.html

3.6.5 Coût de la contre-mesure

L'ensemble des applications présentées a été exécuté sur le Raspberry Pi-3. De part les performances et mémoire limitées de la carte à base de processeur ARM Cortex-M3, seuls les trois codes cryptographiques dédiés (`aes`, `ecc` et `sha`) ont été exécutés sur cette plateforme. Les autres applications ont, en revanche, été exécutées sur la plateforme x86_64 afin d'obtenir des résultats sur deux plateformes pour chacune des applications.

3.6.5.1 Impact en performance

L'impact sur les performances a été mesuré en exécutant les applications 10 fois et en moyennent le temps obtenu en excluant les valeurs extrêmes. La figure 3.5 contient des histogrammes représentant le ratio du temps d'exécution de la version sécurisée par rapport au temps d'exécution de référence de l'application non sécurisée pour les 5 niveaux d'optimisation.

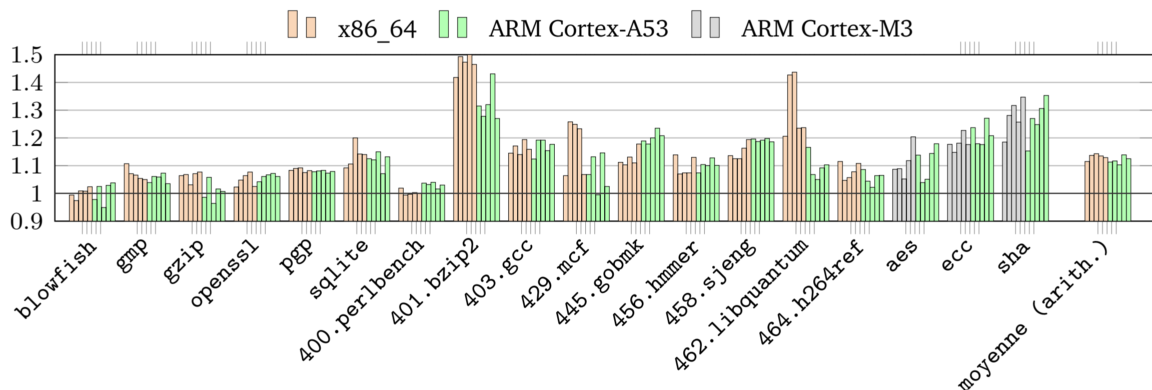


Figure 3.5: Performance relative du code sécurisé par rapport au code original (options de compilation `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`)

Le surcoût apporté est bien évidemment dépendant du temps passé dans les boucles, et est nécessairement plus élevé pour les corps de boucles plus petits. L'exemple de l'application `sha` composée de nombreuses petites boucles indique un surcoût élevé alors que l'application `blowfish` qui n'en contient que quelques grandes ne montre qu'un surcoût moindre (environ 1%). L'impact varie de nul à presque 50% selon les applications avec une moyenne autour des 12,5%. Cela reste très limité en comparaison des protections génériques au niveau instruction. Ces dernières, souvent basées sur de la duplication systématique de toutes les instructions, ont un surcoût en général supérieur à 100% [Barry et al., 2016, Moro et al., 2014, Reis et al., 2005]. S'ils sont appliqués qu'aux régions les plus sensibles du code, ces schémas peuvent apporter un surcoût moins excessif (limité à 40% pour un AES protégé contre les sauts d'instruction dans [Moro et al., 2014]), mais qui reste néanmoins supérieur à notre approche visant un objectif différent où les calculs hors des conditions de sorties ne sont pas protégés.

3.6.5.2 Impact sur la taille de code

La figure 3.6 illustre le ratio de la taille du code sécurisé par rapport à la taille de l'application non sécurisée. Pour le mesurer, nous avons additionné la taille des différents fichiers objets sécurisés.

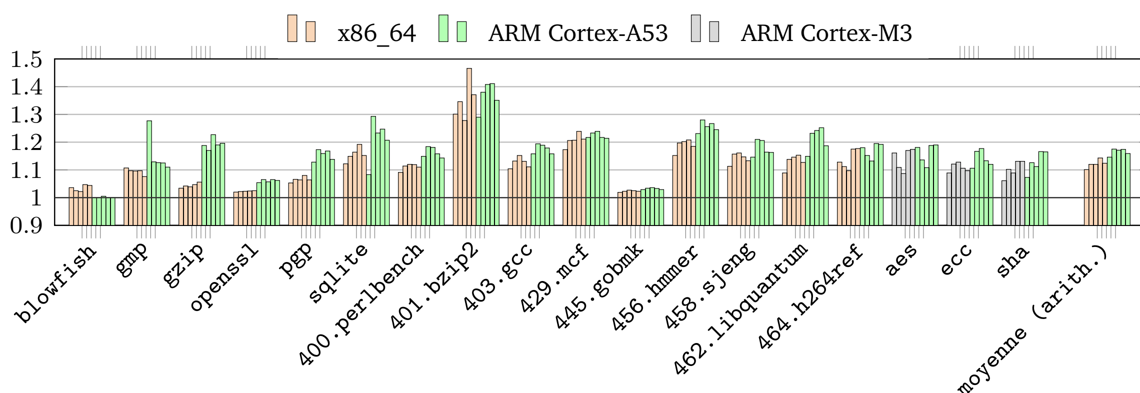


Figure 3.6: Taille relative du code sécurisé par rapport au code original (options de compilation `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`)

L'augmentation en taille varie de 1% à 43%. Elle est fortement liée à la proportion de boucles dans le code à sécuriser, ainsi qu'à la complexité des boucles présentes. Par exemple, `401.bzip2` possède de nombreux nids de boucles aux flots de contrôle complexes générant ainsi un surcoût plus élevé que pour les autres applications dû à la sécurisation des nombreux branchements internes. La moyenne est de 14%, ce qui, de nouveau, valide l'approche dédiées aux boucles lorsque la propriété de sécurité souhaitée est de garantir le nombre d'itérations et le bon chemin de sortie d'une boucle, et non ses calculs.

3.6.5.3 Impact sur le temps de compilation

Nous avons aussi mesuré l'impact de la passe de sécurisation sur le flot de compilation en mesurant la proportion du temps de compilation qu'elle représente. Les proportions du temps de compilation total de la passe de sécurisation ainsi que de différentes passes dédiées aux boucles sont présentées dans le tableau 3.1. Les valeurs ont été déterminées pour le niveau d'optimisation `-O2`.

Table 3.1: Temps dédié à différentes passes en pourcentage du temps de compilation total

Temps de compilation (%)	aes	blowfish	ecc	gmp	gzip	openssl	pgp	sha	sqlite	400.perlbench	401.bzip2	403.gcc	429.mcf	445.gobmk	456.hmmer	458.sjeng	462.libquantum	464.h264ref
Canonicalize natural loops	0.8	0.4	1.2	0.6	0.9	0.6	0.7	0.6	0.3	0.6	0.9	0.5	0.9	0.7	0.7	0.8	0.9	0.8
Loop-invariant code motion	1.8	2.5	2.1	1.8	4.0	1.8	3.4	1.7	3.7	1.7	4.0	2.4	4.1	2.8	3.1	3.1	2.1	3.9
Loop strength reduction	1.7	2.1	3.0	13.0	3.2	1.3	2.9	4.9	0.6	1.6	2.4	0.9	4.4	2.3	4.0	3.4	2.7	5.3
Loop hardening	**0.1**	**0.1**	**0.1**	**0.6**	**0.2**	**0.2**	**0.2**	**0.1**	**8.4**	**1.0**	**0.8**	**1.8**	**0.5**	**0.3**	**0.5**	**0.4**	**0.4**	**0.2**

Le temps passé dans la passe de sécurisation est négligeable par rapport au temps total de compilation pour l'ensemble des applications testées excepté pour `sqlite`. C'est un cas

particulier d'application dont le code source, écrit en C, est entièrement contenu dans un unique fichier de 7MB contenant 200 000 lignes de code. Cela offre davantage de possibilités d'optimisation pour les passes précédentes en particulier au niveau de l'*inlining* de fonctions, créant par conséquent de nombreuses boucles à la fois grandes et complexes (la plus grande étant répartie sur plus de 5 000 lignes de code). De plus, certaines boucles, liées à l'analyse de code SQL, possèdent des *switch* avec de nombreuses sorties (plus de 100) générant des boucles sécurisées excessivement complexes et nécessitant plus de travail de la part de la passe de sécurisation. L'implémentation actuelle recalcule, par simplicité, certains des sous-ensembles utilisés plutôt que de les sauvegarder. Une meilleure gestion des données déjà analysées pourrait faire chuter ce temps de compilation.

3.6.6 Évaluation sécuritaire de l'algorithme PLAF

Pour évaluer la robustesse des applications sécurisées face à des attaques par faute, nous avons développé un simulateur d'injections de faute basé sur le débugger GNU `gdb`, en exploitant ses capacités d'interaction avec des scripts en langage python. Le principe repose sur l'exécution en mode *debug* de l'application ciblée jusqu'à un point d'arrêt défini à l'adresse de l'instruction avant laquelle on souhaite simuler la faute. À ce point d'arrêt, une faute transitoire simulant un saut d'instruction ou une corruption de registre, conformément au modèle de faute prédéfini (voire section 3.2.1), est insérée avant de reprendre l'exécution.

Pour être capable d'analyser l'effet d'une faute, il faut connaître au préalable le nombre d'itérations attendu de chaque boucle ciblée. En effet, il ne suffit pas de comparer la sortie d'une exécution sans faute de celle d'une exécution avec faute : une faute peut cibler une instruction qui n'impacte ni le nombre d'itérations ni la sortie de la boucle mais qui peut avoir un impact sur les données en sortie. Une telle faute ne compromet donc pas la propriété de sécurité visée par notre schéma de sécurisation des boucles.

Afin de déterminer le nombre d'itérations attendu d'une boucle, nous avons instrumenté la passe de sécurisation avec l'ajout d'un compteur de boucle supplémentaire via une variable globale. Cette variable est incrémentée à chaque passage dans l'entête de la boucle (les boucles sécurisables n'ont qu'un seul arc arrière et un seul entête). De plus, l'ajout de cette variable aide à déterminer les plages d'adresses de la boucle sur le code assembleur généré. Le nombre d'itérations attendu est donc obtenu par une première exécution de l'application avec boucles instrumentées et sans injection de faute. L'évaluation de la sécurité se base sur la comparaison de cette valeur avec celle obtenue après l'injection d'une faute.

Le simulateur de fautes, dépendant de l'architecture visée, a été développé pour la cible ARM. Pour des raisons de temps de simulation, nous avons conduit ces évaluations uniquement sur les applications cryptographiques dédiées (`aes`, `ecc` et `sha`) capable d'être exécutées sur

les deux plateformes à base de processeurs ARM. La sécurité de chaque boucle qui a pu être sécurisée a été évaluée en considérant le code original et le code sécurisé.

Afin de couvrir au mieux l'ensemble des fautes possibles du modèle considéré, nous avons simulé des injections de faute en modifiant le moment d'injection d'une simulation à l'autre. Chaque instruction du corps de la boucle a été visée, à la première et dernière itération ainsi qu'à une itération intermédiaire. Pour le modèle de saut d'instruction, un unique saut est donc testé pour toutes les instructions de ces itérations possibles. Pour la corruption de registre, nous avons considéré la corruption de tous les registres généraux, de `r0` à `r12` ainsi que `lr`, avec comme modèle la mise à `0`, à `0xFFFFFFFF` ou à une valeur aléatoire. En effet, il est possible de générer des fautes provoquant des mises à 0 de bits [Roscian et al., 2013b], des mises à 1 de bits [Moro et al., 2013] ou des renversements de bits pouvant induire des valeurs aléatoires [Roscian et al., 2013a]. Toutes les valeurs ne peuvent être exhaustivement testées, une valeur aléatoire différente est donc utilisée pour chaque simulation d'injection de valeur aléatoire.

Le tableau 3.2 présente les résultats obtenus sur les boucles originales ainsi que celles sécurisées de chacune des trois applications considérées. Deux niveaux d'optimisations standard ont été utilisés (`-O2` et `-Oz`) générant des boucles avec des flots de contrôle différents. Par exemple, au niveau `-O2`, les boucles avec sortie unique ont régulièrement le bloc d'entête commme bloc de sortie, alors qu'au niveau `-Oz`, le bloc de sortie est plus souvent le *latch*.

Nous avons classé les fautes en 4 catégories selon les effets visibles obtenus. Elles peuvent être sans effet comme lors d'une corruption d'un registre mort, conduire à une erreur comme lors d'un accès mémoire à une adresse invalide, être détectée ou à l'inverse, provoquer une altération du nombre d'itérations sans être détectée en cas de boucle sécurisée.

L'ensemble de ces simulations a représenté un total de 615 000 injections de fautes sur les versions originales des boucles et plus de 900 000 sur leurs versions sécurisées, réparties sur 115 nids de boucles différents. La sécurisation d'une boucle augmentant la taille de son corps, cela explique le nombre plus important de fautes simulées sur les boucles sécurisées.

Les simulations d'injection de fautes sur les boucles originales ont mené à un total de 29 083 altérations de la propriété de sécurité définie. Seulement 454 fautes sont restées non détectées sur les boucles sécurisées, réduisant par conséquent le nombre d'attaques réussies de 98% malgré une augmentation de la taille des boucles et donc de la surface d'attaque.

Nous avons analysé manuellement les 454 fautes non détectées. Elles s'expliquent toutes par l'un des trois cas suivants tous des à des modifications du code par les passes du *back-end* appliquées après la sécurisation. Le premier cas concerne des lectures mémoire de variables globales. Au niveau IR, l'adresse d'une variable globale n'est pas disponible et ne peut donc pas être dupliquée. Si le registre qui contient cette adresse est corrompu, alors la lecture

Table 3.2: Résultats des simulations de faute ciblant 3 applications cryptographiques et pour chacune la version originale et celle sécurisée

				Saut d'instruction					Corruption de registre				
				Sans effet	Erreur	Détection	Altération	Total	Sans effet	Erreur	Détection	Altération	Total
Application originale	M3	`aes`	O2	399	79	0	55	533	15956	5519	0	911	22386
			Oz	675	108	0	92	875	27870	7253	0	1627	36750
		`ecc`	O2	798	104	0	115	1017	33052	8467	0	1195	42714
			Oz	1884	233	0	223	2340	38861	8658	0	1621	49140
		`sha`	O2	1245	113	0	366	1724	57778	8210	0	6420	72408
			Oz	1194	101	0	279	1574	51809	11926	0	4053	67788
	A53	`aes`	O2	434	57	0	58	549	15796	5005	0	610	21411
			Oz	538	62	0	65	665	19623	5425	0	887	25935
		`ecc`	O2	1641	230	0	259	2130	59959	20721	0	2390	83070
			Oz	461	136	0	129	726	18830	8510	0	974	28314
		`sha`	O2	1185	90	0	341	1616	48159	12576	0	2219	62954
			Oz	1514	64	0	618	2196	67318	14750	0	3576	85644
	Total			**11968**	**1377**	**0**	2600	**15945**	**455011**	**117020**	**0**	26483	**598514**
Application sécurisée	M3	`aes`	O2	533	93	100	1	727	22204	6386	2419	29	31038
			Oz	1641	235	249	0	2125	69040	13424	6435	14	88913
		`ecc`	O2	1028	106	302	8	1444	44694	8680	7020	45	60439
			Oz	1251	66	279	1	1597	51982	7375	5560	47	64964
		`sha`	O2	1625	202	489	0	2316	64268	16178	11063	160	91669
			Oz	1424	208	264	0	1896	57462	13468	5200	55	76185
	A53	`aes`	O2	583	102	122	2	809	18920	4607	2289	12	25828
			Oz	1373	90	225	0	1688	52028	8476	5314	30	65848
		`ecc`	O2	2105	284	539	4	2932	79695	23884	10662	23	114264
			Oz	1353	129	319	1	1802	51191	12728	6346	22	70287
		`sha`	O2	1661	106	560	0	2327	62744	17347	10631	0	90722
			Oz	2210	145	484	0	2839	84508	17954	8040	0	110502
	Total			**16787**	**1766**	**3932**	17	**22502**	**658736**	**150507**	**80979**	437	**890659**

mémoire et son duplicata seront affectés par la faute sans détection possible. En analysant le code source, une partie de ces cas peut être évitée en ajoutant une variable locale à la boucle qui permet de ne pas avoir recours à la lecture en mémoire à chaque itération. Le deuxième cas vient des opérandes immédiats qui ne peuvent être utilisés directement dans l'encodage de l'instruction mais qui nécessitent, à la place, l'utilisation d'un registre qui ne sera pas dupliqué. Le dernier cas provient de la mise en pile d'opérandes invariants pour la boucle. Une valeur corrompue d'un tel opérande peut être utilisée dans la boucle, impactant une condition de sortie, sans être détectée par le bloc bb_{iops} en sortie à cause de la relecture depuis la pile de la valeur non corrompue.

Ces trois cas sont couverts par les explications fournies dans les sections 3.5.3.2 et 3.5.3.3 qui proposent des solutions basées sur la modification du *back-end* afin qu'il n'altère pas la robustesse de la protection. Ainsi, on peut affirmer que le schéma de sécurisation mis au point et son implémentation sont efficaces pour protéger les boucles avec un taux d'élimination de faute de 97% qui pourrait atteindre 100% si les modifications du backend nécessaires présentées dans la section 3.5.3 étaient toutes implémentées.

3.7 Discussion sur une généralisation aux branchements

Le schéma de sécurisation des boucles proposé repose sur la sécurisation des sorties de la boucle. Ces sorties sont représentées par des branchements permettant de quitter la boucle. De plus, en sécurisant une sortie de boucle, il est possible, selon son flot de contrôle interne, d'avoir à sécuriser un ou plusieurs branchements dans la boucle, comme présenté dans la section 3.3.3.1. Ce schéma de sécurisation a donc pour principal objectif de sécuriser certains branchements définis par les analyses arrières. Cette section se propose de discuter de la généralisation de ce principe pour sécuriser un branchement quelconque.

Dans la littérature, plusieurs attaques visant à contourner des mécanismes d'authentification sont basées sur la corruption d'un branchement sensible [Dureuil et al., 2016a, Timmers and Spruyt, 2016]. Ces attaques ne visent pas forcément le branchement directement mais les données permettant de corrompre sa condition. Sécuriser seulement l'instruction de branchement comme dans le schéma de [Moro, 2014], en rendant l'instruction équivalente à une suite d'instructions idempotentes facilement duplicables, ne suffit donc pas forcément.

Pour sécuriser un branchement quelconque, les différences avec ce qui est présenté pour les boucles dans ce chapitre sont les suivantes :

- Sans boucle, la condition d'arrêt de l'algorithme 2 d'analyse arrière de dépendance n'est plus valable et doit être redéfinie. En effet, lors de la sécurisation d'une boucle, toutes les instructions à dupliquer font partie de la boucle. Cette limite d'action au niveau d'une boucle n'est pas généralisable et nécessite de considérer la fonction dans son ensemble.
- Sans boucle, il n'y a pas de notion d'opérande invariant de boucle. Les arguments de la fonction sont toutefois comparables à ces opérandes invariants.
- Pour un branchement quelconque, tous les successeurs sont équivalents, il n'y a pas de notion de successeur appartenant ou non à la boucle traités différemment.

La sécurisation d'un branchement quelconque peut se faire de manière similaire à la sécurisation d'un branchement interne de la boucle présentée dans la section 3.3.3.5 par les étapes suivantes : une analyse arrière de dépendances part de la condition de ce branchement, comme pour les boucles, pour déterminer les instructions participant au calcul de cette condition. L'algorithme 2 est adapté pour continuer l'analyse arrière sans être limité à une boucle (condition à la *ligne 21*), et ainsi remonter plus haut dans le corps de la fonction, jusqu'à l'entrée de la fonction. Lorsqu'un nœud ϕ est rencontré, les branchements impactant la valeur prise par ce nœud ϕ doivent être protégés également de la même manière que les branchements internes d'une boucle. Toutes les instructions ainsi rencontrées sont alors dupliquées. Les blocs de vérification décrits dans la section 3.3.3.5 sont ajoutés entre le bloc des branchements à sécuriser et chacun de ses successeurs. Ceci permet de détecter une erreur

dans l'un des flots de données de la même manière que pour les branchements internes d'une boucle.

En prenant en compte ces différents éléments, cela fournit un schéma capable de sécuriser tout type de branchement conditionnel. En pratique, l'analyse arrière non limitée à une boucle a de fortes chances de se terminer par des accès mémoire ou des arguments de la fonction. Les arguments ne peuvent pas être dupliqués par un tel schéma au niveau IR. Une action possible est de les traiter de la même manière que les opérandes invariants de boucle : on peut sauvegarder leur valeur sur la pile en entrée de fonction et ajouter un bloc de vérification de leur valeur en fin de fonction, de la même façon que dans le bloc bb_{iops}, présenté dans la section 3.3.3.3, qui gère les opérandes invariants. Une autre action possible consiste à agir au niveau du *back-end* pour pouvoir dupliquer les arguments concernés en les copiant dans un second registre dès l'entrée de la fonction. Ces registres contenant les valeurs copiées peuvent alors être utilisés dans le flot de données constitué des instructions dupliquées. L'intervalle de vie de ces registres impose une forte pression sur les registres à l'allocateur de registres, la première option est donc préférable.

Cette variante de l'algorithme PLAF sécurise les branchements conditionnels mais ne se contente pas de sécuriser l'instruction du branchement en elle-même. Elle apporte une sécurisation du flot de contrôle qui tient également compte du flot de données afin d'assurer que le chemin suivi soit le bon dans un contexte d'exécution donné. Les schémas existants d'intégrité du flot de contrôle cherchent le plus souvent à garantir que le chemin suivi à l'exécution est valide par rapport au CFG mais ne tiennent pas compte du contexte d'exécution. Le schéma proposé dans cette section peut être comparé à une version plus légère et ciblée de schémas génériques de duplication comme [Reis et al., 2005]. Ce schéma a été implémenté comme preuve de concept dans le compilateur LLVM, dérivé de l'algorithme PLAF et utilisant le même placement dans le flot de compilation. L'algorithme n'a toutefois pas fait l'objet d'une étude détaillée.

3.8 Conclusion

Dans ce chapitre, nous avons présenté la conception d'un schéma générique de sécurisation des boucles, applicable quelque soit la complexité du flot de contrôle du corps de la boucle. Ce schéma offre une protection contre des attaques par faute qui peuvent être modélisées soit par un saut d'instruction, soit par une corruption de registre. Cette protection garantit l'exécution du bon nombre d'itérations de la boucle ainsi que la sortie de boucle attendue. Elle est basée sur la duplication sélective d'instructions et d'opérandes et sur l'insertion de blocs de vérifications à la fois à l'intérieur et à l'extérieur de la boucle. Pour parvenir à assurer la propriété de sécurité, nous avons constaté qu'il était non seulement nécessaire de sécuriser le

flot de données menant au calcul des conditions de sortie, mais également le flot de contrôle interne dont elles dépendent.

Afin d'appliquer ce schéma de sécurisation de manière automatique, nous avons conçu un algorithme à implémenter au niveau intermédiaire (IR) d'un compilateur. La principale limitation du schéma de protection actuel est l'impossibilité de sécuriser une sortie de boucle lorsqu'un appel de fonction intervient dans le calcul de la condition de cette sortie. Cette limitation est signalée à l'utilisateur, ce qui lui permet de le prendre en compte et éventuellement modifier le code en conséquence. Dans certains codes, il n'est en revanche pas toujours possible de modifier le code source de telle sorte qu'un tel appel de fonction disparaisse. Toutefois, pour les boucles qui ont été sécurisées, le schéma a montré sa capacité à protéger le code contre les fautes dont les effets correspondent au modèle défini tout en ayant un impact mesuré sur les performances et la taille du code. Dans le prochain chapitre, nous proposons un schéma de sécurisation de l'arbre d'appels de fonction et nous discuterons de son utilité en combinaison avec la protection des sorties de boucle dont la condition dépend d'un appel de fonction.

Nous avons fait le choix de l'implémenter dans le compilateur LLVM. De la même manière qu'un ajout de contre-mesure au niveau du code source peut être compromis par les optimisations du compilateur, nous avons constaté que l'ajout d'une contre-mesure à l'intérieur même de ce compilateur nécessite d'ajuster certaines passes d'optimisations afin de ne pas en perdre le bénéfice. Cela montre à quel point la sécurisation de code, reposant sur l'ajout de code non fonctionnel, nécessite de revoir et d'adapter le flot de compilation pour qu'il soit plus orienté sécurité, c'est-à-dire qu'il puisse prendre en compte ce code et ne pas dégrader sa fonction de protection.

Lors des analyses des interactions avec les passes d'optimisation du compilateur, nous avons mis en évidence l'importance de l'adéquation du modèle de faute avec celui de la représentation du code. Un saut d'instruction au niveau assembleur peut en effet correspondre à deux sauts d'instructions au niveau IR. Les différentes traductions et modifications du code apportées par les étapes majeures de la compilation rendent particulièrement délicats les liens entre les modèles de fautes aux différents niveaux.

4 Sécurisation partielle du graphe d'appel

Sommaire

4.1 Introduction **73**
4.2 Motivations **74**
4.2.1 Attaques sur le graphe d'appel 75
4.2.2 Schémas de protection existants 76
4.2.3 Modèle d'attaquant 77
4.3 Schéma de suivi de l'arbre d'appel : *CalleeSimo* **78**
4.3.1 Principe du schéma 78
4.3.2 Algorithmes principaux 80
4.3.3 Suivi des fonctions 83
4.3.4 Protection des arguments 86
4.3.5 Intégration dans le flot de compilation 94
4.3.6 Limites du schéma de protection 95
4.4 Analyse sécuritaire pour architecture ARM **96**
4.4.1 Protection de l'arbre d'appel 96
4.4.2 Protection des arguments 98
4.5 Implémentation et résultats **98**
4.5.1 Implémentation dans LLVM 99
4.5.2 Environnement expérimental 100
4.5.3 Évaluation fonctionnelle 100
4.5.4 Coût de la contre-mesure 101
4.6 Discussion sur la complémentarité avec la sécurisation des boucles **103**
4.7 Conclusion **104**

4.1 Introduction

Dans le chapitre précédent, nous avons présenté la conception d'une première contre-mesure automatiquement appliquée à la compilation et visant des boucles, parties souvent sensibles de codes système ou cryptographiques. Cette protection automatisée répond simultanément à une problématique de sécurisation, de performances, de taille de code, et de coût pour les ingénieurs, tout en se limitant à la protection de ces parties sensibles.

Au delà, le suivi du bon enchainement des fonctions avec les bons paramètres est un élément important, notamment lors du démarrage des plateformes (*secure boot*), de leur mise à jour (*firmware update*), ou lors de protocoles d'authentification ou cryptographiques. En pratique dans l'industrie, des mécanismes de traçage sont souvent utilisés pour pouvoir suivre ce bon enchainement. Ces mécanismes sont souvent ajoutés manuellement, afin d'être adaptés au code ciblé comme, par exemple, compter le nombre d'appels aux sous-fonctions d'un chiffrement AES. Ce chapitre présente une seconde contre-mesure, automatiquement appliquée à la compilation, pour le suivi automatique du graphe d'appel.

La contre-mesure, appelée *CalleeSimo*, s'applique à la compilation et est configurable, ce qui permet de sécuriser à des degrés divers. La protection consiste en un suivi des appels de fonction sensible avec une couverture variable d'attaque, ainsi qu'une protection de la valeur des arguments là encore avec une plage de protection dans le temps variable. Les différentes variantes de protections proposées restent légères, ne couvrant pas toute l'exécution ni toutes les fautes, de la même manière que les traceurs insérés manuellement. L'automatisation a pour but de permettre aux ingénieurs un gain de temps tout en limitant les erreurs dues à des modifications manuelles, et en évitant de faire appel à des schémas de protection génériques plus lourds comme [Reis et al., 2005].

La suite du chapitre s'organise comme suit. La section 4.2 présente l'état de l'art des attaques et contre-mesures relatives à la protection du graphe d'appel, ainsi que le contexte d'attaque dans lequel ce schéma est proposé. Les variantes du schéma de suivi des appels et protection des arguments sont exposées dans la section 4.3 qui détaille les algorithmes correspondants. Une analyse de l'intégration dans le flot de compilation et des limites de ce schéma est également présentée. La section 4.4 présente une analyse sécuritaire du schéma dans le cadre d'une architecture cible implémentant le jeu d'instructions ARMv7 classique des systèmes embarqués. La section 4.5 présente l'implémentation dans le compilateur LLVM des schémas de protection ainsi que des résultats expérimentaux. Avant de conclure, la section 4.6 discute des interactions de *CalleeSimo* et PLAF et explique les bénéfices à combiner les deux. Enfin, la section 4.7 conclut le chapitre.

4.2 Motivations

Avant de détailler l'algorithme de sécurisation du graphe d'appel, nous présentons dans cette section un état de l'art sur les attaques visant les appels de fonction et les schémas de sécurisation applicables à la protection du graphe d'appel. Nous précisons également le modèle d'attaquant considéré dans le reste du chapitre.

4.2.1 Attaques sur le graphe d'appel

Les appels de fonction sont des points sensibles d'un programme car ce sont de potentiels points d'entrée d'un sous-programme. Éviter un appel sensible peut compromettre la sécurité du code de démarrage d'un composant (*secure boot*) [Timmers and Spruyt, 2016], mais aussi la sécurité d'algorithmes cryptographiques [Blömer et al., 2014, Trichina and Korkikyan, 2010] en contournant l'exécution de toute une partie du programme.

En pratique, de nombreux codes sécurisés incluent des mécanismes de vérification de l'authenticité du code exécuté ou du bon déroulement de l'algorithme utilisé. Ces mécanismes sont souvent basés sur une comparaison de résultats de calculs effectuée par une fonction qui, si évitée, peut compromettre la sécurité. Par exemple, dans [Timmers and Spruyt, 2016], les auteurs indiquent que sauter l'appel à la fonction *memcmp* permet de contourner l'authentification d'un démarrage sécurisé (*secure boot*) même si ce n'est pas l'unique point d'attaque. Dans le contexte de l'ECC, les auteurs de [Blömer et al., 2014] montrent que sauter l'appel à l'exponentiation finale permet, grâce à une cryptanalyse, de récupérer le secret de l'algorithme de couplage. De façon générale, les contre-mesures à base de redondance consistent à effectuer deux fois les calculs puis à comparer les résultats. Lorsque l'attaquant a les capacités d'injecter des double fautes, il peut viser l'un des calculs avec une faute et cibler la vérification avec l'autre. Quand cette vérification est réalisée par une fonction à part, ne pas l'exécuter permet donc de contourner la contre-mesure.

Certaines fonctions, sensibles ou non, peuvent avoir des arguments sur lesquels repose la sécurité de l'algorithme sous-jacent. Par exemple, la plupart des algorithmes cryptographiques peuvent être utilisés avec des paramètres différents fournissant des niveau de sécurité variables : le chiffrement AES peut être utilisé avec des niveaux de sécurité de 128, 192 ou 256 bits, et l'algorithme PRESENT [Bogdanov et al., 2007], avec des niveaux de 80 ou 128 bits. De nos jours, le niveau minimum de sécurité recommandé est d'au moins 128 bits [ANSSI, 2014] et les algorithmes à faible niveau de sécurité ($\leq$80 bits) sont proscrits. Si un développeur pense utiliser l'un de ces algorithmes avec un niveau de sécurité de 128 bits et qu'une attaque par faute force à utiliser l'algorithme dans son mode à 80 bits, la sécurité de l'algorithme est compromise. Dans l'industrie, ainsi que dans la plupart des librairies open-source [1], ce niveau de sécurité est souvent passé en argument des fonctions principales des algorithmes. Il est donc possible, en faisant une faute qui modifie la valeur d'un argument d'une de ces fonctions, d'appeler l'algorithme considéré avec le mauvais niveau de sécurité. Un autre exemple particulièrement sensible est le cas de l'ECC où la sécurité des algorithmes dépend de la courbe elliptique utilisée. Dans [Biehl et al., 2000], les auteurs montrent que l'on peut rendre la sécurité nulle en fautant les points de la courbe, entrées de l'algorithme, pour qu'ils se retrouvent sur une courbe faible.

1. Par exemple dans OpenSSL (`http://www.openssl.org`)

4.2.2 Schémas de protection existants

L'intégrité du graphe d'appel peut être assurée par des schémas proposant l'intégrité plus générale du flot de contrôle (*Control Flow Integrity* (CFI)). De nombreux schémas de CFI ont été proposés dans la littérature.

Une première catégorie de schémas cherche à protéger contre les attaques logicielles à base de réutilisation de code, comme *Return-Oriented Programming* (ROP) [Shacham, 2007] ou *Jump-Oriented Programming* (JOP) [Bletsch et al., 2011], en protégeant les sauts indirects. Ces schémas de protection proposent de vérifier la source et la cible de sauts indirects, comme dans les travaux précurseurs de [Abadi et al., 2005]. Ils reposent sur la vérification d'un identifiant encodant la validité du transfert de flot de contrôle. Des systèmes de protection de la pile comme des *shadow stacks* [Vendicator, 2000] peuvent également être utilisés contre une corruption de l'adresse de retour de fonctions. Ces protections se concentrent sur les sauts indirects mais ne suffisent pas à se protéger des possibilités offertes par les attaques par faute visant tout type d'appels, indirects ou non, contrairement aux attaques logicielles où l'attaquant ne peut modifier le code, mais est libre de choisir des entrées susceptibles d'exploiter des vulnérabilités, et donc des sauts indirects.

Une seconde catégorie de schémas cherche à assurer que le chemin exécuté est correct vis à vis du CFG de chaque fonction. Ces schémas sont souvent basés sur un pré-calcul statique de signatures recalculées et vérifiées dynamiquement afin de déterminer si l'exécution a suivi un chemin autorisé dans le code [Oh et al., 2002]. Le calcul dynamique peut être réalisé par du matériel dédié. Dans [de Clercq and Verbauwhede, 2017], les auteurs synthétisent les différentes architectures existantes. Dans le cas où le processeur cible n'offre pas le support matériel nécessaire, des approches logicielles ont également été proposées utilisant un fil d'exécution dédié ou du code effectuant un suivi [Oh et al., 2002, Goloubeva et al., 2005]. Dans [Lalande et al., 2014], un schéma d'intégrité du flot de contrôle est proposé qui permet de détecter une attaque provoquant un saut correspondant à au moins deux instructions au niveau du code source. Ce schéma basé sur l'incrémentation de compteurs entre instructions a ensuite été automatisé à la compilation dans la thèse de Thierno Barry [Barry, 2017]. Ces approches peuvent être utilisées pour tracer les différents appels mais sont exclusivement attachées au CFG. Elles ne prennent pas en compte le flot de données et ne permettent donc pas de protéger la valeur des arguments.

Un schéma dédié à l'intégrité du graphe d'appel est proposé et mis en œuvre dans un outil de réécriture de binaires, avec d'autres schémas de sécurisation de code pour cartes à puce présentés dans [De Keulenaer et al., 2015]. Il est basé sur la vérification locale d'une signature : l'identifiant d'une fonction appelée est copiée dans une variable (registre ou variable globale) dont la valeur est vérifiée à l'entrée de la fonction appelée. De la même façon, un autre identifiant est copié dans une variable à la fin de la fonction appelée, la

valeur de cette variable est vérifiée au retour de la fonction. Ce schéma est proche de celui que nous proposons pour le graphe d'appel mais ne prend pas en compte la corruption des arguments de la fonction. Mis à part des schémas dupliquant l'intégralité du code comme dans [Reis et al., 2005], il n'existe pas, à notre connaissance, de schémas de protection de l'intégrité du graphe d'appel qui prennent en compte la valeur des arguments des fonctions. Une duplication complète implique nécessairement un coût élevé non adapté aux contraintes des systèmes embarqués. À l'inverse, notre schéma de protection du graphe d'appel a pour but d'être léger pour pouvoir être déployé facilement quitte à ne pas protéger contre toutes les attaques. Ce schéma est particulièrement adapté en présence de mécanismes matériels de détection de fautes.

4.2.3 Modèle d'attaquant

Avant de détailler le schéma de protection proposé, nous précisons les capacités de l'attaquant considéré. Les attaques recherchées par l'attaquant considéré sont celles qui profitent de la non exécution d'une fonction, ou de l'exécution avec des valeurs d'arguments corrompues afin de contourner des mécanismes d'authentification ou de retrouver des informations sensibles.

Nous considérons que l'attaquant peut injecter des fautes qui peuvent amener à ne pas exécuter une fonction, comme un saut d'instruction ou une corruption du flot d'exécution dans la fonction appelante permettant de suivre un chemin ne passant pas par l'appel de la fonction. Le schéma de protection proposé se veut peu coûteux et localisé autour de l'appel. Le second type de faute n'est donc pas couvert : un schéma de protection du CFG plus générique et plus coûteux peut être appliqué dans ce cas. Nous considérons également que l'attaquant est capable de sauter une instruction qui fait quitter la fonction (*return*) ou une instruction qui, sautée, provoque une sortie de fonction (lorsque la dernière instruction est un branchement inconditionnel par exemple). En effet, cela peut également altérer le bon enchainement des appels de fonction, même si la probabilité que la suite de l'exécution se déroule correctement est faible (données traduites comme instructions à exécuter, retour de fonction via des valeurs ne correspondant pas à une adresse valide). En cas d'appel indirect, la corruption du registre utilisé pour l'appel peut également amener à appeler une autre fonction que celle attendue. Notre schéma ne propose pas de mécanisme de protection des appels indirects, nous ne considérons donc pas ce type d'attaques. S'il est indispensable de protéger ces appels indirects, des schémas de protection visant à assurer leur intégrité doivent être ajoutés [Abadi et al., 2005, Vendicator, 2000].

Comme pour la sécurisation des boucles, les sauts d'instructions ainsi que corruptions de registres sont susceptibles d'altérer le flot de données de la fonction appelante (cf. section 3.2.1), et donc la valeur d'un argument d'une fonction. La corruption de cette valeur peut avoir lieu a divers endroits dans la fonction appelante ou appelée, c'est pourquoi nous proposons plusieurs variantes à notre schéma de protection de la valeur des arguments.

Finalement, dans ce chapitre, nous considérons un attaquant capable de passer outre les autres protections (notamment matérielles si présentes) et induire la non exécution ou la sortie prématurée d'une fonction ainsi que la corruption de la valeur d'un argument.

4.3 Schéma de suivi de l'arbre d'appel : *CalleeSimo*

Dans cette section, nous présentons le principe puis l'algorithme de sécurisation du graphe d'appel, *CalleeSimo*, que nous avons conçu pour à la fois assurer un suivi des fonctions appelées et protéger leurs arguments. Comme l'algorithme de sécurisation des boucles présenté au chapitre précédent, cet algorithme est également dédié à une représentation intermédiaire (IR) de compilateur, de bas niveau et en forme SSA.

4.3.1 Principe du schéma

Cette section montre le principe général de notre sécurisation du graphe d'appel ainsi que les différents modes proposés.

4.3.1.1 Modes de sécurisation proposés

Selon la puissance de l'attaquant, l'application à sécuriser et les contre-mesures matérielles présentes, le développeur a la possibilité de choisir des niveaux de sécurité (taux de couverture de la protection) en sélectionnant parmi 3 modes de suivi des appels et 3 modes de protection des arguments décrits ci-dessous.

Suivi des appels de fonction Les trois modes de suivi des appels sont décrits en précisant leur objectif de sécurisation :

- ◇ *CallCount* : vise à assurer le bon nombre d'appels d'une fonction sensible, dans le but de garantir qu'aucun appel de fonction a été sauté.
- ◇ *CallSequence* : vise à suivre la bonne séquence d'appels des fonctions sensibles. Par rapport à *CallCount*, ce mode permet de vérifier que l'ordre des appels a bien été respecté.
- ◇ *CallRetSequence* : vise à suivre la bonne séquence d'appels et de retours des fonctions sensibles. Par rapport à *CallSequence*, ce mode permet de s'assurer que la fonction appelée s'est exécutée jusqu'à la fin, et que le retour s'est fait au bon endroit dans la fonction appelante.

Protection des arguments De plus, pour chacun de ces trois modes, viennent s'ajouter trois modes de protection des arguments sensibles des fonctions appelées avec une région de

protection croissante (zone du code où l'argument est protégé) et un coût croissant. Ils sont décrits ci-dessous.

- *CallerCallee* : vise à protéger la valeur de l'argument à travers l'appel.
- *DefCallee* : vise à protéger la valeur de l'argument depuis sa définition dans l'appelant jusqu'à l'entrée de la fonction appelée.
- *DefUses* : vise à protéger la valeur de l'argument depuis sa définition dans l'appelant jusqu'à ses utilisations dans la fonction appelée ;

4.3.1.2 Concept de la sécurisation

Dans la littérature, l'intégrité du flot de contrôle est souvent réalisée en vérifiant des signatures de blocs de base à différents points clés du programme, par exemple à l'entrée de blocs avec différents prédécesseurs. De manière similaire, une protection du graphe d'appel peut être apportée par la vérification de signatures des fonctions appelées (cf. section 4.2.2). Notre schéma se base sur deux signatures logicielles du chemin parcouru, accumulant des identifiants représentant les fonctions sensibles appelées, de façon similaire au schéma de [Oh et al., 2002] qui vise à sécuriser le flot de contrôle interne des fonctions seulement.

Suivi des appels de fonction Pour sécuriser les appels, deux variables globales (*Tracking Global Variables (TGV)*) sont utilisées pour suivre l'enchaînement des appels de fonction. L'une est dédiée aux fonctions appelantes (`TGVcaller`) et l'autre est dédiée aux fonctions appelées (`TGVcallee`). Le schéma de protection repose sur une mise à jour identique, dépendante de la fonction appelée, de ces deux variables de part et d'autre de l'appel. Ces deux variables doivent rester égales sauf dans la plage d'exécution de deux mises à jour correspondantes qui ont lieu autour de l'appel dans les fonctions appelantes et appelées. Une différence de valeurs entre ces deux variables en dehors de cette plage indique que l'appel ne s'est pas réalisé comme prévu ; un appel sauté ou une autre fonction appelée par exemple. Pour détecter une corruption, un bloc de contrôle en fin de fonction appelante vérifie l'égalité des variables et appelle une fonction de gestion d'erreur le cas échéant. La figure 4.1 montre un exemple de mise à jour des variables des deux côtés de l'appel, en utilisant une mise à jour volontairement simpliste qui décrémente les variables avec l'identifiant id_{callee} de la fonction appelée valant 1. Il faut noter que ces variables ne dépendent pas de la fonction à sécuriser : seules deux variables sont utilisées pour l'ensemble des appels des fonctions d'une unité de compilation.

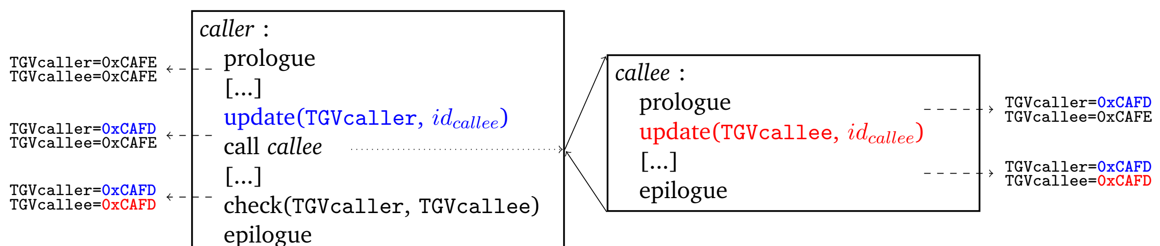


Figure 4.1: Exemple de mise à jour des signatures côté appelant (bleu) et côté appelé (rouge)

Protection des arguments de fonction Pour sécuriser les arguments sensibles d'une fonction, le schéma de protection proposé utilise deux autres variables globales : `TGVargcaller` mise à jour dans les fonctions appelantes et `TGVargcallee` mise à jour dans celles appelées. Chaque mise à jour de ces variables dépend de la valeur courante de l'argument à protéger.

Les mises à jour doivent rester cohérentes : toute mise à jour avec la valeur d'un argument dans la fonction appelante doit être équilibrée par une mise à jour dans la fonction appelée. Si l'argument est corrompu entre les deux mises à jour correspondantes, les deux variables ne sont pas mises à jour avec la même valeur et la différence provoquée permet de détecter la faute. La région entre les deux mises à jour correspondantes définit par conséquent la région de protection de l'argument. En effet, si la valeur de l'argument est altérée avant ces régions, les deux variables sont mises à jour avec une valeur corrompue et dans ce cas, aucune différence ne pourra être détectée. Le principe de protection est relativement simple mais son automatisation pour tous les codes, quelque soit le flot de contrôle, l'est moins. Un algorithme de mise en œuvre automatique est développé dans la section 4.3.4.

En l'occurrence, à cause des appels imbriqués, les deux variables ne peuvent être identiques dans tous les cas. Elles peuvent avoir des valeurs différentes en entrée d'une fonction, mais cette différence doit être retrouvée à la sortie de la fonction. Cela permet d'insérer des vérifications tout au long du code quelque soit l'imbrication des appels en ne comparant pas les valeurs des deux variables en fin de fonction comme c'est le cas pour le suivi des appels, mais en comparant la valeur de cette différence en entrée et sortie de fonction.

4.3.1.3 Prérequis

Afin d'appliquer automatiquement le schéma de sécurisation proposé, des modifications sont nécessaires à la fois dans les fonctions appelées et appelantes. Au niveau du code source, ces fonctions peuvent être définies dans différents fichiers ou unités de compilation. À la compilation, réaliser de telles modifications interprocédurales nécessite, en général, d'avoir à disposition le code des différentes fonctions impliquées. En particulier, les modifications nécessaires à la sécurisation d'un site d'appel dépendent d'analyses qui parcourent le code des deux fonctions concernées. Il est donc indispensable pour pouvoir sécuriser une fonction sensible d'avoir dans la même unité de compilation la fonction sensible et l'ensemble des fonctions pouvant l'appeler. Dans la suite, on suppose que toutes les fonctions sont disponibles dans la même unité de compilation. La section 4.3.6 explique les limitations liées à ce prérequis et comment les gérer en pratique.

4.3.2 Algorithmes principaux

L'algorithme proposé pour implémenter le schéma de protection agit sur une unité de compilation, appelée module ici en référence à la terminologie utilisée dans le compilateur LLVM. Il

est décrit dans l'algorithme 3. En premier lieu, les variables globales ainsi que le gestionnaire d'erreur sont créés ou récupérés grâce à la fonction *getOrCreateErrorHandler* (*ligne 2*). Ensuite, les analyses et insertions de mises à jour de variables à effectuer du côté des fonctions appelées sont réalisées pour l'ensemble des fonctions (*lignes 3 et 4*). Les insertions de mises à jour et de blocs de vérification du coté des fonctions appelantes ne sont réalisées que dans un second temps après avoir parcouru l'ensemble des fonctions appelées (*lignes 5 et 6*). Enfin, une dernière étape de nettoyage supprime du module le code inutile (*ligne 7*), par exemple des fonctions devenues mortes.

Algorithm 3: Algorithme principal

$M \leftarrow$ Calleesimo(M)
- **Input:** Module M
- **Output:** Secured module M

```
1  getOrCreateErrorHandler(M)
2  foreach function f of M do
3  |   addCalleeSignatureUpdates(M, f)
4  foreach function f of M do
5  |   addCallerSignatureUpdates(M, f)
6  cleanup()
```

4.3.2.1 Analyses et transformations des fonctions appelées

L'ensemble des transformations à appliquer à une fonction $callee$ appelée depuis le module est décrit dans l'algorithme 4. Pour la fonction *callee* du module traitée, cet algorithme détermine si cette fonction est sensible (*ligne 1*) et récupère la liste des arguments à sécuriser (*ligne 2*). Dans le cas où il n'y a rien à sécuriser, il est inutile d'aller plus loin (*ligne 3*).

Algorithm 4: Sécurisation d'un appel à une fonction $callee$ du côté de la fonction appelée

$callee \leftarrow$ addCalleeSignatureUpdates$(callee)$
- **Input:** Function $callee$
- **Output:** Hardened function $callee$

```
1  sensitive = isSensitiveCallee(callee)
2  ListArgs = getSensitiveArguments(callee)
3  if (isEmpty(ListArgs) and not sensitive) then return callee
4  cloned = Clone(callee)
5  addCalleeUpdatesForMode(cloned, getFuncMode(callee))
6  foreach arg ∈ ListArgs do
7  |   addArgCalleeUpdatesForMode(cloned, arg, getArgMode(arg))
8  return callee
```

Quand la fonction doit être sécurisée, elle est d'abord clonée (*ligne 4*) afin de garder intacte la fonction originale. Ce clonage est nécessaire dès lors qu'un site d'appel à cette fonction ne peut pas être traité, comme par exemple, dans le cas où la fonction est visible de l'extérieur du module et donc potentiellement appelée depuis l'extérieur. Le site d'appel externe n'étant pas traité, il est essentiel de conserver une version originale de la fonction qui sera appelée par tous les sites d'appel externes, au prix d'un coût en taille de code potentiellement important car les deux versions coexistent. Cette pratique est déjà utilisée dans d'autres passes de

transformation inter-procédurales comme, par exemple, la passe d'inlining partiel dans le compilateur LLVM.

Les mises à jour de la variable `TGVcallee` dépendent du mode de suivi de la fonction appelée, récupéré par la fonction `getFuncMode` (*ligne 5*), et sont gérées par la fonction `addCalleeUpdatesForMode` (*ligne 5*) dont les ajouts sont décrits dans la section 4.3.3. Si de plus, la fonction possède des arguments sensibles, les mises à jour de la variable `TGVargcallee` nécessaires à la protection de leur valeur dépendent du mode de protection de chaque argument sensible (*ligne 7*) et sont gérées par la fonction `addArgCalleeUpdatesForMode` décrite dans la section 4.3.4.

4.3.2.2 Analyses et transformations des fonctions appelantes

Une fois les modifications effectuées pour l'ensemble des fonctions appelées, l'algorithme traite des modifications à effectuer du côté des fonctions appelantes. En considérant une fonction *caller* du module, l'algorithme 5 présente les différentes étapes à réaliser.

Algorithm 5: Sécurisation des appels à des fonctions sensibles du côté de la fonction appelante

```
caller ← addCallerSignatureUpdates(caller)
    Input: Function caller
    Output: Hardened function caller
 1  ListCalls = getListOfCalls(caller)
 2  foreach call ∈ ListCalls do
 3      callee = getCallee(call)
 4      if isSensitive(callee) then
 5          addCallerUpdatesForMode(caller, call, callee, getFuncMode(callee))
 6      ListArgs = getSensitiveArguments(callee)
 7      foreach arg ∈ ListArgs do
 8          addArgCallerUpdatesForMode(caller, call, arg, getArgMode(arg))
 9      if atLeastOneUpdate(call) then setCall(call, getClone(callee))
10  if noUpdatedCall(caller) then return caller
11  end = getOrCreateUniqueExit(caller)
12  createCallCheckAtEnd(caller, end)
13  if atLeastOneSensitiveArg(caller) then
14      createArgCheck(caller, end)
15  return caller
```

Pour chaque appel *call* à une fonction, si la fonction appelée *callee* est sensible, les mises à jour de la variable `TGVcaller` sont gérées par la fonction `addCallerUpdatesForMode` (*ligne 5*) dont les modifications sont décrites dans la section 4.3.3. De la même façon que précédemment, si la fonction appelée possède des arguments à protéger, les mises à jour de la variable `TGVargcaller` sont gérées par la fonction `addArgCallerUpdatesForMode` (*ligne 8*) expliquée dans la section 4.3.4. Pour un appel donné, si l'algorithme est entré dans l'un des cas de mise à jour de signature, la fonction appelée est soit sensible, soit possède des arguments sensibles et a donc nécessairement été clonée. Dans ce cas, l'appel à la fonction originale est remplacé par un appel à la version clonée et sécurisée de la fonction (*ligne 9*).

Enfin, si la fonction $caller$ ne possède aucun appel à une fonction sensible, elle n'a pas été modifiée et ne nécessite aucun ajout. En revanche, dès que l'un des appels est sécurisé, un bloc de contrôle des signatures est ajouté à la fin de la fonction appelante (*lignes 11 à 14*). Ce bloc contrôle d'abord la cohérence entre les 2 variables `TGVcaller` et `TGVcallee`. Si de plus, des fonctions appelées possèdent des arguments sensibles, il contrôle aussi la cohérence des 2 variables dédiées aux arguments `TGVargcaller` et `TGVargcallee`. Ces deux dernières variables doivent garder une différence identique au début et à la fin d'une fonction. La fonction `createArgCheck` ajoute le calcul de cette différence en entrée de la fonction $caller$, puis de son recalcul dans le bloc de contrôle pour s'assurer qu'il n'y a pas eu de corruption des arguments pendant l'exécution de la fonction.

4.3.3 Suivi des fonctions

Cette section détaille comment gérer, en fonction du mode choisi, les mises à jour nécessaires des deux variables `TGVcallee` et `TGVcaller` pour assurer le suivi des fonctions sensibles. En effet, l'insertion des mises à jour se fait à des points différents du programme selon le mode de suivi de la fonction appelée concernée.

4.3.3.1 Modes de suivi des fonctions

Les deux premiers modes *CallCount* et *CallSequence* positionnent les mises à jour de la même manière et fournissent le schéma le plus simple. Leur but étant de suivre les appels pour s'assurer qu'ils n'ont pas été sautés, `TGVcaller` est mise à jour dans l'appelant juste avant l'instruction d'appel et `TGVcallee` dans l'appelé juste à l'entrée. La différence entre ces deux modes repose dans la fonction de mise à jour pour suivre le nombre ou l'ordre des appels, qui est discutée dans la section suivante. En présence de sécurisation d'un de ces deux modes, l'exécution de l'instruction d'appel est donc entourée des deux mises à jour qui s'équilibrent. La figure 4.2 représente ces deux cas.

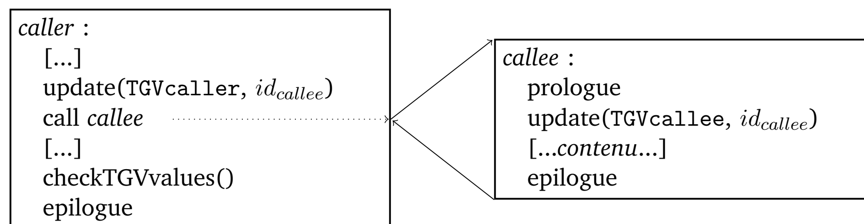


Figure 4.2: Position des mises à jour des variables pour les modes *CallCount* et *CallSequence*

Le mode de suivi *CallRetSequence* vise à protéger les appels comme les retours de fonctions. Le principe est donc d'entourer l'instruction d'appel et l'instruction de retour. `TGVcaller` est donc mise à jour dans l'appelant avant et après l'instruction d'appel. Aussi, `TGVcallee` est mise à jour à l'entrée et avant chaque instruction de retour possible dans la fonction appelée. Les deux variables sont ainsi équilibrées. La figure 4.3 représente ce dernier cas.

Les fonctions addCallerUpdatesForMode et addCalleeUpdatesForMode sont responsables de ces ajouts.

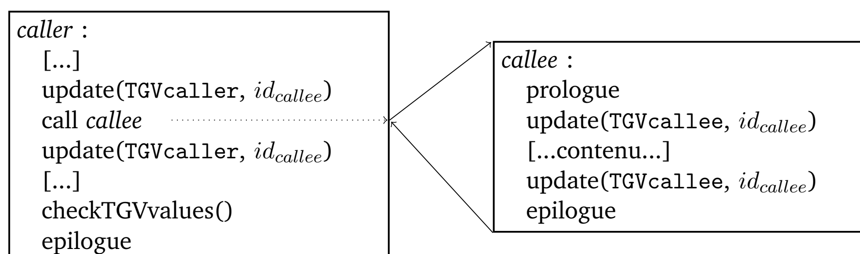


Figure 4.3: Position des mises à jour des variables pour le mode *CallRetSequence*

4.3.3.2 Mise à jour des variables de suivi

Nous discutons dans cette section des propriétés de la fonction de mise à jour. En effet, jusqu'ici, la mise à jour des variables de suivi des appels était manipulée comme une boîte noire. Or, de ses propriétés dépendent les paramètres de sécurité voulus. De plus, quel que soit le mode de sécurité choisi, si de nombreuses fonctions sont à sécuriser, de nombreuses mises à jour sont nécessaires. Il est donc important de minimiser le coût de la mise à jour.

Les mises à jour des variables TGVcaller et TGVcallee doivent permettre de différencier l'appel à une fonction f de l'appel à une fonction g. Pour cela, un identifiant id_f propre à chaque fonction f sensible appelée est défini. Pour le mode *CallCount* où l'on souhaite juste enregistrer le nombre d'appels de chaque fonction, la mise à jour de ces variables peut simplement être une opération arithmétique de base comme TGV $\leftarrow$ TGV $+ id_f$. Cependant, pour les modes *CallSequence* et *CallRetSequence* dans lesquels on souhaite tracer les séquences d'appels, il est nécessaire de pouvoir différencier entre un appel de f suivi de g d'un appel de g suivi de f. La fonction de mise à jour ne doit donc pas être commutative. Une fonction de mise à jour possible est : TGV $\leftarrow$ ror(TGV, 19) $+ id_f$ où ror(x,19) est la rotation des bits du mot x de 19 bits vers la droite. Lorsque l'on cherche à identifier le nombre d'appels successifs d'une même fonction f, il est préférable que la période de cette fonction de mise à jour soit suffisamment grande pour ne pas générer de collisions sur le nombre d'appels possible. Que l'architecture cible soit 32 ou 64 bits [2], 19 étant premier avec les valeurs de la forme 2^n, la dispersion des bits liée à la rotation et la retenue générée par l'addition rendent la période de cette fonction de mise à jour très grande (supérieure à 2^{24}). Le schéma générique ne vise pas une architecture en particulier mais, en cas de cible implémentant le jeu d'instructions ARMv7, la mise à jour peut être exécutée en une seule instruction de type `add rx, ry, rz, ror #19` grâce à la flexibilité sur le seconde opérande [3]. Cela fournit une mise à jour peu coûteuse en taille et en temps.

2. En cas d'architecture 8 bits, on peut utiliser des rotations de 3 bits au lieu de 19, fournissant alors une période plus faible
3. http://infocenter.arm.com/help/topic/com.arm.doc.kui0100a/armasm_cihbeage.htm

4.3.3.3 Cas statique

Le schéma proposé se base sur des mises à jour de variables au niveau de chaque site d'appel. Ce schéma générique permet de sécuriser les appels à une fonction indépendamment de la complexité du flot de contrôle de la fonction appelante. Pour certaines applications ou cas d'usage, il est toutefois possible de connaître statiquement le nombre et/ou la séquence des appels de fonction.

En guise d'illustration, le listing 4.1 montre l'exemple d'un chiffrement AES. Le nombre de rondes de l'AES (variable `numRounds`) dépend du niveau de sécurité : 10 pour l'AES-128, 12 pour l'AES-192 et 14 pour l'AES-256. Dans le cas général, si la variable `numRounds` est globale ou argument de la fonction, il n'est pas forcément possible de connaitre statiquement le nombre d'appels. Dans ce cas générique de l'AES, on peut utiliser la contre-mesure PLAF précédente pour s'assurer du bon nombre de tours de boucle. Si la variable `numRounds` est fixée, dans le cas d'un AES-128 par exemple, on connait statiquement le nombre et la séquence des appels effectués par la fonction.

```
addRoundKey();
for (i=0; i<numRounds - 1; i++) {
    SubBytes();
    ShiftRows();
    MixColumns();
    addRoundKey();
}
SubBytes();
ShiftRows();
addRoundKey();
```

Listing 4.1: Structure d'un chiffrement AES

Dans les cas où une telle information statique est disponible, on peut améliorer le schéma de protection du graphe d'appel, en termes à la fois de sécurité apportée et de surcoût induit : les mises à jour de la variable dans les fonctions appelées restent inchangées, mais les mises à jour dans la fonction appelante peuvent être toutes mutualisées en une unique mise à jour. Cette unique mise à jour doit être placée dans un bloc dominant ou post-dominant tous les appels concernés. Il est par exemple possible de la placer juste avant le bloc de contrôle en sortie de fonction appelante.

Cette mutualisation possède deux avantages. Premièrement, la sécurisation ne se limite pas à entourer le site d'appel mais prend en compte l'intégralité de la fonction appelante. En effet, si une corruption du flot de contrôle dans la fonction appelante évite toute une branche contenant plusieurs appels, le schéma dynamique présenté précédemment ne permet pas de la détecter car la protection est localisée autour de l'appel, dans le bloc de base le contenant. Deuxièmement, on évite une mise à jour à chaque appel. Une seule mise à jour étant suffisante pour la fonction appelante, le coût est donc presque divisé par 2 (les mises à jour dans les fonctions appelées étant toujours présentes). Cela n'est toutefois possible que lorsque les

fonctions appelées sont des feuilles de l'arbre d'appel ou qu'elles ont elles aussi cette propriété, permettant ainsi de pré-calculer les valeurs attendues à la compilation.

Dans l'exemple de l'AES-128 précédent, cela revient à effectuer une unique mise à jour en fin de fonction qui compense les 10 *SubBytes* et *ShiftRows*, les 11 *addRoundKey* et les 9 *MixColumns*. De plus, cette sécurisation fournit une alternative à la protection des boucles pour s'assurer d'avoir bien effectué le bon nombre de tours de boucle. En effet, altérer le flot de contrôle de cette fonction peut permettre un mauvais nombre d'appels aux sous-fonctions, qui sera détecté par la différence entre les deux variables. La précision de la protection est plus fine que PLAF dans ce cas car sauter un seul de ces appels est également détecté.

Cette version du schéma *CalleeSimo* est restreint aux cas où le nombre et éventuellement la séquence des appels sont statiquement connus dans la fonction appelante, ce qui peut paraître limité. Ce cas est néanmoins fréquent dans les algorithmes cryptographiques qui, pour des raisons de protection envers les attaques par observation, s'efforcent d'avoir un flot de contrôle indépendant des entrées de l'algorithme.

4.3.4 Protection des arguments

Après avoir détaillé l'insertion des mises à jour nécessaires pour le suivi des fonctions et les propriétés de la fonction de mise à jour associée, cette section présente l'insertion des mises à jour des variables `TGVargcaller` et `TGVargcallee` nécessaires pour la protection des arguments. La position de ces mises à jour est dépendante du mode de protection de chacun des arguments à protéger. La difficulté est de toujours équilibrer les mises à jour des deux variables indépendamment du flot de contrôle des fonctions appelantes et appelées. Comme expliqué précédemment dans la section 4.3.1.2, la région dans laquelle la valeur de l'argument est protégée correspond à la région définie entre les deux mises à jour correspondantes de l'argument. Les modes de protection proposés protègent des régions de plus en plus grandes autour de l'appel. Tous les modes de protection fonctionnent en insérant une mise à jour de `TGVargcaller` dans la fonction appelante et en insérant une mise à jour de `TGVargcallee` correspondante dans la fonction appelée. Leur différence réside dans les analyses et modifications supplémentaires nécessaires pour ne pas compromettre l'équilibre entre ces deux variables. Le tableau 4.1 montre informellement le début de la région dans la fonction appelante et la fin de la région dans la fonction appelée pour les différents modes.

Table 4.1: Définition des régions de protections selon le mode

Mode	Début de la région	Fin de la région
CallerCallee	Juste avant l'appel...	jusqu'à l'entrée de la fonction
DefCallee	Juste après la définition...	jusqu'à l'entrée de la fonction
DefUses	Juste après la définition...	jusqu'aux utilisations

La fonction `addArgCallerUpdatesForMode` insère, dans la fonction appelante, une mise à jour de la variable `TGVargcaller` juste avant l'appel pour le mode *CallerCallee*. Pour les deux autres modes protégeant à partir de la définition de l'argument, les modifications à apporter au code de la fonction appelante sont plus complexes et nécessitent de prendre en compte le flot de contrôle. Ces modifications sont détaillées dans la section 4.3.4.1, représentées par la fonction `addArgCallerUpdatesFromDef`. De façon similaire du coté de la fonction appelée, la fonction `addArgCalleeUpdatesForMode` insère une mise à jour de la variable `TGVargcallee` juste à l'entrée de la fonction pour les modes *CallerCallee* et *DefCallee*. Les modifications à apporter au code de la fonction appelée pour le mode *DefUses* sont également complexes et dépendantes du flot de contrôle. Elles sont présentées dans la section 4.3.4.2, représentées par la fonction `addArgCalleeUpdatesToUses`. Les modifications dépendantes d'analyses du flot de contrôle nécessitent d'ajouter deux types de mises à jour pour respecter la cohérence des schémas :

- ◇ *COMPENSATE* : des *mises à jour de compensation* d'une variable sont des mises à jour inverses permettant d'annuler une autre mise à jour ; dans le cas où la fonction de mise à jour ne peut être (facilement) inversée, une mise à jour de l'autre variable peut être réalisée pour compenser ;
- ◇ *ACCUMULATE :* des *mises à jour supplémentaires* d'une variable peuvent être nécessaires sur certains chemins pour que chaque chemin d'exécution possible contienne le même nombre de mises à jour de chacune des variables.

4.3.4.1 Protection des arguments depuis leur définition

Les modes *DefCallee* et *DefUses* proposent de protéger la valeur de l'argument depuis sa définition dans la fonction appelante. Il est alors nécessaire de mettre à jour la variable `TGVargcaller` juste après la définition de cet argument, définissant ainsi l'entrée dans la région de protection de l'argument. Toute corruption de sa valeur entre la mise à jour dans l'appelant et l'entrée de la fonction appelée sera donc détectée. En revanche, l'ajout de cette mise à jour à la définition nécessite de prendre en compte tous les chemins d'exécution possibles dans la fonction appelante pour s'assurer qu'aucun d'eux ne créé de déséquilibre entre les deux variables. En effet, l'appel d'une fonction ne post-domine pas nécessairement la définition de l'argument, ce qui peut engendrer une incohérence des variables quand l'exécution ne passe pas par l'appel à la fonction. Le but des analyses et transformations présentées dans cette section est de s'assurer que les deux variables sont mises à jour autant de fois chacune par rapport à cet argument, quel que soit le chemin parcouru dans la fonction appelante. Ainsi, la différence entre ces deux variables est bien la même en entrée de fonction appelante et à la fin de celle-ci, au niveau duquel est inséré le bloc de vérification de ces variables.

L'algorithme 6 présente les modifications nécessaires dans le cas de protection d'un argument depuis sa définition. Si l'argument sensible concerné est une constante dans la fonction

appelante, la constante n'a pas de définition au niveau IR. L'algorithme met alors à jour la variable `TGVargcaller` avant l'appel avec cette constante (*lignes 1 à 3*) et se termine directement. Il est possible que la constante soit directement utilisée comme opérande immédiat des instructions de mise à jour, ce qui permet de détecter une corruption menant à une valeur corrompue de l'argument. Dans ce cas, cela élargit la région de protection de l'argument avant sa définition et le placement de l'instruction de mise à jour n'a pas d'impact sur la région couverte. Dans les autres cas, l'algorithme détermine le bloc de base contenant l'instruction définissant l'argument sensible. Si cet argument est lui-même argument de la fonction appelante, une telle instruction n'existe pas et la première instruction de la fonction appelante est alors sélectionnée (*lignes 4 et 5*). Une mise à jour de `TGVargcaller` est insérée après l'instruction de définition sélectionnée (*ligne 7*).

Algorithm 6: Insertions de mises à jour nécessaires du côté de la fonction appelante pour la protection d'argument sensible depuis sa définition

```
caller ← addArgCallerUpdatesFromDef(caller, arg)
    Input: Function caller, Value arg, Instruction call
    Output: Function caller secured relatively to arg
1   if isConstant(arg) then
2       addUpdateBefore(call, TGVargcaller, arg)
3       return caller
4   argdef = getDefinitionInstructionOrFirst(arg)
5   defBB = getBasicBlock(argdef)
6   callBB = getBasicBlock(call)
7   addUpdateAfter(argdef, TGVargcaller, arg)
8   Edges = getEdgesNeedingUpdates(defBB, callBB)
9   foreach edge = (from, to, type) ∈ Edges do
10      newBB = createAndInsertBasicBlockBetween(from, to)
11      b = createUncondBranch(newBB, to)
12      if type == ACCUMULATE then
13          addUpdateBefore(b, TGVargcaller, arg)
14      else
15          inverseUpdateBefore(b, TGVargcaller, arg)
16  return caller
```

Le problème que peut présenter cette variante de la protection des arguments est lié à la potentielle complexité du flot de contrôle de la fonction appelante. La définition de l'argument domine, par définition de la forme SSA, l'appel à la fonction l'utilisant mais tout autre type de flot de contrôle est possible. Des chemins peuvent passer par la définition sans passer par l'appel, ou des boucles peuvent par exemple contenir l'appel mais pas la définition. La mise à jour de `TGVargcaller` à la définition de l'argument est associée à celle effectuée sur `TGVargcallee` dans l'appel utilisant l'argument. Mais, s'il existe des chemins qui passent par la définition sans passer par l'appel, il est nécessaire de compenser cette mise à jour à la définition en ajoutant sur ces chemins des mises à jour de compensation de `TGVargcaller`. De la même manière, sur les chemins qui passent par l'appel sans passer par la définition (boucles internes par exemple), il est nécessaire d'ajouter des mises à jour supplémentaires de `TGVargcaller` afin de contrebalancer les multiples mises à jour de `TGVargcallee`.

La fonction `getEdgesNeedingUpdates` (*ligne 8*) est en charge de l'analyse du flot de contrôle et de la détermination des points d'insertion de telles mises à jour supplémentaires et de compensation. Elle est détaillée dans le prochain paragraphe. Pour chacun de ces arcs, un nouveau bloc contenant uniquement la mise à jour correspondante est créé et inséré dans le CFG de la fonction entre le bloc source et le bloc destination de cet arc (*lignes 9 à 13*). Le but de l'analyse du flot de contrôle est de déterminer sur quels arcs placer des mises à jour supplémentaires et de compensation des variables en fonction du bloc contenant la définition et de celui contenant l'appel. L'algorithme 7 décrit les étapes de cette analyse et est expliqué ci-dessous. Afin de mieux comprendre cette analyse, un exemple de flot de contrôle complexe illustrant tous les cas possibles de l'analyse est présenté en figure 4.4. Cet exemple n'est pas directement tiré du graphe d'un code source réel, mais regroupe l'ensemble des cas rencontrés sur de vrais codes.

Algorithm 7: Analyse du flot de contrôle pour déterminer les arcs sur lesquels des mises à jour doivent être ajoutées

```
Edges ← getEdgesNeedingUpdates(def, call)
    Input: BasicBlock def, BasicBlock call
    Output: Set of edges to update Edges
1   Edges = []
2   Worklist = {def}
3   while Worklist is not empty do
4       current = dequeue(Worklist)
5       foreach successor S of current do
6           if S == call then continue
7           if edge(current → S) is a loop back-edge then continue
8           if call is not accessible from S without backedges then
9               Append(Edges, Edge(current, S, ACCUMULATE))
10              continue
11          Append(Worklist, S)
12  Loop L = getInnerLoopContainingOrNone(call)
13  while L is a Loop do
14      if def ∉ L then
15          Append(Edges, Edge(getLatch(L), getHeader(L), COMPENSATE)
16      L = getOuterLoopOrNone(L)
17  return Edges
```

L'analyse part du bloc contenant la définition de l'argument (*ligne 2*) qui permet nécessairement d'atteindre le bloc contenant l'appel. En parcourant récursivement les successeurs, elle cherche l'ensemble des blocs pouvant encore atteindre le bloc contenant l'appel (*lignes 5 et 6*). Ce sont les blocs en bleu dans la figure 4.4. Dès que l'on ne peut plus l'atteindre, il faut compenser la mise à jour de `TGVargcaller` effectuée à la définition de l'argument (*lignes 8 à 10*) : l'exécution est passée par la définition mais pas par l'appel. Les arcs quittant la région de ces blocs capables d'atteindre l'appel sont donc ceux sur lesquels il faut insérer des mises à jour de compensation. Ils sont représentés en rouge sur la figure 4.4. On peut constater qu'il n'est possible de sortir de cette région (blocs en bleu) que par des arcs rouges ou par l'appel. Lors de cette recherche, il ne faut pas considérer les arcs arrière de boucles contenant l'appel. En effet, si l'on inclut les arcs arrière de ces boucles dans la recherche, tous les blocs de la

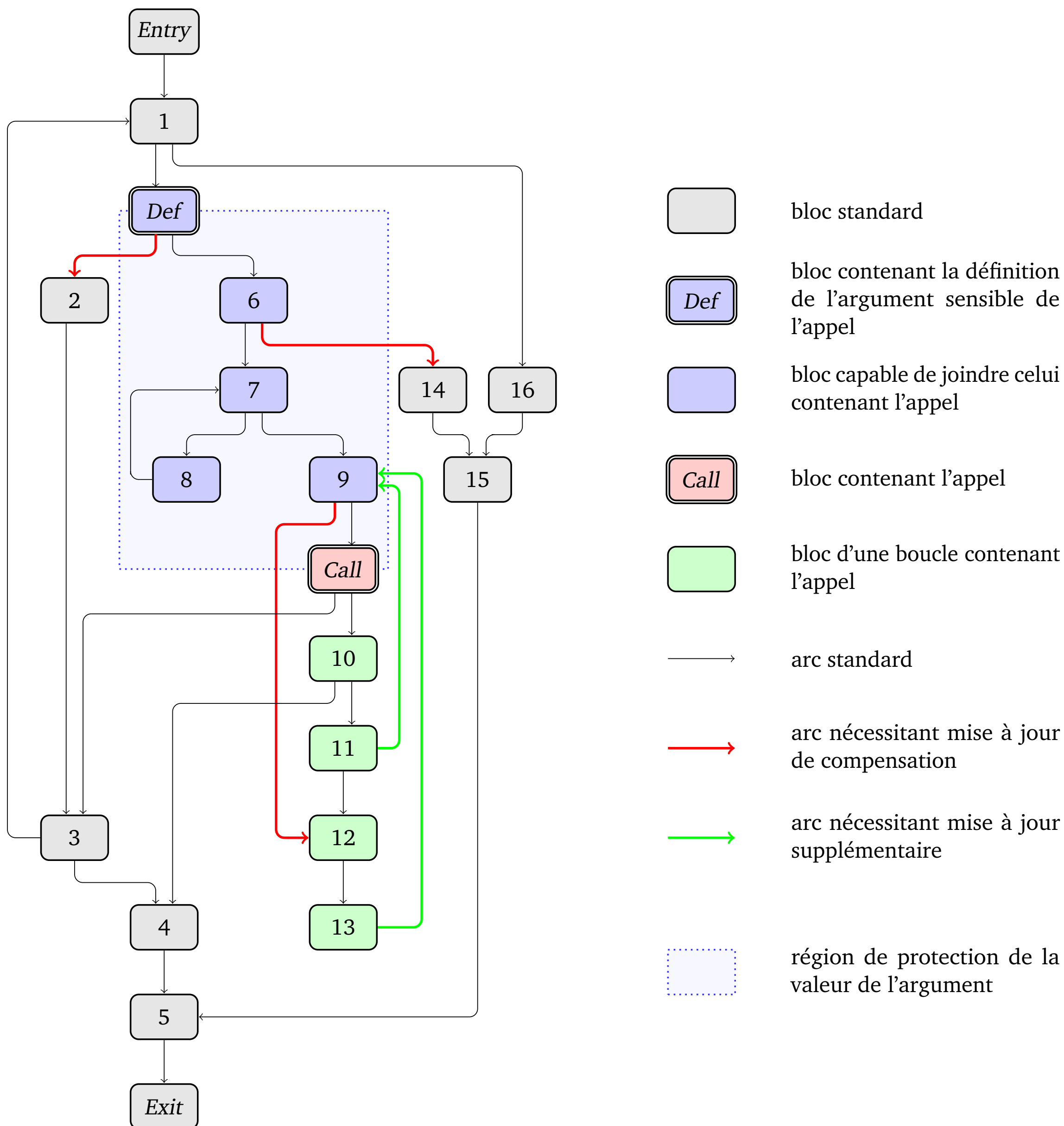


Figure 4.4: Exemple de flot de contrôle présentant les arcs sur lesquels ajouter des mises à jour pour une protection de la valeur de l'argument depuis sa définition

boucle (blocs en vert sur la figure) pourraient permettre d'atteindre l'appel, ainsi que les blocs 2 et 3. Sur l'exemple présenté, les blocs 10 à 13 feraient partie de cette région permettant d'atteindre l'appel, avec l'arc $10 \to 4$ comme arc de sortie. Or, il est possible d'exécuter dans la fonction le chemin *Entry* $\to 1 \to$ *Def* $\to 6 \to 7 \to 9 \to$ *Call* $\to 10 \to 4 \to 5 \to$ *Exit*. Dans ce chemin, on mettrait à jour deux fois la variable `TGVargcaller` (dans le bloc *Def* et dans l'arc $10 \to 4$) et une seule fois la variable `TGVargcallee` (dans la fonction appelée), aboutissant à une incohérence. C'est pourquoi dans l'algorithme 7, en déterminant si le bloc de l'appel est atteignable, on ignore les arcs arrière des boucles (*ligne 8*), traités à part (*lignes 12 à 16*).

Dans le cas de boucle (ou nid de boucles) contenant l'appel mais pas la définition de l'argument, il est possible d'exécuter, pour une seule mise à jour de `TGVargcaller`, plusieurs fois la fonction appelée et donc de mettre à jour plusieurs fois la variable `TGVargcallee`. Il faut alors contrebalancer ces multiples mises à jour de `TGVargcallee` en ajoutant des mises à jour supplémentaire de `TGVargcaller` sur les arcs arrière (en vert sur la figure 4.4) de telles boucles, quand elles existent (*lignes 12 à 16*). Avec des mises à jour supplémentaires sur les arcs ainsi trouvés, les deux variables auront des valeurs cohérentes quelque soit le chemin parcouru dans la fonction appelante. Plusieurs parcours possibles sont illustrés en exemple dans la figure 4.5 représentant divers chemins possibles dans le CFG.

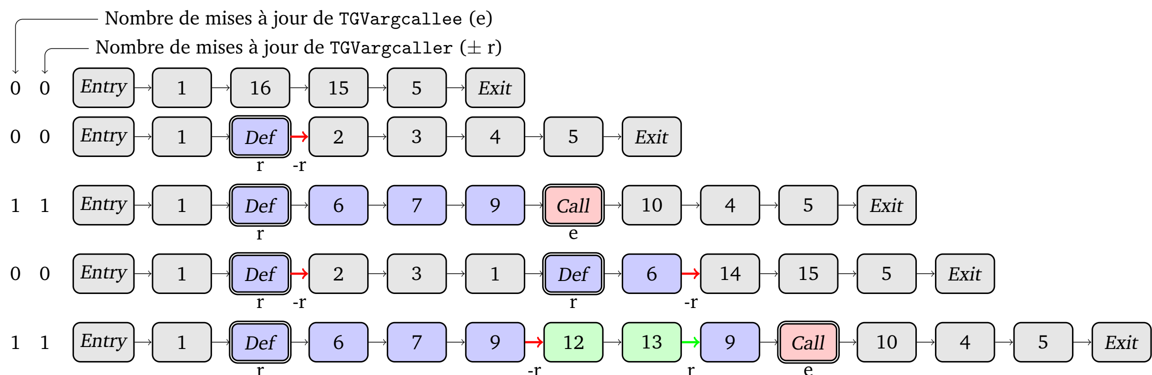


Figure 4.5: Exemples de parcours dans la fonction présentée en figure 4.4 avec le nombre de mises à jour de variables associées

4.3.4.2 Protection des arguments jusqu'à leurs utilisations

Le mode *DefCallee* précédent permet d'étendre la région de protection de la valeur de l'argument, dans la fonction appelante, en amont de l'appel à partir de sa définition. Quant à lui, le mode *DefUses* permet en plus d'étendre la région en aval de l'appel, dans la fonction appelée, jusqu'aux utilisations de l'argument. Plus précisément, la région de protection couvre l'ensemble des blocs du CFG de la fonction permettant d'atteindre une utilisation de l'argument dans la fonction appelée. L'algorithme 8 présente les modifications nécessaires dans la fonction appelée pour implémenter ce schéma pour un argument arg. Le but de cet algorithme est de s'assurer que la variable `TGVargcallee` est mise à jour exactement une fois quel que soit le chemin parcouru dans la fonction appelée. Cela permet d'être cohérent avec les modifications apportées dans les fonctions appelantes, qui nécessitent une mise à jour de `TGVargcallee` dans la fonction appelée. Afin de visualiser ces modifications sur un exemple, le graphe utilisé dans la figure 4.4 est réutilisé dans la figure 4.6 en prenant comme exemple un argument utilisé dans plusieurs blocs. La région de protection de l'argument est représentée en bleu sur cette figure.

L'algorithme commence par déterminer l'ensemble des blocs contenant une utilisation de l'argument qui définissent la fin de la région de protection (*ligne 1*). Ces blocs correspondent aux utilisations *finales* de l'argument, i.e. ceux qui ne permettent pas de joindre une autre utilisation et sont représentés en rouge sur la figure 4.6. Pour une instruction u de cet

Algorithm 8: Insertions de mises à jour nécessaires en cas de vérification d'argument sensible jusqu'à ses utilisations

```
caller ← addArgCalleeUpdatesToUses(callee, arg)
    Input: Function callee, Value arg
    Output: Function callee secured relatively to arg
1   Uses = getListOfLastBlocksUsing(arg)
2   foreach u ∈ Uses do
3       addUpdateAfter(u, TGVargcallee, arg)
4   entryBB = getEntryBlock(callee)
5   Edges = getEdgesNeedingUpdatesFromList(entryBB, Uses)
6   foreach edge = (from, to, type) ∈ Edges do
7       newBB = createAndInsertBasicBlockBetween(from, to)
8       b = createUncondbranch(newBB, to)
9       if type==ACCUMULATE then
10          addUpdateBefore(b, TGVargcallee, arg)
11      else
12          inverseUpdateBefore(b, TGVargcallee, arg)
13  return callee
```

ensemble utilisant l'argument, il est nécessaire de mettre à jour la variable `TGVargcallee` juste après u, définissant ainsi la fin de la région de protection de l'argument (*lignes 2 et 3*). Toute corruption de sa valeur entre l'entrée de la fonction et cette utilisation sera donc détectée. Comme pour l'analyse précédente du côté de la fonction appelante, l'ajout de mise à jour au niveau de ces utilisations nécessite de prendre en compte tous les chemins d'exécution possibles dans la fonction appelée pour s'assurer qu'aucun d'eux ne créé de déséquilibre entre les deux variables. S'il existe des chemins ne passant par aucune de ces utilisations finales, il est nécessaire d'insérer des mises à jour supplémentaires de `TGVargcallee` afin que chaque chemin possible passe une fois par une mise à jour (arcs rouges sur la figure 4.6). Aussi, les chemins passant par les boucles dont l'arc arrière a pour source un bloc hors de la région de protection et pour destination un bloc de cette région peuvent contenir plusieurs mises à jour de `TGVargcallee`. Il faut donc insérer des mises à jour de compensation de `TGVargcallee` dans ces arcs arrière (arcs verts sur la figure 4.6). La boucle $\{9, \mathit{Use2}, 10, 11\}$ possède une utilisation mais est entièrement contenue dans la région de protection, il n'est donc pas nécessaire d'ajouter de mise à jour sur son arc arrière $11 \rightarrow 9$.

L'analyse, gérée par la fonction `getEdgesNeedingUpdatesFromList`, détermine où placer ces mises à jour. Elle est similaire à celle, décrite dans la section précédente 4.3.4.1. Comme l'argument est nécessairement défini à l'entrée et n'est pas constant (paramètre), l'analyse part du bloc d'entrée de la fonction (*lignes 4 et 5*) et cherche tous les blocs de la région permettant d'atteindre une des utilisations finales. Lors du parcours du graphe, si un bloc hors de cette région est atteint, l'arc quittant la région nécessite une mise à jour supplémentaire (*lignes 9 et 10*). Enfin, une mise à jour de compensation est insérée sur les arcs arrières des boucles contenant une utilisation finale (*ligne 12*).

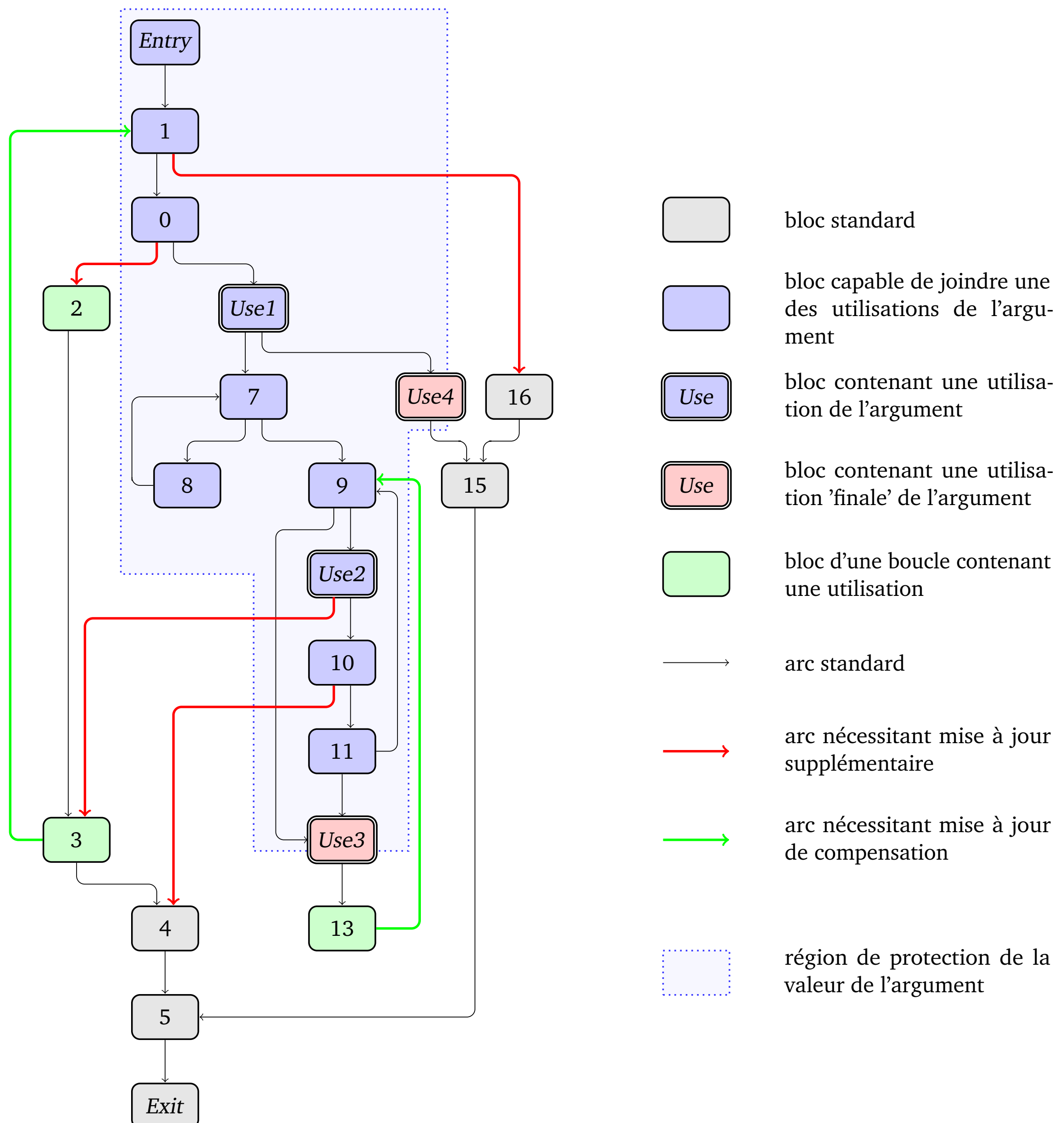


Figure 4.6: Exemple de flot de contrôle présentant les arcs sur lesquels ajouter des mises à jour pour une protection de la valeur de l'argument jusqu'à ses utilisations

Grâce à cet algorithme, tous les chemins possibles dans la fonction appelée contiennent une mise à jour de la variable `TGVargcallee`. Si une fonction appelée est de plus appelante de fonctions sensibles, elle possède un bloc de vérification des valeurs de variables en fin de fonction. La différence entre les variables `TGVargcaller` et `TGVargcallee` n'est donc pas identique en début et fin de fonction, mais l'écart est connu à la compilation, correspondant à exactement une mise à jour de `TGVargcallee` avec arg. Dans ce cas, il est donc nécessaire d'intégrer cette mise à jour dans le calcul de la différence en entrée de fonction.

4.3.4.3 Mise à jour des variables de protection des arguments

Les propriétés requises pour le suivi des appels de fonction ne sont pas les mêmes que pour la protection des arguments. Les mises à jour des variables `TGVargcaller` et `TGVargcallee` dépendent de la valeur de l'argument à protéger. À cause des schémas complexes liés à la variété des flots de contrôle et des lieux possibles pour les définitions, appels et utilisations, si plusieurs fonctions possèdent des arguments sensibles, il est impossible de connaître l'ordre des mises à jour. Le listing 4.2 montre une inversion de l'ordre dans le cas du mode *DefCallee* avec deux arguments définis et utilisés dans deux ordres différents. La fonction de mise à jour doit donc nécessairement être commutative. Afin de pouvoir insérer des mises à jour de compensation, il est préférable que la fonction de mise à jour soit simple à inverser. Pour des raisons de simplicité, la fonction pour la mise à jour liée à un argument arg peut donc être : $\texttt{TGV} \leftarrow \texttt{TGV} + arg$. Dans ce cas, la mise à jour inverse est donc : $\texttt{TGV} \leftarrow \texttt{TGV} - arg$.

```
x = ...;                    // TGVargcaller ← update(TGVargcaller, x)
y = ...;                    // TGVargcaller ← update(TGVargcaller, y)
f(y);                       // TGVargcallee ← update(TGVargcallee, y)
g(x);                       // TGVargcallee ← update(TGVargcallee, x)
```

Listing 4.2: Mises à jour des variables dans deux ordres différents

Pour la protection des arguments, on ne cherche pas à tracer la séquence des arguments utilisés. Ce n'est donc pas critique d'avoir une fonction non commutative. On veut seulement pouvoir vérifier que les arguments ont la bonne valeur. Toutefois, on peut constater qu'une corruption d'un argument compensée par une corruption inverse d'un autre argument ne serait pas détecté ($+1$ sur un argument et -1 sur l'autre).

4.3.5 Intégration dans le flot de compilation

Cette section décrit les interactions possibles entre *CalleeSimo* et les optimisations du *middle-end* de LLVM, dans le but d'intégrer une passe de compilation implémentant le schéma de protection proposé au niveau du *middle-end* et rester ainsi indépendant au maximum de l'architecture cible. Dans le chapitre 3, nous avons vu que les optimisations du compilateur interagissent avec l'application de contre-mesure même lorsque l'application a lieu à la compilation. Au niveau du *middle-end*, la plupart des optimisations agissent au niveau d'un bloc de base ou d'une fonction. Peu d'optimisations sont inter-procédurales. Le schéma de suivi du graphe d'appel repose sur des mises à jour de variables globales et de modifications dépassant le cadre d'une unique fonction. Il est donc moins impacté par le flot de compilation que l'algorithme PLAF.

Cependant, comme décrit dans la section 3.5.1.1, la passe d'optimisation *register promotion* est essentielle pour éliminer les fausses dépendances mémoires. Pour ce schéma, cela permet une plus grande précision sur la définition des arguments sensibles, et ainsi une plus grande

région de protection qui, sinon, ne débuterait au mieux qu'au chargement mémoire précédent l'appel. La passe *Common Subexpression Elimination* (CSE) serait susceptible de détecter que les mêmes mises à jour peuvent être effectuées sur différents chemins et chercherait donc à les simplifier, déplaçant ainsi la limite de la région de protection. Toutefois, l'utilisation de variables globales (visibles à l'extérieur du module) empêche cette optimisation de s'appliquer. Si une optimisation plus agressive permettait tout de même une telle simplification, l'ajout d'attribut `volatile` au chargement des variables globales (`TGV`) la bloquerait.

Les passes d'*inlining* et d'*inlining partiel* font partie des rares optimisations inter-procédurales. Elles remplacent des appels de fonction, en partie ou entièrement, par le corps de la fonction directement au site d'appel et peuvent interférer avec la sécurisation du graphe d'appel. Si une fonction est inlinée, il n'y a plus d'appel sensible étant donné qu'il est remplacé par le contenu de la fonction. Cela ne pose pas de problème dans le cas général car cela supprime le point sensible. En revanche cela peut interférer avec le schéma de suivi proposé dans le cas statique présenté en section 4.3.3.3.

L'*inlining* est une des premières passes d'optimisation car elle permet, à cette position dans le flot de compilation, aux optimisations internes aux fonctions d'être plus efficaces. Les passes d'optimisation interagissant avec la sécurisation du graphe d'appel sont donc placées tôt dans le flot de compilation. Pour minimiser un éventuel impact non détecté des autres optimisations et appliquer ce schéma sur du code déjà optimisé, nous avons décidé de le placer à la fin du *middle-end*. Il n'y a pas de passe d'optimisation critique dans le *back-end* concernant le graphe d'appel, contrairement à la sécurisation des boucles qui repose sur de la duplication à grain fin et qui est impactée par la passe *Loop Strength Reduction* (LSR) dépendante de la cible et donc placée dans le *back-end*. Pour des raisons pratiques et de portabilité, il est donc plus judicieux de la placer à la fin du *middle-end* plutôt que de la placer au début du *back-end* comme pour la sécurisation des boucles.

Pour compléter cette analyse au niveau du *middle-end*, une analyse des interactions avec les passes en aval dans le *back-end* serait nécessaire afin de s'assurer que ce dernier n'altère pas le code généré. Cela est laissé en perspective par manque de temps dans la thèse.

4.3.6 Limites du schéma de protection

Le schéma de sécurisation proposé ne compromet pas la fonctionnalité du programme en cas d'appel d'indirects, grâce au clonage des fonctions modifiées. Les appels directs sécurisés sont remplacés par un appel à la fonction clone intégrant le suivi, mais les appels indirects continuent d'appeler la version originale non sécurisée de la même manière que le font les fonctions définies dans d'autres unités de compilation. Une des limitations de ce schéma est qu'il n'est pas possible, sans information supplémentaire du développeur, de sécuriser les appels indirects de façon générale. La passe *Called Value Propagation* cherche à déterminer,

pour un site d'appel indirect donné, l'ensemble des fonctions qui peuvent être appelées dynamiquement à ce site d'appel. Lorsque son analyse détermine avec succès toutes ces fonctions, il serait possible d'étendre le schéma pour assurer que la fonction appelée est une fonction autorisée pour le site d'appel concerné. Une autre solution pour étendre ce schéma serait de définir au niveau du code source une annotation décrivant, pour un site d'appel indirect donné, un ensemble de fonctions autorisées.

Une autre limitation de ce schéma est la nécessité de regrouper en une unique unité de compilation l'ensemble du code du programme contenant les appels à sécuriser. Dans le compilateur LLVM, il est possible d'interrompre le flot de compilation avant le *back-end* et de générer, pour chaque fichier source, un fichier IR optimisé correspondant. Les différents fichiers IR générés lors de la compilation d'une application ou librairie peuvent alors être rassemblés en un seul fichier sur lequel la passe d'optimisation implémentant ce schéma peut être appliquée. Ce n'est donc pas une limitation intrinsèque du schéma de sécurisation dans les cas où tout le code source est disponible (application ou librairie à sécuriser), mais plutôt une limitation pratique qui implique de modifier les scripts de compilation d'une application pour intégrer cette compilation en deux étapes, ce qui n'est pas toujours aisé dans un contexte industriel impliquant de multiples acteurs et une maintenance de longue durée.

4.4 Analyse sécuritaire pour architecture ARM

Dans cette partie, nous analysons la robustesse du schéma de protection proposé. Les capacités de l'attaquant considéré sont décrites dans la section 4.2.3. Un appel de fonction au niveau IR ne décrit pas comment chaque architecture gère les conventions d'appels, spécifiques à chaque architecture, et est trop éloigné du code final pour analyser directement l'apport de ce schéma au niveau IR. Nous avons donc choisi d'étudier la robustesse face à des sauts d'instructions et corruptions de registre au niveau assembleur, en sélectionnant le jeu d'instruction ARMv7, en cohérence avec PLAF.

4.4.1 Protection de l'arbre d'appel

Le schéma de protection de l'arbre d'appel est basé sur un équilibre entre mises à jour de variables à l'extérieur et à l'intérieur de la fonction sensible appelée. Toute corruption qui amène à exécuter l'une des mises à jour mais pas l'autre provoquera une différence détectable entre ces deux variables. À la sortie d'une fonction appelant une fonction sensible, des blocs de vérification permettent de contrôler cette différence.

Si la vérification est exécutée, toute corruption de registre ou saut d'instruction altérant l'une des mises à jour, que l'altération concerne le chargement d'une variable, sa mise à jour ou sa recopie en mémoire, provoquera une différence entre les deux variables et sera ainsi

détectée. Dans la fonction appelante, le saut de l'instruction d'appel empêche la fonction de s'exécuter ainsi que la mise à jour qu'elle contient. La différence entre les deux variables est donc détectée.

La protection face à un saut de l'instruction de retour dans la fonction appelée est plus complexe et dépend du schéma de suivi utilisé. Dans l'ISA ARMv7, l'appel à une fonction, *branch and link*, copie l'adresse de l'instruction suivante dans le registre `lr`. Si une fonction doit en appeler une autre, elle sauvegarde dans le prologue la valeur de ce registre sur la pile, via l'instruction `push {rx,..., lr}`. Le retour d'une fonction se fait en recopiant la valeur du registre `lr` dans le registre `pc` afin de continuer l'exécution dans la fonction appelante (soit directement via l'instruction `bx lr`, soit en le relisant depuis la pile via l'instruction `pop {rx,...,pc}`).

Dans le cas où l'instruction de retour d'une fonction *callee* n'est pas la dernière instruction du code de la fonction, la sauter revient à continuer dans cette même fonction *callee* et n'a pas d'impact sur l'enchainement des fonctions exécutées. Si les retours sont suivis (mode *CallRetSequence*), la faute sera néanmoins détectée car une deuxième mise à jour avant le retour réellement effectué sera exécutée. En revanche, lorsque le retour est la dernière instruction de la fonction *callee*, la sauter provoque l'exécution du code placé derrière. Ce code peut correspondre à des données alors traitées comme instructions et ainsi provoquer un comportement indéfini. Ce code peut aussi être celui d'une autre fonction *fall*. Lorsque cette fonction *fall* est exécutée correctement (i.e. pas de comportement incohérent), son retour va copier dans le registre `pc` la valeur qui était contenue dans le registre `lr` à l'entrée de cette fonction *fall*. Cette valeur dépend de la fonction *callee*. Si la fonction *callee* n'appelle pas d'autre fonction et n'a pas utilisé le registre `lr` comme registre de travail, la valeur du registre `lr` correspond à l'adresse de l'instruction suivant l'appel dans la fonction appelante et alors l'exécution va continuer normalement comme si le retour provenait de la fonction *callee* elle-même. En revanche, si la fonction *callee* en appelle d'autres, la valeur du registre `lr` correspond alors à l'adresse de l'instruction après le dernier appel exécuté dans la fonction *callee*. De plus, le registre `lr` peut servir de variable temporaire et ne pas nécessairement contenir une adresse. À cause d'optimisations de type *tail call optimization*, il arrive que la valeur du registre `lr` fasse directement revenir dans la fonction appelante. Pour ce dernier cas, on ne peut pas toujours assurer qu'une telle corruption peut être détectée, mais l'exploitation d'une telle faute ne semble pas triviale.

Lorsque le saut d'un appel provoque un comportement indéfini, par une mauvaise valeur du registre `lr` ou par l'exécution de données, il est peu probable que l'exécution continue sans erreur. Dans ce cas hypothétique, il est impossible de savoir dans quelle fonction l'exécution va continuer. Toutes les fonctions n'appellent pas nécessairement de fonctions sensibles et ne possèdent donc pas de blocs de vérification. Pour s'assurer d'une telle détection, il est utile de rajouter au moins un bloc de vérification à la fin de la fonction principale du code (*main*). Ce schéma ne cherche pas à couvrir à 100% contre toutes les fautes considérées, une étude

sécuritaire fine avec des métriques qui elles-mêmes n'existent pas, est laissée en question ouverte de ces travaux.

4.4.2 Protection des arguments

Le schéma de protection des arguments est également basé sur l'équilibre entre les mises à jour de deux variables. À l'entrée d'une fonction, les variables peuvent avoir des valeurs différentes mais doivent retrouver cette différence initiale à la sortie de la fonction. Selon le mode de protection utilisé, les régions de protection des arguments sont de taille variable.

Choisissons le mode *DefCallee* pour exemple, les autres modes fonctionnant de façon similaire. Toute faute, corruption de registre ou saut d'instruction, modifiant la valeur que l'argument possède à sa définition, altère de la même façon les deux mises à jour de variables et ne peut donc être détectée. De la même façon, toute faute dans la fonction appelée impacte l'utilisation de cet argument, mais les deux mises à jour ayant été exécutées avant que la faute ait lieu, les deux mises à jour sont effectuées normalement et ne peuvent donc pas permettre de détecter de faute. En revanche, une faute modifiant la valeur de l'argument après qu'il a été défini mais avant d'appeler la fonction altère la mise à jour effectuée dans la fonction appelée qui est alors différente de celle effectuée à la définition de l'argument. Cette différence permet donc de détecter une faute dans cette région déterminée entre la définition de l'argument et l'appel à la fonction.

Ce schéma résiste aux choix effectués par l'allocateur de registre, notamment en cas de *spill* de registres. En effet, les mises à jour ne dépendent que de la valeur de l'argument et non du registre qui le contient. Par conséquent, même si le registre utilisé lors de la définition est *spillé* ou n'est pas le même que celui utilisé au moment de l'appel, toute faute altérant la valeur de l'argument lors de l'appel (valeur mise en pile ou dans un registre) est détectée si celle utilisée pour la mise à jour dans l'appelant était correcte. Les mises à jour de compensation ajoutées sur les différents arcs du CFG de la fonction appelante ne permettent pas de différencier si l'appel a bien été effectué ou si l'un de ces arcs a été pris en cas de corruption altérant le flot de contrôle. Il est donc bien nécessaire de suivre l'appel pour protéger la valeur de l'argument, et donc de combiner la protection des arguments avec le suivi des appels.

4.5 Implémentation et résultats

Cette section présente la mise en œuvre du schéma de suivi des appels dans le compilateur LLVM ainsi que les résultats expérimentaux.

4.5.1 Implémentation dans LLVM

L'algorithme *CalleeSimo* présenté en section 4.3 a été, comme l'algorithme PLAF, implémenté comme passe de compilation dans LLVM version 6.0. Les modifications à apporter dépassant le cadre d'une unique fonction, la passe de compilation agit au niveau d'un module entier (unité de compilation) et non au niveau d'une fonction. La passe désactivée par défaut, s'active via l'ajout d'une option `-call-tracking=std` à l'appel du driver de compilation de `clang`.

4.5.1.1 Sélection des arguments et fonctions sensibles

Comme les boucles, toutes les fonctions d'un programme ne sont pas sensibles et n'ont pas besoin d'être sécurisées. De plus, pour celles qui le sont, il est nécessaire de fournir l'information au compilateur du mode de sécurisation choisi. Pour cela, un attribut de fonction similaire à celui utilisé pour les boucles peut être associé à chaque fonction sensible. Cet attribut se présente sous la forme `__attribute__((annotate(mode)))` où `mode` correspond au mode à appliquer parmi *CallCount*, *CallSequence* ou *CallRetSequence*. La sécurisation des valeurs d'arguments s'effectue de la même façon en ajoutant au type de la variable, dans le prototype de la fonction, un attribut de type de la forme `__attribute__((annotate(mode)))` où `mode` est l'un des trois définis pour le suivi des arguments, à savoir *CallerCallee*, *DefCallee* ou *DefUses*.

Pour simplifier les expérimentations visant à mesurer l'impact sur les performances et taille de code après une validation fonctionnelle, d'autres options ont été ajoutées au driver de compilation. Les options `-call-tracking=random -ct-rand=mode -ct-threshold=XX` permettent de sélectionner aléatoirement les fonctions à sécuriser, indépendamment de l'annotation utilisateur. Pour chaque fonction, une valeur aléatoire est tirée et la fonction est considérée sensible si cette valeur est inférieure au seuil défini par la valeur `XX` et est suivie avec le mode défini par l'option `-ct-rand`. Des options équivalentes (`-call-tracking-args=random -ct-rand-args=mode -ct-threshold-args=XX`) sont disponibles pour protéger les arguments de façon aléatoires. Elles intègrent le bloc de vérification dans les fonctions appelantes. Ces options permettent de tester les différentes variantes de la passe de sécurisation sur des codes conséquents sans avoir à sélectionner dans le code source des arguments sensibles et à ajouter manuellement les attributs associés.

4.5.1.2 Gestion des attributs de variables

La gestion des attributs de variable par LLVM diffère de celle des attributs de fonction même si la sémantique est la même. En effet, LLVM attache cet attribut à l'adresse de la variable ce qui empêche des passes d'optimisation comme *register promotion* de simplifier les accès mémoire par des variables scalaires quand cela est possible. Afin de garder possibles les optimisations classiques, nous avons ajouté une passe au début du *middle-end* qui convertit ces attributs en

appels virtuels temporaires. Ainsi, les optimisations classiques ne sont plus contraintes par ces attributs et peuvent fonctionner normalement tout en gardant l'information sur la sensibilité de l'argument pour la passe de sécurisation. La passe additionnelle de gestion des attributs a été ajoutée au flot de compilation et est également automatiquement appelée quand la sécurisation du graphe d'appel est activée. Cette passe est surtout un artefact pour contourner une limitation de LLVM dans la gestion de ces attributs. Comme elle doit être placée avant toutes les passes d'optimisations afin que celles-ci soient efficaces, nous avons placé cette passe au début du *middle-end*.

4.5.2 Environnement expérimental

Les expérimentations ont pour but d'évaluer la non altération fonctionnelle du code ainsi que l'estimation du coût de la contre-mesure. Pour évaluer la capacité à sécuriser tout type de fonction et d'argument, les expériences présentées dans cette section ont été réalisées avec l'option de compilation `-call-tracking=random` permettant de considérer aléatoirement les fonctions comme sensibles ou non, ainsi que l'option `-call-tracking-args=random` équivalente pour les arguments.

Compte tenu de la limitation qui contraint à avoir en une seul unité de compilation tout le code à sécuriser, nous avons restreint les codes testés à deux exemples :

- l'application `sqlite` utilisée pour tester l'algorithme plaf (cf. section 3.6.2.1) qui présente l'avantage d'avoir un code source regroupé en un unique fichier,
- la librairie de cryptographie sur courbes elliptiques `libecc-anssi` de l'*Agence nationale de la sécurité des systèmes d'information* (ANSSI) [4], pour laquelle le script de compilation a été adapté afin de générer les fichiers IR de chaque fichier source pour les réunir ensuite en un seul fichier.

L'environnement matériel est le même que celui utilisé pour les boucles dans la section 3.6.2.2, à savoir les architectures ciblées sont ARMv7 et x86_64 afin de compléter la validation.

4.5.3 Évaluation fonctionnelle

Les codes testés étant trop imposants pour être implémentés sur la carte à base de Cortex-M3, la validité fonctionnelle a été vérifiée sur le Raspberry Pi-3 et la plateforme X86_64. Le protocole expérimental de validation utilise le même script pour valider le bon fonctionnement de l'application `sqlite` sécurisée que celui utilisé pour la validation de l'algorithme PLAF (cf. section 3.6.2.1), et la librairie `libecc-anssi` possède une campagne de validation auto-testante dont nous avons pu nous servir pour valider le fonctionnement de ce deuxième code testé. Les tests réalisés avec les 5 principaux niveaux d'optimisation (`-O1`, `-O2`, `-O3`, `-Os`,

4. https://github.com/ANSSI-FR/libecc

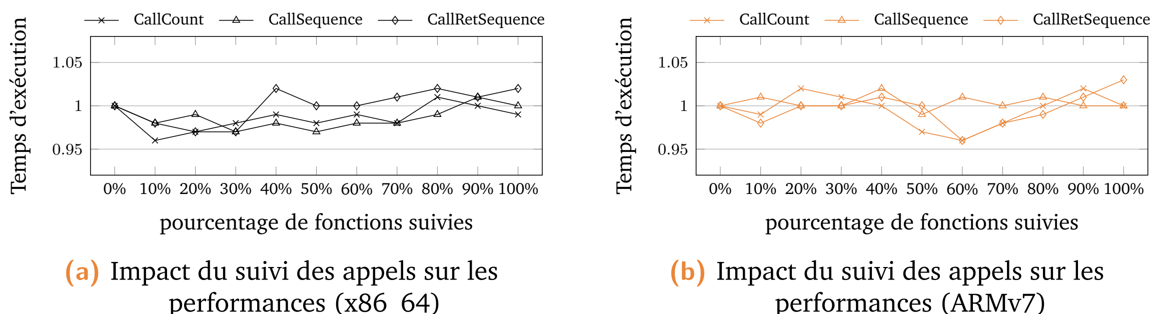
-Oz) n'ont reporté aucune erreur, validant la fonctionnalité des applications sécurisées. De plus, tous les tests considérant les fonctions et arguments comme étant sensibles de manière aléatoire présentés dans la section suivante ont été validés fonctionnellement.

4.5.4 Coût de la contre-mesure

Pour mesurer l'impact sur les performances et la taille du code de l'application de la contre-mesure, nous avons appliqué les différents modes de sécurisation disponibles sur l'exemple de la librairie `libecc-anssi` décrite précédemment. Plusieurs seuils de sécurisation aléatoire ont été utilisés afin de montrer l'impact en fonction du nombre d'arguments ou de fonctions à sécuriser : de 0 à 100% par pas de 10%. En pratique seules quelques fonctions et quelques arguments sont considérés sensibles. Il est donc plus probable que le coût réel de la sécurisation d'une application soit proche des faibles pourcentages. Néanmoins, les autres valeurs donnent des estimations en cas de proportion plus importante de fonctions à sécuriser et fournissent également une borne supérieure du coût.

4.5.4.1 Impact sur les performances

L'impact sur les performances a été mesuré comme pour la contre-mesure PLAF (cf. section 3.6.5) en exécutant les applications 10 fois et en moyennant le temps obtenu en excluant les valeurs extrêmes. La figure 4.7 présente le surcoût en temps d'exécution induit par le suivi des appels en fonction de la proportion de fonctions à suivre et du mode utilisé pour le suivi. De la même manière, la figure 4.8 présente le surcoût en temps d'exécution induit par la protection des arguments en fonction de la proportion d'arguments à protéger et du mode utilisé la protection.



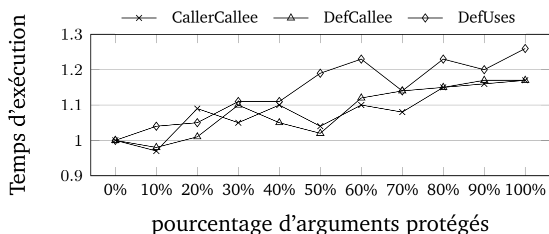
(a) Impact du suivi des appels sur les performances (x86_64)

(b) Impact du suivi des appels sur les performances (ARMv7)

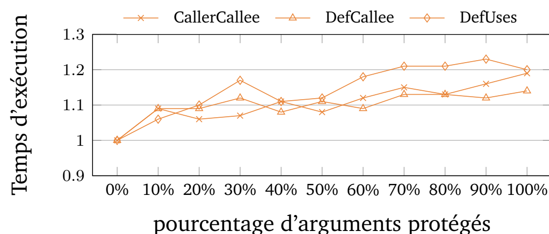
Figure 4.7: Impact du suivi des appels de fonction sur les performances en fonction du pourcentage de fonctions à sécuriser (-Os, ratio entre code sécurisé et code original)

On peut constater que les mises à jour de variables pour le suivi des fonctions n'a quasiment aucun impact sur les performances, la faible variations des courbes est plutôt due à l'imprécision de la mesure (moyenne sur 10 exécutions). Les résultats obtenus sont similaires sur les deux plateformes. La protection des arguments apporte un surcoût allant jusqu'à 25% du temps d'exécution lorsque l'intégralité des arguments sont sécurisés, avec une progression

légèrement sous-linéaire. La protection des arguments ajoute davantage de mises à jour dans le code mais, hors boucle, une seule est exécutée, limitant ainsi l'impact sur les performances. Toutefois, le nombre d'argument à sécuriser est plus élevé que le nombre de fonctions (2,3 arguments par fonction en moyenne sur cet exemple). Il est donc normal que le coût soit plus élevé, il reste cependant limité.



(a) Impact de la protection des arguments sur les performances (x86_64)

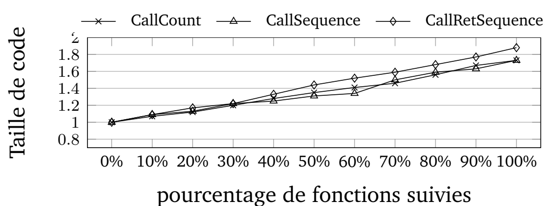


(b) Impact de la protection des arguments sur les performances (ARMv7)

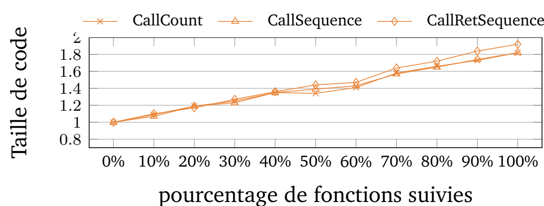
Figure 4.8: Impact de la sécurisation des arguments sur les performances en fonction du pourcentage d'arguments à sécuriser (-Os, ratio entre code sécurisé et code original)

4.5.4.2 Impact sur la taille du code

La figure 4.9 présente l'impact du suivi des appels de fonction sur la taille du code en fonction du nombre de fonctions à suivre et du mode utilisé pour tracer ces fonctions. Le ratio présenté est celui entre la taille du code sécurisé et celle du code original. La taille de code étant un élément important dans les systèmes embarqués, nous présentons les résultats obtenus au niveau d'optimisation -Os. De la même manière, la figure 4.10 illustre le ratio entre la taille du code où les arguments sont protégés par rapport à celle du code original. Elle montre l'impact, sur la taille de code, de la proportion d'arguments à sécuriser dans les trois modes de sécurisation proposés, au niveau d'optimisation -Os.



(a) Impact du suivi des fonctions sur la taille de code (x86_64)

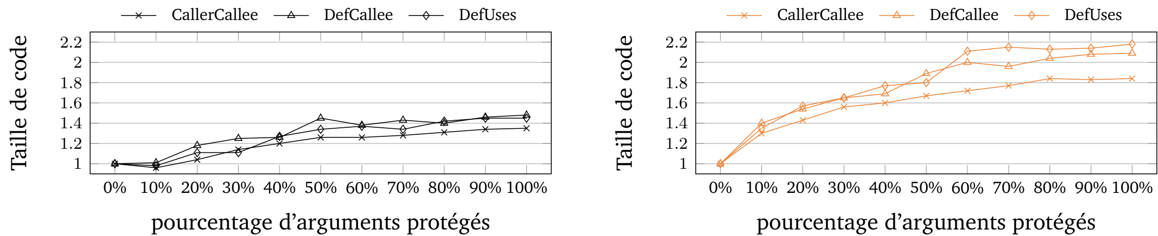


(b) Impact du suivi des fonctions sur la taille de code (ARMv7)

Figure 4.9: Impact du suivi des appels de fonction sur la taille de code en fonction du pourcentage de fonctions à sécuriser (-Os, ratio entre code sécurisé et code original)

L'ajout de mises à jour pour le suivi des fonctions impacte de façon linéaire la taille du code en allant de 0% à 90% de taille en plus en fonction du pourcentage de fonction suivies. Ceci s'explique en partie par la présence dans la libraire de certaines fonctions originales et de leur version sécurisée. L'impact en taille de code de la sécurisation des arguments est plus varié, allant de 0% à 50% de surcoût sur architecture x86_64 et de 0% à 120% sur architecture ARMv7. Les schémas reposant sur des analyses complexes du flot de contrôle, de multiples

blocs peuvent être ajoutés dans les cas proches des 100% d'arguments à protéger, expliquant les chiffres plus élevés pour ces valeurs. Les exemples de librairies cryptographiques expliqués en section 4.2.1 n'ont souvent qu'un seul argument à sécuriser par fonction. Les résultats correspondants aux faibles valeurs de pourcentage d'arguments à protéger sont donc les plus réalistes.



(a) Impact de la protection des arguments sur la taille de code (x86_64)

(b) Impact de la protection des arguments sur la taille de code (ARMv7)

Figure 4.10: Impact de la sécurisation des arguments sur la taille de code en fonction du pourcentage d'arguments à sécuriser (`-Os`)

Lorsque les fonctions doivent être suivies et les arguments protégés en même temps, l'application simultanée des deux schémas coute moins cher que s'ils avaient dû être appliqués successivement, réduisant le surcout de 25% sur ARM et de 30% sur x86_64 dans le pire cas (*CallRetSequence* + *DefUses* avec 100% de fonctions et d'arguments sécurisés).

4.6 Discussion sur la complémentarité avec la sécurisation des boucles

Les contre-mesures présentées dans ce chapitre et dans le précédent ont pour but de sécuriser de façon automatique et légère des parties sensibles de codes. Les appels de fonction sont l'un des points sensibles d'un programme. En plus de cette sensibilité, la présence d'appels de fonction dans l'ensemble des instructions à dupliquer de la contre-mesure PLAF est la principale cause empêchant la sécurisation des boucles de pouvoir s'appliquer automatiquement à la compilation. Disposer de ces deux contre-mesures dans le même compilateur permet de choisir avec précision quelle contre-mesure utiliser pour quel code, et offre aussi la possibilité de cumuler les deux contre-mesures.

Plus précisément, la contre-mesure PLAF effectue une analyse arrière de dépendances afin de déterminer les instructions à dupliquer. Lorsque cette analyse rencontre un appel de fonction, la sécurisation de la sortie de boucle ne peut avoir lieu. La limitation vient de l'impossibilité de dupliquer l'appel à la fonction sans information spécifique. Cet appel pourrait alors être protégé par le schéma *CalleeSimo* présenté dans ce chapitre afin de donner un moyen de s'assurer qu'il a bien été exécuté et que ses argument avaient les bonnes valeurs. Il serait alors envisageable d'interagir avec l'algorithme PLAF afin de continuer l'analyse arrière à

partir de la définition de chacun de ces arguments afin de les dupliquer, via la duplication des instructions participant à leur calcul. Un bloc de vérification comparant les valeurs des arguments originaux et dupliqués serait ainsi ajouté juste après chaque définition d'argument afin de détecter une éventuelle corruption de leur valeur. Avec un tel schéma, il serait donc possible de sécuriser plus largement les boucles (avec prise en compte des sorties de boucles dont la condition dépend d'appel de fonction) : les instructions dupliquées par le schéma PLAF jusqu'aux définitions des arguments protègent contre des corruptions des arguments en amont de l'appel dans la boucle. Le schéma *CalleeSimo* assure que les arguments ont la bonne valeur jusqu'à l'appel (voire jusqu'à leurs utilisations dans la fonction appelée), et les instructions dupliquées par PLAF de l'appel à la fonction jusqu'à la condition de sortie de la boucle protègent contre les corruptions des valeurs en aval de l'appel. La limitation de cette combinaison de protection est que la valeur de retour de la fonction appelée n'est pas protégée : des fautes réalisées durant l'exécution de cette fonction peuvent corrompre cette valeur de retour. Néanmoins, la combinaison de ces contre-mesures permet d'étendre la protection des boucles avec une protection partielle des appels de fonction impactant une condition de sortie de la boucle.

Cette utilisation complémentaire des deux passes de compilation apportant deux schémas de contre-mesure distincts met en avant la capacité à combiner les contre-mesures de manière intelligente. Proposer plusieurs contre-mesures à la compilation permet en effet de repousser les limites des schémas de protection unitaires.

4.7 Conclusion

Dans ce chapitre, nous avons présenté la conception d'un schéma générique de suivi de l'arbre d'appel intégrant une protection des arguments de fonction. Ce schéma fournit plusieurs variantes tant au niveau du suivi des appels qu'au niveau de la protection des arguments, afin de fournir au développeur des schémas de protection avec coût et couverture variables lui permettant ainsi de s'adapter à ses contraintes. Les objectifs de sécurisation couverts par le suivi des appels sont d'assurer le bon nombre d'appels, la bonne séquence d'appels ou encore la bonne séquence d'appels et de retours. Les variantes de protection des arguments couvrent des régions de protection de la valeur de l'argument en termes de surface d'attaque plus ou moins grandes, soit autour de l'appel, depuis la définition de l'argument jusqu'à l'entrée de la fonction appelée ou encore jusqu'à ses utilisations dans la fonction appelée. L'élargissement de la région de protection d'un argument nécessite une analyse du flot de contrôle des fonctions appelantes et appelées qui serait délicate à appliquer manuellement. Toutes ces variantes peuvent être directement utilisées par le développeur en ajoutant une annotation au niveau du code source dans le prototype de la fonction à tracer ou du type de l'argument à protéger.

Ces mécanismes de traçage des appels sont souvent ajoutés manuellement dans l'industrie. Nous apportons une automatisation de ce schéma de protection, au travers d'algorithmes que nous avons conçus pour être implémentés au niveau IR d'un compilateur et que nous avons mis en œuvre au sein du compilateur LLVM. Le coût relativement faible et adaptable de la contre-mesure permet une intégration légère dans des codes sensibles de systèmes embarqués. La version actuelle ne protège pas les appels indirects mais pourrait être modifiée en intégrant des annotations manuelles du développeur afin de fournir des informations essentielles à la compilation pour assurer que les fonctions dynamiquement appelées sont statiquement autorisées à être appelées pour un site d'appel donné.

Les deux contre-mesures présentées dans ce chapitre et le précédent considèrent des attaquants aux capacités similaires mais toutefois pas identiques. La protection apportée est fortement liée aux capacités de l'attaquant. Les analyses effectuées sur cette contre-mesure et les simulations réalisées sur la contre-mesure PLAF ont permis de mettre en évidence l'efficacité des protections proposées contre les attaquants définis, à modifications du back-end près. Les modèles de fautes utilisés sont des modèles théoriques représentant les effets observables des fautes que l'attaquant considéré peut réaliser. Ces modèles sont représentatifs des effets observés sur les composants utilisés dans l'industrie [Moro et al., 2013]. Le modèle de faute dépend toujours de la cible et du moyen d'injection, donc de l'attaquant. Les modèles considérés sont les plus communs rencontrés et considérés par les industriels concernés par les attaques en faute. Toutefois, la réalité ne colle jamais à 100% à un modèle théorique. Seule une caractérisation réelle d'une cible au travers d'expérimentations permet de statuer sur le type de faute possible et sur l'efficacité d'une protection. Le chapitre suivant se place au niveau d'un attaquant cherchant à exploiter des failles suite à des injections réelles. Nous considérons des processeurs complexes pour tester les contre-mesures et comprendre les effets possibles sur ces processeurs.

5 Caractérisation des fautes au niveau ISA

Sommaire

5.1 Introduction **107**
5.2 Environnement d'attaque **109**
5.2.1 Composant cible des attaques 109
5.2.2 Banc d'attaque et paramétrage 109
5.2.3 Processus expérimental 112
5.3 Première analyse de sensibilité aux fautes **114**
5.3.1 Codes cibles 115
5.3.2 Campagne d'attaque sur les boucles standards 117
5.3.3 Campagne d'attaque sur les boucles sécurisées 118
5.4 Caractérisation de fautes **121**
5.4.1 Méthodologie 121
5.4.2 Code initial : séquence de `nops` 122
5.4.3 Séries d'additions 122
5.4.4 Résumé des effets et classification 129
5.5 Retour sur les boucles **130**
5.5.1 Analyse des fautes sur boucles 130
5.5.2 Répétabilité des fautes 132
5.5.3 Classification des fautes sur les boucles 133
5.6 Conclusion **136**
5.7 Annexe des attaques **138**

5.1 Introduction

Dans les chapitres précédents, nous avons proposé des contre-mesures logicielles insérées automatiquement à la compilation pour des boucles et des appels de fonctions sensibles. Leur but est de protéger ces parties sensibles contre une unique faute dont les effets se modélisent par un saut d'instruction ou une corruption de registre. Ces modèles de faute ont été choisis car ils sont représentatifs d'effets observés sur des plateformes sécurisées, soumises à certification, comme les cartes à puce ou les passeports [Moro, 2014].

Les attaques sur ces architectures ont été largement étudiées depuis une vingtaine d'années, poussées par les exigences de certification des produits et la recrudescence des attaques due à l'essor du paiement par carte bancaire. La diversification des produits et des architectures sous-jacentes, embarquant des données sensibles, pose aujourd'hui la question de l'effet des attaques physiques sur ces plateformes (téléphone mobile, IoT). De plus, l'automatisation des contre-mesures permet une application systématique de sécurisation dans les systèmes d'exploitation, souvent partagés entre plusieurs plateformes, nécessitant une évaluation de l'efficacité des protection sur ces plateformes variées. De récents travaux ont montré la possibilité d'exploiter des injections de fautes sur des architectures complexes, notamment en utilisant des injections laser ou par perturbation de la tension d'alimentation [Vasselle et al., 2017, Timmers et al., 2016, Timmers and Mune, 2017].

Dans ce chapitre, nous nous intéressons aux effets des injections d'impulsions EM sur des processeurs plus complexes et à leur capacité à altérer des codes divers : nous montrons que l'on peut fauter le nombre d'itérations de boucles sensibles protégées ou non. Plus précisément, nos injections en boîte noire visent une architecture superscalaire à base de processeur ARM Cortex-A9 représentative des architectures ciblées par ces attaques récentes. Nous proposons une méthodologie de caractérisation des fautes sur ces composants et nous investiguons les effets observés avec cette démarche méthodologique pour pallier le manque d'information sur le processeur cible : à partir de codes simples, nous tirons des conclusions sur les effets possibles et nous accroissons la complexité des codes visés. Cela nous permet de caractériser au niveau ISA les fautes possibles : des fautes simples et des fautes multiples combinant des modèles de fautes déjà rencontrés sur processeurs simples et d'autres jamais reportés dans la littérature à notre connaissance. Notamment, nous observons des substitutions d'opérandes ou l'ajout d'arcs dans le CFG. De plus, notre étude met en évidence une forte dépendance des valeurs corrompues avec les valeurs présentes dans d'autres instructions en cours de traitement. L'ensemble de cette caractérisation permet d'expliquer pourquoi les protections des boucles ne sont pas suffisantes, même en superposant deux contremesures.

La suite du chapitre est composée d'une description de l'environnement d'attaque dans la section 5.2. Des campagnes d'injection illustrant la possibilité d'injecter des fautes avec cet environnement sur des codes avec et sans protection sont présentées dans la section 5.3. La section 5.4 utilise une démarche incrémentale afin de caractériser les effets observés de ces fautes. Enfin, nous analysons, dans la section 5.5, les effets observés sur des codes avec et sans ajout de protections logicielles grâce à cette caractérisation que nous enrichissons. Cela nous permet de conclure, dans la section 5.6, sur la complexité des effets observables sur des architectures complexes, ainsi que sur l'importance du modèle de faute dans la conception d'une contre-mesure dont l'efficacité est réduite face à des effets plus complexes que ceux considérés. Les résultats de ce chapitre ont été acceptés pour publication et sont présentés à la conférence *ARES 2019* [1] en Août 2019 à Canterbury (UK).

1. https://www.ares-conference.eu/

5.2 Environnement d'attaque

Cette section détaille l'environnement expérimental utilisé pour réaliser des campagnes d'injections de fautes, à savoir le banc d'attaque, le processeur visé et leur articulation autour d'un ordinateur de contrôle, ainsi que le contexte logiciel dans lequel s'insère le code ciblé par les attaques.

5.2.1 Composant cible des attaques

L'ensemble des sessions d'attaques a été réalisé sur un SoC largement répandu dans le contexte de l'automobile ainsi que dans l'IoT. Il est composé d'un processeur iMX6, basé sur un double-coeur 32 bits ARM Cortex-A9 MPCore qui implémente le jeu d'instructions ARMv7a, dont la fréquence peut atteindre 1 GHz. La fréquence utilisée pour les différentes expérimentations est de 80 MHz. La micro-architecture du Cortex-A9 possède plusieurs propriétés qui le différencie des processeurs souvent étudiés, comme dans les micro-contrôleurs des études précédentes [Moro et al., 2013, Yuce et al., 2016] :

- pipeline à 8 étages à latence variable ;
- décodeur d'instructions superscalaire de degré 2 ;
- renommage de registres avec écriture dans le désordre ;
- prédicteur de branchements, avec un *Branch Target Buffer* (BTB) d'au moins 512 entrées ;
- tampon de 64 octets dédié aux boucles qui contourne le cache d'instructions (*loop buffer*).

La hiérarchie mémoire possède 3 niveaux : deux caches de données et d'instructions de niveau 1 de 32 Ko chacun, un cache unifié de niveau 2 de 256 Ko, ainsi que différentes interfaces de mémoires externes comme de la mémoire Flash `NAND`, de la mémoire Flash `NOR` et de la mémoire RAM DDR3. Ce SoC intègre des blocs multimédia, de gestion du réseau, de la gestion de l'énergie ainsi que des paramètres de sécurité comme une TrustZone [2]. Il est intégré dans une carte de développement incluant des ports de communication série (UART) et des signaux *General Purpose Input-Output* (GPIO) nécessaires aux interactions avec le banc d'attaque. Une photographie de ce SoC est présentée en figure 5.1.

5.2.2 Banc d'attaque et paramétrage

Pour réaliser des attaques par injection d'impulsions électromagnétiques, nous avons utilisé le banc d'attaque présenté en figure 5.2. Ce banc est composé de plusieurs éléments distincts :

2. https://www.arm.com/files/pdf/TrustZone-and-FIDO-white-paper.pdf

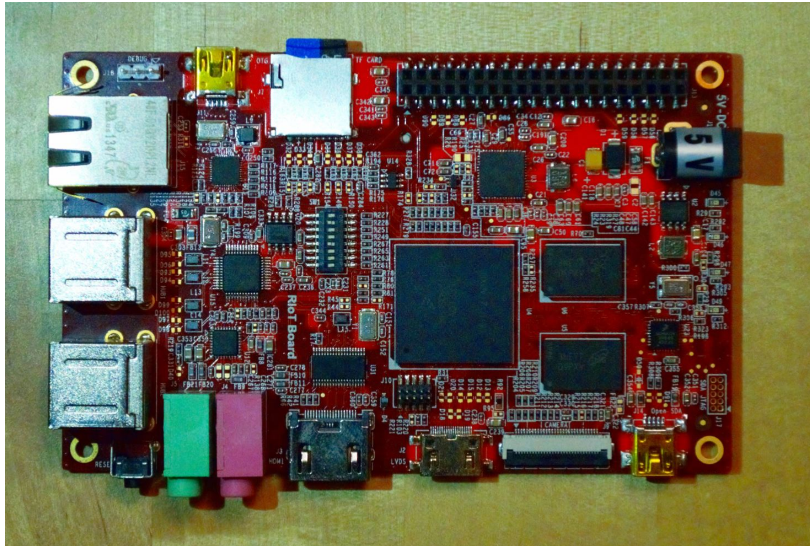


Figure 5.1: Photographie du SoC choisi pour les expérimentations

- une table XYZ motorisée dont la coordonnée Z n'est pas modifiée au cours des campagnes d'injection, mais qui peut être réglée manuellement à l'initialisation ;
- un générateur d'impulsions capable de générer des impulsions d'une durée variant de 6 ns à 150 ns et dont la tension peut être fixée à une valeur entre -400 V et 400 V ;
- un oscilloscope numérique 8 bits capable d'acquérir simultanément 4 signaux, via des canaux différents, avec une bande passante de 4 GHz et une fréquence d'échantillonnage pouvant aller jusqu'à 40 GS/s ;
- une *sonde d'injection* en forme de bobine de cuivre entourant un cœur de ferrite afin de concentrer le champ électromagnétique ;
- une *sonde de mesure* d'émanations électromagnétiques également composée d'un cœur de ferrite entourée de fils de cuivre ;
- un ordinateur qui synchronise les différentes actions nécessaires avec les signaux GPIO de la carte ;

Afin de pouvoir injecter une faute dans un circuit à l'aide d'impulsions électromagnétiques grâce à ce banc d'attaque, plusieurs paramètres doivent être configurés, comme le placement de la sonde d'injection, l'instant et la durée de l'injection, ainsi que sa polarité et sa tension [Dureuil et al., 2016b].

Le but de notre démarche n'est pas d'étudier l'impact des différents paramètres mais de rechercher une configuration qui permet d'obtenir une faute avec une probabilité suffisamment élevée afin d'effectuer nos campagnes d'attaques dans cette configuration. Cette démarche est classique pour faire face au nombre trop important de configurations qui ne peuvent être exhaustivement testées [Dureuil et al., 2016b].

Les configurations utiles pour étudier les effets d'injections EM sont celles avec un pourcentage de fautes visibles raisonnable. Réduire la tension d'injection réduit la proportion de faute.

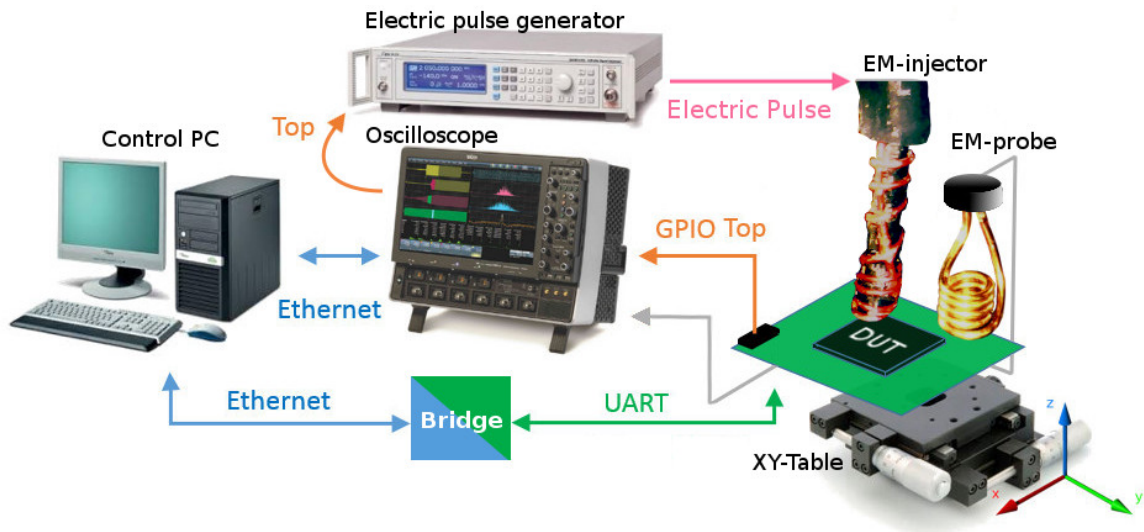
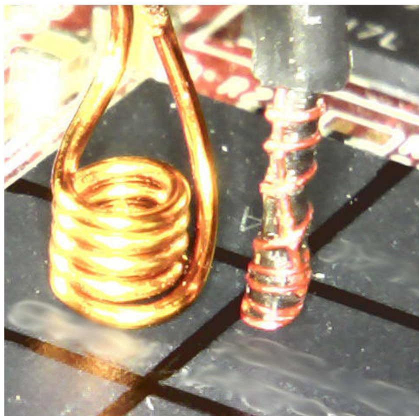


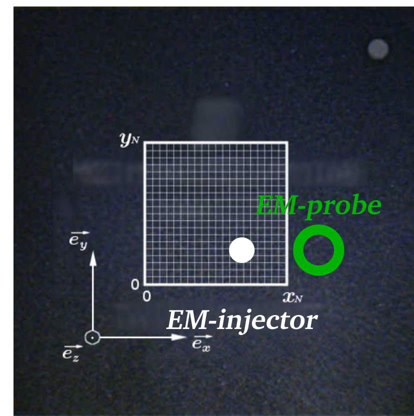
Figure 5.2: Schéma du banc d'attaque électromagnétique

En revanche, augmenter la tension augmente le pourcentage d'injections après lesquelles le composant ne répond plus ; l'effet d'une telle faute n'est pas analysable et n'est donc pas souhaité. Il est donc important de définir une tension d'injection qui fournit une proportion de faute visible raisonnable.

Cet environnement d'attaque a précédemment été utilisé dans les travaux de thèse de Fabien Majéric [Majéric, 2018]. Son étude a montré l'existence d'une zone sensible autour du point d'intérêt (x, y) présenté en figure 5.3 en configurant une durée d'injection de 6 ns et une tension de +310 V.



(a) Photographie des sondes d'injection et de mesure au dessus du SoC ciblé



(b) Positionnement des sondes au-dessus du SoC ciblé

Figure 5.3: La sonde de mesure et la sonde d'injection utilisées

Pour confirmer que cette configuration est intéressante, nous avons effectué un balayage spatial autour de ce point d'intérêt en gardant cette tension de +310 V de référence et en utilisant un pas de 50 μm dans les axes X et Y. La proportion d'injections où la carte n'a pas

répondu est représenté sur la figure 5.4. On constate qu'en s'écartant du point d'intérêt, cette proportion augmente trop (coin supérieur gauche) ou alors diminue au point de devenir nulle (coin inférieur droit). Dans ces deux cas, la proportion de fautes visibles n'est pas optimale, ce qui confirme que le point d'intérêt initial est une position adaptée pour la suite de notre étude.

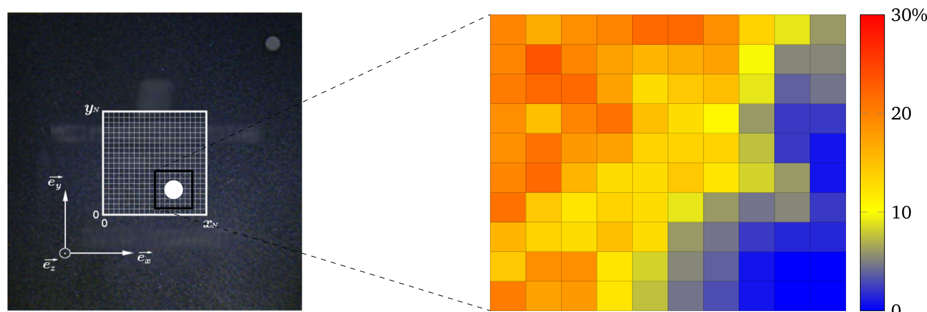


Figure 5.4: Proportion de non-réponse (inexploitable) du composant ciblé après injection électromagnétique en fonction du positionnement en x et y de la sonde d'injection

Dans la suite, nous appelons *campagne d'attaque* une série d'expériences ciblant le même code. Chaque expérience correspond à une unique impulsion électromagnétique émise avec les mêmes paramètres (tension, durée, position), et dont l'instant d'injection peut varier. Dans ces campagnes, cette variation de l'instant d'injection est réalisée dans un intervalle de temps préalablement déterminé. Ceci est expliqué dans la section suivante.

5.2.3 Processus expérimental

5.2.3.1 Détermination de l'intervalle pour injection

L'instant d'injection d'une impulsion EM est défini grâce à une analyse de la trace électromagnétique reçue depuis la sonde et du signal GPIO. Afin de détecter la fenêtre temporelle dans laquelle réaliser des injections, le code ciblé est instrumenté de la façon suivante : On lève le signal GPIO, puis une première séquence de 200 `nops` s'exécute avant le code cible lui même, suivi d'une seconde séquence de 200 `nops` avant de redescendre le signal GPIO. Lors de l'exécution de ces séquences de `nops`, le processeur consomme moins et le rayonnement électromagnétique émis est plus faible. Cela permet de détecter, sur la trace, les régions correspondant à ces séquences et donc la région d'exécution du code visée. La figure 5.5 fournit un exemple de trace EM obtenue lors de l'exécution d'un code composé d'une boucle de copie mémoire comportant 8 itérations. Dans cet exemple, une impulsion EM a été injectée à la cinquième itération.

Sur cette figure, on peut observer les différentes itérations de la boucle (correspondant aux petits pics), les séquences de `nops` (parties plus lisses) ainsi que l'injection de l'impulsion générant un puissant pic dépassant largement l'échelle de la figure. La combinaison de ces

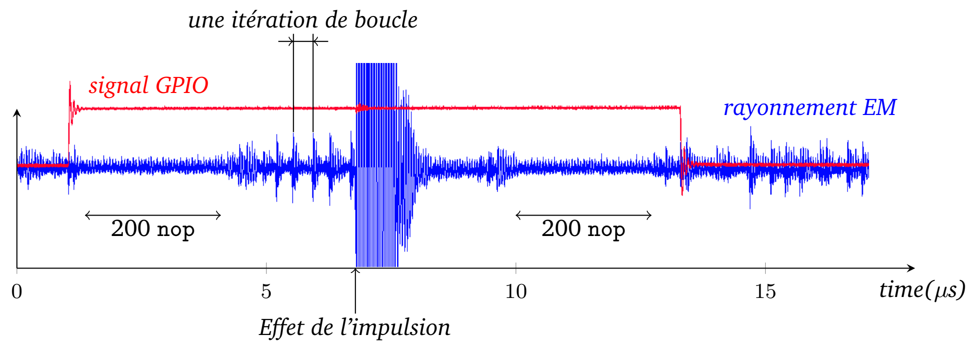


Figure 5.5: Trace électromagnétique d'injection d'impulsion EM lors de l'exécution d'une boucle et signal GPIO correspondant

séquences de `nops` et du signal GPIO permet de déterminer deux éléments de configuration : à la fois le délai entre la levée du signal GPIO et l'injection de l'impulsion EM, et la durée de l'intervalle pour couvrir une portion significative de l'exécution du code ciblé. Par exemple, dans le cas d'une boucle, il est raisonnable de couvrir plusieurs itérations. Les campagnes d'attaques balayent l'intervalle défini en injectant des fautes avec un pas de 5 ns jusqu'à en couvrir l'intégralité. Finalement, lors de nos expériences, le délai entre la levée du signal GPIO et l'instant d'injection est le seul paramètre qui varie au cours d'une campagne d'injections. Cette variation permet d'injecter des impulsions EM alors que le processeur exécute diverses instructions du code ciblé.

5.2.3.2 Instrumentation du code visé

Pour observer et analyser les effets d'une injection, on a besoin de récupérer des informations. Pour cela, chaque programme ciblé se termine par l'ajout d'une série d'instructions permettant de sauvegarder le contenu des registres généraux (de `r0` à `r14`) ainsi que le contenu des plages mémoires utilisées par le programme dans une *mémoire tampon de sortie* dédiée. Cette mémoire est ensuite envoyée à l'ordinateur de traitement. Pour faciliter l'analyse des informations obtenues, nous avons fait plusieurs choix d'implémentation :

- Les registres généraux sont initialisés avec des valeurs spécifiques, *en V*, comme présenté dans le listing 5.1. Le but est de n'avoir que des valeurs à faible poids de Hamming, toutes distinctes, sans bit commun à 1 et sans relation arithmétique triviale entre deux registres. Il est ainsi plus facile d'analyser les valeurs en cas de corruption (multiple) de registres.
- Le contenu des plages mémoire utilisées par le programme ciblé est également initialisé avec des valeurs spécifiques de façon similaire et dépendant du code cible.
- Les codes ciblés sont écrits en langage C et sont compilés avec LLVM version 6.0 au niveau d'optimisation `-O2`. Les codes assembleurs générés sont ensuite instrumentés et réécrits afin d'utiliser au maximum les registres disponibles et ainsi garder le plus possible de valeurs utiles en fin de programme, contrairement au compilateur dont l'allocateur de registre cherche à minimiser le nombre de registres utilisés.

```
 r0 = 0x80000001;                  // 0b10000000000000000000000000000001
 r1 = 0x40000002;                  // 0b01000000000000000000000000000010
 r2 = 0x20000004;                  // 0b00100000000000000000000000000100
 r3 = 0x10000008;                  // 0b00010000000000000000000000001000
 r4 = 0x08000010;                  // 0b00001000000000000000000000010000
 r5 = 0x04000020;                  // 0b00000100000000000000000000100000
 r6 = 0x02000010;                  // 0b00000010000000000000000001000000
 r7 = 0x01000080;                  // 0b00000001000000000000000010000000
 r8 = 0x00800100;                  // 0b00000000100000000000000100000000
 r9 = 0x00400200;                  // 0b00000000010000000000001000000000
r10 = 0x00200400;                  // 0b00000000001000000000010000000000
r11 = 0x00100800;                  // 0b00000000000100000000100000000000
r12 = 0x00081000;                  // 0b00000000000010000001000000000000
```

Listing 5.1: Initialisation *en V* des registres généraux avant exécution du code cible

5.2.3.3 Processus d'analyse

Le déroulement d'une campagne d'attaques est le suivant : le code ciblé est exécuté dans son environnement de test sans injection d'impulsion EM afin de récupérer, grâce à l'instrumentation, les valeurs attendues dans la mémoire tampon de sortie. Les exécutions suivantes sont soumises à des injections d'impulsions EM pendant l'intervalle défini précédemment, et les valeurs résultantes sont comparées avec celles attendues afin d'analyser l'effet de l'injection. Enfin, grâce à cette comparaison, les exécutions soumises à injections d'impulsions EM sont classées dans l'une des trois catégories suivantes :

- *sans effet* : le contenu de la mémoire tampon de sortie est identique à celle de référence. L'injection n'a donc aucun effet visible sur le programme ;
- *faute injectée* : le contenu de la mémoire tampon de sortie diffère de celle de référence pour au moins une valeur. L'injection a réussi à provoquer une faute dont l'impact peut être analysé ;
- *muet* : le composant ne répond plus. Le statut à la fois du composant et de la mémoire tampon de sortie est indéfini. Dans ce cas, le composant a besoin d'être réinitialisé.

Chacune des exécutions avec faute injectée peut être analysée afin d'identifier les différents effets de l'injection EM au niveau ISA. Nous concentrons donc notre étude sur ces cas de fautes injectées. Les cas où la carte est rendue muette pourraient toutefois être classés par l'analyse de la trace EM associée, avec moins de précision.

5.3 Première analyse de sensibilité aux fautes

Dans cette partie, nous présentons les résultats d'une campagne d'attaques sur un exemple de code ayant pour but de valider la capacité à injecter des fautes avec le banc d'attaque et la configuration choisie. Si les essais ont montré qu'il est relativement facile de rendre la carte muette sans ajuster les paramètres d'injection avec précision, injecter des fautes est plus rare, mais reste possible avec les paramètres précédemment décrits. Les exemples de code

choisis sont basés sur des boucles. Ainsi, nous pouvons viser des codes non sécurisés mais également leur appliquer une contre-mesure logicielle existante ([Reis et al., 2005]) ainsi que notre algorithme de protection des boucles. Cela nous permet d'évaluer la capacité de l'environnement à fauter même en présence de protections logicielles, la sécurité recherchée pour les boucles étant la garantie du bon nombre d'itérations et de la bonne sortie le cas échéant.

5.3.1 Codes cibles

Pour cette première analyse, nous avons sélectionné deux boucles représentatives de codes sensibles et à la fois relativement simples et de petite taille. Cela nous permet d'analyser les résultats dans un temps raisonnable, pour ne pas rendre cette analyse manuelle trop fastidieuse.

5.3.1.1 Boucle de type *memcpy*

La première boucle choisie, dont le code source est décrit dans le listing 5.2, provient d'une fonction qui copie un tampon mémoire dans un autre. Elle est similaire à beaucoup de boucles que l'on peut trouver dans les mises à jour de microcode et qui sont susceptibles d'être la cible d'attaques de type *buffer overflow* [Nashimoto et al., 2016]. Elle sera notée *Boucle1* dans la suite.

```
void Boucle1(char *dst, char *src, int n) {
    for (int i=0; i<n; i++) {
        dst[1+i] += src[i];
    }
}
```

Listing 5.2: Code source de la boucle de type *memcpy*

Afin de faciliter l'analyse des effets des fautes produites par l'injection d'impulsions EM, au lieu de copier le contenu du tampon source dans le tampon destination, le contenu de ce dernier est incrémenté par les valeurs du tampon source. Les tampons `src` et `dst` sont initialisés avant l'appel à la fonction et leur contenu est recopié après l'exécution de la fonction. Comme indiqué dans la section 5.2.3, le code généré après compilation est instrumenté manuellement. Le code assembleur final est présenté dans le listing 5.3, où les registres sont initialisés à l'entrée de la fonction et sauvegardés juste avant le retour. Un compteur supplémentaire est ajouté afin de pouvoir tracer le nombre d'itérations de la boucle. Ce compteur de traçage (`j`) utilise le registre `r5`. Il est initialisé avec la valeur 7 (*ligne 8*) et est incrémenté de 3 à chaque itération (*ligne 11*), afin d'utiliser des constantes différentes de celles utilisées pour le compteur de boucle original.

Le lecteur attentif observera sur le listing 5.3 qu'au niveau d'optimisation `-O2`, le compilateur a renversé le sens du compteur de boucle initial `i` (*lignes 4, 7 et 12*). Cela illustre l'effet typique

```
1   Boucle1:
2      push {r4, r5, r6, r7, r8, r9, r10, r11, lr}
3      [...]                        ; initialisation des registres en V
4      cmp r2, #0                   ; i = 0?
5      beq .exit
6      add r3, r0, #1               ; p_dst = dst[1]
7      mov r0, r2                   ; i = n
8      mov r5, #7                   ; j = 7
9   .loopbody:
10     ldrb lr, [r1], #1            ; p_src++
11     adds r5, r5, #3              ; j += 3
12     subs r0, r0, #1              ; i--
13     ldrb r4, [r3]
14     add r4, r4, lr
15     strb r4, [r3], #1            ; p_dst++
16     bne .loopbody
17  .exit:
18     [...]                        ; sauvegarde des registres
19     pop {r4, r5, r6, r7, r8, r9, r10, r11, pc}
```

Listing 5.3: Code assembleur de la boucle de type *memcpy* du listing 5.2 après instrumentation

de passes d'optimisations du compilateur capables de modifier les variables d'induction, et susceptible d'interférer avec l'ajout de sécurisation à la compilation (cf. section 3.5.1.2).

5.3.1.2 Boucle de type *memcmp*

La seconde boucle choisie est tirée d'une fonction de comparaison de tampons mémoire avec une sortie prématurée en cas de différence. Le code source de cette fonction est représenté dans le listing 5.4. De telles boucles sont sensibles dans des schémas d'authentification ou de vérification de code PIN [Dureuil et al., 2016a]. Elle sera notée *Boucle2* dans la suite.

```c
void Boucle2(int *src1, int *src2, int *dst, int n) {
    for (int i=0; i<n; i++) {
        dst[i] = (src1[i] != src2[i]);
        if (dst[i]) {
            break;
        }
    }
}
```

Listing 5.4: Code source de la boucle de type *memcmp* avec double sortie

Afin de garder la trace des différentes comparaisons lors des itérations successives de la boucle, le résultat de la comparaison de deux éléments i des tampons sources est sauvegardé dans l'élément i d'un troisième tampon. Cette boucle est structurellement plus complexe que la première car elle possède deux sorties. De plus, la sortie prématurée de la boucle dépend de la comparaison de deux données lues en mémoire alors que l'unique sortie de la première boucle ne dépend que d'un compteur incrémenté de façon monotone. Le code assembleur correspondant à cette deuxième fonction après instrumentation est présenté dans le listing 5.5.

```
Boucle2:
    push {r4, r5, r6, r7, r8, r9, r10, r11, lr}
    [...]                       ; initialisation des registres en V
    mov lr, r0
    mov r0, #0                                          ; n = 0
    cmp r3, #1
    mov r6, #7                                          ; j = 7
    blt .exit
.loopbody:
    ldr r4, [r1, r0, lsl #2]                            ; src1[i]
    ldr r5, [lr, r0, lsl #2]                            ; src2[i]
    cmp r5, r4
    mov r4, #0
    movwne r4, #1
    str r4, [r2, r0, lsl #2]                            ; dst[i]
    bne .exit                                           ; break
.elseblock:
    add r0, r0, #1                                      ; i++
    add r6, r6, #3                                      ; j += 3
    cmp r0, r3                                          ; i < n?
    blt .loopbody
.exit:
    [...]                       ; sauvegarde des registres
    pop {r4, r5, r6, r7, r8, r9, r10, r11, pc}
```

Listing 5.5: Code assembleur de la boucle de type *memcmp* du listing 5.4 après instrumentation

5.3.2 Campagne d'attaque sur les boucles standards

La première campagne d'attaques a été réalisée sur les deux exemples de boucle précédents, avec un intervalle d'injection des impulsions EM couvrant au moins une itération. Le pas de 5 ns et le nombre d'injections réalisées assurent d'avoir visé chaque instruction exécutée de l'itération ciblée.

Chaque exécution soumise à une impulsion est rangée dans l'une des trois catégories suivantes, définies dans la section 5.2.3.3, en fonction des données obtenues à la fin de l'exécution : sans effet, pas de réponse ou faute injectée. Un des objectifs étant de vérifier s'il est possible ou non de modifier le nombre d'itérations des boucles considérées, les exécutions avec fautes injectées sont divisées en deux sous-catégories :

- une *faute inoffensive* : les données obtenues en sortie diffèrent des données de référence mais indiquent un nombre d'itérations inchangé ;
- une *faute exploitable* : les données obtenues en sortie diffèrent des données de référence et leurs valeurs témoignent, à l'aide du compteur de traçage ajouté, d'une altération du nombre d'itérations.

Le tableau 5.1 montre le nombre d'exécutions entrant dans chaque catégorie pour les deux codes *Boucle1* et *Boucle2*. La proportion de fautes exploitables est comparable aux proportions mentionnées dans des campagnes d'injections réalisées sur des composants plus simples [Timmers et al., 2016].

Table 5.1: Classification des résultats d'injection de fautes électromagnétiques

code	sans faute	muet	faute injectée	
			faute exploitable	faute inoffensive
Boucle1	12 663 *(93.0%)*	403 *(2.9%)*	555 *(4.1%)*	
			87 *(15.7%)*	468 *(84.3%)*
Boucle2	14 287 *(95.0%)*	341 *(2.4%)*	372 *(2.6%)*	
			299 *(80.4%)*	73 *(19.6%)*

On peut constater qu'environ 15% des fautes injectées sur la première boucle et jusqu'à 80% des fautes injectées sur la deuxième boucle altèrent la propriété de sécurité. Dans cette dernière, la plupart des instructions sont impliquées dans les conditions de sorties de la boucle alors que, dans la première boucle, seules les instructions incrémentant le compteur le sont, comparant sa valeur à celle finale et le branchement basé sur cette comparaison sortant de la boucle. La surface d'attaque est donc plus grande sur la deuxième boucle, ce qui peut expliquer les proportions différentes observés entre ces deux codes. Ces deux exemples montrent qu'il est possible d'injecter des fautes visant des boucles simples sur une micro-architecture complexe avec cet environnement d'attaque.

Dans les chapitres précédents, nous avons considéré pour la conception de contre-mesure deux modèles de fautes : le saut d'instruction et la corruption de registre. Nous avons réalisé une première analyse du contenu de la mémoire tampon de sortie à la fin des exécutions avec faute injectée en considérant ces modèles. Nous avons cherché à identifier les contenus de la mémoire et des registres qui pouvaient être la conséquence d'une faute ainsi modélisée. Nous avons pu établir que 8% des cas sont explicables par un saut d'instruction et que 14% s'expliquent par une corruption de registre. En revanche, les 78% des résultats restants ne peuvent être directement expliqués par ces modèles. D'autres modèles sont nécessaires et sont proposés en section 5.5 puis utilisés pour analyser les résultats.

Ces résultats montrent qu'il est possible d'altérer le nombre d'itérations des boucles exécutées en l'absence de protection. Les codes sensibles étant souvent protégés, nous avons inclus des contre-mesures logicielles pour sécuriser ces boucles et la prochaine section décrit les résultats des campagnes sur les boucles sécurisées.

5.3.3 Campagne d'attaque sur les boucles sécurisées

Les deux exemples de boucles précédents ont été compilés en activant la passe de sécurisation des boucles dans le compilateur LLVM (cf. chapitre 3). La contre-mesure PLAF duplique les instructions impliquées dans les conditions de sortie de la boucle, donc, en particulier les compteurs de boucles. Comme précédemment, les tampons mémoire sont initialisés en amont de l'appel à la fonction, et le code assembleur a été manuellement modifié pour insérer l'initialisation des registres ainsi que l'instrumentation du code. Le listing 5.6 montre le code

sécurisé de la première boucle ainsi que les modifications et instrumentations apportées pour faciliter les analyses.

```
Boucle1:
       push {r4, r5, r6, r7, r8, r9, r10, r11, lr}
    [...]                                  ; initialisation des registres en V
       sub sp, sp, #8
       cmp r2, #0
       str r2, [sp, #4]                    ; Sauvegarde de n, invariant de boucle
       beq .exit
       add r3, r0, #1
       mov lr, r2
       mov r6, #69                         ; Initialisation du compteur de traçage (r6)
       add r0, r2, r2
       add r0, r0, #7                      ; Initialisation du compteur dupliqué r0=2n+7
.loopbody:
       sub r0, r0, #2                      ; Décrémentation de 2 du compteur dupliqué
       ldrb r4, [r1], #1
       add r6, r6, #3                      ; Incrémentation de 3 du compteur instrumenté
       ldrb r5, [r3]
       add r4, r5, r4
       strb r4, [r3], #1
       subs lr, lr, #1
       beq .bb_out
.bb_in:
       cmp r0, #7
       bne .loopbody
.errorHandler:
       ldr r8, =nbfail                     ; L'erreur est gérée via r8 et r9 inutilisés ailleurs
       movw r9, #44510
       str r9, [r8]
       b .exit
.bb_out:
       cmp r0, #7
       bne .errorHandler
.bb_iops
       ldr r7, [sp, #4]                    ; Intégrité des opérandes invariants vérifiée via r7
       teq r2, r7
       bne .errorHandler
.exit:
       add sp, sp, #8
    [...]                                  ; sauvegarde des registres
       pop {r4, r5, r6, r7, r8, r9, r10, r11, pc}
```

Listing 5.6: Code assembleur sécurisé de la boucle de type *memcpy* du listing 5.2 après instrumentation

Les modifications incluent :

- L'ajout d'un compteur de traçage dans un registre dédié (`r6`). Ce compteur est initialisé à une constante non déjà présente, et incrémenté de 3 à chaque itération. La valeur 3 diffère de l'incrément initial (1), devenu -1 après optimisation du compilateur. Ainsi, les instructions d'incrémentation/décrémentation du compteur original et de traçage n'ont rien en commun (`add r6, r6, #3` contre `subs lr, lr, #1`)
- De la même façon, le compteur dupliqué de la contre-mesure est initialisé et décrémenté de façon différente de l'original. Ainsi, dans le bloc bb_{in}, il est comparé à une constante non nulle.
- La vérification de l'intégrité des opérandes invariants utilise le registre `r7` non utilisé ailleurs dans le code.

◇ Le bloc de gestion d'erreur met une valeur spécifique dans une variable globale afin de détecter lors de l'analyse post-exécution une modification de cette variable. À cette fin, il utilise les registres `r8` et `r9` non utilisés ailleurs.

L'ensemble de ces modifications éloigne le code exécuté du code compilé original mais fournit une aide essentielle à l'analyse des données de sortie obtenues. Dans la suite, les codes des deux boucles sécurisées sont nommées *Boucle1-plaf* et *Boucle2-plaf*.

L'exemple *Boucle2* possède un flot de contrôle plus complexe dû aux deux sorties de boucle présentes. En plus de la sécurisation des boucles avec l'algorithme PLAF, nous avons souhaité tester et comparer les effets d'une deuxième contre-mesure sur cet exemple. Parmi les schémas cherchant à se protéger contre des modèles de fautes similaires, nous avons choisi la contre-mesure *SoftWare Implemented Fault Tolerance* (SWIFT) présentée dans [Reis et al., 2005] conçue pour être résistante à des corruptions du flot de contrôle et de registres provenant de faute unique mono-bit (Single Event Upset [Bar-El et al., 2006]). Ce schéma repose sur une protection du flot de contrôle à base de signatures : l'ensemble des instructions est dupliqué et chaque bloc de base possède sa propre signature utilisant deux variables dédiées. La première variable, appelée *Global Signature Register* (GSR), contient la signature du bloc en cours d'exécution et est mise à jour au bloc suivant grâce à un `xor` avec une valeur déterminée statiquement. La seconde variable, appelée *Run-Time Signature* (RTS), est utilisée pour compenser les valeurs statiques quand les blocs ont plusieurs prédécesseurs. Les signatures sont alors contrôlées à l'entrée des blocs de base. Cette contre-mesure a été initialement conçue pour l'architecture IA64 possédant un jeu d'instructions prédiquées et un degré élevé (nombre d'instructions exécutées par cycle). Nous avons manuellement adapté ce schéma pour le jeu d'instructions ARMv7. La boucle sécurisée est nommée *Boucle2-swift* dans la suite.

En plus des deux classes de fautes inoffensives et fautes exploitables présentées précédemment, une nouvelle classe appelée *fautes détectées* est introduite à cause de la présence de mécanismes de détection de fautes ajoutés par les contre-mesures. La figure 5.6 présente la répartition des fautes injectées pour les campagnes d'attaques réalisées sur les trois boucles sécurisées considérées, selon leur exploitabilité par un attaquant et leur détection par les contre-mesures. Les résultats montrent que les contre-mesures sont bien capables de détecter une partie des fautes : 33% des fautes injectées dans le cas de la *Boucle1-plaf*, 70% dans le cas de la *Boucle2-plaf* et 20% dans le cas de la *Boucle2-swift*.

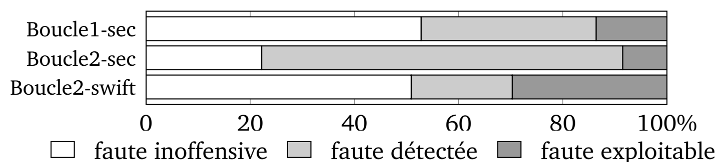


Figure 5.6: Répartition des fautes injectées sur boucles sécurisées

Ces résultats sont cohérents avec notre analyse d'explication des fautes par l'injection de saut d'instruction et corruption de registre. En revanche, ces résultats montrent également que de nombreuses fautes ne sont pas détectées, ce qui pousse à penser que les impulsions électromagnétiques peuvent générer des effets plus complexes au niveau logiciel qui ne sont pas couverts par ces contre-mesures. Afin de comprendre plus en détail les comportements observés, nous cherchons à caractériser les effets des fautes au niveau ISA dans la section suivante.

5.4 Caractérisation de fautes

Cette section est consacrée à la caractérisation des effets observés lors des injections de fautes par impulsions électromagnétiques sur notre plateforme SoC. La méthodologie utilisée est expliquée en 5.4.1, suivi des observations et analyses dans les sections suivantes.

5.4.1 Méthodologie

Les analyses des résultats présentées dans la section précédente ne sont que partielles car elles ne considèrent que les deux modèles de faute de saut d'instruction et de corruption de registre pour expliquer les données fautées. Même avec des exemples de boucles relativement simples, l'analyse des résultats fautés pour chaque injection est loin d'être triviale. Pour mieux comprendre les effets observés possibles, nous avons temporairement laissé de côté les exemples de boucle pour revenir à des codes plus simples n'utilisant qu'un ensemble limité d'éléments micro-architecturaux.

Il n'existe pas de méthode générique pour caractériser les effets des fautes injectées. Une pratique couramment utilisée est de commencer par cibler une séquence de `nops`. Cette méthode a été appliquée à la caractérisation des fautes par injection laser sur un microcontrôleur AVR 8-bit ATtiny841 [Kelly et al., 2017], par perturbation de la tension d'alimentation [Spruyt, 2012], ou par impulsions électromagnétique sur un microcontrôleur ATmega128 à base ARM Cortex-M3 [Moro, 2014].

Nous avons également commencé par cibler une séquence de `nops`, puis nous avons augmenté progressivement la complexité des codes ciblés en stimulant de plus en plus d'éléments micro-architecturaux (aucun registre utilisé, puis un, puis plusieurs par exemple). La nature des effets observés est dépendante du code exécuté, il faut donc être vigilant sur les modifications apportées au code. L'objectif est double : valider sur un code plus complexe les observations et analyses effectuées sur les codes précédents, et comprendre de nouveaux comportements mis en évidence par les ajouts apportés par les nouveaux codes considérés. Ainsi, notre méthode est d'inventer à chaque étape un nouveau code afin de comprendre les effets inexpliqués par les analyses des résultats des injections sur les codes précédents. Pour les différents

codes testés dans cette section, nous avons sauvegardé à la fin de l'exécution le contenu des registres pour l'analyse, de manière similaire à ce qui a été présenté dans la section 5.2.3. Les prochaines sections présentent les codes testés, les résultats obtenus sur ces codes ainsi que les analyses possibles à chaque étape.

5.4.2 Code initial : séquence de `nops`

Le premier code testé est une séquence de 1000 `nops`. C'est le code le plus simple à analyser car la moindre modification est plus facilement visible : à l'issue d'une telle séquence, aucune valeur ne doit avoir changé.

La campagne d'attaques sur ce code a consisté en une série de 3000 injections, parmi lesquelles la carte a été muette 142 fois, mais aucune faute n'a été observée. Au niveau ISA, aucun registre, ni aucune instruction ne semble avoir été corrompu. Ce résultat est particulièrement inattendu étant donné les premiers résultats de corruptions de registres sur les boucles. De cette première observation, il est déjà possible de déduire deux points essentiels :

- Les fautes ne provoquent visiblement pas de corruption directe des registres. Si leurs valeurs sont altérées, cela semble être de manière indirecte, via l'exécution d'instructions par le processeur par exemple.
- Un remplacement d'instruction par n'importe quelle autre est fortement improbable. En effet, si l'un des `nops` est remplacé par une autre instruction du jeu d'instructions, il y a de fortes chances que la valeur de l'un des registres soit modifiée (peu d'instructions n'ont pas de registre destination). En particulier, cela exclut un remplacement d'instruction avant l'étage de décodage du pipeline.

La documentation fournie par ARM précise que l'implémentation du `nop` dépend des architectures [3]. Cette instruction peut être remplacée par une instruction alternative qui ne fait rien comme `mov r0, r0`. Il est aussi possible que le `nop`, une fois décodé, soit supprimé du pipeline avant son exécution. En l'absence d'information sur les détails d'implémentation du `nop` sur cette plateforme, il est difficile de savoir quels éléments du processeur sont ciblés par l'injection. Par construction, une séquence de `nops` ne permet pas de mettre en évidence des fautes de type saut d'instruction, ce qui nous amène aux campagnes suivantes.

5.4.3 Séries d'additions

À la suite de cette première étape de caractérisation, nous avons réalisé des campagnes d'attaques sur des codes contenant des opérations arithmétiques.

3. http://infocenter.arm.com/help/topic/com.arm.doc.dui0489h/Cjafcggi.html

5.4.3.1 Addition mono-registre

La première campagne a ciblé un code exécutant 1000 fois la même instruction `add r0, r0, #1`. Après la campagne sur les `nops`, cela représente un des codes les plus simples possibles, n'utilisant qu'un seul registre et qu'un seul type d'instruction. En incrémentant un unique registre, nous essayons de corréler les effets de corruptions de registres avec les opérandes des instructions. Pour cette campagne, l'ensemble des registres `r0` à `r12` a été initialisé avant l'exécution du code avec les valeurs `ri = 0x10001 << i`.

Cette campagne a été limitée à une série de 1000 injections. La carte est devenue muette 56 fois, proportion équivalente à la séquence de `nops`, et 10 fautes injectées ont été recensées. La première observation est que sur ces 10 fautes, seul le registre `r0` a été corrompu. Lors de l'expérience précédente sur les `nops`, aucun registre n'était utilisé et aucun n'a été corrompu. Ces résultats sont en accord et laissent penser que seul un registre utilisé dans une instruction peut être corrompu. De plus, cela indique que le `nop` n'est pas implémenté par la micro-architecture comme une instruction utilisant `r0` comme `add r0, r0, #0`.

Nous avons également analysé les valeurs corrompues de `r0`. Sa valeur initiale étant `0x10001`, la valeur de `r0` à la fin de l'exécution sans faute injectée est `0x103E9` (`0x10001` + 1000). Les valeurs corrompues obtenues sont les suivantes :

1. la valeur `0x103E8` a été obtenue une fois, ce qui s'explique par un simple saut d'instruction parmi les 1000 additions de la séquence.
2. Cinq valeurs correspondent au quadruple de la valeur attendue à un petit écart près : `0x406ED`, `0x4074D`, 2 fois `0x407C5` et `0x4083D`. Ces valeurs peuvent s'expliquer de manière simple par la sélection, comme opérande, du registre `r0` à la place de la constante `#1`. Deux de ces instructions `add r0, r0, #1` consécutives seraient alors remplacées par `add r0, r0, r0`. De plus, l'écart observé peut s'expliquer par la place des instructions fautées dans la séquence : si les `k` et `k+1`-ièmes instructions de la séquence sont fautées de cette manière, la valeur de `r0` avant la faute est `0x10001 + k`, et sa valeur à la fin de l'exécution est donc 4(`0x10001+k`)+(998-`k`). Ainsi, les valeurs finales de `r0` observées correspondent à un remplacement des 257 et 258$^{\text{èmes}}$ instructions, 289 et 290$^{\text{èmes}}$, 329 et 330$^{\text{èmes}}$ et enfin 369 et 370$^{\text{èmes}}$. Grâce à la trace EM, il est possible d'estimer l'instant d'injection et constater que cet instant coïncide avec la place de l'instruction fautée dans la séquence : l'instant d'injection est visible à environ un quart de la trace totale de l'exécution pour le cas 257-258/1000, et à un tiers pour le cas 329-330/1000.
3. Deux valeurs similaires au cas précédent ne peuvent s'expliquer de la même façon : `0x202D8` et `0x202E9`. En effet, l'écart entre ces valeurs et le double de la valeur initiale ne correspondent pas à un tel remplacement. En revanche, la valeur initiale de `r1` étant `0x20002`, ces valeurs peuvent être expliquées par la sélection comme opérande du registre `r1` à la place du registre `r0`. Une des instructions `add r0, r0, #1` serait alors

remplacée par `add r0, r1, #1` : la 257ème instruction pour obtenir la valeur `0x202E9` et la 274ème pour la valeur `0x202D8`.

4. La valeur `0x503EB` a été obtenue une fois. Ce cas peut s'expliquer par le remplacement de la constante `#1` par le registre `r1`. Deux instructions `add r0, r0, #1` serait alors remplacées par `add r0, r0, r1`.
5. La valeur `0x10380` a été obtenue une fois. Cette valeur ne correspond pas à un remplacement simple comme précédemment, mais peut éventuellement être expliquée par de multiples renversements de bit dans le registre `r0`.

Pour les cinq valeurs du deuxième cas, on pourrait aussi imaginer que l'instruction `add r0, r0, #1` ait été remplacée par l'instruction `sll r0, r0, #1`. Un tel remplacement semble néanmoins moins probable étant donné les valeurs observées : des remplacements de l'opcode de l'instruction auraient dû générer une distribution plus large de résultats.

Pour éliminer une éventuelle hyper-sensibilité du registre `r0`, nous avons réalisé exactement la même campagne d'attaques en remplacant `r0` par `r5` dans le code : nous avons considéré une séquence de 1000 instructions `add r5, r5, #1`. Cette expérience a amené à des résultats similaires : sur 1000 injections, la carte est devenue muette 33 fois et 16 fautes ont été recensées. Surtout, seul le registre `r5` a été corrompu, ce qui montre encore que les registres inutilisés ne sont pas impactés. En revanche, les valeurs obtenues diffèrent fortement de l'expérience précédente. La valeur initiale de r5 étant `0x200020`, sa valeur finale en l'absence de faute est `0x200408`. La valeur `0x200407` a été observée une fois et s'explique par un simple saut d'instruction parmi les 1000 additions de la séquence. La valeur `0x220408` a été observée 14 fois et peut s'expliquer par un double remplacement de `add r5, r5, #1` par `add r5, r5, r0` (la place des instructions fautées dans la séquence n'influe pas sur la valeur finale de `r5`). Cette valeur peut également être expliquée par le renversement du bit 21 du registre `r5`. Enfin, la valeur `0x408` a été observée une fois et peut s'expliquer par le renversement du bit 25 du registre `r5`.

Les résultats de ces deux campagnes nous ont poussé à évaluer l'impact du choix du registre sur les effets des fautes qui semblent varier entre les deux campagnes. De plus, la plupart des valeurs obtenues peuvent être expliquées de différentes manières et ces codes simples ne permettent pas de les différencier, même si certains effets semblent moins probables que d'autres (remplacement de `add` par `sll`). À ce stade, on peut juste constater que 96% des valeurs obtenues s'expliquent par un remplacement d'opérande dans une instruction, mais la nature précise de ce remplacement reste à identifier. De multiples effets micro-architecturaux peuvent en être la cause : la corruption peut avoir lieu dans les registres eux-mêmes, dans les multiplexeurs et sélecteurs qui gèrent les banques de registres ou dans la logique. Certaines valeurs obtenues semblent indiquer des corruptions multiples, potentiellement corrélées. Il est toutefois impossible avec seulement ces deux expériences de connaître le nombre de registres ou d'instructions impactées simultanément. Les campagnes suivantes ont pour but d'aider à éclaircir ce point.

5.4.3.2 Additions multi-registre indépendantes

La campagne suivante considère un code, présenté dans le listing 5.7, comportant 100 séquences de 10 additions de la forme `add rx, rx, Cx` utilisant chacune un registre `rx` et une constante `Cx` différents. Cette répétition séquentielle (ni saut ni boucle) de blocs de 10 additions possède plusieurs avantages.

```
add r0, r0, #2
add r1, r1, #3
add r2, r2, #5
add r3, r3, #7
add r4, r4, #11
add r5, r5, #13
add r6, r6, #17
add r7, r7, #19
add r8, r8, #23
add r9, r9, #29
add r0, r0, #2                ; répétition du bloc de 10 instructions
[...]
```

Listing 5.7: Code assembleur de la campagne d'attaques sur additions multi-registre

- L'addition est le seul type d'instruction utilisé, le code reste peu complexe.
- Plusieurs registres sont utilisés mais de façon indépendante. Une corruption unique de l'un des registres n'en impacte pas un autre et cela facilite grandement l'analyse des résultats.
- 10 instructions séparent la répétition d'une même instruction incrémentant le même registre. Cela limite les effets temporels associés à l'exécution dans le désordre ou au pipeline du processeur (8 étages).
- Chaque registre est incrémenté d'une valeur qui lui est propre. Pour réduire les chances de collisions, nous avons choisi des nombres premiers comme incréments, comme indiqué dans le listing 5.7. Cela permet de limiter les collisions dans les résultats et ainsi le nombre d'explications possibles du même résultat.

Toujours pour limiter les risques de collisions, nous avons aussi choisi d'initialiser les registres avant l'exécution du code avec le schéma *en V* présenté dans la section 5.2 (cf. listing 5.1).

Le but de cette campagne est d'estimer le nombre de registres et/ou d'instructions qui peuvent être impactés en une seule injection ainsi que les potentielles interactions entre instructions exécutées simultanément. Comme observé dans les campagnes précédentes, seuls les registres utilisés (`r0` à `r9`) ont été impactés.

Le nombre de fautes affectant chacun des registres est présenté dans la figure 5.7. On peut observer qu'aucun registre ne se démarque particulièrement des autres : les 10 registres utilisés ont eu la même sensibilité aux corruptions.

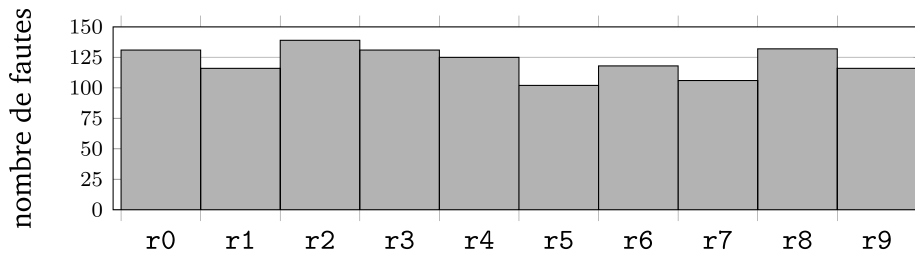


Figure 5.7: Nombre de fautes affectant chaque registre (r0 à r9)

En revanche, dans la majorité des cas, plusieurs registres ont été corrompus en une seule injection. La figure 5.8 représente la distribution du nombre de registres corrompus par faute. Le nombre moyen de registres corrompus par faute est de 3,8 et la figure montre qu'en cas de multiple corruption de registre, un nombre pair de registre est très majoritairement corrompu.

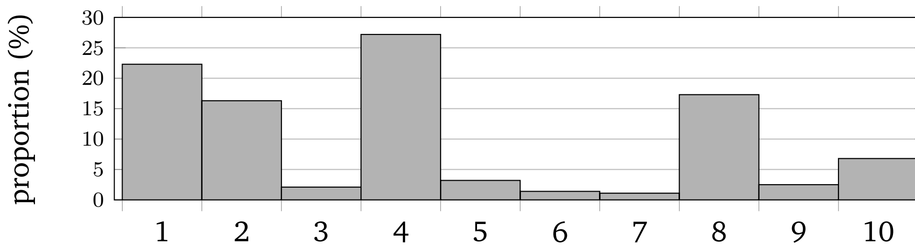


Figure 5.8: Distribution du nombre de registres corrompus simultanément

Bien que le code soit assez simple, l'analyse des données de sortie reçues devient complexe. Grâce à la configuration initiale des registres et du choix des instructions, certains comportements au niveau ISA sont toutefois explicables. Certaines valeurs obtenues ont été fréquemment observés, appelé motif de corruption. Le tableau 5.2 illustre 5 de ces motifs. La ligne référence donne la valeur attendue et chaque ligne exemple i illustre un motif de corruption.

Dans l'exemple 1, le registre `r7` possède à la fin une valeur (`0x010007E0`) qui diffère d'exactement 12 de la valeur attendue (`0x010007EC`). Or, 12 correspond également à la différence entre la constante `C7` = 19 utilisée pour incrémenter `r7` et la constante `C3` = 7 utilisée pour incrémenter `r3`. Au niveau ISA, cela peut s'expliquer par une corruption d'opérande (ici l'immédiat), en remplaçant une instruction `add r7, r7, #19` par `add r7, r7, #7`.

Le deuxième exemple montre un cas où le registre `r4` possède à la fin de l'exécution une valeur (`0x040004CE`) proche de celle attendue pour `r5`. Cela peut s'expliquer par le remplacement d'une instruction `add r4, r4, #11` par `add r4, r5, #13` (corruption de 2 opérandes). Plus précisément, la valeur s'explique exactement par la corruption de la 52$^{\text{ème}}$ instruction `add r4, r4, #11`. En effet, `0x04000020` $+ 51 \times 11 + 49 \times 13 =$ `0x040004CE`.

Table 5.2: Exemples de valeurs de registre corrompus obtenues après injection de faute

registre valeur de référence	r0 0x800000C9	r1 0x4000012E	r2 0x200001F8	r3 0x100002C4	r4 0x0800045C
exemple 1	–	–	–	–	–
exemple 2	–	–	–	–	0x040004CE
exemple 3	–	–	0x200001F3	0x100002CB	–
exemple 4	0x000000B7	–	–	–	–
exemple 5	0x800000C7	–	–	0x100002CB	–

registre valeur de référence	r5 0x04000534	r6 0x020006E4	r7 0x010007EC	r8 0x008009FC	r9 0x00400D54
exemple 1	–	–	0x010007E0	–	–
exemple 2	–	–	–	–	–
exemple 3	–	–	–	–	–
exemple 4	–	0x000006E4	–	–	–
exemple 5	–	–	0x010007D9	0x00400CDE	0x00400D71

Dans l'exemple 3, la valeur obtenue pour le registre r2 est celle attendue minorée de 5 et celle du registre r3 est celle attendue majorée de 7. Or, la constante `C2` pour incrémenter le registre `r2` est 5 et la constante `C3` pour incrémenter `r3` est 7. Une explication possible est que l'une des instructions `add r2, r2, #5` est sautée et que l'une des instructions `add r3, r3, #7` est rejouée. La présence de nombreux résultats correspondants à ce motif de corruption explique la majorité de nombres pairs de registres fautés dans la figure 5.8.

L'exemple 4 montre un cas où le registre `r6` a ses 16 bits de poids fort à zéro, mais les valeurs des 16 bits de poids faible ne sont pas altérées. Plusieurs cas de bits de poids fort mis à zéro sont observés pour lesquels les valeurs des bits de poids faible peuvent être expliquées de deux façons différentes : soit par une mise à zéro complète du registre, soit parce qu'ils correspondent à leur valeur attendue avec les 16 bits de poids fort mis à zéro.

Enfin, l'exemple 5 montre l'un des nombreux cas combinant plusieurs corruptions similaires à celles des exemples précédents : le registre `r8` a une valeur proche de celle attendue pour `r9`, la valeur du registre `r0` est minorée de 2 (=`C0`) et celle de `r3` majorée de 11 (=`C3`), la valeur du registre `r7` est minorée de 19 (=`C7`) et celle de r9 majorée de 29 (=`C9`). Les valeurs obtenues dans cet exemple peuvent s'expliquer par 2 sauts et 2 rejeux d'instruction et la corruption des opérandes source d'une cinquième instruction. Un autre cas rencontré plusieurs fois de corruption particulièrement complexe est présenté en annexe de ce chapitre (cf. section 5.7).

Ces exemples montrent que divers motifs de corruption semblent se produire, dont des substitutions d'opérandes et des sauts et rejeux d'instructions. Certaines corruptions peuvent aussi s'expliquer par des renversements de bits à l'intérieur des registres, mais étant donné la répétabilité des effets présentés ici, il semble peu probable que les valeurs obtenues soient uniquement dues à des renversements pseudo-aléatoires de bits. Les opérandes corrompus

d'instructions ont le plus souvent des valeurs apparaissant dans les opérandes d'autres instructions en cours de traitement (exemples 1 et 2 du tableau 5.2). De plus, les corruptions de registre multiples ne correspondent pas nécessairement à des registres destination d'instructions consécutives dans le programme. Cela semble fortement lié à l'exécution superscalaire et dans le désordre du processeur qui traite en parallèle différentes instructions. Les résultats d'instructions corrompues s'expliquent très fréquemment par des remplacements de l'un de leurs opérandes avec des valeurs disponibles car étant opérandes d'autres instructions en cours d'exécution.

5.4.3.3 Isolation des effets

Dans la précédente campagne, nous avons montré que plusieurs instructions pouvaient être affectées par une unique injection de faute. Nous avons suspecté des effets de bords liés au pipeline du processeur : une seule perturbation peut affecter plusieurs instructions en cours de traitement [Yuce et al., 2016].

Pour confirmer ces effets, nous avons réalisé plusieurs campagnes d'attaques courtes. Le jeu d'instructions ARMv7 offre plusieurs instructions de type barrière susceptibles de limiter les effets du pipeline, dont notamment :

- `dsb` (*Data Synchronization Barrier*) qui agit comme une barrière mémoire : aucune instruction dans l'ordre du programme ne peut être exécutée avant que l'instruction `dsb` ne soit terminée [4].
- `isb` (*Instruction Synchronization Barrier*) qui vide le pipeline de telle sorte que les instructions qui suivent l'`isb` sont nécessairement rechargées depuis la mémoire ou le cache d'instructions [4].

Nous avons réalisé deux campagnes d'attaques visant des codes correspondant à ceux du listing 5.7 : le premier code visé contient une instruction `dsb` entre chaque instruction exécutée, le deuxième code contient une instruction `isb` entre chaque instruction. Nous avons observé une augmentation du nombre moyen de registres fautés pour ces deux codes, passant de 3.8 à 5.6 pour le `dsb` et 5.2 pour l'`isb`. Cela ne réduit donc pas les effets multiples. Au contraire, les instructions pouvant activer plus de parties du pipeline, l'effet d'isolation recherché n'a pas été obtenu. Sans information additionnelle sur les détails de l'action précise de ces instructions au cœur du processeur, il est difficile d'expliquer plus précisément ces résultats.

Après ces expériences infructueuses, nous avons cherché à séparer les instructions d'une autre façon. Nous avons réalisé trois campagnes d'attaques supplémentaires en considérant le code présenté dans le listing 5.7 dans lequel nous avons inséré 1, 2 puis 4 `nop`s entre chaque instruction exécutée. Dans les trois cas, nous avons pu constaté une réduction du

4. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489c/CIHGHHIE.html

nombre moyen de registres corrompus simultanément. Les valeurs sont regroupées dans le tableau 5.3.

Table 5.3: Nombre moyen de registres corrompus par injection en fonction des instructions insérées entre chaque addition du listing 5.7

instruction insérée	∅ (réf.)	1 `dsb`	1 `isb`	1 `nop`	2 `nops`	4 `nops`
# registres corrompus	3.8	5.6	5.2	3.2	1.8	1.7

Ces campagnes confirment ainsi qu'une simple faute peut affecter plusieurs instructions en cours de traitement à la fois et montre que les espacer par des `nops` réduit le nombre d'additions originales fautées simultanément, confirmant ainsi l'impact du pipeline. Lorsque les contre-mesures considèrent des fautes n'altérant qu'une seule d'instruction, une recommandation peut alors être d'insérer des `nops` entre les instructions qui ne doivent pas être altérées simultanément. Il est également possible d'agir au niveau de l'ordonnancement des instructions pour les séparer [Barry et al., 2016].

5.4.4 Résumé des effets et classification

Afin de synthétiser les informations apportées par les différentes campagnes d'attaques présentées jusqu'ici, nous résumons les effets observés et mettons en évidence leurs caractéristiques :

- Un registre non utilisé par le processeur a une très faible probabilité d'être corrompu ;
- Tous les registres semblent avoir la même sensibilité aux fautes malgré une position fixe de la sonde d'injection électromagnétique ;
- La valeur d'un registre vivant peut être corrompue (bits modifiés voire mise à zéro des bits de poids fort) ;
- L'ensemble des registres vivants peuvent être corrompus en même temps et l'insertion de `nop` entre instructions réduit le nombre de registres et/ou instructions impactés simultanément.

De plus, les instructions peuvent être corrompues de différentes manières. Elles peuvent être sautées (pas d'écriture, pas d'opération), être rejouées, ou avoir un ou plusieurs des leurs opérandes substitués principalement par ceux d'autres instructions en cours de traitement.

Ces observations générales et les comportements rencontrés et analysés sont spécifiques à l'environnement d'injection de fautes utilisé. Avant de confronter et d'appliquer ces remarques à des codes plus complexes, nous avons voulu classifier ces résultats d'injection en fonction du type d'effet observé au niveau ISA :

- *Saut d'instruction* : une instruction n'a pas été exécutée.
- *Mise à zéro de la partie haute* : les 16 bits de poids fort du registre sont nuls.

- *Corruption de registre* : le registre a été corrompu et la valeur obtenue est soit dérivée d'une valeur existante (à quelques bits près), soit sans relation particulière avec d'autres valeurs présentes.
- *Substitution d'opérande source* : un opérande constant ou un registre source d'une instruction est remplacé. La plupart du temps, la valeur corrompue provient de l'un des opérandes d'une autre instruction.

Dans la plupart des cas, les résultats obtenus ne peuvent être directement expliqués par une corruption unique entrant dans l'un de ces 4 cas. En revanche, ils peuvent être expliqués par une combinaison de ces effets, généralement corrélés. Ces *effets composés* sont aussi classés en 4 cas :

- Un saut d'une instruction combiné avec le rejeu d'une autre.
- Multiple corruption de registres dont les valeurs corrompues sont corrélées.
- Multiple faute suivant les effets précédemment cités : multiple sauts d'instructions ou corruptions de registres aux valeurs indépendantes.
- *Fautes mixtes* : combinaison d'effets distincts sans corrélation apparente.

En plus de ces classes, on constate que les instructions affectées ne sont pas consécutives dans le programme à cause de l'exécution dans le désordre. L'effet précis d'une faute est difficile à prédire, mais des sorties corrompues identiques sont apparues à plusieurs reprises, même avec des moments d'injection différents. Certaines fautes ont donc une probabilité non négligeable de se reproduire, i.e. aboutissant aux mêmes valeurs corrompues. Mieux armés avec les analyses de cette section et la classification des effets possibles, nous pouvons maintenant revenir sur les résultats restés inexpliqués sur les exemples de boucles intrinsèquement plus complexes. La prochaine section est dédiée à ces explications.

5.5 Retour sur les boucles

La grande différence entre les exemples de boucles testés et les codes qui ont servi à caractériser les fautes possibles est la variété des instructions utilisées. En plus d'opérations arithmétiques, les boucles contiennent des accès mémoire (lecture et écriture) ainsi que des branchements. De plus, les registres n'évoluent plus de façon indépendante les uns des autres, ce qui complique fortement l'analyse des fautes à partir des valeurs récupérées en fin d'exécution. Par exemple, la corruption d'un seul registre peut provoquer une modification du flot de contrôle et affecter la valeur finale de plusieurs registres par propagation des dépendances.

5.5.1 Analyse des fautes sur boucles

Pour tenter d'analyser les effets des fautes sur les boucles présentés dans la section 5.3, nous sommes partis de la classification des fautes issues des campagnes sur des codes plus simples.

Sur les codes des boucles, les fautes se propagent. Les instructions produisant des valeurs pour les suivantes et influençant le flot de contrôle, il est commun de pouvoir expliquer une sortie fautée par plusieurs effets différents au niveau ISA [Moro et al., 2013]. Pour expliquer les résultats des campagnes sur les boucles, nous avons cherché si un modèle de faute, selon la classification, pouvait expliquer le résultat en choisissant l'ordre suivant (de complexité croissante) : nous avons considéré en premier lieu un saut d'instruction unique, puis la substitution d'opérande, la corruption de registre, les effets composés et finalement les fautes mixtes. Les résultats auraient pu être rangés différemment, avec un autre ordre, toutefois notre but est d'expliquer le plus de cas possibles pour avoir un modèle au niveau ISA des effets observés. Nous avons pu comprendre et classifier une majorité des valeurs corrompues reçues. La présence d'instructions de nature différente a mis en évidence d'autres effets possibles que nous présentons ci-dessous.

Lors de l'analyse des sorties fautées, nous avons observés que des valeurs lues en mémoire étaient corrompues alors que la valeur stockée en mémoire était intacte. Les valeurs lues corrompues étaient soit proches de la valeur d'un registre vivant avec quelques bits d'écart, soit une valeur qui semblait aléatoire, sans relation visible avec le code exécuté. Cependant, les valeurs se sont répétées lors de différentes injections. De par la répétition de ces valeurs à motif spécifique, nous en avons déduit qu'elles venaient probablement d'autres emplacements mémoire non récupérés et que ces valeurs étaient sûrement dues à une substitution d'opérande dans l'instruction `load` correspondante. Le cas des valeurs proches de celles de registres vivants peut être représenté par une corruption du registre destination du `load`, et ainsi ne pas altérer la valeur en mémoire. Nos expériences ne permettent pas d'exclure une corruption du transfert mémoire lui-même. Ces cas ayant été plusieurs fois rencontrés, ils mettent en évidence une sensibilité des lectures mémoire. Nous nommons ces effets *corruption de lecture mémoire* dans la suite.

Le deuxième effet rencontré est nettement plus original et nécessite de considérer le CFG de la fonction. Certaines injections sur les versions sécurisées des boucles ont donné lieu à des valeurs ne s'expliquant que par le saut depuis la fin d'un bloc de base de la fonction directement à l'entrée d'un autre bloc de la fonction. Lorsque ce bloc d'arrivée n'est pas une destination légale du branchement initial, cet effet est nommé *arc magique*. Aucune des valeurs obtenues ne s'explique par l'arrivée au milieu d'un bloc de base, confortant cette image d'arc magiquement ajouté dans le CFG. Ce comportement semble indiquer une corruption du BTB ou d'un autre mécanisme intervenant dans la prédiction de branchement, plutôt que des effets directs sur le registre PC : il aurait été source d'une plus grande variété d'adresses destination (des sauts au milieu de blocs de base auraient dû être visibles). Comme exemple, le flot de contrôle de la fonction *Boucle2-plaf* est présenté dans la figure 5.9 avec les cas d'arcs magiques constatés (provenant de différentes fautes observées), représentés par des tirets rouges.

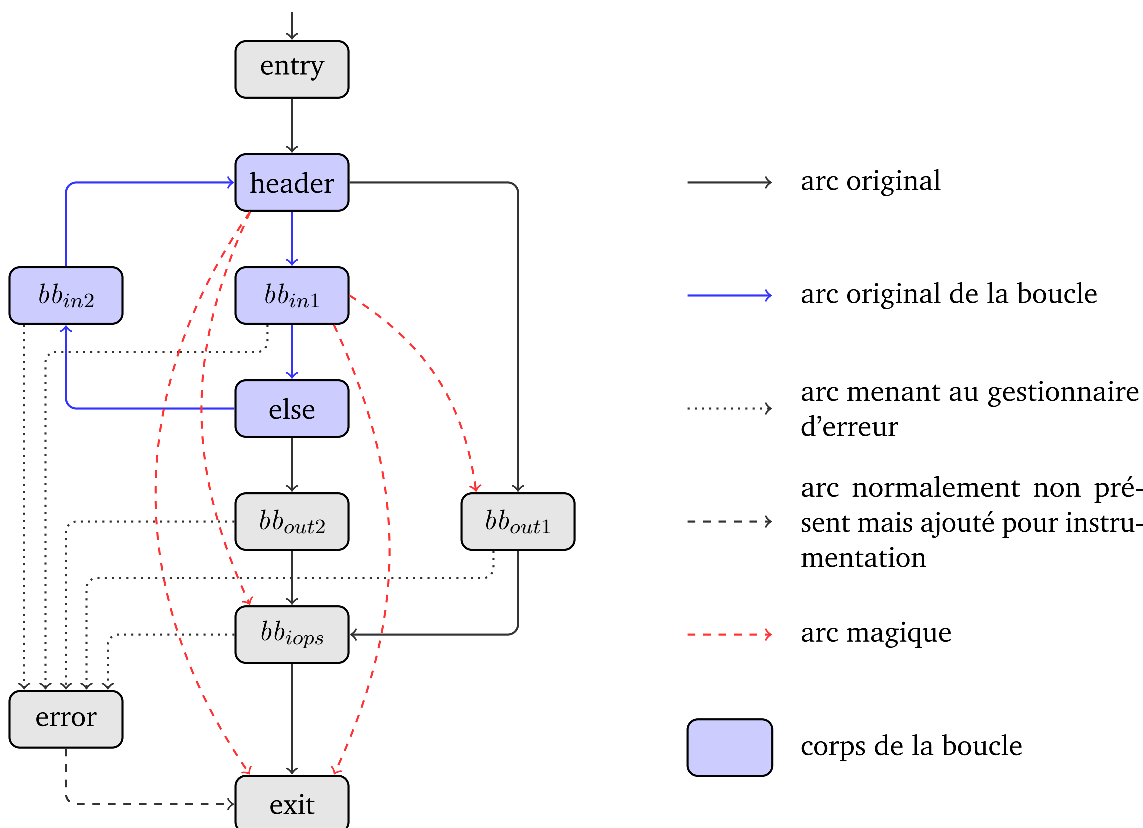


Figure 5.9: Flot de contrôle de la fonction *Boucle2-plaf* avec les différents arc magiques observés

Ajouter un arc magique peut introduire un saut permettant une sortie prématurée de la boucle qui contourne les blocs de vérification ajoutés par les contre-mesures. La protection basée sur SWIFT ne détecte pas ces effets alors que ce schéma protège contre certaines corruptions du flot de contrôle. Une raison possible est que l'ajout de vérifications au début de chaque bloc créé artificiellement des branchements dans ces blocs. Il est alors possible que ces arcs magiques provoquent le contournement des vérifications susceptibles de détecter la faute. L'accès au banc d'attaque étant restreint, nos expériences en nombre limité n'ont pas suffit pour expliquer le choix de la destination par rapport au code ayant été chargé dans le pipeline. Clairement, la présence de sauts dans les boucles stimule le mécanisme de prédiction de branchement et mériterait des analyses et expériences supplémentaires pour mieux mettre en avant les sources de la faute (adresse présente dans le pipeline, dans le BTB, . . .). L'ajout possible d'arc est un puissant moyen d'attaque même si contrôler avec précision la destination du saut semble difficile voire impossible à ce niveau.

5.5.2 Répétabilité des fautes

En plus de la difficulté à contrôler les effets obtenus par injection de fautes, la répétabilité des fautes possibles observées est très variable. On a remarqué que certains motifs de valeurs corrompues ont été observés qu'une seule fois alors que d'autres se répètent à de nombreuses

reprises tout au long d'une même campagne. En effet, certains motifs de valeurs corrompues apparaissent jusqu'à 30 fois, mais en moyenne, 40% des fautes sont uniques. Ceci met en avant une faible répétabilité des fautes au sein d'une même campagne d'attaques.

Aussi, lorsque la même campagne est rejouée deux fois, les résultats peuvent varier malgré des campagnes suffisamment longues. À titre de signature d'une campagne, nous avons regroupé les fautes par nombre de registre corrompus de la même manière que pour la campagne sur les additions multi-registres présentée en figure 5.8. Nous avons réalisé deux campagnes identiques sur le code de la *Boucle1-plaf* et les signatures associées des campagnes sont présentées en figure 5.10. On constate en effet que la même campagne rejouée dans les mêmes conditions fournit des résultats différents mettant en évidence la difficulté à reproduire des fautes entre campagnes distinctes. Ces différences peuvent être expliquées par des modifications de l'environnement pas toujours contrôlées. En particulier, la sonde d'injection est susceptible de subir de faibles mouvements. De même, la température peut affecter le comportement du composant ciblé [NXP, 2018].

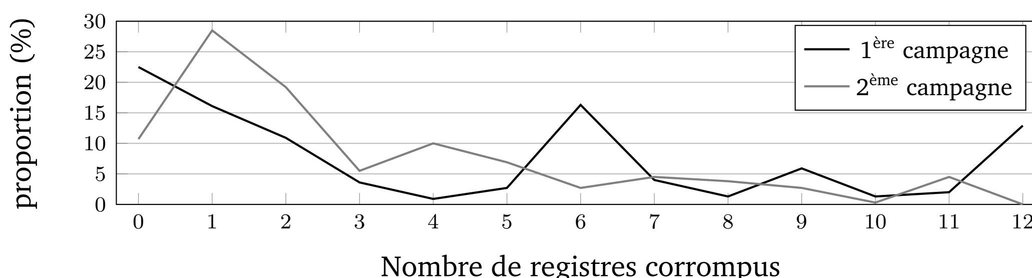


Figure 5.10: Distribution du nombre de registres corrompus simultanément pour la même campagne d'attaques sur le code de la *boucle1-plaf* jouée deux fois

5.5.3 Classification des fautes sur les boucles

Avec l'ensemble des modèles classifiés précédemment, cette section présente les résultats de la classification des fautes sur les boucles. Les exemples de boucles sécurisées utilisés en section 5.3 ont montré qu'ils laissaient tous une portion de fautes exploitables non détectées. Pour cette raison, nous avons ajouté à la *Boucle2* du listing 5.4 un troisième type de sécurisation en combinant de manière coûteuse les 2 contre-mesures. Nous avons d'abord appliqué notre schéma PLAF de protection des boucles et ensuite protégé le code résultant par le schéma de SWIFT. Cette version est appelée *Boucle2-plaf+swift*. Le tableau 5.4 résume pour l'exemple des versions de *Boucle2* le nombre total d'instructions dans la fonction, ainsi que le nombre d'instructions contenues à l'intérieur de la boucle pour chacune des versions.

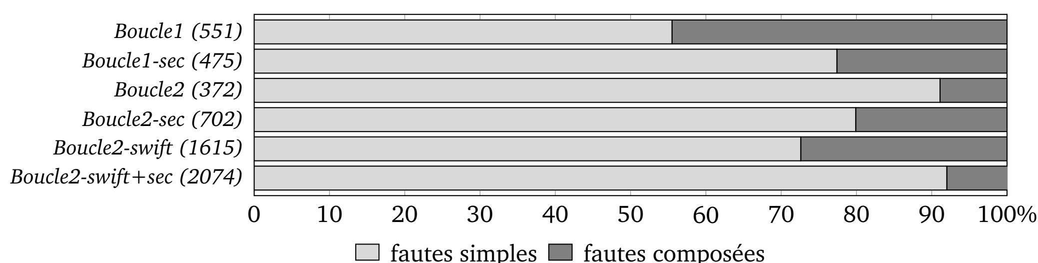
Le nombre indiqué tient compte de l'instrumentation avec un compteur supplémentaire de traçage. Comme l'algorithme PLAF ajoute plusieurs blocs de vérification, en particulier en cas de sortie multiple, le flot de contrôle de la boucle est nettement plus complexe que la boucle

Table 5.4: Nombre d'instructions dans la boucle et la fonction pour chaque version de *Boucle2* du listing 5.4

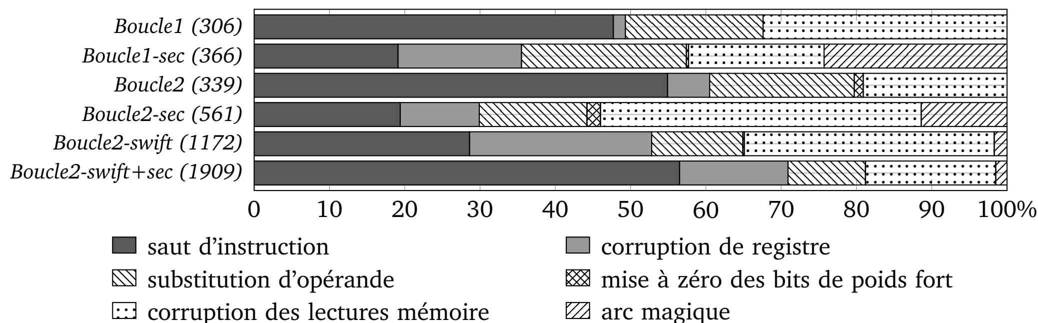
version	*Boucle2*	*Boucle2-plaf*	*Boucle2-swift*	*Boucle2-plaf+swift*
Boucle	10	22	36	47
Fonction	15	59	57	183

initiale. Lui appliquer ensuite le schéma de sécurisation de SWIFT devient donc très coûteux ($\times 12,2$).

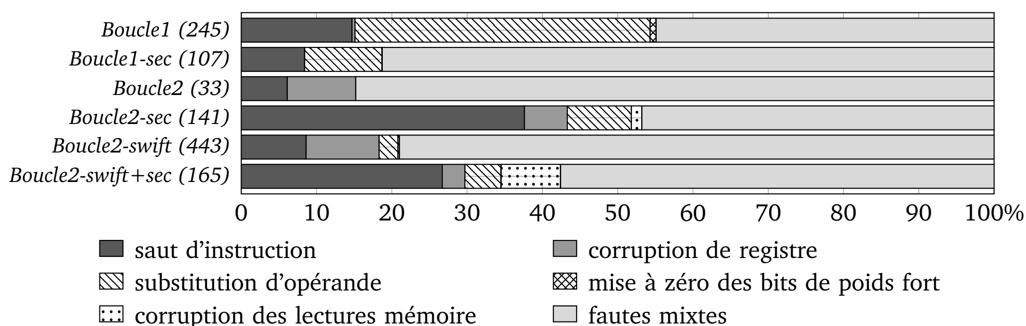
Pour l'ensemble des campagnes d'attaques, la proportion de fautes injectées et de carte muette est sensiblement la même : environ 4% de fautes et de 2 à 6% de mutisme de la carte. La figure 5.11 montre la distribution des fautes injectées en fonction des modèles de fautes des classifications précédentes.



(a) Répartition des fautes selon leur multiplicité



(b) Détails de la répartition des fautes simples



(c) Détails de la répartition des fautes composées

Figure 5.11: Répartition des fautes réussies selon les modèles de fautes et multiplicité des effets (avec le nombre total de fautes réussies) pour chaque campagne d'attaques sur boucle

Pour plus de visibilité, la figure sépare les cas de résultats obtenus en sortie s'expliquant par des fautes simples (figure 5.11b) de ceux s'expliquant par des fautes composées (figure 5.11c). Le banc d'attaque n'ayant pas toujours été à notre disposition le temps souhaité, les campagnes d'attaques ont été réalisées sur des durées variables. Les valeurs absolues peuvent donc difficilement être comparées entre deux campagnes. Des pourcentages sont reportés plutôt que des valeurs qui sont toutefois indiquées dans la légende. La proportion de fautes injectées s'expliquant par l'un des modèles de faute de la classification dépend du protocole d'attribution d'une classe comme expliqué dans la section 5.5.1. La proportion de saut d'instruction est potentiellement surestimée à cause de la démarche d'attribution des modèles commençant par celui-ci.

Les résultats montrent que dans chaque campagne, tous les modèles de fautes sont présents. La répartition des fautes varie fortement d'une campagne à l'autre, mais, la figure 5.11a montre que, dans tous les cas, la proportion de fautes composées est loin d'être négligeable. Un effet précis semble difficile à obtenir, mais réaliser des campagnes d'attaques suffisamment longues permet d'observer une large variété d'effets. Il semble donc probable qu'un attaquant soit capable, à force d'injecter des fautes, d'obtenir au moins une fois l'effet qu'il souhaite.

Si ces résultats fournissent des informations sur le large éventail d'effets possibles suite à l'injection d'impulsions EM, ils ne donnent pas directement d'information sur la ou les raisons pour lesquelles les fautes ne sont pas détectées par les contre-mesures. Pour tenter de dégager des raisons, la figure 5.12 montre, pour les 4 versions des deux boucles qui intègrent des protections, la répartition des fautes selon le modèle de la classification précédente expliquant leur effet. Dans cette figure, seules les fautes exploitables et non détectées sont représentées, c'est-à-dire ni les inoffensives ni les détectées.

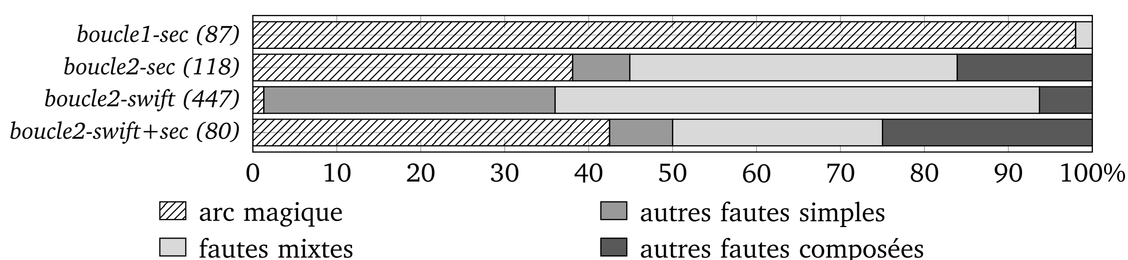


Figure 5.12: Répartition des fautes exploitables non détectées selon les modèles de la classification expliquant les résultats fautés

En comparant les figures 5.11b et 5.12, on constate que la plupart des fautes simples (hors arcs magiques) sont détectées ou inoffensives. Celles qui restent non détectées sont le plus souvent dues à une substitution du registre destination d'une instruction qui amène à une double corruption de registre permettant de contourner les mécanismes de détection d'erreur. En effet, dans un tel cas de corruption, le registre initialement destination n'est pas mis à jour comme prévu et le registre de destination réel est corrompu. De telles doubles corruptions,

issues d'effet simple, peuvent altérer de manière similaire le flot de données original et celui dupliqué rendant, dans ce cas précis, la contre-mesure à base de duplication inefficace.

La figure 5.12 montre surtout que les fautes exploitables non détectées sont majoritairement dues à des effets complexes comme les fautes composées et les arcs magiques, donc pas couvertes par les contre-mesures. En revanche, les fautes composées qui affectent de manière différente les calculs originaux des calculs dupliqués sont détectées : 48% des fautes composées détaillées dans la figure 5.11c ne sont plus présentes dans la figure 5.12. Les 52% restants contournent les mécanismes de détection et représentent une part importante des fautes non détectées. Elles peuvent affecter les calculs et le flot de contrôle, expliquant leur non détection. Quant aux arcs magiques, ils représentent une réelle menace pour toutes les versions sécurisées des boucles. Notre contre-mesure dédiée aux boucles ne peut détecter de telles corruptions du flot de contrôle. Parmi les campagnes sur les versions sécurisées de la *Boucle2*, un tiers des fautes mixtes sont une combinaison d'une faute de type arc magique et d'un deuxième effet.

On peut aussi voir que superposer les contre-mesures ne suffit pas à protéger contre ce type d'effets composés. Il reste en effet 80 fautes non détectées dans la campagne sur la *Boucle2-plaf+swift*. La superposition de contre-mesures ne semble pas suffire et peut être très cher, avec un coût qui peut augmenter exponentiellement. Comme l'indique les valeurs du tableau 5.4, le surcoût sur les exemples *Boucle2-plaf* et *Boucle2-swift* est d'un facteur 4, alors qu'il est d'un facteur 12 pour l'exemple *Boucle2-plaf+swift*. Superposer davantage de contre-mesures ne garantirait pas de détecter toutes les fautes et le coût serait véritablement prohibitif.

Tous ces résultats mettent en avant la variété et la complexité des effets observés après injection d'impulsions EM. Malgré les multiples campagnes d'attaques effectuées pendant les périodes où le banc d'attaque nous a été dédié, certains effets complexes ne sont pas entièrement compris. Les travaux réalisés préconisent donc une investigation plus profonde des effets des fautes sur des micro-architectures complexes afin d'en fournir une caractérisation plus complète.

5.6 Conclusion

Dans ce chapitre, nous avons utilisé un environnement d'injection de fautes par impulsions électromagnétique qui a permis de mettre en évidence la capacité à injecter, sur des codes à base de boucles, des fautes exploitables par un attaquant sur une architecture complexe à base de processeur superscalaire à exécution dans le désordre. Sans parcourir de façon exhaustive l'ensemble des paramètres du banc d'attaque, nous avons réussi à injecter des

fautes capables de corrompre la bonne exécution de boucles sensibles avec ou sans ajout de protection logicielle.

Afin de comprendre les effets observés, nous avons suivi une méthodologie de caractérisation pas à pas au niveau ISA. De petits codes en petits codes, nous avons mis en évidence des effets et points particuliers permettant d'éliminer ou de valider des hypothèses sur les éléments matériels fautés. En commençant par des codes très simples mais conçus de façon à différencier au maximum les effets visibles, nous avons pu analyser et comprendre les effets sur des exemples de boucles sécurisées. Grâce à cette méthodologie, nous avons identifié différents modèles de fautes, allant du saut d'instruction ou de la corruption de registre classiquement observés sur des processeurs plus simples, jusqu'à la mise en évidence de modèles spécifiques à des micro-architectures complexes. Nous avons aussi vu leur dangerosité face à des protections logicielles non adaptées. Parmi ces nouveaux modèles, nous avons observé et classifié des effets comme des sauts et rejeux d'instructions corrélés, des substitutions d'opérandes, des ajouts d'arcs magiques dans le CFG, ainsi que des compositions corrélées ou non de ces effets.

La contre-mesure PLAF, déployée sur les boucles sensibles, a été conçue pour être résistante à un saut ou une corruption de registre. Les simulations réalisées dans le chapitre 3 ont validé cette robustesse, à effets du *back-end* près et montrent une protection théorique contre un modèle de faute choisi. Cette contre-mesure étant indépendante de la cible, elle a pu être testée dans les codes visés par les injections d'impulsions EM. Les résultats montrent qu'en pratique, elle protège en partie, validant son utilité à rendre plus difficile la tâche en réduisant fortement la surface d'attaque d'une boucle sensible. Cependant, elle ne garantit pas un niveau de sécurité comparable à celui obtenu lors des simulations lorsque les effets des impulsions EM ne se ramènent pas à un simple saut d'instruction ou corruption de registre.

Définir un modèle de faute au niveau ISA ne suffit pas nécessairement pour la conception d'une contre-mesure logicielle. Il est important de disposer de modèles de fautes réalistes. Pour adapter des contre-mesures existantes à des processeurs complexes, il est nécessaire d'investiguer les effets des attaques sur ces cibles. À défaut, les contre-mesures doivent être déployées en connaissance de cause. Des élements micro-architecturaux, comme le cache d'instructions, peuvent compromettre la sécurisation [Rivière et al., 2015]. Ils modifient également la façon dont le code est exécuté (profondeur de pipeline, degré d'exécution superscalaire, mécanismes de prédiction de branchements) et donc, la façon dont il est fauté, ce qui est mis en évidence par la variété d'effets complexes mixtes ou combinés observés. Pour une sécurité accrue, cela pose la question de l'articulation entre les protections logicielles et matérielles.

5.7 Annexe des attaques

Cas répété de faute à effet complexe

Lors de la campagne d'attaques présentée dans le listing 5.7, un cas particulier de résultat lié à des effets complexes corrélés est apparu à de multiples reprises affectant divers registres. Quelques uns de ces exemples sont présentés dans le tableau 5.5 et sont tous basés sur le même schéma où 8 instructions successives ont été corrompues.

Table 5.5: Valeurs obtenues lors de cas spécifiques à effets complexes corrélés

registre	r0	r1	r2	r3	r4
référence	0x800000C9	0x4000012E	0x200001F8	0x100002C4	0x0800045C
exemple 1	0x800000DE	0x40000148	0x200001F5	-	-
exemple 2	-	**0xC0000186**	**0xA000024E**	0x100002C0	0x08000456
exemple 3	0x800000DE	-	-	**0x9000031E**	**0x880004B2**

registre	r5	r6	r7	r8	r9
référence	0x04000534	0x020006E4	0x010007EC	0x008009FC	0x00400D54
exemple 1	**0x8400059E**	**0x8200074A**	0x010007E6	0x008009F6	0x00400D4A
exemple 2	0x0400052E	0x020006DE	0x010007E6	0x008009F6	-
exemple 3	0x0400052E	0x020006DE	0x010007E6	0x008009F6	0x00400D4A

Pour comprendre ces cas à effets complexes, détaillons les valeurs du premier exemple. Les valeurs obtenues des 6 registres consécutifs `r7`, `r8`, `r9`, `r0`, `r1` et `r2` sont proches des valeurs attendues : les écarts sont respectivement de -6, -6, -10, 21, 26 et -3. Or les constantes utilisées pour incrémenter les registres `r0` à `r9` sont 2, 3, 5, 7, 11, 13, 17, 19, 23, 29. On peut alors constater que chacun des écarts correspond à une corruption d'une instruction en remplaçant l'incrément par celui utilisé par l'instruction précédente de 2 places dans le programme :

- ◇ remplacement de `add r7, r7, #19` par `add r7, r7, #13`;
- ◇ remplacement de `add r8, r8, #23` par `add r8, r8, #17`;
- ◇ remplacement de `add r9, r9, #29` par `add r9, r9, #19`;
- ◇ remplacement de `add r0, r0, #2` par `add r0, r0, #23`;
- ◇ remplacement de `add r1, r1, #3` par `add r1, r1, #29`;
- ◇ remplacement de `add r2, r2, #5` par `add r2, r2, #2`;

Les valeurs des registres `r5` et `r6` correspondent chacune à une corruption d'une instruction en remplaçant l'incrément par la valeur de `r0`. On peut même déterminer que, dans cet exemple, ce sont les 59$^{\text{ème}}$ instructions incrémentant `r5` et `r6` qui sont corrompues. En effet, à cette étape, la valeur de `r0` est 0x80000001 $+ 59 \times 2 =$ 0x80000077, et on peut vérifier que :

$$\texttt{0x04000020} + 99 \times 13 + \texttt{0x80000077} = \texttt{0x8400059E}$$
$$\texttt{0x02000040} + 99 \times 17 + \texttt{0x80000077} = \texttt{0x8200074A}$$

Ce type d'effet complexe corrélé a été observé 34 fois, à différents instants d'injection, avec différents registres impactés comme le montrent les 3 exemples présentés dans le tableau 5.5. Cela confirme d'autant plus la possibilité de corrompre les instructions de façon à ce que leurs opérandes soient remplacés par ceux d'autres instructions en cours de traitement mise en évidence dans la section 5.4.4.

6 Conclusion générale

6.1 Bilan du travail réalisé

Dans l'industrie des systèmes embarqués, la sécurité des composants est un élément essentiel afin de garantir la confidentialité des données sensibles qu'ils possèdent et pour la certification de ces composants. Dans ce contexte, l'objectif de cette thèse était de proposer des schémas de protection contre les attaques par faute, ciblant des parties sensibles de code, mis en œuvre à la compilation de façon à être déployés automatiquement tout en étudiant les difficultés inhérentes à cette automatisation dans le compilateur.

Dans le but de répondre à cette problématique, nous avons dans un premier temps présenté les différents moyens à disposition d'un attaquant pour pouvoir injecter des fautes ainsi que les effets qu'elles peuvent produire à différents niveaux de représentation (physique, circuit, micro-architecture, ISA ou applicatif). Nous avons également présenté l'état de l'art des schémas de protection et leur mise en œuvre visant à limiter ou empêcher de telles attaques. Si des protections matérielles existent et sont utilisées en pratique, elles manquent de flexibilité et ne suffisent pas à fournir une protection suffisante. Les contre-mesures logicielles offrent une solution plus souple mais souvent coûteuse. Par rapport à l'état de l'art de ces protections logicielles, nous avons choisi de nous positionner sur une approche de schémas légers mis en œuvre à la compilation pour limiter les coûts de deux façons : des schémas ciblés limitant l'ajout de code fournissent des codes sécurisés moins volumineux et plus performant, réduisant ainsi le coût du composant. Des schémas automatisés ajoutés à la compilation permettent un gain de temps de développement tout en limitant les erreurs liées aux approches manuelles, limitant ainsi les coûts de développement des produits sécurisés.

Dans le chapitre 3, nous avons présenté un premier schéma de contre-mesure (PLAF) visant à protéger les boucles. Les boucles sont des parties souvent sensibles des codes à sécuriser, notamment dans la cryptographie. Le schéma propose de garantir que le bon nombre d'itérations a été exécuté et que la bonne sortie de boucle a été prise en présence d'attaques par fautes modélisables au niveau ISA par un saut d'instruction ou une corruption de registre. En se concentrant sur les seules instructions impactant cette propriété, nous proposons un schéma à faible coût compatible avec les contraintes de performances et de ressources limitées des systèmes embarqués. Toutefois, la présence d'appels de fonction parmi ces instructions est un point limitant du schéma. Il ne peut être appliqué que partiellement dans ce cas, limitant d'autant la plage de protection du schéma. Implémenté dans la représentation intermédiaire

du compilateur LLVM, ce schéma générique peut être déployé de manière automatique uniquement sur les boucles que le développeur annote comme étant sensibles et ne dépend pas de l'architecture ciblée. Nous avons étudié les interactions entre l'insertion d'un schéma de protection à la compilation et le reste des optimisations réalisées par le compilateur. Nous en avons conclu qu'il est possible d'intégrer ce type de schéma au niveau IR du compilateur afin de garder cette indépendance vis à vis de l'architecture cible, mais que des modifications du *back-end* sont requises afin de ne pas altérer la protection apportée. Nous avons étudié, pour une architecture ARM, les modifications nécessaires et avons réalisé des expériences sur une cible implémentant le jeu d'instructions ARMv7 validant la sécurité apportée par un tel schéma. Néanmoins, l'utilisation sur d'autres cibles nécessiterait une étude approfondie du *back-end* associé afin de s'assurer que, dans ce changement de contexte, d'autres optimisations ne viennent pas impacter la sécurisation apportée.

Le chapitre 4 propose une seconde contre-mesure (*Calleesimo*) dédiée à la protection du graphe d'appel à la compilation. L'idée derrière ce schéma est de présenter une manière d'automatiser le traçage d'appels de fonction et de protéger la valeur de leurs arguments. Ce type de protection est souvent utilisé en pratique et déployé manuellement dans l'industrie, en particulier dans les séquences de démarrage des produits. Le schéma a été conçu autour de plusieurs variantes correspondant à des besoins de suivi variés que le développeur peut librement choisir à l'aide d'annotations sur le code source. L'élargissement de la plage de protection de la valeur des arguments de fonction nécessite une analyse du flot de contrôle, afin de déterminer ou placer le code nécessaire à la protection, qui profite pleinement des analyses disponibles dans le compilateur et qui semble délicat à déployer manuellement.

Les schémas de protection précédents ainsi que de nombreuses contre-mesures logicielles de la littérature considèrent des modèles de faute simples largement observés sur des architectures de micro-controlleurs présents dans l'univers de la carte à puce. Les récentes attaques contre des architectures plus complexes à base de processeurs superscalaires et multi-cœurs nous a poussé à étudier le comportement de telles architectures dans le contexte des attaques en faute. Le chapitre 5 présente des campagnes d'attaques, utilisant un banc d'injection d'impulsions électromagnétiques, qui nous ont permis de décrire des effets de ces fautes au niveau ISA sur un composant à base de processeur ARM Cortex-A9. Nous en avons déduit de nouveaux modèles de faute (substitution d'opérandes et ajout d'arc dans le CFG) et des combinaisons variées d'effets de modèles existants. Avec cet environnement, nous avons également constaté que malgré la possibilité d'observer des effets plus complexes que ceux définis pour la conception de la contre-mesure PLAF, une majorité de fautes ont été détectées même parmi celles aux effets complexes. Cependant, certaines fautes restent non détectées même en empilant les contre-mesures logicielles. Cette étude montre les limitations des contre-mesures actuelles pour les architectures complexes et la nécessité d'investiguer en profondeur et à différents niveaux les effets visibles d'attaques par faute sur ces plateformes de plus en plus ciblées par ces attaques.

6.2 Perspectives

L'implémentation de deux contre-mesures au niveau de la représentation intermédiaire du compilateur LLVM a mis en avant la difficulté d'utiliser l'environnement de compilation pour ajouter de la sécurité. Les différentes propriétés de sécurité que les schémas de protection cherchent à préserver ne font pas partie de la sémantique du code que le compilateur se doit de conserver. Sans intégrer dans la totalité du flot de compilation les contraintes liées à ces propriétés de sécurité, il est difficile de s'assurer que les optimisations en aval ne compromettent pas les modifications apportées.

Une première possibilité serait d'utiliser au mieux les informations de debug disponibles pour y transporter les informations nécessaires à la sécurisation sans la compromettre. Cela nécessite toutefois de pouvoir et devoir agir au niveau des principales passes d'optimisation des différents *back-end* selon les architectures ciblées afin de traiter ces informations transmises et limiter ou éviter les interactions. Ce thème fait l'objet d'une thèse en cours d'étude par Son Tuan Vu [1]. Ces informations peuvent être utilisées pour vérifier la présence des contre-mesures dans le code binaire, pour des simulations ou encore pour des outils de vérification formelle. Toutefois, l'utilisation d'un compilateur classique nécessite une étape de vérification de la présence des protections dans le code final. La vérification de protections étant également une étape coûteuse dans le développement, des outils d'analyse et des méthodes de vérifications automatiques sont autant d'éléments importants à considérer qui font l'objet de recherches comme la thèse de Jean-Baptiste Bréjon [2]

À plus long terme, la conception d'une infrastructure de compilation orientée sécurité qui intègre nativement la gestion de ces propriétés de sécurité serait une piste intéressante. En effet, cela éviterait de devoir modifier l'ensemble des passes existantes d'un compilateur mais nécessiterait un travail conséquent, probablement similaire à la conception d'une infrastructure comme LLVM qui, après 15 ans de développement, a aujourd'hui remplacé GCC dans une majorité d'applications industrielles. À défaut de telle infrastructure, les deux contre-mesures proposées peuvent être considérées comme deux éléments d'un compilateur orienté sécurité mais nécessitant d'être attentif aux interactions lors de l'ajout d'une nouvelle contre-mesure. D'autres schémas pourrait ainsi être intégrés de façon similaire afin de fournir au développeur un panel de sécurisation possible. Cela permettrait de pouvoir, de manière automatisée, adapter différentes contre-mesures aux types de codes ou aux différents modèles d'attaquant selon les codes. Il est ainsi possible de considérer des modèles d'attaquant différents par blocs, selon les parties du code à sécuriser qui ne requièrent pas toujours le même niveau de protection. Toutefois, les deux contre-mesures proposées se complètent et ne rentrent pas en conflit, ce qui n'est pas toujours vrai pour des contre-mesures quelconques. Dans le cas de multiples contre-mesures, il faut donc étudier le positionnement des différentes passes

1. https://www.lip6.fr/actualite/personnes-fiche.php?ident=D2144
2. https://www.lip6.fr/actualite/personnes-fiche.php?ident=D1770

de sécurisation dans le flot de compilation, notamment lorsqu'elles visent le même code, le but étant de s'assurer que le code généré possède les protections et si possible éviter une sur-protection coûteuse de certains parties de code.

Comme cela a été montré dans cette thèse, la compréhension et le choix des modèles de faute considérés sont importants pour la conception des contre-mesures. Sur des architectures complexes, nous avons déterminé de nouveaux modèles de faute en se basant sur une analyse fine de codes simples méthodiquement écrits. Néanmoins, l'accès au banc d'attaque étant limité, nous n'avons pas pu compléter cette analyse avec des codes de test utilisant d'autres types d'instruction. Une caractérisation précise des accès mémoire et des branchements fournirait sûrement de nouvelles données essentielles à la compréhension approfondie des effets des fautes sur ces architectures complexes. Une caractérisation d'effets provenant d'autres moyens d'injection compléterait également cette analyse. Ces architectures complexes sont de plus en plus amenées à être sécurisées, il est donc urgent de mener ces efforts de caractérisation. Les modèles de fautes étant plus complexes sur ces architectures, il semble également nécessaire d'adapter les schémas de protections existants. Enfin, il convient de proposer des solutions satisfaisantes lorsque l'ajout de contre-mesures logicielles devient trop coûteux. Notre expérience préliminaire indique que ce pourrait bien être d'autant plus le cas que les architectures sont complexes. Se pose alors la question de la complémentarité entre contre-mesures logicielles et contre-mesures matérielles. En effet, des schémas mixtes logiciel-matériel peuvent apporter de bons compromis en termes de sécurité et de coût. Cette double sécurisation logicielle-matérielle est la méthode que nous recommandons pour le déploiement de contremesures pour les composants de l'IoT.

Bibliographie

[Abadi et al., 2005] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. (2005). Control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*, pages 340–353. (Citations pages 76 et 77.)

[Aciiçmez et al., 2007] Aciiçmez, O., Gueron, S., and Seifert, J.-P. (2007). New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, Cryptography and Coding'07, pages 185–203, Berlin, Heidelberg. Springer-Verlag. (Citation page 8.)

[Agosta et al., 2013] Agosta, G., Barenghi, A., Maggi, M., and Pelosi, G. (2013). Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. (Citation page 30.)

[Agoyan et al., 2010] Agoyan, M., Dutertre, J.-M., Naccache, D., Robisson, B., and Tria, A. (2010). When clocks fail : On critical paths and clock faults. In Gollmann, D., Lanet, J.-L., and Iguchi-Cartigny, J., editors, *Smart Card Research and Advanced Application*, pages 182–193, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 11.)

[Akkar et al., 2003] Akkar, M.-L., Goubin, L., and Ly, O. (2003). Automatic integration of counter-measures against fault injection attacks. (Citation page 21.)

[ANSSI, 2014] ANSSI (2014). Référentiel général de sécurité. https://www.ssi.gouv.fr/uploads/2014/11/RGS_v-2-0_B1.pdf. (Citation page 75.)

[Aumüller et al., 2003] Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., and Seifert, J.-P. (2003). Fault attacks on rsa with crt : Concrete results and practical countermeasures. In Kaliski, B. S., Koç, ç. K., and Paar, C., editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 260–275, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 11.)

[Balakrishnan et al., 2008] Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T. (2008). *WYSINWYX : What you see is not what you execute*, volume 32, pages 202–213. (Citation page 3.)

[Balasch et al., 2011] Balasch, J., Gierlichs, B., and Verbauwhede, I. (2011). An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 105–114. IEEE. (Citations pages 11, 14 et 15.)

[Bar-El et al., 2006] Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., and Whelan, C. (2006). The sorcerer's apprentice guide to fault attacks. *Proc. of the IEEE*, 94(2) :370–382. (Citations pages 10, 17, 18 et 120.)

[Barenghi et al., 2012] Barenghi, A., Breveglieri, L., Koren, I., and Naccache, D. (2012). Fault injection attacks on cryptographic devices : Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11) :3056–3076. (Citation page 14.)

[Barenghi et al., 2010] Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., and Regazzoni, F. (2010). Countermeasures against fault attacks on software implemented AES. In *5th Workshop on Embedded Systems Security*, WESS '10, pages 7 :1–7 :10. ACM. (Citations pages 18 et 19.)

[Barry, 2017] Barry, T. (2017). *Sécurisation à la compilation de logiciels contre les attaques en fautes*. PhD thesis, Université de Lyon. (Citation page 76.)

[Barry et al., 2016] Barry, T., Couroussé, D., and Robisson, B. (2016). Compilation of a countermeasure against instruction-skip fault attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, CS2 '16, pages 1–6. (Citations pages 15, 29, 64, 65 et 129.)

[Bayrak et al., 2015] Bayrak, A. G., Regazzoni, F., Novo, D., Brisk, P., Standaert, F.-X., and Ienne, P. (2015). Automatic application of power analysis countermeasures. *IEEE Trans. Comp.*, 64(2) :329–341. (Citation page 29.)

[Berthomé et al., 2012] Berthomé, P., Heydemann, K., Kauffmann-Tourkestansky, X., and Lalande, J. (2012). High level model of control flow attacks for smart card functional security. In *2012 Seventh International Conference on Availability, Reliability and Security*, pages 224–229. (Citations pages 14 et 15.)

[Biehl et al., 2000] Biehl, I., Meyer, B., and Müller, V. (2000). Differential Fault Attacks on Ellitic Curve Cryptosystems. In Bellare, M., editor, *Advances in Cryptology (CRYPTO 2000)*, volume 1880 of *LNCS*, pages 131–146. Springer. (Citation page 75.)

[Biham and Shamir, 1999] Biham, E. and Shamir, A. (1999). Differential fault analysis of secret key cryptosystems. *Proceedings of Advances in Cryptology - CRYPTO 1997*, 1294. (Citation page 14.)

[Bletsch et al., 2011] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. (2011). Jump-oriented programming : A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA. ACM. (Citation page 76.)

[Blömer et al., 2014] Blömer, J., Gomes da Silva, R., Günther, P., Kramer, J., and Seifert, J.-P. (2014). A practical second-order fault attack against a real-world pairing implementation. *Proceedings - 2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014*, pages 123–136. (Citation page 75.)

[Blömer and Seifert, 2003] Blömer, J. and Seifert, J.-P. (2003). Fault based cryptanalysis of the advanced encryption standard (aes). In Wright, R. N., editor, *Financial Cryptography*, pages 162–181, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 16.)

[Bogdanov et al., 2007] Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J. B., Seurin, Y., and Vikkelsoe, C. (2007). Present : An ultra-lightweight block cipher. In Paillier, P. and Verbauwhede, I., editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 75.)

[Boneh et al., 1997] Boneh, D., DeMillo, R. A., and Lipton, R. J. (1997). On the importance of checking cryptographic protocols for faults. In Fumy, W., editor, *Advances in Cryptology — EUROCRYPT*, pages 37–51. Springer. (Citations pages 1, 8 et 16.)

[Boneh et al., 2001] Boneh, D., DeMillo, R. A., and Lipton, R. J. (2001). On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14 :101–119. (Citation page 8.)

[Breier and Jap, 2015] Breier, J. and Jap, D. (2015). Testing feasibility of back-side laser fault injection on a microcontroller. In *Proceedings of the WESS'15 : Workshop on Embedded Systems Security*, WESS'15, pages 5 :1–5 :6, New York, NY, USA. ACM. (Citation page 13.)

[C. May and H. Woods, 1978] C. May, T. and H. Woods, M. (1978). A new physical mechanism for soft errors in dynamic memories. In *16th International Reliability Physics Symposium*, pages 33–40. (Citation page 8.)

[Chen et al., 2017] Chen, Z., Shen, J., Nicolau, A., Veidenbaum, A., Ghalaty, N. F., and Cammarota, R. (2017). Camfas : A compiler approach to mitigate fault attacks via enhanced simdization. Cryptology ePrint Archive, Report 2017/1083. https://eprint.iacr.org/2017/1083. (Citations pages 19 et 30.)

[Ciet and Joye, 2005] Ciet, M. and Joye, M. (2005). Elliptic curve cryptosystems in the presence of permanent and transient faults. *Designs, Codes and Cryptography*, 36(1) :33–43. (Citation page 19.)

[Clark et al., 2007] Clark, N., Hormati, A., Yehia, S., Mahlke, S., and Flautner, K. (2007). Liquid simd : Abstracting simd hardware using lightweight dynamic mapping. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 216–227. (Citation page 30.)

[Cojocar et al., 2018] Cojocar, L., Papagiannopoulos, K., and Timmers, N. (2018). *Instruction Duplication : Leaky and Not Too Fault-Tolerant !*, pages 160–179. (Citation page 29.)

[Colombier et al., 2018] Colombier, B., Menu, A., Dutertre, J., Moëllic, P., Rigaud, J., and Danger, J. (2018). Laser-induced single-bit faults in flash memory : Instructions corruption on a 32-bit microcontroller. *IACR Cryptology ePrint Archive*, 2018 :1042. (Citations pages 14 et 15.)

[Criteria, 2017] Criteria, C. (2017). Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 5. https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R5.pdf. (Citations pages 2 et 35.)

[de Clercq and Verbauwhede, 2017] de Clercq, R. and Verbauwhede, I. (2017). A survey of hardware-based control flow integrity (CFI). *CoRR*, abs/1706.07257. (Citation page 76.)

[De Keulenaer et al., 2015] De Keulenaer, R., Maebe, J., De Bosschere, K., and De Sutter, B. (2015). Link-time smart card code hardening. *International Journal of Information Security*, pages 1–20. (Citations pages 22, 36 et 76.)

[Dehbaoui et al., 2012a] Dehbaoui, A., Dutertre, J., Robisson, B., Orsatelli, P., Maurine, P., and Tria, A. (2012a). Injection of transient faults using electromagnetic pulses – practical results on a cryptographic system. Cryptology ePrint Archive, Report 2012/123. (Citation page 12.)

[Dehbaoui et al., 2012b] Dehbaoui, A., Dutertre, J., Robisson, B., and Tria, A. (2012b). Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15. (Citation page 12.)

[Dehbaoui et al., 2013] Dehbaoui, A., Mirbaha, A., Moro, N., Dutertre, J., and Tria, A. (2013). Electromagnetic glitch on the AES round counter. In *COSADE*. (Citations pages 3, 8 et 36.)

[Dehbaoui et al., 2010] Dehbaoui, A., Ordas, T., Lomné, V., Maurine, P., Torres, L., and Robert, M. (2010). Incoherence analysis and its application to time domain em analysis of secure circuits. (Citation page 8.)

[Demirci and Selçuk, 2008] Demirci, H. and Selçuk, A. A. (2008). A meet-in-the-middle attack on 8-round AES. In *Fast Software Encryption : 15th International Workshop, FSE*, pages 116–126. (Citation page 36.)

[Dottax et al., 2009] Dottax, E., Giraud, C., Rivain, M., and Sierra, Y. (2009). On second-order fault analysis resistance for CRT-RSA implementations. In *Third IFIP WG 11.2 International Workshop on Information Security Theory and Practice*, pages 68–83. (Citation page 35.)

[Dugardin et al., 2016] Dugardin, M., Guilley, S., Danger, J.-L., Najm, Z., and Rioul, O. (2016). Correlated extra-reductions defeat blinded regular exponentiation. In *Proceedings of the 18th International Conference on Cryptographic Hardware and Embedded Systems — CHES 2016 - Volume 9813*, pages 3–22, New York, NY, USA. Springer-Verlag New York, Inc. (Citation page 8.)

[Dureuil et al., 2016a] Dureuil, L., Petiot, G., Potet, M., Le, T., Crohen, A., and de Choudens, P. (2016a). FISSC : A fault injection and simulation secure collection. In *SAFECOMP*, pages 3–11. (Citations pages 8, 16, 36, 37, 50, 70 et 116.)

[Dureuil et al., 2016b] Dureuil, L., Potet, M.-L., de Choudens, P., Dumas, C., and Clédière, J. (2016b). From code review to fault injection attacks : Filling the gap using fault model inference. In *Smart Card Research and Advanced Applications Conference (CARDIS)*, pages 107–124. Springer. (Citations pages 10, 12 et 110.)

[Dutertre et al., 2014] Dutertre, J., Castro, S. D., Sarafianos, A., Boher, N., Rouzeyre, B., Lisart, M., Damiens, J., Candelier, P., Flottes, M., and Natale, G. D. (2014). Laser attacks on integrated circuits : From cmos to fd-soi. In *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. (Citation page 12.)

[Dutertre et al., 2011] Dutertre, J., Fournier, J. J. A., Mirbaha, A., Naccache, D., Rigaud, J., Robisson, B., and Tria, A. (2011). Review of fault injection mechanisms and consequences on countermeasures design. In *2011 6th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. (Citations pages 10 et 13.)

[Eide and Regehr, 2008] Eide, E. and Regehr, J. (2008). Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 255–264, New York, NY, USA. ACM. (Citation page 21.)

[El-Baze et al., 2016] El-Baze, D., Rigaud, J., and Maurine, P. (2016). A fully-digital EM pulse detector. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 439–444. (Citation page 18.)

[El Mrabet, 2009] El Mrabet, N. (2009). *What about vulnerability to a fault attack of the Miller's algorithm during an identity based protocol ?*, pages 122–134. Springer, Berlin, Heidelberg. (Citation page 36.)

[Espitau et al., 2018] Espitau, T., Fouque, P., Gérard, B., and Tibouchi, M. (2018). Loop-abort faults on lattice-based signature schemes and key exchange protocols. *IEEE Trans. Computers*, 67(11) :1535–1549. (Citation page 36.)

[Genkin et al., 2017] Genkin, D., Shamir, A., and Tromer, E. (2017). Acoustic cryptanalysis. *Journal of Cryptology*, 30(2) :392–443. (Citation page 8.)

[Goloubeva et al., 2005] Goloubeva, O., Rebaudengo, M., Reorda, M. S., and Violante, M. (2005). Improved software-based processor control-flow errors detection technique. In *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, pages 583–589. (Citation page 76.)

[Guillen et al., 2017] Guillen, O. M., Gruber, M., and De Santis, F. (2017). Low-cost setup for localized semi-invasive optical fault injection attacks. In Guilley, S., editor, *Constructive Side-Channel Analysis and Secure Design*, pages 207–222, Cham. Springer International Publishing. (Citation page 10.)

[Guthaus et al., 2001] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). Mibench : A free, commercially representative embedded benchmark suite. In *IEEE*, WWC '01, pages 3–14. (Citation page 62.)

[He et al., 2016] He, W., Jakub, B., and Bhasin, S. (2016). Cheap and cheerful : A low-cost digital sensor for detecting laser fault injection attacks. pages 27–46. (Citation page 18.)

[Hillebold, 2014] Hillebold, C. (2014). *Compiler-Assisted Integrity against Fault Injection Attacks*. Master's thesis, Graz university of Technology. (Citation page 19.)

[Holloway and Smith, 2000] Holloway, G. and Smith, M. D. (2000). An extender's guide to the optimization programming interface and target descriptions. the machine-suif documentation set. Technical report, Harvard University. (Citation page 55.)

[Hutter and Schmidt, 2014] Hutter, M. and Schmidt, J. (2014). The temperature side channel and heating fault attacks. *IACR Cryptology ePrint Archive*, 2014 :190. (Citation page 12.)

[Karaklajic et al., 2013] Karaklajic, D., Schmidt, J.-M., and Verbauwhede, I. (2013). Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12) :2295–2306. (Citation page 35.)

[Kelly et al., 2017] Kelly, M. S., Mayes, K., and Walker, J. F. (2017). Characterising a cpu fault attack model via run-time data analysis. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 79–84. (Citation page 121.)

[Kocher et al., 1999] Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Advances in Cryptology — CRYPTO' 99*, volume 1666 of *LNCS*, pages 388–397. Springer. (Citations pages 1 et 8.)

[Kocher, 1996] Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Koblitz, N., editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 8.)

[Lac et al., 2016] Lac, B., Beunardeau, M., Canteaut, A., Fournier, J. J.-A., and Sirdey, R. (2016). A First DFA on PRIDE : from Theory to Practice. In *The 11th International Conference on Risks and Security of Internetand Systems - CRISIS 2016*, Lectures Notes in Computer Science, Roscoff, France. (Citation page 16.)

[Lac et al., 2017] Lac, B., Canteaut, A., Fournier, J., and Sirdey, R. (2017). Dfa on ls-designs with a practical implementation on scream. pages 223–247. (Citation page 16.)

[Lalande et al., 2014] Lalande, J.-F., Heydemann, K., and Berthomé, P. (2014). Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In *Computer Security - ESORICS*, volume 8713 of *LNCS*, pages 200–218. Springer. (Citations pages 19, 21, 37 et 76.)

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). Llvm : A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86. (Citations pages 3 et 25.)

[Majéric, 2018] Majéric, F. (2018). *Etude d'attaques matérielles et combinées sur les « System-on-Chip »*. Theses, Université Jean Monnet. (Citation page 111.)

[Majéric et al., 2016] Majéric, F., Bourbao, E., and Bossuet, L. (2016). Electromagnetic security tests for SoC. In *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 265–268. (Citation page 12.)

[Mohamed et al., 2011] Mohamed, M. S. E., Bulygin, S., and Buchmann, J. (2011). Using sat solving to improve differential fault analysis of trivium. In Kim, T.-h., Adeli, H., Robles, R. J., and Balitanas, M., editors, *Information Security and Assurance*, pages 62–71, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 16.)

[Morin, 2006] Morin, H. (2006). Les puces ne garantissent pas la sécurité des échanges en ligne. *Le Monde*. (Citation page 8.)

[Moro, 2014] Moro, N. (2014). *Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués*. PhD thesis, UPMC, Paris, France. (Citations pages 15, 19, 20, 22, 34, 35, 37, 70, 107 et 121.)

[Moro et al., 2013] Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., and Encrenaz, E. (2013). Electromagnetic fault injection : Towards a fault model on a 32-bit microcontroller. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. (Citations pages 12, 14, 15, 68, 105, 109 et 131.)

[Moro et al., 2014] Moro, N., Heydemann, K., Encrenaz, E., and Robisson, B. (2014). Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3) :145–156. (Citations pages 35, 64 et 65.)

[Mrabet, 2013] Mrabet, N. E. (2013). Side channel attacks against pairing over theta functions. Cryptology ePrint Archive, Report 2013/386. http://eprint.iacr.org/2013/386. (Citation page 36.)

[Muller, 2003] Muller, F. (2003). *A New Attack against Khazad*, pages 347–358. Springer, Berlin, Heidelberg. (Citation page 36.)

[Nashimoto et al., 2016] Nashimoto, S., Homma, N., Hayashi, Y.-i., Takahashi, J., Fuji, H., and Aoki, T. (2016). Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering*. (Citations pages 3, 9, 36 et 115.)

[NXP, 2018] NXP (2018). i.mx 6dual/6quad applications processor for consumer products. https://www.nxp.com/docs/en/data-sheet/IMX6DQCEC.pdf. (Citation page 133.)

[Oh et al., 2002] Oh, N., Shirvani, P. P., and McCluskey, E. J. (2002). Control-flow checking by software signatures. *IEEE Trans. Reliability*, 1(51) :111–122. (Citations pages 19, 37, 76 et 79.)

[Ordas et al., 2015] Ordas, S., Guillaume-Sage, L., Tobich, K., Dutertre, J.-M., and Maurine, P. (2015). *Evidence of a Larger EM-Induced Fault Model*, pages 245–259. Springer. (Citation page 12.)

[Page and Vercauteren, 2006] Page, D. and Vercauteren, F. (2006). A fault attack on pairing-based cryptography. *IEEE Trans. Comp.*, 55(9) :1075–1080. (Citation page 36.)

[Patranabis et al., 2017] Patranabis, S., Chakraborty, A., and Mukhopadhyay, D. (2017). Fault tolerant infective countermeasure for aes. *Journal of Hardware and Systems Security*, 1(1) :3–17. (Citation page 17.)

[Patrick et al., 2017] Patrick, C., Yuce, B., Farhady Ghalaty, N., and Schaumont, P. (2017). Lightweight fault attack resistance in software using intra-instruction redundancy. pages 231–244. (Citation page 19.)

[Proy et al., 2017] Proy, J., Heydemann, K., Berzati, A., and Cohen, A. (2017). Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Transactions on Architecture and Code Optimization*, 14(4) :36. (Citation page 34.)

[Put et al., 2005] Put, L. V., Chanet, D., Bus, B. D., Sutter, B. D., and Bosschere, K. D. (2005). Diablo : a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, pages 7–12. (Citation page 22.)

[Quisquater and Samyde, 2002] Quisquater, J.-J. and Samyde, D. (2002). Eddy current for magnetic analysis with active sensor. (Citation page 12.)

[Ramalingam, 1999] Ramalingam, G. (1999). Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.*, 21(2) :175–188. (Citation page 39.)

[Rastello and Bouchez-Tichadou, 2019] Rastello, F. and Bouchez-Tichadou, F. (2019). Ssa-based compiler design. (Citation page 25.)

[Reis et al., 2005] Reis, G., Chang, J., Vachharajani, N., Rangan, R., and August, D. (2005). SWIFT : Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254. (Citations pages 37, 64, 65, 71, 74, 77, 115 et 120.)

[Rivière et al., 2015] Rivière, L., Najm, Z., Rauzy, P., Danger, J.-L., Bringer, J., and Sauvage, L. (2015). High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. (Citations pages 15, 29 et 137.)

[Roche et al., 2011] Roche, T., Lomné, V., and Khalfallah, K. (2011). Combined fault and side-channel attack on protected implementations of aes. In Prouff, E., editor, *Smart Card Research and Advanced Applications*, pages 65–83, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 17.)

[Roscian et al., 2013a] Roscian, C., Dutertre, J., and Tria, A. (2013a). Frontside laser fault injection on cryptosystems - application to the aes' last round -. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 119–124. (Citation page 68.)

[Roscian et al., 2013b] Roscian, C., Sarafianos, A., Dutertre, J.-M., and Tria, A. (2013b). Fault model analysis of laser-induced faults in sram memory cells. In *Proceedings of the 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC '13, pages 89–98, Washington, DC, USA. IEEE Computer Society. (Citation page 68.)

[Schmidt and Hutter, 2007] Schmidt, J.-M. and Hutter, M. (2007). Optical and EM fault-attacks on CRT-based RSA : Concrete results. In *15th Austrian Workhop on Microelectronics (Austrochip 2007)*. (Citation page 12.)

[Shacham, 2007] Shacham, H. (2007). The geometry of innocent flesh on the bone : Return-into-libc without function calls (on the x86). In De Capitani di Vimercati, S. and Syverson, P., editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press. (Citation page 76.)

[Skorobogatov, 2009] Skorobogatov, S. (2009). Local heating attacks on flash memory devices. pages 1–6. (Citation page 12.)

[Skorobogatov and Anderson, 2003] Skorobogatov, S. P. and Anderson, R. J. (2003). Optical fault induction attacks. In Kaliski, B. S., Koç, ç. K., and Paar, C., editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 2–12, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 12.)

[Soucarros et al., 2011] Soucarros, M., Canovas-Dumas, C., Clédière, J., Elbaz-Vincent, P., and Réal, D. (2011). Influence of the temperature on true random number generators. pages 24–27. (Citation page 12.)

[Spruyt, 2012] Spruyt, A. (2012). Building fault models for microcontrollers. (Citation page 121.)

[Timmers and Mune, 2017] Timmers, N. and Mune, C. (2017). Escalating privileges in linux using voltage fault injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8. (Citations pages 11, 32 et 108.)

[Timmers and Spruyt, 2016] Timmers, N. and Spruyt, A. (2016). Bypassing secure boot using fault injection. Black Hat Europe. (Citations pages 4, 8, 9, 16, 70 et 75.)

[Timmers et al., 2016] Timmers, N., Spruyt, A., and Witteman, M. (2016). Controlling PC on ARM using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. (Citations pages 15, 16, 108 et 117.)

[Trichina and Korkikyan, 2010] Trichina, E. and Korkikyan, R. (2010). Multi fault laser attacks on protected crt-rsa. In *Proceedings of the 2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC '10, pages 75–86, Washington, DC, USA. IEEE Computer Society. (Citations pages 14 et 75.)

[Tunstall et al., 2011] Tunstall, M., Mukhopadhyay, D., and Ali, S. (2011). Differential fault analysis of the advanced encryption standard using a single fault. In Ardagna, C. A. and Zhou, J., editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 224–233, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 16.)

[Tupsamudre et al., 2014] Tupsamudre, H., Bisht, S., and Mukhopadhyay, D. (2014). Differential fault analysis on the families of simon and speck ciphers. pages 40–48. (Citation page 16.)

[van Woudenberg et al., 2011] van Woudenberg, J. G. J., Witteman, M. F., and Menarini, F. (2011). Practical optical fault injection on secure microcontrollers. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 91–99. (Citation page 16.)

[Vasselle et al., 2017] Vasselle, A., Thiebeauld, H., Maouhoub, Q., Morisset, A., and Ermeneux, S. (2017). Laser-induced fault injection on smartphone bypassing the secure boot. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 41–48. (Citations pages 1, 4, 9, 16, 32, 62 et 108.)

[Vendicator, 2000] Vendicator (2000). A s̈tack smashingẗechnique protection tool for linux. (Citations pages 76 et 77.)

[Verbauwhede et al., 2011] Verbauwhede, I., Karaklajic, D., and Schmidt, J. (2011). The fault attack jungle - a classification model to guide you. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 3–8. (Citation page 13.)

[Viera et al., 2018] Viera, R. A. C., Dutertre, J., Maurine, P., and Bastos, R. P. (2018). Standard CAD tool-based method for simulation of laser-induced faults in large-scale circuits. In *Proceedings of the 2018 International Symposium on Physical Design, ISPD 2018, Monterey, CA, USA, March 25-28, 2018*, pages 160–167. (Citation page 12.)

[Vigilant, 2008] Vigilant, D. (2008). Rsa with crt : A new cost-effective solution to thwart fault attacks. In Oswald, E. and Rohatgi, P., editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, pages 130–145, Berlin, Heidelberg. Springer Berlin Heidelberg. (Citation page 19.)

[Yen and Joye, 2000] Yen, S.-M. and Joye, M. (2000). Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9) :967–970. (Citation page 16.)

[Yuce et al., 2016] Yuce, B., Ghalaty, N. F., Santapuri, H., Deshpande, C., Patrick, C., and Schaumont, P. (2016). Software fault resistance is futile : Effective single-glitch attacks. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58. (Citations pages 15, 35, 109 et 128.)

[Yuce et al., 2018] Yuce, B., Schaumont, P., and Witteman, M. (2018). Fault attacks on secure embedded software : Threats, design, and evaluation. *Journal of Hardware and Systems Security*, 2(2) :111–130. (Citations pages 9, 10 et 155.)

[Zussa et al., 2014] Zussa, L., Dehbaoui, A., Tobich, K., DUTERTRE, J.-M., Maurine, P., Guillaume-Sage, L., Clédière, J., and Tria, A. (2014). Efficiency of a glitch detector against electromagnetic fault injection. In *DATE : Design, Automation and Test in Europe*, Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, pages 1–6, Dresden, Germany. IEEE. (Citation page 18.)

[Zussa et al., 2012] Zussa, L., Dutertre, J.-M., Clédière, J., Robisson, B., and Tria, A. (2012). Investigation of timing constraints violation as a fault injection means. (Citation page 11.)

Table des figures

2.1 Vue d'ensemble des effets d'une faute aux différents niveaux ([Yuce et al., 2018]) 10
2.2 Taille comparative du faisceau laser par rapport à la technologie ciblée 13
2.3 Flot de compilation et représentations du code associées 24
2.4 Structure modulaire du compilateur LLVM 26
2.5 Transformations d'un code source (1) par le *front-end* en IR (2), optimisé par le *middle-end* au niveau d'optimisation -Os (3), transformé par le *back-end* pour architectures implémentant l'ISA ARMv7 en code assembleur (4) 28

3.1 Exemple de boucle avec plusieurs blocs sortants et plusieurs blocs de sortie (à gauche), et sa version sécurisée (à droite) 40
3.2 Schéma présentant les blocs du CFG contenant des branchements à sécuriser en fonction d'un nœud ϕ 44
3.3 Les deux plateformes embarquées cibles des évaluations des contre-mesures . 63
3.4 Répartition des boucles en fonction de la réussite de la sécurisation pour les options de compilation `-O1`, `-O2`, `-O3`, `-Os` et `-Oz` 63
3.5 Performance relative du code sécurisé par rapport au code original (options de compilation `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`) 65
3.6 Taille relative du code sécurisé par rapport au code original (options de compilation `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`) 66

4.1 Exemple de mise à jour des signatures côté appelant (bleu) et côté appelé (rouge) 79
4.2 Position des mises à jour des variables pour les modes *CallCount* et *CallSequence* 83
4.3 Position des mises à jour des variables pour le mode *CallRetSequence* 84
4.4 Exemple de flot de contrôle présentant les arcs sur lesquels ajouter des mises à jour pour une protection de la valeur de l'argument depuis sa définition . . . 90
4.5 Exemples de parcours dans la fonction présentée en figure 4.4 avec le nombre de mises à jour de variables associées 91
4.6 Exemple de flot de contrôle présentant les arcs sur lesquels ajouter des mises à jour pour une protection de la valeur de l'argument jusqu'à ses utilisations . . 93
4.7 Impact du suivi des appels de fonction sur les performances en fonction du pourcentage de fonctions à sécuriser (`-Os`, ratio entre code sécurisé et code original) 101

4.8 Impact de la sécurisation des arguments sur les performances en fonction du pourcentage d'arguments à sécuriser (`-Os`, ratio entre code sécurisé et code original) . . . 102
4.9 Impact du suivi des appels de fonction sur la taille de code en fonction du pourcentage de fonctions à sécuriser (`-Os`, ratio entre code sécurisé et code original) . . . 102
4.10 Impact de la sécurisation des arguments sur la taille de code en fonction du pourcentage d'arguments à sécuriser (`-Os`) . . . 103

5.1 Photographie du SoC choisi pour les expérimentations . . . 110
5.2 Schéma du banc d'attaque électromagnétique . . . 111
5.3 La sonde de mesure et la sonde d'injection utilisées . . . 111
5.4 Proportion de non-réponse (inexploitable) du composant ciblé après injection électromagnétique en fonction du positionnement en x et y de la sonde d'injection 112
5.5 Trace électromagnétique d'injection d'impulsion EM lors de l'exécution d'une boucle et signal GPIO correspondant . . . 113
5.6 Répartition des fautes injectées sur boucles sécurisées . . . 120
5.7 Nombre de fautes affectant chaque registre (`r0` à `r9`) . . . 126
5.8 Distribution du nombre de registres corrompus simultanément . . . 126
5.9 Flot de contrôle de la fonction *Boucle2-plaf* avec les différents arc magiques observés . . . 132
5.10 Distribution du nombre de registres corrompus simultanément pour la même campagne d'attaques sur le code de la *boucle1-plaf* jouée deux fois . . . 133
5.11 Répartition des fautes réussies selon les modèles de fautes et multiplicité des effets (avec le nombre total de fautes réussies) pour chaque campagne d'attaques sur boucle . . . 134
5.12 Répartition des fautes exploitables non détectées selon les modèles de la classification expliquant les résultats fautés . . . 135

Liste des tableaux

3.1 Temps dédié à différentes passes en pourcentage du temps de compilation total 66
3.2 Résultats des simulations de faute ciblant 3 applications cryptographiques et pour chacune la version originale et celle sécurisée 69

4.1 Définition des régions de protections selon le mode 86

5.1 Classification des résultats d'injection de fautes électromagnétiques 118
5.2 Exemples de valeurs de registre corrompus obtenues après injection de faute . 127
5.3 Nombre moyen de registres corrompus par injection en fonction des instructions insérées entre chaque addition du listing 5.7 129
5.4 Nombre d'instructions dans la boucle et la fonction pour chaque version de *Boucle2* du listing 5.4 134
5.5 Valeurs obtenues lors de cas spécifiques à effets complexes corrélés 138

RÉSUMÉ

La sécurité des systèmes embarqués contenant des données sensibles est un enjeu crucial. La disponibilité de ces objets en fait une cible privilégiée pour les attaques physiques, nécessitant l'ajout de protections matérielles et logicielles. La recherche de réduction des coûts de développement pousse les industriels à opter pour du déploiement automatique de protections. L'objet de la thèse consiste à étudier l'intégration de contre-mesures logicielles contre les attaques par faute dans les outils de développement, en particulier dans le compilateur, afin d'automatiser l'application de contre-mesures variées. Pour cela, nous proposons deux schémas de protection génériques et automatiquement déployables contre ces attaques : un dédié à la sécurisation des boucles et le deuxième à la sécurisation du graphe d'appel. Ces schémas spécifiques, intégrés dans un même compilateur (LLVM) permettent la sécurisation de parties sensibles et choisies du code limitant ainsi leur surcoût en performances. Les fautes exploitables variant d'un composant à l'autre, nous proposons également une caractérisation des effets des fautes au niveau du jeu d'instructions sur une plateforme intégrant un processeur superscalaire typique des téléphones mobiles. Ces travaux montrent la nécessité d'étudier les injections de faute sur des plateformes complexes, de concevoir de nouveaux schémas de protection adaptés, et de continuer à intégrer dans un même compilateur plus de schémas de sécurisation.

MOTS CLÉS

compilateur, logiciel embarqué, attaques par canaux auxiliares, cryptographie.

ABSTRACT

The security of embedded systems containing sensitive data has become a main concern. These widely deployed devices are subject to physcial attacks, requiring protections both in hardware and software. The race for higher productivity and shorter time to market in the deployment of secure systems pushes for automatic solutions. This thesis studies the integration of software countermeasures against fault attacks in development tools, with a special focus on the compiler. The goal is to enable the automatic application, at compilation time, of a wide range of countermeasures. We propose two protection schemes against these attacks which can be automatically deployed: one scheme dedicated to loop control flow and the second dedicated to the protection of the call graph. These schemes, integrated in the LLVM compiler framework, allow to focus security application on sensitive areas of the targeted code, thus limitating the overhead. Faults that can be exploited are different from a device to another, we thus also provide an ISA-level characterization of fault effects on a superscalar processor representative of mobile phones. This work highlights the need of studying fault effects on more complex platforms, leading to the design of new protection schemes and automating their compilation-time application.

KEYWORDS

compiler, embedded applications, side-channel attacks, cryptography.